Technical Report

UCAM-CL-TR-670
ISSN 1476-2986

Number 670

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

On the application of program analysis and transformation to high reliability hardware

Sarah Thompson

July 2006

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 2006 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St Edmund's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Abstract

Safety- and mission-critical systems must be both *correct* and *reliable*. Electronic systems must behave as intended and, where possible, do so at the first attempt – the fabrication costs of modern VLSI devices are such that the iterative design/code/test methodology endemic to the software world is not financially feasible. In aerospace applications it is also essential to establish that systems will, with known probability, remain operational for extended periods, despite being exposed to very low or very high temperatures, high radiation, large G-forces, hard vacuum and severe vibration.

Hardware designers have long understood the advantages of formal mathematical techniques. Notably, model checking and automated theorem proving both gained acceptance within the electronic design community at an early stage, though more recently the research focus in validation and verification has drifted toward software. As a consequence, the newest and most powerful techniques have not been significantly applied to hardware; this work seeks to make a modest contribution toward redressing the imbalance.

An abstract interpretation-based formalism is introduced, *transitional logic,* that supports formal reasoning about dynamic behaviour of combinational asynchronous circuits. The behaviour of majority voting circuits with respect to single-event transients is analysed, demonstrating that such circuits are not SET-immune. This result is generalised to show that SET immunity is impossible for all delay-insensitive circuits.

An experimental hardware partial evaluator, HarPE, is used to demonstrate the 1st Futamura projection in hardware – a small CPU is specialised with respect to a ROM image, yielding results that are equivalent to compiling the program into hardware. HarPE is then used alongside an experimental non-clausal SAT solver to implement an automated transformation system that is capable of repairing FPGAs that have suffered cosmic ray damage. This approach is extended to support automated configuration, dynamic testing and dynamic error recovery of reconfigurable spacecraft wiring harnesses.

Author Publications

Parts of this research have been published in the following papers (in chronological order):

S. THOMPSON AND A. MYCROFT, *Abstract Interpretation of Combinational Asynchronous Circuits,* In Proc. 11th International Symposium on Static Analysis (SAS 2004), R. Giacobazzi, ed., LNCS 3148, Springer Verlag, August, 2004

S. THOMPSON AND A. MYCROFT, *Abstract Interpretation of Combinational Asynchronous Circuits (Extended Version),* Science of Computer Programming, Elsevier Science, To Appear

S. THOMPSON AND A. MYCROFT, *Statically Analysing the Dynamic Behaviour of Asynchronous Circuits by Abstract Interpretation,* In Proc. PREP 2004, Hatfield, UK, 2004

S. THOMPSON AND A. MYCROFT, *Sliding Window Logic Simulation,* In Proc. 15th UK Asynchronous Forum, Cambridge, January, 2004

S. THOMPSON AND A. MYCROFT, *Abstract Interpretation in Space: SET Immunity of Majority Voting Logic,* In Proc. APPSEM II Workshop, Frauenchiemsee, Germany, September 2005

S. THOMPSON AND A. MYCROFT, G. BRAT AND A. VENET, *Automatic In-Flight Repair of FPGA Cosmic Ray Damage,* In Proc. 1st Disruption in Space Symposium, Marseille, July, 2005

S. THOMPSON AND A. MYCROFT, *Bit-Level Partial Evaluation of Synchronous Circuits,* In Proc. ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06) Charleston, South Carolina, January 9-10, 2006

S. THOMPSON AND A. MYCROFT, *Self-Healing Reconfigurable Manifolds,* In Proc. Designing Correct Circuits (DCC'06), Vienna, March 2006

Acknowledgements

Thanks are due to: NASA Ames/Mission Critical Technologies Inc for accommodating me during the Summer of 2004, Guillaume Brat, Arnaud Venet, Jason Lohn, Gregory Larchev and Rick Alena for many useful discussions at Ames, Patrick and Radhia Cousot, Charles Hymans and others from the Abstract Interpretation group at École Normale Supérieure/École Polytechnique for their hospitality during my visits in Spring 2004 and Autumn 2005, Jim Lyke and others of the US Air Force Office of Scientific Research at Kirtland AFB, Mary Sheeran for her useful comments about shuffle networks at DCC'06, and of course my colleagues in the CPRG group at the Computer Laboratory. Thanks are also due to Ganesh Sittampalam and Mark Snellgrove for their comments on late drafts of my thesis, and of course to my supervisor, Professor Alan Mycroft, for believing in me in the first place and for all his help and encouragement over the last three years.

My work was financially supported by Big Hand Ltd., EPSRC, Intel, NASA/MCT Inc., by a St Edmund's College Charter Studentship and by the US Air Force Office of Scientific Research (European Office of Aerospace Research & Development).

Oh, doseybat made me do it, so it's all her fault, really.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

"Historians a thousand years hence will say that with Apollo we enabled man to extend his arena of activity beyond his own planet and to make himself at home wherever he pleases. If I were 10 or 15 years old today I would very definitely commit myself and my life to the space programme. There are tremendous opportunities, tremendous challenges out there, and it's a very interesting world ahead. I just envy the kids who have a chance of going on where we leave off."

– Dr. Wernher von Braun, interview with Reginald Turnill, 1975 [138] pp. 379.

1.1 Motivation

Space is an extreme environment, by almost any practical measure. Hard vacuum poses significant challenges – for example, many familiar engineering materials are unusable, metal parts tend to weld themselves to each other spontaneously, lubricants freeze or evaporate, and convective cooling becomes impossible. Radiation in the form of charged particles, neutrons and high energy photons arrive from interstellar space and from the Sun – some cosmic rays (atomic nuclei moving at close to the speed of light) possess vast kinetic energy, approaching that of a solidly hit golf ball, and can easily penetrate two metres of lead shielding. Any spacecraft surfaces facing the Sun may reach temperatures that would melt many materials commonly used on Earth, yet surfaces facing in the opposite direction must simultaneously withstand temperatures approaching absolute zero. Furthermore, despite a need to withstand high G-forces and large amounts of vibration during launch, spacecraft must nevertheless be as light as possible. Spacecraft design, as a direct consequence, is certainly one of the most challenging areas of modern engineering.

This thesis explores a number of areas relevant to the design of electronic systems within spacecraft. Though this has been our primary motivation, most of our contributions are also applicable to the the design of high reliability electronics for general applications.

1.1.1 Applying Program Analysis to Hardware

Though much is already known about the practicalities of electronic design for space applications, much of this is in the form of best practice [77] – engineer's 'folk knowledge,' effectively. Formalising the design and analysis of space electronics, ideally to the extent that behavioural constraints can be mathematically verified, is clearly highly desirable.

Many techniques that are currently under active research in the program analysis world have a long history of application to the analysis of electronic circuits. In particular, theorem proving and model checking were (and arguably still are) more widely used for electronic design validation and verification than for software analysis. However, the research focus in recent years has been almost exclusively on software, so the most powerful and advanced techniques currently available have seldom been applied to hardware. The theoretical work described in Part II of this thesis is aimed to be a small step toward redressing this imbalance.

1.2 Thesis Structure and Contributions

This thesis is logically grouped into five parts: *I. Introduction,* of which this chapter forms part, *II. Foundations,* which addresses theoretical concerns that underpin the later chapters, *III. Applications,* which reports results from applying this theoretical work to practical problems in space science, *IV. Conclusions,* which summarises our results and finally *V. Appendices.*

Individual chapters are summarised as follows:

1. **Introduction.**

2. **Background.** In this chapter, concepts that underpin the later chapters are described, including abstract interpretation, partial evaluation, synchronous and asynchronous digital circuits. A brief overview of relevant space science issues is also included, with particular reference to environmental effects on electronic systems.

3. **Transitional Logics.** We introduce a family of many-valued logics that are capable of supporting formal reasoning about a wide class of asynchronous circuits. A formal semantics based upon an abstract interpretation is given and used for the purpose of providing soundness and completeness proofs.

   Contribution: In this chapter we introduce the first abstract interpretation-based analysis of asynchronous circuits. We also coin the term *achronous analysis* to describe a wide class of analyses, all of which adopt an independent attribute model and abstract away details of absolute time. A number of pre-existing analysis techniques are discussed, and are shown to be subsumed by our approach.

4. **Bit-level Partial Evaluation of Synchronous Circuits.** This chapter describes a technique by which offline partial evaluation can be applied to purely synchronous digital circuits. A loop unrolling strategy is given that is capable of transforming circuits into versions that maintain equivalent behaviour whilst executing in fewer cycles. Extending unrolling until a fixed point is reached is shown to be equivalent

to the first Futamura projection. An embedded hardware description language, *HarPE,* which was implemented specifically to support experimentation in this area, is also described.

Contribution: We demonstrate the application of bit-level partial evaluation with loop unrolling to hardware, and also the first example of a 1st Futamura projection in hardware.

5. **Repairing Cosmic Ray Damage in FPGAs with Non-Clausal SAT solvers.** In this chapter, a technique for generating FPGA bitstreams that work around radiation induced permanent latch-up damage is given. An adapted version of HarPE is used to *flatten* (reduce to purely combinational form) a model of a small FPGA, then a non-clausal SAT solver is used to work around simulated injected faults.

   Contribution: We demonstrate the first application of SAT solvers to the rapid generation of formally correct work-around bit streams – previous work (*e.g.,* Lohn *et al* [86, 85, 83], Stoica *et al* [120]) was based upon genetic algorithms, which had limitations in terms of CPU requirements, completeness and correctness.

6. **Reconfigurable Manifolds.** In this chapter, we extend the concepts of the previous chapter to encompass reconfigurable spacecraft wiring harnesses. Though logically similar in concept to a macroscopic FPGA, a practical reconfigurable manifold would be more likely to show resemblance to a circuit-switched telephone exchange, partly due to the computational cost of computing routing layouts for FPGA architectures, and partly in order to accommodate other important signal types (*e.g.,* analogue, power and microwave).

   Contribution: This chapter reports the results of an early-stage study into the feasibility of constructing reconfigurable manifolds. At the time of writing, no spacecraft has yet flown with such hardware on board, though small-scale prototypes have been ground tested by the US Air Force Office of Scientific Research [90]. A number of issues are examined including manifold architectures, dynamic discovery of interconnections, self-repair and graceful degradation.

7. **SET Immunity in Delay-Insensitive Circuits.** In this chapter, transitional logic is applied to the analysis of the effect of single-event transients on delay insensitive circuits. 3-way and 5-way voting circuits are analysed in detail and are shown to be unable to reject single-event transients, though they are shown (as expected) to be immune to permanent failures. The result is generalised to the entire class of delay-insensitive circuits constructed from fundamental gates.

   Contribution: We present a formal proof that no delay-insensitive circuit, whatever its architecture, can be immune from the effects of single-event transients.

8. **Conclusions.**

9. **Future Work and Open Questions.**

1.3 Assumptions

It is assumed that readers of this thesis will already have some background in both program analysis and in the basics of electronic design. Prior knowledge of spacecraft engineering is not generally assumed, though a grounding in the basic science would be helpful.

Chapter 2

Background

"The flight was extremely normal. . . for the first 36 seconds then after that got very interesting."

– Pete Conrad, Apollo 12 commander, regarding the launch during which two electrical discharges almost ended the mission.

Mankind's conquest of space began in 1957 with the launch of the 83kg, basketball-sized Sputnik 1 into an elliptical orbit. Its on-board electronics were limited to a simple radio transmitter that emitted a periodic bleep, but nevertheless this was a crucial first step that garnered the necessary political will to take the exploitation of space further. Just 12 years later, Neil Armstrong became the first person in history to set foot on the Moon. Such progress in such a short period is quite astonishing, and it seems clear that the success of the Apollo programme was, in no small part, due to the great talent exhibited by Wernher von Braun, his team and the thousands of contractors who were involved. Nevertheless, it is quite apparent that corners were cut in America's haste to beat the Russians to the Moon – the first landing by Apollo 11's lunar module occurred despite numerous computer failures during the descent. The *Eagle,* in fact, landed with just 20 seconds of remaining fuel reserves [138].

Apollo's replacement by the Shuttle system (see Fig. 2.1) began with the launch of *Columbia* on the STS-1 mission on 20th April, 1981 at 7:00:03 a.m. EST. Despite Shuttle's now-acknowledged limitations, it nevertheless represented a huge advance over previous technology, particularly with regard to its flight systems electronics. Though low-powered by modern standards, its 5-way redundant on-board computers were vastly superior to Apollo's primitive hardware, and to date have proven incredibly reliable[1].

[1]NASA are, apparently, considering the possibility of continuing to use the Shuttle computer architecture in the upcoming CEV/CLV programme, because the performance gains of moving to a newer architecture are not regarded as justified given the excellent reliability of the existing systems.

Photo: NASA

Figure 2.1: The Shuttle *Discovery,* seen from the ISS during Return to Flight

2.1 Space: The ultimate extreme environment

Space is an utterly unforgiving environment. All spacecraft must be able to endure hard vacuum, extremes of temperature, high G forces and hard radiation whilst remaining reliable and functional. Achieving this in satellites and space probes is challenging; manned spaceflight increases the difficulty substantially because, ultimately, astronauts' lives critically depend upon the reliability of their equipment. In this section, we summarise the main issues that must be overcome by spacecraft designers.

2.1.1 Vacuum

The vacuum of deep space is near-perfect, though it is not entirely empty – any cubic metre of space is likely to contain a significant number of charged particles. In low-earth orbit (see Fig. 2.2), particle density is sufficient to exhibit significant drag that causes orbits to gradually decay. A stark example of this effect was Skylab, which burned up in the atmosphere on 11th July, 1979 after its orbit had deteriorated over a 5 year period.

Engineering materials must be carefully selected before they can be safely incorporated into spacecraft designs. Outgassing is a frequent problem [142], where chemical components of a material that would normally be stable at ground level can sublime due to the low ambient pressure. In some cases, the emitted gases may themselves be a problem, but more commonly it is the effect on the material itself that is of concern.

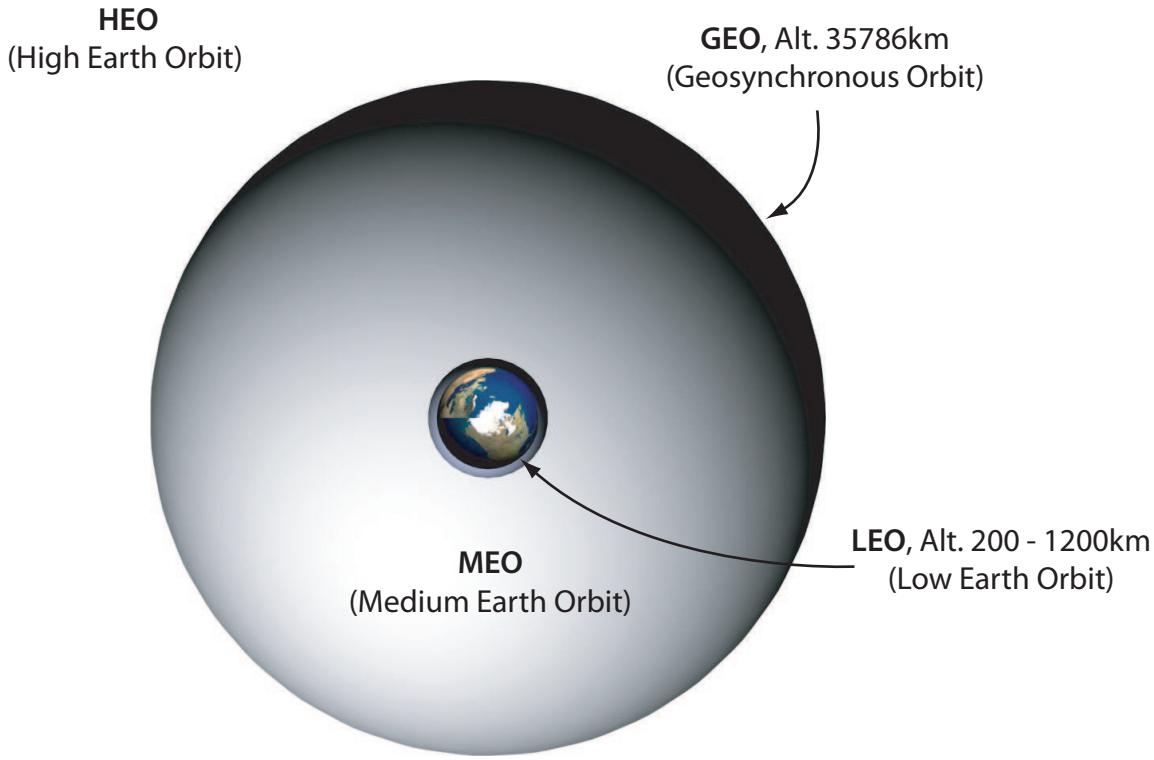


Figure 2.2: LEO, MEO, GEO and HEO

If a plasticiser sublimes out of a plastic component, it may become brittle, changing its mechanical properties and perhaps risking failure.

Astronauts must, of course, be protected from exposure to vacuum at all costs. In an accident during the ground-based testing of a space suit at NASA Johnson Space Center in 1965, a test subject was briefly exposed to a near-vacuum (approximately 1psi/0.05bar). The test subject lost consciousness after approximately 14 seconds, then regained consciousness again when the air pressure was pumped back up to an equivalent of an altitude of 15000 feet/4500 metres. He is not said to have suffered any long-term damage, but reported that his last conscious memory was of the water on his tongue beginning to boil.

2.1.1.1 The Charging Problem

The spacecraft charging problem first came to light in the early 1970s, after investigation of the anomalous behaviour of several satellites, as well as the total loss of the DSCS II 9431 military communications satellite. A number of missions have flown that were specifically intended to study the problem, beginning with a joint NASA/USAF investigation in the late 1970s: in 1979, the SCATHA (Spacecraft Charging AT High Altitudes) mission [93] collected important data. More recently, in July 1990, CRRES (Combined Release and Radiation Effects Spacecraft) was launched into a geosynchronous transfer orbit [64].

In a nutshell, the problem was found to be caused by incident charged particles im-

pacting the surface of the satellite and thereby transferring their charge to the spacecraft itself. The effect is at its worst during geomagnetic storms (solar flares), where the results of differential charging between adjacent parts can exceed the breakdown voltage of the vacuum, causing arcing, sputtering and other unwanted effects. High energy electrons, mostly an issue for spacecraft within the Van Allen belts, can penetrate deep within a spacecraft to cause dielectric charging within, for example, PCB substrates.

Current practice is to limit the effect of the problem by carefully grounding all spacecraft components with several, redundant, low resistance ground connections. Some satellites also incorporate ion guns that can counteract charging directly by emitting positively charged particles to adjust the spacecraft's net charge.

2.1.2 Radiation

The most serious environmental challenge that faces the designers of space borne electronic systems is radiation, mostly in the form of energetic particles from the following sources:

Heavy ions trapped in the Earth's magnetosphere The Earth's magnetic field interacts with any moving charged particles in the vicinity – Lorentz forces [89] can cause some particles to become trapped, where they continue to orbit at significant velocities. This radiation source was studied extensively during the SAMPEX (Solar Anomalous and Magnetospheric Particle Explorer) mission [15], which concluded that the particles' energy was insufficient to cause noticeable effects on satellites [77].

Electrons and protons trapped in the Van Allen belts The Van Allen belts consist of charged particles from the solar wind that have been trapped by the Earth's magnetosphere. The large outer Van Allen belt sits at an altitude of approximately 10,000 to 65,000km, consisting of relatively low energy (< 1MeV) electrons and positively charged ions. Trapped protons are primarily a feature of the much smaller inner Van Allen belt, which lies at an altitude of approximately 1000-5000km, and are sufficiently energetic that they can affect electronics, particularly modern small-geometry devices.

On 9th July, 1962, in the Starfish Prime test (see Fig 2.3), a Thor rocket carrying a 1.45 megaton W49 thermonuclear warhead[2] was detonated at an altitude of 400km above Johnston Atoll in the Pacific ocean. Charged particles from the detonation significantly increased the intensity of the Van Allen belts, which caused the total failure of 7 satellites, including Telstar, the world's first communications satellite. The detonation also caused an electromagnetic pulse within the atmosphere that damaged electrical equipment 1500km away in Hawaii. The resulting charged particles, mostly electrons, persisted within the belts for several years.

Solar flares A solar flare is a violent explosion in the Sun's upper atmosphere that emits very large quantities of charged particles travelling at close to the speed of light, as

[2]Approximately 100 times the yield of the bomb that destroyed Hiroshima on 6th August, 1945.

Photo: nuclearweaponarchive.org

Figure 2.3: Starfish Prime, as seen from Honolulu, Hawaii

well as electromagnetic radiation across the entire spectrum, from radio frequencies to gamma rays. Typical particle energies for protons are of the order of several hundred MeV, with heavy ion energies ranging from 10 MeV/n to several hundred GeV/n.

Charged particles from solar flares are a significant danger, both to space electronics and also to astronauts. The moon landings were timed carefully to coincide with solar minimum – the missions concerned were lucky not to encounter any flares, because the radiation shielding within the Apollo spacecraft was likely to have been inadequate to prevent excessive radiation exposure. Future long-duration manned deep space missions, particularly any manned Mars exploration programme, must be designed to provide a solar flare 'storm shelter,' capable of protecting the astronauts until the flare has passed.

Cosmic rays Galactic cosmic rays originate from sources outside the solar system, and include ions of all elements with atomic numbers up to 92 (uranium). Velocities are extremely high, with energies up to the hundreds of GeV/n, though particle flux is low. Since shielding against particles of these energies is generally impractical, cosmic rays are a significant potential cause of damage to electronics.

The intensity of all of these sources is affected by the level of solar activity, which varies on a cycle with a period of between 9 and 13 years. The typically 11 year solar cycle may be divided into *solar maximum,* lasting approximately 7 years, and *solar minimum,* lasting approximately 4 years. Solar maximum is characterised by frequent solar flare activity, whereas during solar minimum flares are rare. Cosmic ray flux follows an opposite cycle, and is at its greatest during solar minimum.

Due to the shielding effects of the Earth's magnetosphere, radiation levels vary dramatically with a spacecraft's orbit. In low earth orbit (LEO), radiation levels are compa-

rable to those experienced to high altitude aircraft, and are generally relatively easy to deal with. Manned spaceflight has to-date mostly concentrated on LEO missions for this reason – Shuttle and the International Space Station (ISS) operate exclusively in LEO. Radiation levels on the ISS are sufficiently low that astronauts are able to use standard laptop PCs[3]. Manned spaceflight tends to avoid MEO because of the Van Allen belts – the trajectories of the manned Apollo lunar missions were specifically designed to avoid the areas of densest radiation flux, resulting in a (relatively) harmless total dose for the astronauts.

Unmanned spacecraft do use MEO and GEO extensively, however. The Navstar GPS (Global Positioning System), operated by the 50th Space Wing's (US Air Force Space Command) 2nd Space Operations Squadron, consists of a constellation of typically 24 satellites on circular, 20200km altitude, MEO orbits. Most communications satellites are on geosynchronous (GEO) or geostationary (GSO) orbits, also known as Clarke orbits after their original proposer, Sir Arthur C. Clarke [36]. Satellites on both MEO and GEO/GSO orbits must be designed with radiation hardening techniques[4].

2.1.3 Temperature

It is important to clearly state that the vacuum of space does not possess a temperature in the conventional sense of the word – in its usual terrestrial context, temperature is a bulk measure of the vibration of the molecules within a solid, liquid, gas or plasma. In space, particle density is sufficiently low as to be irrelevant for most purposes when considering the behaviour of satellites, although more correctly the solar wind should be regarded as a very low density plasma.

Surfaces that face toward the sun are heated by electromagnetic radiation, primarily infrared light. A perfect black body in solar orbit, facing the sun, at the same distance as the Earth would typically stabilise at a temperature of 280K (7°C). A similar surface facing away from the sun would fall to approximately 5K (-268°C) – in deep space, far from stars or galaxies, the temperature would tend toward the cosmic microwave background at approximately 2.7K (-270°C). Most space-qualified electronic devices are characterised only to operate between -55°C and +125°C, so effective thermal management is essential. Since electronic components generate heat, particularly any fast CMOS VLSI devices, cooling can be a difficult problem, since it can only be achieved by radiation and not convection[5].

2.1.4 G-forces and Vibration

Launch G-forces vary depending upon the launch system. The Saturn V vehicle used in much of the Apollo programme would typically develop over 4G, whereas Shuttle is a

[3]NASA purchased over 70 laptops from different manufacturers, then tested each one in a radiation chamber. Apparently, the one that proved most reliable, a relatively low-powered IBM Thinkpad, is now standard equipment *(Source: personal conversation with Rick Alena, NASA Ames, August 2004)*.

[4]Note that, in the space community, correct English usage is taken to be 'on orbit,' rather than the colloquial 'in orbit.' We adopt the former convention.

[5]Convection is only possible in a fluid – indeed, it is the absence of convection that allows a Thermos™-style vacuum flask to maintain its contents' temperature for long periods.

little gentler at approximately 3.5G. The Pegasus XL launch vehicle, often used for small satellites, develops approximately 2.5G. Vibration during launch can be considerable – though a launch may appear sedate from a distance, rocket motors do not burn perfectly evenly, with fluctuations in thrust causing nontrivial levels of vibration.

All spacecraft must therefore be carefully designed to withstand both the G-forces and vibration of launch. Large printed circuit boards typically need to be supported at multiple points across their surface, and individual components need to be strongly attached so that their soldered joints can not shear off.

2.2 Program Analysis and Transformation

It would be impossible within the scope of this thesis to attempt to thoroughly review all of program analysis and transformation. We therefore concentrate specifically upon abstract interpretation and partial evaluation, which underpin the work described in Chapters 3 and 4 respectively. Readers wishing for a broader reference are better referred to the standard texts [104, 72].

2.2.1 Abstract Interpretation

Abstract interpretation [42, 43] is a long-established technique, most commonly applied to software, that allows abstract properties of systems to be determined.

As a simple example[6], consider the 'law of signs' in integer arithmetic. It is possible, knowing only the signs of a and b to know with certainty the sign of the result of the integer expression $a \times b$. The sign of the result of the addition $a + b$ may be determined in some cases, but not all. This can be thought of as a very simple kind of abstract interpretation. We might define an abstract multiplication operator $\otimes$, and an abstract addition operator $\oplus$ as follows:

$\otimes$	$-$	0	$+$	$\bigstar$
$-$	$+$	0	$-$	$\bigstar$
0	0	0	0	0
$+$	$-$	0	$+$	$\bigstar$
$\bigstar$	$\bigstar$	0	$\bigstar$	$\bigstar$

$\oplus$	$-$	0	$+$	$\bigstar$
$-$	$-$	$-$	$\bigstar$	$\bigstar$
0	$-$	0	$+$	$\bigstar$
$+$	$\bigstar$	$+$	$+$	$\bigstar$
$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$

where $-$ represents any negative integer, 0 represents zero, $+$ represents any positive integer and $\bigstar$ represents any integer whatsoever.

The multiplication $1543 \times -783 = -1208619$ in the concrete world maps to the abstract multiplication $+ \otimes - = -$. Using this technique, it is possible to determine with certainty the sign of the result of a multiplication without actually needing to carry out the multiplication itself. However, the addition $-344 + 762 = 418$ maps to the abstract addition $- \oplus + = \bigstar$, since the abstract values $-$, 0, $+$ and $\bigstar$ do not carry enough information for a more accurate result to be determined. Nevertheless, in many cases

[6]The material in this section is an extended version of an example that was originally included in the author's publication [131]

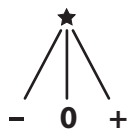


Figure 2.4: Hasse diagram for $Z^\sharp$ with respect to subset inclusion $\subseteq$

this approach is still sufficient to fully predict the sign of an integer expression involving addition and multiplication, without any requirement to perform the actual arithmetic.

It is traditional in abstract interpretation to define a *concrete domain* that closely models the real world, and an *abstract domain* that in some useful way approximates the concrete domain. Functions are then defined that have counterparts on each domain – often, the concrete version of the function is difficult to compute efficiently, but the (usually less accurate) abstract version has comparatively trivial complexity. Abstract interpretation allows formal proofs to be created that demonstrate the correctness of the abstraction, as well as proving that related abstract and concrete functions faithfully model each other's behaviour.

In this thesis, we consider the original approach to abstract interpretation, pioneered in the late 1970s by Cousot & Cousot [42, 43], which draws heavily on the theory of Galois connections. Other approaches that have been introduced more recently that are based on the Moore-Penrose pseudo-inverse or on Lagrangian relaxation are acknowledged though for brevity they are not described here since they are not required by our definition of transitional logics in Chapter 3.

2.2.1.1 Concrete and Abstract Domains

Formally, abstract interpretation models both concrete and abstract domains as partially ordered sets. In the example described in Section 2.2.1, the concrete domain would initially appear to be the integers $\mathbb{Z}$, partially ordered by $\leq$, but this is not the case – the reason for this becomes clear on closer examination of the abstract domain

$$Z^\sharp \stackrel{\text{def}}{=} \{-, 0, +, \bigstar\}.$$

Whilst it is possible to relate $-, 0$ and $+$ by a simple numeric $\leq$ ordering, this gives no effective means of handling $\bigstar$ sensibly. The ordering that is actually important in this case, as in most applications of abstract interpretation to program analysis, is subset inclusion $\subseteq$. Since $\bigstar$ is defined as representing any integer whatsoever, is clear that $- \subseteq \bigstar$, $0 \subseteq \bigstar$ and $+ \subseteq \bigstar$, with all other cases being incomparable. This may be represented pictorially as a *Hasse diagram* as shown in Fig. 2.4 – in such diagrams, values that are connected by lines are partially ordered such that the upper value is greater than or equal to the lower value, and values that are not connected by lines are incomparable.

Whilst it is possible to represent any member of $\mathbb{Z}$ by a single (best) member of $Z^\sharp$, the converse is not generally feasible, with only 0 having a unique representation. Our concrete domain, then, needs to be able to represent sets of *possible* values, *i.e.,*

$$Z \stackrel{\text{def}}{=} \wp(\mathbb{Z}) \backslash \{\{\}\}$$

with values in Z partially ordered on subset inclusion[7]. It now becomes possible to convert values from Z to values in $Z^\sharp$ and vice-versa: 0 is represented by the singleton set $\{0\}$, $+$ by the set $\{1, 2, 3, \dots\}$, $-$ by $\{\dots, -3, -2, -1\}$ and $\bigstar$ by $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$ (*i.e.,* by $\mathbb{Z}$ itself). If our abstraction is valid, then beginning with any set of possible values $\hat{s} \in Z$, converting this to a value in $Z^\sharp$ and back again is always guaranteed to return a set of values in Z that contains $\hat{s}$.

2.2.1.2 Abstraction and Concretisation Functions

We can now define an *abstraction function,* $\alpha : Z \to Z^\sharp$ as follows:

$$\alpha(\hat{n}) \stackrel{\text{def}}{=} \begin{cases} -, & \text{if } \forall n \in \hat{n} \,.\, n < 0, \\ 0, & \text{if } \hat{n} = \{0\}, \\ +, & \text{if } \forall n \in \hat{n} \,.\, n > 0, \\ \bigstar, & \text{otherwise.} \end{cases}$$

and a concretisation function $\gamma : Z^\sharp \to Z$ as follows:

$$\gamma(v) \stackrel{\text{def}}{=} \begin{cases} \{n \in \mathbb{Z} \mid n < 0\}, & \text{if } v = -, \\ \{0\}, & \text{if } v = 0, \\ \{n \in \mathbb{Z} \mid n > 0\}, & \text{if } v = +, \\ \mathbb{Z}, & \text{if } v = \bigstar. \end{cases}$$

This gives a formal way to take any value $\hat{n}$ in Z, which of course may itself be a set of possible values, and then abstract it by applying the abstraction function α to yield a value in $Z^\sharp$. If we now concretise this again by applying γ, we get another value $\hat{m} \in Z$. If we have defined our domains and our abstraction and concretisation functions correctly we can guarantee that $\hat{n} \subseteq \hat{m}$.

2.2.1.3 Galois Connections and Galois Insertions

Given a pair of partially ordered domains D and $D^\sharp$, and a pair of functions $\alpha : D \to D^\sharp$ and $\gamma : D^\sharp \to D$, if it can be shown that $\alpha \circ \gamma(\hat{x}) \sqsubseteq \hat{x}$ and $\gamma \circ \alpha(\hat{x}) \sqsupseteq \hat{x}$ then a *Galois connection* may be said to exist between the domains[8]. If it is also the case that $\alpha \circ \gamma(\hat{x}) = \hat{x}$, this is known as a *Galois insertion.* Since $\hat{x} = \hat{y}$ trivially implies $\hat{x} \subseteq \hat{y}$, all Galois insertions are also Galois connections.

Returning to our example, we can use this to formally prove that a Galois insertion exists for its concrete and abstract domains:

[7] Note that we exclude the empty set from Z because it has no direct counterpart in $Z^\sharp$. An alternative might be to instead add a corresponding bottom element $\bot$ to $Z^\sharp$, such that $\bot \subseteq -$, $\bot \subseteq 0$, $\bot \subseteq +$ and $\bot \subseteq \bigstar$, though for clarity (as is common practice) we prefer to omit bottom elements from both domains.

[8] In this thesis, as is common practice in program analysis, we assume that our Galois connections are monotone, with the concrete and abstract domains ordered in the same direction. This is different from Galois' original definition, and also to the definition used outside program analysis, where the domains are typically ordered in opposite directions. Our approach tends to make our definitions simpler, but does not otherwise gain or lose any power with respect to the original definition.

Theorem 2.2.1. *The functions $\langle \alpha, \gamma \rangle$ form a Galois insertion between Z and $Z^\sharp$. First we prove by cases that $\alpha \circ \gamma(\hat{x}) = \hat{x}$:*

1. $\alpha \circ \gamma(-) = \alpha\{n \in \mathbb{Z} \mid n < 0\} = -$
2. $\alpha \circ \gamma(0) = \alpha\{0\} = 0$
3. $\alpha \circ \gamma(+) = \alpha\{n \in \mathbb{Z} \mid n > 0\} = +$
4. $\alpha \circ \gamma(\bigstar) = \alpha\mathbb{Z} = \bigstar$

Now we prove that $\gamma \circ \alpha(\hat{x}) \sqsupseteq \hat{x}$. Proof by cases:

1. *Assume that $\alpha(\hat{x}) = -$. We know from this that $\forall x \in \hat{x} \;.\; x < 0$, so $\gamma \circ \alpha(\hat{x}) = \{n \in \mathbb{Z} \mid n < 0\} \supseteq \hat{x}$.*
2. *Assume that $\alpha(\hat{x}) = 0$. Therefore, $\hat{x} = \{0\}$, so $\gamma \circ \alpha\{0\} = \gamma(0) = \{0\} = \hat{x}$.*
3. *Assume that $\alpha(\hat{x}) = +$. We know from this that $\forall x \in \hat{x} \;.\; x > 0$, so $\gamma \circ \alpha(\hat{x}) = \{n \in \mathbb{Z} \mid n > 0\} \supseteq \hat{x}$.*
4. *Assume that $\alpha(\hat{x}) = \bigstar$. Since $\gamma\bigstar = \mathbb{Z}$, we can know that $\gamma \circ \alpha(\hat{x}) = \mathbb{Z} \supseteq \hat{x}$.*

We can therefore be sure that, in our example, $Z^\sharp$ is a valid abstraction of Z – the Galois connection guarantees that, given correct operators, any abstract prediction will be *safe,* because it must contain any corresponding concrete result. This is known as *over-approximation* in the abstract interpretation literature.

2.2.1.4 Proving the Correctness of Abstract Operators

Knowing that we have a valid abstraction is not generally sufficient – it is normally necessary to also prove the correctness (and possibly completeness) of abstract functions with respect to their concrete counterparts [42, 43]. Given a Galois connection $\langle D, D^\sharp, \alpha, \gamma \rangle$ and a pair of functions $f : D \to D$ and $f^\sharp : D^\sharp \to D^\sharp$, if either of the (equivalent) relations $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ or $f \circ \gamma \sqsubseteq \gamma \circ f^\sharp$ can be shown to hold, the function $f^\sharp$ can be said to be *correct* with respect to f.

Though it is feasible to prove the correctness of our example's $\oplus$ and $\otimes$ operators, this is somewhat long-winded due to the number of cases that need to be handled[9]. In the interests of clarity, we will instead define and prove the correctness of concrete and abstract versions of a unary minus operator $\ominus$.

We begin by defining unary minus for the concrete domain Z. Note that we can't use the standard unary minus (*e.g.,* $-n$), because we must handle sets of possible values rather than just singleton integers. Our definition is as follows:

$$\ominus(\hat{n}) \stackrel{\text{def}}{=} \{-n \mid n \in \hat{n}\}$$

The abstract version of unary minus is defined by tabulation:

[9]Since $\otimes$ and $\oplus$ both take two arguments, this further complicates matters. See Theorem 3.4.4 in Section 3.4.1 and its accompanying discussion for an example of such a proof.

$\ominus^\sharp$	
$-$	$+$
0	0
$+$	$-$
$\bigstar$	$\bigstar$

Proving correctness is straightforward in this case:

Theorem 2.2.2. *The function $\ominus : Z \to Z$ is correct with respect to $\ominus^\sharp : Z^\sharp \to Z^\sharp$. Proof by cases:*

1. $\ominus \circ \gamma(-) = \ominus\{n \in \mathbb{Z} \mid n < 0\} = \{n \in \mathbb{Z} \mid -n < 0\} = \{n \in \mathbb{Z} \mid n > 0\} = \gamma(+) = \gamma \circ \ominus^\sharp(-)$
2. $\ominus \circ \gamma(0) = \ominus\{0\} = \{0\} = \gamma(0) = \gamma \circ \ominus^\sharp(0)$
3. $\ominus \circ \gamma(+) = \ominus\{n \in \mathbb{Z} \mid n > 0\} = \{n \in \mathbb{Z} \mid -n > 0\} = \{n \in \mathbb{Z} \mid n < 0\} = \gamma(-) = \gamma \circ \ominus^\sharp(+)$
4. $\ominus \circ \gamma(\bigstar) = \ominus \circ \mathbb{Z} = \mathbb{Z} = \gamma(\bigstar) = \gamma \circ \ominus^\sharp(\bigstar)$

2.2.1.5 Proving Completeness

If an abstraction is said to be *complete,* it includes all possible behaviours of the concrete system and no more. Formal completeness [58, 59, 60, 56, 57, 101, 119] proofs tend to be a little more complex than correctness proofs, though completeness always implies correctness. To prove completeness for $\langle f, f^\sharp \rangle$ it is first necessary to define an ideal, most-accurate-possible version of $f^\sharp$ as follows:

$$f^\sharp_{best} \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma$$

and then to prove that $f^\sharp = f^\sharp_{best}$. If we can also prove

$$f \circ \gamma = \gamma \circ f^\sharp$$

then we have γ-completeness (also called forward-completeness). If we can prove

$$\alpha \circ f = f^\sharp \circ \alpha$$

then we have α-completeness (also called backward-completeness). Note that these two kinds of completeness are orthogonal – it is possible for $\langle f, f^\sharp \rangle$ to possess either, neither or both α-completeness and γ-completeness.

Completeness in abstract interpretation was first explored in the early 1990s [101, 119], though distinguishing α- and γ-completeness is due to Giacobazzi *et al* [57]. Note that most of the literature concerning completeness in abstract interpretation concentrates on α- (backwards) completeness.

Section 3.4.1 contains fully worked completeness proofs for the operators of our transitional logic.

2.2.1.6 Independent and Relational Attribute Models

In 1980, Jones and Muchnick [73] introduced the concept of *independent* and *relational attribute models*.

In the independent attribute model, values are modeled entirely separately from each other. A typical example might be an abstract domain that models integer ranges as intervals, where the sum $[1,3]+[5,8]$ evaluates to the range $[6,11]$. Assuming an underlying semantics based on sets of integers, this calculation corresponds to

$$\{m+n \mid m \in \{1,2,3\} \wedge n \in \{5,6,7,8\}\} = \{6,7,8,9,10,11\}.$$

Note that m is independent from n, and both can appear in any combination.

In the relational attribute model, values are assumed to be related to each other such that they may not necessarily appear in all possible combinations. For example, if the m and n in the above example were constrained to only appear in certain combinations, the result is more exact, *e.g.,*

$$\{m+n \mid (m,n) \in \{(1,6),(3,5),(2,8)\}\} = \{7,8,10\}.$$

Independent attribute models tend to be inexact, though they are well suited to program analysis since they can typically be represented in space that is linear in the number of values. In contrast, relational attribute models can give exact results, but are rarely directly implementable due to a tendency toward exponential space blowup. In abstract interpretation, it is common for a concrete domain with a relational attribute model to be modeled by an abstract domain with an independent attribute model, thereby enabling a compact, efficient and provably-correct representation to be used to perform analyses without state space blowup problems.

2.2.1.7 Discussion

Abstract interpretation is extremely powerful, something that is often not apparent on first encountering the technique. It makes it possible to prove correct many kinds of analyses, particularly those that depend upon abstraction. In practical terms, these correct abstractions can drastically reduce the amount of CPU time necessary to perform analyses, and in many cases can make it possible to rapidly generate safe approximations in situations where exact results would be take unfeasibly long to compute or even be undecidable.

2.2.2 Partial Evaluation

Partial evaluation was first described by Lombardi in the 1960s [88, 87], though its underlying principle is acknowledged to rely upon Kleene's *s-m-n theorem* [79]. It has since become well known as an effective and efficient program transformation technique. Interested readers are referred to the excellent reference work by Jones, Gomard and Sestoft [72] – for brevity, we shall provide only a brief overview of the technique as it is generally applied to the partial evaluation of software. Special considerations for partially evaluating hardware are specifically discussed in Chapter 4.

Given a program p written in some language L, and some input for that program i, an interpreter I_L can execute the program and determine its result r:

$$r = I_L(p, i)$$

If the input i is known completely before the program is run, a clever optimisation might be to run the program p exactly once (assuming of course that it terminates), then define a new program p' that simply returns r immediately without performing any other computation first, *i.e.,*

$$r = I_L(p')$$

which in practice would be likely to provide a very considerable speedup.

Partial evaluation is primarily concerned with cases where i is only partially known beforehand – though an optimisation as drastic as the above is not generally feasible, specialising p given what is known about i can nevertheless be used to generate a *residual* program that in many cases may be much faster than the original. Normally, code size increases as a consequence of the transformation, so partial evaluation may therefore be seen as a tradeoff between code size and performance. In some special cases, the optimisations that partial evaluation allows may cause the residual program to be smaller than the original program[10].

A *partial evaluator* is a program PE_L that accepts another program written in some language L, along with input $i \cup j$, where i is known and j is unknown, and generates a specialised program

$$p' = PE_L(p, i)$$

such that $I_L(p(i \cup j)) = I_L(p'(j))$ for all j.

Things start to get interesting if the program being partially evaluated is itself an interpreter:

$$PE_M(I_L, p)$$

In this case, the program is an interpreter for language L that is itself written in language M. The partially defined input to I_L, in this case, is a program p, written in L. The effect of this transformation is equivalent to compiling p into the language M – one can also view this transformation as allowing interpreter I_L to perform the function of compiler $C_{L \to M}$:

$$C_{L \to M}(p) = PE_M(I_L, p)$$

In the literature, this transformation is referred to as the first Futamura projection [54].

If a partial evaluator is self-applicable[11] (*e.g.,* if we have a partial evaluator PE_L that

[10] Code size reduction is not generally explicitly discussed in the software PE literature, though it clearly occurs. Whilst loop unrolling normally increases code size, it also exposes the internals of loops, making it possible to apply a wider range of optimisations. If the expansion inherent in unrolling is exceeded by the code size reduction from optimisation, it is possible for the residual program to be smaller than the original. A clear example would be specialisation of a function designed to raise a number to a specified power by iterative multiplication. Specialising the power to 3, for example, would (given a suitably sophisticated partial evaluator) yield a residual program consisting only of two multiplications if the loop is unrolled to a fixed point, which would clearly result in far smaller and faster code. An example of this effect in the context of hardware PE is given in Section 4.6.3.3.

[11] This is not generally feasible or desirable in hardware partial evaluation, but is mentioned here for completeness.

is itself written in L), some surprising results are possible. If we specialise the partial evaluator against an interpreter

$$C_{L\to M} = PE_M(PE_M, I_L)$$

this yields a true stand-alone compiler from L to M that does not require the presence of a partial evaluator.

If we specialise the partial evaluator against itself

$$CC_M = PE_M(PE_M, PE_M)$$

this yields a compiler-compiler CC_M that can transform an interpreter into a compiler:

$$C_{L\to M} = CC_M(I_L).$$

2.2.2.1 Partial Evaluation of a Toy Imperative Language

In this section, we will examine the partial evaluation of a simple imperative language that supports expressions, assignment, `if/else` and `while` statements.

Expressions and Assignment. The partial evaluation of expressions is straightforward. Consider the numeric expression

$$(a \times b) + c$$

where a is to be specialised and b and c are unknown. If a is specialised to take the value 10, the expression can be rewritten as

$$(10 \times b) + c$$

thereby avoiding the requirement to allocate memory for a or to retrieve its value. If a is specialised as 1, the multiplication can be eliminated:

$$b + c$$

If a is specialised as 0, then the entire expression can be reduced to c.

Most compilers do this by default, though in this context the transformation is usually called *strength reduction* in the literature.

Assignment. Partial evaluation of assignment is straightforward: first, partially evaluate the expression. If the result can be completely determined, then the assigned variable may now be regarded as part of the known input for the program and the assignment statement eliminated. Otherwise, the assignment is emitted normally.

Conditionals. In a conditional of the form

$$\texttt{if } c \texttt{ then } s_1 \ [\texttt{else } s_2] \texttt{ endif}$$

a partial evaluator would first partially evaluate the conditional expression c. If its result is completely known, then the `if` can be eliminated – if c is $true$, then the statement is replaced with s_1. If c is known to be $false$, the statement is replaced with s_2 if it is given, or otherwise the entire construct can be safely ignored.

Note that our hardware partial evaluator, HarPE takes a substantially different approach to handling assignment and conditionals than is described here (see Section 4.4.5.1).

While Loops. The main way in which while loops may be partially evaluated is *loop unrolling*, whereby loops may be flattened partially or completely.

Loop unrolling typically exploits the following identity:

```
while (c)                    =    if (c)
   s                                 s
endwhile                             while (c)
                                        s
                                     endwhile
                                  endif
```

Applying the rule once unwinds one instance of the loop body, opening it up to specialisation by the techniques mentioned above. Repeated application allows the loop to be unrolled to any desired extent – if this is continued until a fixed point is reached[12] (*i.e.,* until either the inner conditional can be determined to be $false$, or if partial evaluation of s eliminates all code), then the loop can be eliminated entirely.

One of the hardest problems in partial evaluation is deciding whether or not to unroll a particular loop, and if so, to what extent. Techniques from program analysis, particularly abstract interpretation, have much to offer here, though further discussion is beyond the scope of this chapter.

2.2.2.2 Discussion

This section has merely touched the surface of partial evaluation as applied to software; far more sophisticated techniques exist than those that were described for our toy language. It also became apparent whilst studying partial evaluation in hardware that there are some subtle but important differences in comparison with software PE – in particular, self-application is not generally feasible or desirable.

Though partial evaluation has been studied for some time, given its obvious power, it is a little disappointing that it has not found mainstream application beyond its influence (albeit in restricted form) on compiler optimisation. Nevertheless, the technique is there to be exploited – given our findings in Chapter 4, it would be tempting to hope that it might find applications in hardware that may at some point filter back into the software world, in the way that this has happened with model checking.

[12]Note that repeated unrolling is not guaranteed to reach a fixed point within acceptable time or space limits, or indeed at all.

2.3 Boolean SAT

In a nutshell, a Boolean SAT problem is a Boolean formula with a (possibly large) number of unbound variables, which evaluates to *true* for some variable assignment[13]. As a trivial example, the expression

$$(a \wedge b) \vee c$$

has several possible *solutions* that cause it to evaluate to T:

a	b	c
T	T	F
F	F	T
F	T	T
T	F	T
T	T	T

Any of these solutions may be said to *satisfy* the expression.

In contrast, the expression

$$(a \wedge \neg a) \wedge (b \vee c)$$

has no variable assignments that cause it to evaluate to T – as a consequence, the expression may be said to be *unsatisfied,* often abbreviated as UNSAT.

Finding solutions to SAT problems is computationally hard in the general case, and is known to be NP-complete [40]. Indeed, equivalence to Boolean SAT is often used as a means of proving a problem's NP-completeness. Given a particular variable assignment, evaluating a Boolean expression is typically linear in the size of the expression. Treating the expression as a black box, it is potentially necessary to enumerate all possible input assignments in order to find one that causes the expression to evaluate to T, so in this case the complexity of the search (assuming a deterministic machine) is exponential, of the order of $O(2^N)$, in the number of variables. A nondeterministic machine could of course evaluate all possible assignments in parallel and therefore find a solution in polynomial time.

It is a characteristic of NP-complete problems that they can normally be freely converted into each other in polynomial time. An NP-complete problem that cannot easily be attacked in its native form can often be transformed into an alternative form – converting problems of other kinds into Boolean SAT problems is a commonly adopted strategy.

2.3.1 SAT solvers

A *SAT solver* is a program that is capable of automatically finding solutions to Boolean SAT problems. Typically, SAT solvers do not treat expressions as a black box – rather, they tend to adopt heuristics that attempt to gain leverage on the problem by exploiting information from the expression itself. In practice, SAT problems typically vary considerably in difficulty, and only in more extreme cases tend toward the exponential $O(2^N)$

[13] In this thesis, for notational convenience in describing *transitional logics* (see Chapter 3) we adopt the convention of representing *true* by T and *false* by F, rather than the more common (though potentially ambiguous) 0 and 1.

upper bound. Modern SAT solvers are capable of attacking very large expressions with thousands of variables whilst maintaining acceptable run times.

Algorithms for solving SAT problems were first studied in the early 1960s. The Davis-Putnam decision procedure [46, 47] is still widely used, albeit in modified form. In this approach, a recursive search is carried out, whereby each variable is in turn assigned a value and the expression is checked to see whether it is satisfied, or if it is unsatisfiable. If it is satisfied, the search terminates and the current variable assigments constitute a solution to the SAT problem. If it is unsatisfiable, the search backtracks. If the search completes before a solution is found, the algorithm terminates with an UNSAT result.

An alternative approach based on stochastic search was introduced by Selman *et al* [114] in 1992. The GSAT algorithm is remarkably simple, yet it is capable of solving many hard SAT problems rapidly. In outline, the GSAT algorithm is as follows:

1. Initialise all variables to random initial values
2. Evaluate the expression. If the result is T, a solution has been found, so the loop terminates.
3. Taking each variable in turn, flip its state (*i.e.,* change F to T and vice-versa), then note the number of *clauses* (see Section 2.3.2) that evaluate to T as a consequence. Return each variable to its initial state after each count.
4. Choose the variable that most increases the number of satisfied clauses, then flip it permanently.
5. If a predetermined number of attempts has been exceeded, go back to step 1 (this is typically called a *restart* in the literature), otherwise go to step 2.

A substantial improvement was made by Selman *et al* [112] in 1994. The WALKSAT algorithm is broadly similar to GSAT, though at each step it concentrates only on the variables of a single, randomly chosen, unsatisfied clause. In outline, the algorithm is as follows:

1. Choose a clause at random that is currently unsatisfied
2. Depending on whether a random number exceeds a suitably chosen $temperature$ parameter, either:
   (a) Randomly choose a variable that appears within the clause and flip it, or
   (b) Attempt flipping each variable that appears within the clause in turn, noting the number of unsatisfied clauses that result in each case, then choose the one flip that results in the lowest number of unsatisfied clauses.

Both WALKSAT and GSAT perform credibly in comparison with current Davis-Putnam-inspired algorithms, though unlike Davis-Putnam they are *incomplete,* in that they fail to terminate if they are presented with an unsatisfiable expression.

At the time of writing, SAT solver technology is a very popular research area. The *SAT Live!* web site [3] carries links to current research and to downloadable SAT solvers that are in the public domain. An annual competition [1] evaluates current solvers against a large collection of SAT problems – the winners of recent competitions are generally good places to start when selecting a SAT solver for a new project.

2.3.2 Clausal SAT

Most contemporary SAT solvers require formulas to be presented in *clausal form,* also termed *conjunctive normal form* (CNF). Formulas in CNF are strictly of the form:

$$(\cdots \vee \cdots \vee \cdots) \wedge (\cdots \vee \cdots \vee \cdots) \wedge \cdots$$

A subexpression of the form $(\cdots \vee \cdots \vee \cdots)$ is known as a *clause,* and may only contain variables or their negations.

In some cases, SAT problems are already expressed naturally in clausal form. However, it often is necessary to transform problems into clausal form from an original form that is expressed as an arbitrary formula or in some cases as a directed graph.

The standard technique for transforming a Boolean expression into CNF is *multiplying out*. First, the expression is transformed to *negation normal form*, where negation is pushed out to the leaves of the expression by exhaustively applying the following rewrite rules:

$$\begin{aligned} \neg\neg a &\rightarrow a \\ \neg(a \wedge b) &\rightarrow (\neg a) \vee (\neg b) \\ \neg(a \vee b) &\rightarrow (\neg a) \wedge (\neg b). \end{aligned}$$

All expressions may be transformed to equivalent expressions in NNF in space that is linear with respect to the size of the original expression.

Next, the distributive law is used to push instances of $\wedge$ out to the top level:

$$a \vee (b \wedge c) \rightarrow (a \vee b) \wedge (a \vee c)$$

This approach to conversion to CNF has the serious problem that, in worst case, it may cause exponential blowup in the size of the resulting expression[14]. Better techniques exist that preserve satisfiability whilst generating an expression linear in the size of the original expression, though this comes at the expense of introducing extra variables to the expression [137].

2.3.3 Non-Clausal SAT

It has been noted by Thiffault *et al* [126] and others (including ourselves, see Appendix B) that flattening an expression to CNF destroys a significant amount of structural information that might otherwise be exploited. Non-clausal SAT solvers, including Thiffault's NOCLAUSE as well as our NNF-WALKSAT, can solve SAT problems in a form close to that of the problem's original structure. Since circuits are rarely expressed in a form that lends itself well to conversion to CNF, non-clausal SAT is particularly well suited to the kinds of SAT problem that typically arise in hardware verification.

[14]This is a consequence of the repetiton of a on the right hand side of the rewrite rule.

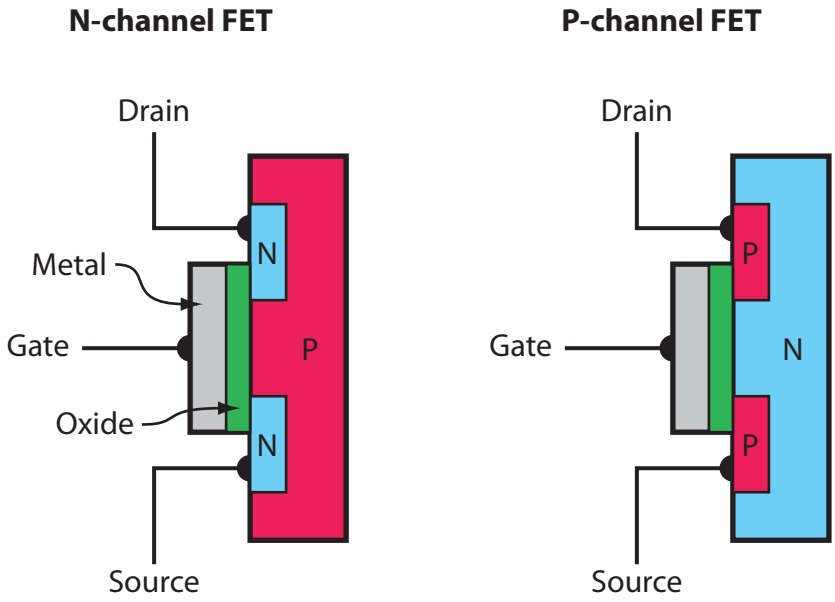


Figure 2.5: Construction of n- and p-channel FETs

2.3.4 Discussion

SAT solvers are rapidly gaining popularity, to the extent that they appear as an essential component of a large proportion of modern verification techniques. In some cases, the SAT solver's solution is the desired result, as is the case with our work on repairing cosmic ray damage in FPGAs (see Chapter 5). Often, as is the case with *bounded model checking* [21], a successful result is for the formula to be UNSAT, but if a solution is found, it represents a counter example that illustrates the cause of the verification failure.

2.4 Digital Electronics

Perhaps the most important understanding that must be reached about digital electronics is that the term is actually a misnomer – fundamentally, *all* electronic circuits are analogue. 'Digital' is actually a convenient abstraction that is adopted by designers in order to render the complexity inherent in large circuits tractable.

Modern silicon devices are most commonly constructed with the CMOS (complimentary metal oxide semiconductor) process. Fig. 2.5 shows the typical construction of n- and p-channel FETs (field-effect transistors). FETs behave somewhat like voltage controlled resistors – when they are turned off, the resistance across their source and drain is very large, in some cases in the teraohms. When turned on, this resistance reduces to near zero ohms. Typically, CMOS chip production processes are carefully tweaked to ensure that an n-channel FET will be turned on completely when its gate voltage approaches the positive supply rail (V_{cc}), and will turn off completely when its gate voltage approaches ground ($0V$). Conversely, p-channel FETs exhibit an opposite response, and are turned on when their gate input approaches $0V$ and off when the voltage approaches V_{cc}. It is important to understand that these switching curves are *not* discontinuous as

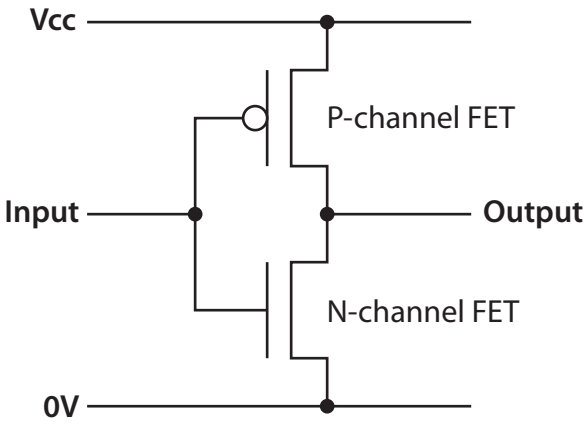


Figure 2.6: CMOS inverter circuit

the voltage varies, nor is the response instantaneous in time. Gate capacitance, in conjunction with finite drive current and non-zero resistance, limits the effective slew rate of the gate voltage, which restricts switching speed, and the solid-state physics of the devices themselves imply transfer curves with finite steepness.

2.4.1 Logic Gates

Fig. 2.6 represents perhaps the simplest circuit that is in common use: a CMOS inverter, constructed from a pair of FETs that have as-far-as-possible mirror image transfer curves. The behaviour of this circuit is as follows:

Input at or near $0V$ The upper p-channel FET turns on, and the lower n-channel FET turns off, creating a low-resistance path from V_{cc} to the output.

Input at or near V_{cc} The upper p-channel FET turns off, and the lower n-channel FET turns on, creating a low-resistance path from $0V$ to the output.

It is extremely important to understand that if the input voltage is not at some level close to $0V$ or V_{cc}, then *both* FETs are likely to be turned on to some extent. Normally, this only occurs very briefly (generally measurable in picoseconds for modern CMOS devices) during switching, but this can also occur for longer periods as a consequence of certain unwanted fault conditions. When both FETs are on, this creates a current path directly between OV and V_{cc} – this consumes power as a consequence of Ohm's law, whilst also causing localised heating of the transistors and associated wiring. This characteristic, along with the current necessary to overcome load capacitance during switching, is jointly responsible for the tendency for the required supply current of CMOS devices to vary proportionally with clock rate, and also why stopping (or substantially slowing) the clock can often reduce power requirements to almost zero.

Figs. 2.7 shows the circuit most commonly used to implement a logical NAND operation[15]. When bearing in mind the transfer curves of the component FETs, it is easy to

[15]A logical AND gate can, if necessary, be constructed by connecting an inverter in series with the output of a NAND gate.

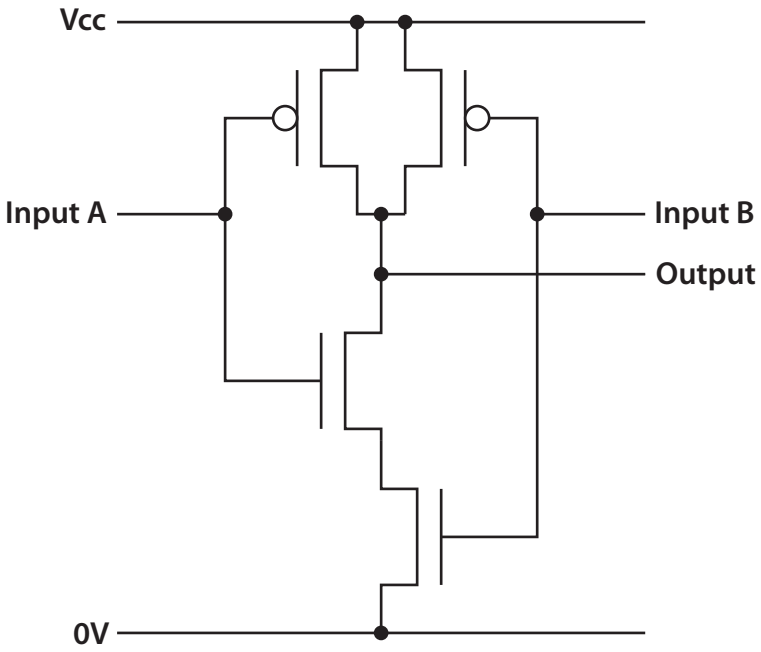


Figure 2.7: CMOS NAND gate circuit

visualise how this circuit validly implements the necessary logic operator – if either input is held to V_{cc}, the output is disconnected from $0V$ and connected to V_{cc}. Only when both inputs are at $0V$ do both of the p-channel FETs turn off, and both n-channel FETs turn on providing a path from $0V$ to the output.

Just as n-channel and p-channel devices can be regarded as 'digital' by assuming that their gate voltages are either $0V$, V_{cc} or rapidly transitioning between these states, it is also possible to further abstract the design problem by considering circuits at the gate level. Gates behave, broadly speaking, like their corresponding operators in classical Boolean logic[16], and are usually represented diagrammatically with simplified circuit symbols (see Fig. 2.8).

2.4.2 Combinational Circuits

Typically, any Boolean expression can be directly represented as a network of gates – if there are no cycles in the network, this is known as a *combinational circuit*. When an input of a combinational circuit changes, this causes a cascade of switching in the gate network that eventually settles, with the circuit's output faithfully representing the result of the Boolean expression that is being modelled.

It is important to reiterate that combinational circuits are still fundamentally analogue, and their ability to model logic is purely a convenient abstraction. In practice, as device geometries get smaller, as noise margins reduce and as clock rates increase, this abstraction becomes a decreasingly safe assumption. It is a common saying in electronic

[16]Though note the caveat that this does not necessarily apply to dynamically changing signals – see Chapter 3 for further discussion

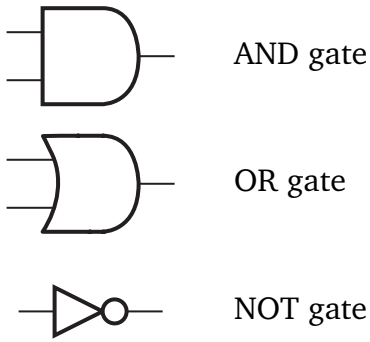


Figure 2.8: Circuit symbols for standard gates

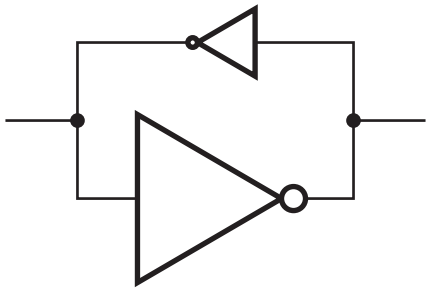


Figure 2.9: CMOS SRAM cell

engineering that, as frequencies rise, electronics starts to increasingly resemble plumbing. At high frequencies, electrons tend to migrate to the surface of conductors, and at microwave frequencies, electrons have a tendency not to stay inside wires at all, to the extent that waveguides (quite literally, copper pipes) are often used instead.

2.4.3 Memory

All CMOS memory devices either require some kind of feedback or must depend on analogue FET behaviour – as a consequence, no purely combinational circuit can implement memory. The simplest possible CMOS memory device is the DRAM (dynamic random access memory) cell, which essentially stores one bit of digital data in the charge on the gate of a single transistor [49]. Clearly, a DRAM cell is a fundamentally analogue device – gate charge leaks away quite rapidly, so in order to retain memory contents for more than a few milliseconds, it is necessary to periodically refresh the charge.

Fig. 2.9 depicts a SRAM (static random access memory) cell, which typically consists of a pair of inverter circuits connected in a ring. The tendency of this circuit is to maintain the voltage at the output either at OV or at V_{cc} indefinitely – no refresh circuitry is required. Like a DRAM cell, however, SRAM is also fundamentally analogue – in order to change the value stored in the memory cell, it is necessary to overpower the feedback path. Generally, this is achieved by arranging that the transistors in the backwards-facing inverter have less current sourcing capability than the transistor(s) driving the input.

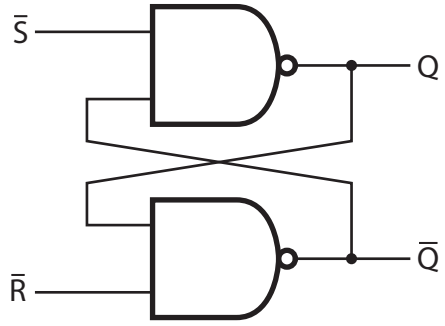


Figure 2.10: S-R flip flop

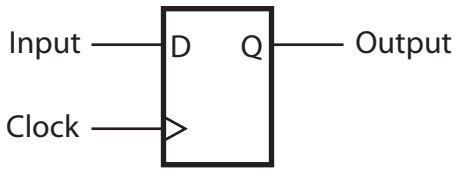


Figure 2.11: D-type flip flop circuit symbol

Predictably, the current consumption of SRAM devices tends to be directly proportional to how often data is written – the supply current necessary to maintain its contents is often so low that a small lithium or mercury battery is sufficient for several years of standby operation.

2.4.3.1 Flip-flops

DRAM and SRAM circuits are normally used to implement large capacity memories where individual bits are read and written comparatively infrequently – though they require comparatively little power and silicon area, they are often too slow for more demanding applications. In such cases, *flip-flops* are more commonly used, which use logic gates for switching rather than over-powering.

The simplest such circuit, the S-R flip-flop, is shown in Fig. 2.10. In this circuit, the output $\overline{Q}$ normally takes the opposite logic value to Q – the overbar notation is commonly used in electronics to indicate 'active low,' rather than the more usual 'active high' convention. When $\overline{R}$ and $\overline{S}$ are both at V_{cc}, the flip-flop retains the current output levels indefinitely. Bringing $\overline{R}$ to $0V$ *resets* the flip flop, *i.e.,* forces Q to $0V$ and $\overline{Q}$ to V_{cc}. Bringing $\overline{R}$ to V_{cc} *sets* the flip flop, *i.e.,* forces Q to V_{cc} and $\overline{Q}$ to $0V$. However, if for any reason $\overline{R}$ and $\overline{S}$ are allowed to simultaneously reach $0V$, both outputs go to V_{cc}, which is a condition that designers normally try to avoid.

The S-R flip flop has fallen out of favour in recent years, along with a number of other formerly-popular devices such as T-type and JK flip-flops, mostly due to the widespread adoption of VLSI design methodologies that tend to favour the use of D-type flip flops in preference, where D abbreviates 'delay.' The standard circuit symbol shown in Fig. 2.11. A D-type flip flop is controlled primarily by its clock input – Under most circumstances, it maintains the level at its Q output indefinitely, but when a positive edge (*i.e.,* an input

that transitions cleanly from $0V$ to V_{cc}) occurs at the clock input, it copies the level at its D input to its output. This behaviour tends to be very useful in practice – D-type flip-flops allow clocked circuits to be constructed straightforwardly, without requiring so much detailed consideration of timing relationships as is the case with most other kinds of flip flop.

2.4.4 Synchronous Circuits

Just as combinational circuits allow a significant abstraction away from analogue behaviour to be made, whose predictions can be relied upon, the *synchronous paradigm* provides a further abstraction that makes design yet simpler. In a nutshell, synchronous circuits allow gates to be connected to form combinational circuits, with feedback allowed via D-type flip flops, all of which are clocked synchronously by a single, global clock signal. In a strictly synchronous design, no gates whatsoever are allowed in the clock path. It is assumed that a clock rate is chosen such that, at worst case, all signals will have settled to their new values at all D inputs strictly before the next clock edge.

It should be stressed that it is not really the circuit that is synchronous – rather, it is the abstraction underlying the design technique that is important here. It is quite possible to build circuits based only on gates and D-type flip flops that are not inherently synchronous, which do not have behaviour that can be safely predicted by the usual synchronous abstraction. Particular examples include so-called gated clock synchronous circuits, where for reasons usually associated with reduction in power consumption, clock signals driving subcircuits or even individual flip flops are gated in order to disable them when they are not required. Such designs break the strict synchronous design rules, which means that correct functionality can only be relied upon if detailed timing analysis is carried out on the gated clock signals – it is no longer sufficient to simply assume that there will exist a global clock frequency sufficiently slow to allow the circuit to function correctly.

2.4.5 Asynchronous Circuits

Circuits that are designed at a level of abstraction above that of analogue though are not compliant with purely synchronous design rules are typically known as *asynchronous circuits* in the literature [118], though in recent times the term has more commonly been used to describe specific design abstractions that can be used to design clockless (or multiply-clocked) circuits. Some of the more common asynchronous design methodologies are summarised as follows:

Globally Asynchronous, Locally Synchronous (GALS) In this approach [34], several locally pure-synchronous circuits are interconnected such that correct operation is independent of local clock rates. Typically, asynchronous handshake protocols are adopted that allow communication to be synchronised without any need for a common clock. GALS circuits tend to be easier to design than self-timed circuits and are often also more space- and power-efficient than their purer counterparts.

Self-timed circuits are essentially clockless, in the sense that they do not require global or even local clock signals in order to function. Instead, computation proceeds

as values arrive on wires, with carefully designed protocols ensuring that gates are given sufficient time to settle to new values before new values are allowed to arrive. There are several approaches to self-timed circuit design, of which the most common are as follows:

Dual rail. What would typically be a single wire in the synchronous paradigm becomes a pair of wires in the dual rail paradigm [48], where $false$ is represented by a pulse on one wire, and $true$ by a pulse on the other. This approach allows wire pairs to carry timing information with them – data is inherently packetised, represented by edges rather than absolute values. Computation is carried out such that the edges representing results are only emitted once the input data edges have arrived and have been processed – as a consequence, feedback is possible, and such circuits naturally find their own ideal speed of operation, though there are no clocks as-such.

Bundled Data. In this approach [109], signalling at a global level is similar to dual rail logic, but computation is carried out locally in a manner closer to the synchronous paradigm than to the pure dual rail approach. This is claimed to reduce gate count and increase performance, which follows from the tendency for dual rail gates to be large and complex in comparison with conventional gates, though it reintroduces a requirement to carefully consider delay characteristics in order to ensure that circuits behave as intended.

Self-timed circuits have some interesting characteristics that are not typically shared by synchronous circuits. Their self-timed nature affords considerable advantages in terms of hardness to certain environmental constraints. If ambient temperature increases, this tends to increase resistances generally, so self-timed circuits slow down to compensate. Interestingly, varying the supply voltage can be used to control both speed and power consumption, with circuits tending to function correctly regardless. It is likely that, for this reason, self-timed circuits could be very effective in extreme environment applications, though at the time of writing little is known about how they might behave in practice.

2.4.6 Radiation Effects

Returning to the issues described in Section 2.1, radiation effects on digital electronics are a significant problem, particularly in aerospace applications, but also in many safety- and mission-critical terrestrial applications.

Total Dose Effects Digital circuits that are exposed to radiation for long periods typically exhibit gradually degraded performance, which is a consequence of cumulative physical damage to the chip itself. FETs affected by this process cease to be able to switch as effectively as they did initially – when turned off, their resistance may be lower than ideal, and similarly, when turned on, their resistance may be higher than expected. As a consequence, leakage current across the ladder structures found in CMOS gates gets steadily worse, thereby increasing overall power requirements. The degradation

also limits the switching current that can be delivered to the gates of downstream transistors, which reduces the chip's maximum achievable clock rate. Eventually, damage may accumulate that is severe enough to render the device non-functional.

Single-Event Effects Single-event effects (SEEs) are caused by impacts from isolated, highly energetic charged particles, generally cosmic rays and heavy ions from solar flares. Typically, a charged particle at these energies will pass through the chip's packaging and substrate, but due to its extremely high velocity (often close to the speed of light), it leaves a trail of charge behind it that affects the gates of any nearby transistors. In most cases, this charge dissipates within approximately 600 picoseconds[17], and may cause a transient voltage spike known as a single-event transient (SET). If such a transient affects the value stored in a memory element, this is typically referred to as a single-event upset (SEU). In some cases, *permanent latch-up* occurs, where a gate is permanently damaged by having both its n- and p-type transistors turned on simultaneously for an extended period, resulting in gate rupture due to the excessive heating caused by the current spike. This effect must be distinguished from the more usual latch-up mechanism familiar to designers of ground-based electronics, where latch-up effects are more usually caused by parasitic thyristor structures that are a consequence of typical contemporary VLSI design approaches. In conventional latch-up, an over-voltage condition at the input of a FET causes a parasitic thyristor to turn on, thereby permanently 'latching' the input to the relevant supply rail. Normally, this can be cleared by power cycling the chip, unlike permanent latch-up which cannot be resolved without replacing the device. Conventional latch-up can also be caused by single-event effects, either indirectly as a consequence of voltage spikes or directly as a consequence of a charged particle impact on or close to a parasitic thyristor. Spacecraft designers are concerned by both kinds of latch-up, and therefore design circuits to accommodate both conditions. Conventional latch-up is normally dealt with either by adopting latch-up-immune fabrication technologies such as Silicon-on-Insulator (SOI), or (particularly when commercial off-the-shelf parts are used in the design) by watchdog circuits that incorporate power cycling into their reset procedures. Permanent latch-up is normally accommodated through modular redundancy. In this dissertation, we are primarily concerned with permanent latch-up damage – this should be assumed unless otherwise specified in the text.

2.4.7 Discussion

This section has provided a very brief overview of digital electronics, particularly as seen from the point of view of spacecraft design. Engineering digital circuits for reliability is a complex problem, and typically requires deeper understanding of circuit behaviour than can be relied upon from design paradigms that were not intended for the purpose. To date, circuit design for radiation-hard circuits has largely been seen as a black art, depending much more upon the experience and skill of design engineers than on formal techniques. Much of the work reported in Part III of this thesis is aimed at redressing

[17]The duration 600ps was quoted by USAF engineers at a meeting with the author at Kirtland AFB in December 2005, and is otherwise unpublished.

this problem – it is always preferable to have some means of *proving* that a circuit will be reliable under given circumstances, rather than to have to rely only upon testing.

Part II

Foundations

Chapter 3

Transitional Logics

The work described in this chapter was published previously in [130, 133].

3.1 Introduction

Most contemporary design approaches assume an underlying *synchronous* paradigm, where a single global signal drives the clock inputs of every flip flop in the circuit. As a consequence, nearly all synthesis, simulation and model checking tools assume synchronous semantics. Designs in which this rule is relaxed are generally termed *asynchronous circuits*.

In a synchronous model, glitches (also known as static and dynamic hazards) do not cause problems unless they occur on a wire used as a clock input; with purely synchronous design rules[1] this cannot occur. However, such safety restrictions are *not* enforced by the semantics of either Verilog or VHDL – it is quite easy, deliberately or otherwise, to introduce unsafe logic into a clock path.

We present a technique, based upon abstract interpretation [42, 43], that allows the glitch states of asynchronous circuits to be identified and reasoned about. The approach taken involves a family of extended, multi-value *transitional* logics with an underlying dense continuous time model, and has applications in synthesis, simulation and model checking.

Our logics are extended with extra values that capture transitions as well as steady states, with an ability to distinguish *clean*, glitch-free signals from *dirty*, potentially glitchy ones. As a motivating example, consider the circuits shown in Fig. 3.1.I and 3.1.II, represented respectively by the expressions $(a \wedge c) \vee (\neg a \wedge b)$ and $(a \wedge c) \vee (\neg a \wedge b) \vee (b \wedge c)$. With respect to steady-state values for a, b and c, both circuits would appear to be identical, with the former representing a circuit that might result from naïve optimisation of the latter. Our technique can straightforwardly illustrate differences in their dynamic behaviour, however. Consider the critical case $a = \uparrow_0$ and $b = c = \mathsf{T}_0$ (see Fig. 3.1.III), representing b and c being wired to $true$ for all time, and a clean transition from $false$ to $true$ on a (this notation is defined fully in Section 3.3):

[1] Exactly one hazard-free global clock driving the clock inputs of all flip flops in the circuit.

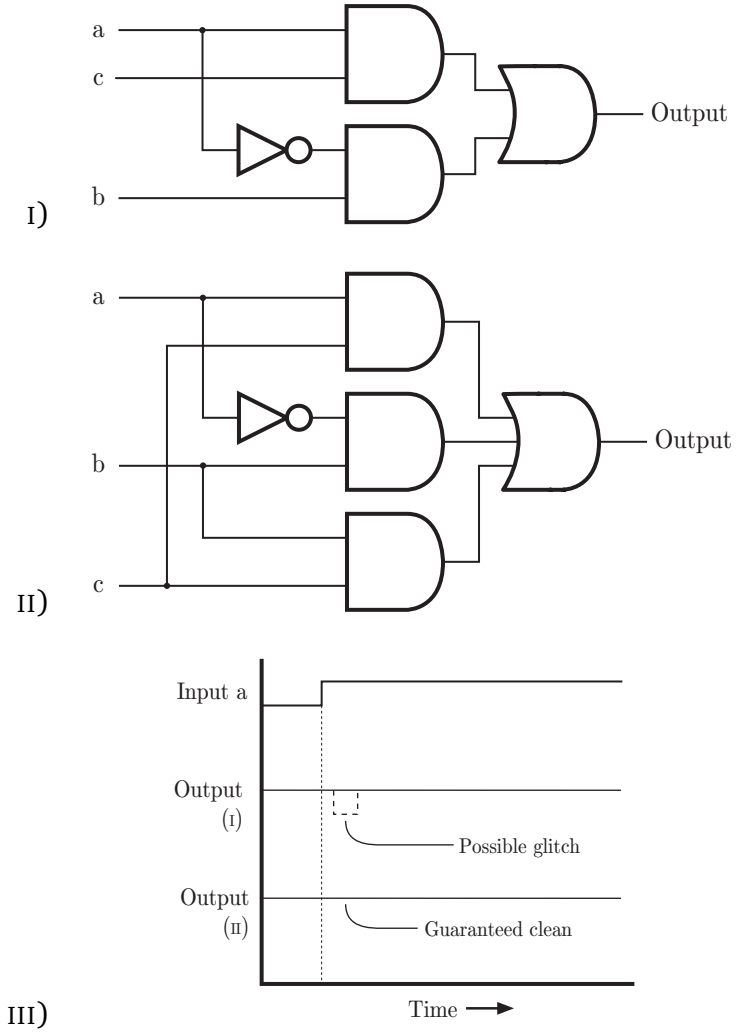


Figure 3.1: Comparison of dynamic behaviour

	$(a \wedge c) \vee (\neg a \wedge b)$			$(a \wedge c) \vee (\neg a \wedge b) \vee (b \wedge c)$	
$=$	$(\uparrow_0 \wedge \mathsf{T}_0) \vee (\neg\uparrow_0 \wedge \mathsf{T}_0)$		$=$	$(\uparrow_0 \wedge \mathsf{T}_0) \vee (\neg\uparrow_0 \wedge \mathsf{T}_0) \vee (\mathsf{T}_0 \wedge \mathsf{T}_0)$	
$=$	$\uparrow_0 \vee \downarrow_0$		$=$	$\uparrow_0 \vee \downarrow_0 \vee \mathsf{T}_0$	
$=$	$\mathsf{T}_{0..1}$	(I)	$=$	T_0	(II)

The result T_0 may be interpreted as '*true* for all time, with no glitches'. However, $\mathsf{T}_{0..1}$ represents '*true* with zero or one glitches', clearly demonstrating the poorer dynamic behaviour of the smaller circuit.

3.1.1 Achronous Analysis

Our transitional logics are all *achronous,* in that they do not consider absolute timing information, though they are nevertheless capable of reasoning about hazards in asyn-

chronous circuits. More formally, an *achronous analysis* adopts an independent attribute model [73], whereby signals are considered separately without regard to absolute time. In contrast, a *non-achronous analysis* may take into account absolute time. For example, given two signals: s_1 that transitions cleanly from 1 to 0 at time 5.0 and s_1 that transitions cleanly from 0 to 1 at time 7.5, a non-achronous analysis of $s_1 \wedge s_2$ could determine that the result is '0 for all time', whereas an achronous analysis (since it must by definition disregard absolute time) must conclude that the result may either be 0 for all time or a single positive pulse.

3.1.2 Hardware Components

In this chapter we consider four[2] basic building blocks: (perfect – zero delay) AND-gates, (perfect) NOT-gates, delay elements (whose delays may depend on time, and environmental factors like temperature, and thus are non-deterministic in a formal sense), and *inertial delay* elements. The difference between an ordinary delay and an inertial delay is that in the former the number of transitions on its input and output are equal, but in the latter a short-duration pulse from high-to-low and back (or vice versa) may be removed entirely from the output.

Of course, real circuits are not so general, in particular no practically realisable circuit of non-zero size can have zero-delay. Hence real-life circuits all correspond to combinations of the above gates with some form of delay element. For the point of designing synchronous hardware all that matters is the maximum delay which can occur from a circuit, so the exact positioning of the delays is often of little importance. When circuits are used asynchronously (*e.g.,* for designing self-timed circuits without a global clock or, more prosaically, when their output is being used to gate a clock signal locally) then their glitch behaviour is often critically important. This leads to two models (the *delay-insensitive* (DI) and *speed-independent* (SI) models) of real hardware. In the SI model logic elements may have delays, but wires do not; in the DI model both logic elements and wires have associated delay. One well-known fact about DI models is that it is impossible to construct an *isochronic fork*, whereby the transitions in output from a given gate will arrive delayed contemporaneously at two other gates. Reasoning in the DI model has become much more important recently as wire delays (*e.g.,* due to routing) have become dominant over single-gate element delays in modern VLSI technologies [99].

Ordinary circuits may be embedded in our model as follows. In the SI model each physical logic gate at the hardware level is seen as a perfect logic gate whose *output* is then passed through a delay element. In the DI model, each physical logic gate is seen as a perfect logic gate whose *input(s)* first pass through separate delays. In essence, the SI and DI models of a circuit are translations of a physical circuit into idealised circuits composed solely of our four perfect elements.

Now consider the circuit in Fig. 3.2.I. Seen as a perfect logic element, its output is always false regardless of the value of its input signal. Seen as an SI circuit (*i.e.,* delays on the output of the AND and NOT, as shown in Fig. 3.2.II), given an input F_1 which starts at false then transitions to true and back, the circuit will be false at all times except (possibly) for a small period just after the rising edge of the input, when the upper AND-

[2] A perfect OR-gate can be constructed from perfect AND- and NOT-gates using de Morgan's law.

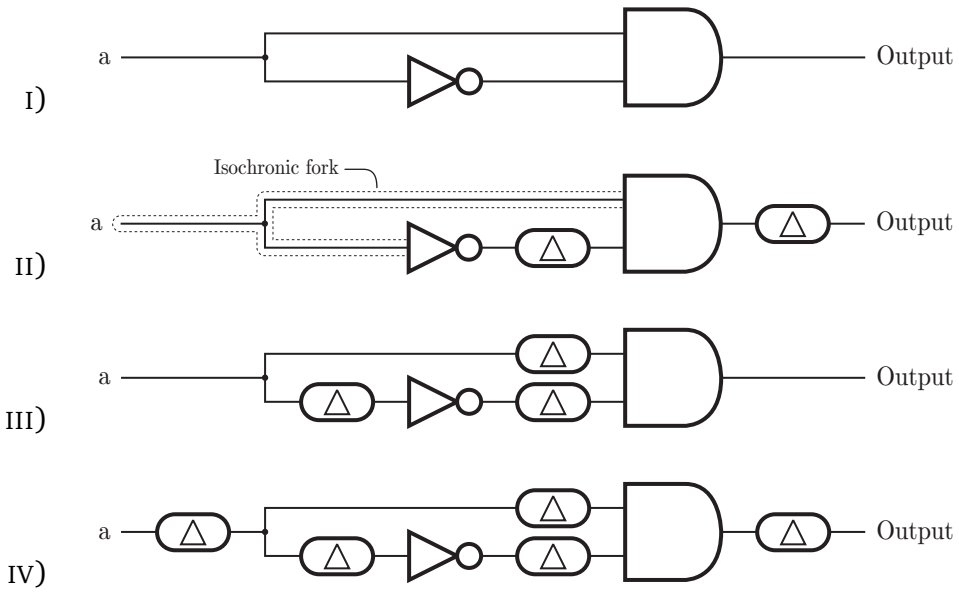


Figure 3.2: Delay models of the circuit $a \wedge \neg a$

input will already be $true$, but before the delayed NOT-output has yet become $false$. Thus the output, at this level of modelling, is $\mathsf{F}_{0..1}$ – an unpredictable choice between F_0 and F_1 – if we assume an inertial delay and F_1 if we assume a non-inertial delay[3].

In contrast, Fig. 3.2.III illustrates the DI model, where the separate delays on both inputs to the AND-gate mean that the same input signal F_1 may result in small positive pulses on both the rising and falling edge of the input; thus the output is described as $\mathsf{F}_{0..2}$ (*i.e.,* F_0 or F_1 or F_2). It is important to note that any of these three possible outputs may occur; delays may vary with time and temperature, and can also differ on whether an input signal is rising or falling.

Our abstract interpretation framework enables us to formally deduce the above behaviours of the circuit shown in Fig. 3.2. Our reasoning is *correct* because of the abstract interpretation framework[4]. In some situations our reasoning is also *complete* in that all abstractly-predicted behaviours may be made to happen by choosing suitable delay functions for the delay elements. For example, in the DI model, our abstraction of the above circuit maps abstract signal F_1 onto $\mathsf{F}_{0..2}$, but the SI model cannot produce F_2 however (positive) delay intervals are chosen.

3.1.3 Abstract Interpretation Basis

In general in abstract interpretation we start with a most precise abstract model from which we make further abstractions which are guaranteed correct by the abstract in-

[3]This argument assumes positive delays; at times later in the chapter we also allow (non-physically realisable) delays by negative time.

[4]In the abstract interpretation literature, the term *soundness* is sometimes used synonymously with correctness. In this thesis, we adopt the latter alternative in order to avoid confusing soundness (correctness) in abstract interpretation with soundness in mathematical logic.

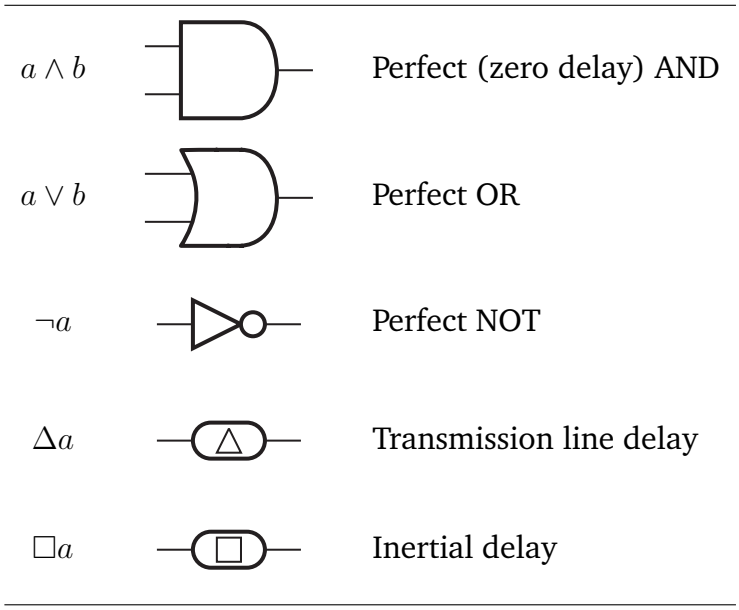


Figure 3.3: Circuit symbols

terpretation formulation. Our most precise model, $\wp(\mathbb{S})$, represents signals on wires as *sets* of functions from dense real time to the Booleans. Nondeterminism is captured straightforwardly – given a signal $\hat{s} \in \wp(\mathbb{S})$, a particular (deterministic) waveform s is represented by $\hat{s}$ if and only if $s \in \hat{s}$.

Our most precise abstract model is constructed by removing absolute timing information from signals in $\wp(\mathbb{S})$, resulting in values in $\wp(\mathbb{T})$. Operators on $\wp(\mathbb{T})$ are correct with respect to their concrete counterparts, so analyses carried out within our abstract framework are guaranteed to encompass all possible timing relationships, including any possible best- and worst-case behaviours. This has advantages and disadvantages: it means that we may predict multiple possible behaviours, some of which may not be possible in reality due to concrete timing constraints, though our predictions are always safe. The accuracy of the $\wp(\mathbb{T})$ abstraction depends largely on how much is knowable about timing relationships in the concrete domain – where detailed timing information can be determined, $\wp(\mathbb{T})$ may contain many false positive results. Our model is far more predictive when timing information is limited, however. When timing information is completely unknown, $\wp(\mathbb{T})$ is a complete abstraction when applied to delay insensitive circuits. Much related work in hardware analysis makes a similar assumption – see Section 3.9 for further discussion.

3.2 Concrete Domain

Definition 3.2.1. Concrete time $\mathbb{R}$ *is continuous, linear and dense, having no beginning or end.*

Definition 3.2.2. *A* signal *is a total function in* $\mathbb{S} : \mathbb{R} \to \{0, 1\}$ *from concrete time to the Boolean values. In order to avoid nonsensical behaviour, we restrict* $\mathbb{S}$ *to those functions that*

are finitely piecewise constant[5]*, i.e., there exists* $\{k_1, \ldots, k_n\}$ *which uniquely determines, and is determined by, a signal* $s \in \mathbb{S}$ *such that*

$$\begin{aligned} s(k_i) &= \neg s(k_{i+1}) && \forall 1 \leq i < n; \\ s(x) &= s(k_i) && \forall k_i \leq x < k_{i+1}; \\ s(-\infty) = s(x) &= \neg s(k_1) && \forall x < k_1; \\ s(+\infty) = s(x) &= s(k_n) && \forall x \geq k_n. \end{aligned}$$

The function $\Psi s \stackrel{\text{def}}{=} \{k_1, \ldots, k_n\}$ represents the bijection which returns the set of times at which signal s has transitions; $|\Psi s|$ represents the total number, n, of transitions made by s. As a further notational convenience, we denote the values of s at the beginning and end of time respectively as $s(-\infty)$ and $s(+\infty)$.

We model nondeterministic signals as members of the set $\wp(\mathbb{S})$; *e.g.,* delaying signal s by time δ, where $\delta_{min} \leq \delta \leq \delta_{max}$, gives $\{\lambda\tau.s(\tau - \delta) \mid \delta_{min} \leq \delta \leq \delta_{max}\}$.

3.3 Abstract Domain

3.3.1 Deterministic Traces

Definition 3.3.1. *A* deterministic trace $t \in \mathbb{T}$ *characterises a deterministic signal* $s \in \mathbb{S}$*, retaining the transitions but abstracting away the times at which they occur. Traces are denoted as finite lists of Boolean values bounded by angle brackets '*$\langle \ldots \rangle$*', and must contain at least one element – the empty trace '*$\langle\rangle$*' is not syntactically valid.*

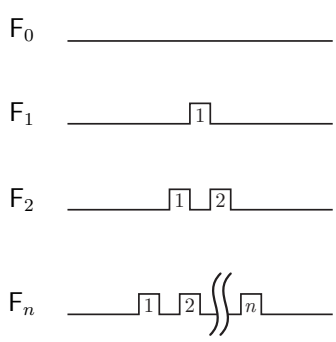
A *singleton trace*, denoted $\langle 0 \rangle$ or $\langle 1 \rangle$, represents a signal that remains at 0 or 1 respectively for all time. For traces with two or more elements, *e.g.,* $\langle a, \ldots, b \rangle$, a is the value at the beginning of time and b is the value at the end of time.

The trace $\langle 0, 1, 0 \rangle$ represents a signal that at the start of time takes the value 0, then at some later time switches cleanly to 1, then back to 0 again before the end of time. The instants at which these transitions occur are undefined, although their time order must be preserved.

Values within traces may be discriminated only by their transitions. Therefore, the trace $\langle 0, 0, 0, 0, 1, 1, 1 \rangle$ is equivalent to the trace $\langle 0, 1 \rangle$. It follows from this that all traces may be reduced to a form that resembles an alternating sequence $\langle \ldots, 0, 1, 0, 1, 0, 1, \ldots \rangle$. Any such sequence can be completely characterised by its start and end values, along with the number of intervening full cycles[6]. A convenient shorthand notation that takes advantage of this is defined in Fig. 3.4.

[5]Note that we do not consider signals that contain an infinite number of transitions, *e.g.,* clocks that oscillate for all time. We can, however, reason about such signals by 'windowing' them within finite intervals (windows) $[p, q]$ of $\mathbb{R}$, resulting in signals that are themselves finitely piecewise constant.

[6]It is of course also possible to represent traces completely in terms of their first (or last) element and their length. However, the representation chosen here turns out to be more convenient, *e.g.,* comparing $\uparrow_0$ with $\uparrow_4$ makes it immediately obvious that both represent traces that eventually transition from 0 to 1, with $\uparrow_0$ being 'cleaner' than $\uparrow_4$. The utility of this approach will become clear later.



F_0 The trace $\langle 0 \rangle$ that is 0 for all time.

F_1 The trace $\langle 0, 1, 0 \rangle$ that has 0 at the beginning and end, containing exactly one pulse.

F_2 The trace $\langle 0, 1, 0, 1, 0 \rangle$ that begins and ends with 0, containing exactly two pulses.

F_n The trace $\langle 0, 1_1, 0, 1_2, 0, \ldots, 0, 1_n, 0 \rangle$ that begins and ends with 0, containing exactly n positive-going pulses.

T_0 The trace $\langle 1 \rangle$ that is 1 for all time.

T_1 The trace $\langle 1, 0, 1 \rangle$ that has 1 at the beginning and end, containing exactly one pulse.

T_2 The trace $\langle 1, 0, 1, 0, 1 \rangle$ that begins and ends with 1, containing exactly two pulses.

T_n The trace $\langle 1, 0_1, 1, 0_2, 1, \ldots, 1, 0_n, 1 \rangle$ that begins and ends with 1, containing exactly n negative-going pulses.

$\uparrow_0$ The trace $\langle 0, 1 \rangle$ that cleanly transitions from 0 to 1.

$\uparrow_1$ The trace $\langle 0, 1, 0, 1 \rangle$ that transitions from 0 to 1 through exactly one intervening cycle.

$\uparrow_2$ The trace $\langle 0, 1, 0, 1, 0, 1 \rangle$ that transitions from 0 to 1 through exactly two intervening cycles.

$\uparrow_n$ The trace $\langle 0, 1_1, 0, \ldots, 0, 1_n, 0, 1 \rangle$ that transitions from 0 to 1 through exactly n intervening cycles.

$\downarrow_0$ The trace $\langle 1, 0 \rangle$ that cleanly transitions from 1 to 0.

$\downarrow_1$ The trace $\langle 1, 0, 1, 0 \rangle$ that transitions from 1 to 0 through exactly one intervening cycle.

$\downarrow_2$ The trace $\langle 1, 0, 1, 0, 1, 0 \rangle$ that transitions from 1 to 0 through exactly two intervening cycles.

$\downarrow_n$ The trace $\langle 1, 0_1, 1, \ldots, 1, 0_n, 1, 0 \rangle$ that transitions from 1 to 0 through exactly n intervening cycles.

Figure 3.4: Shorthand notation: deterministic traces

3.3.2 Nondeterministic Traces

Following the approach taken in Section 3.2.2, we represent nondeterministic traces $\hat{t} \in \wp(\mathbb{T})$ as sets of deterministic traces[7].

The need for this extra structure is demonstrated by the following example. Let us attempt to specify the meaning of the expression $\langle 0,1\rangle \wedge \neg\langle 0,1\rangle$, which represents the effect of feeding a clean transition from 0 to 1 to the a input of the circuit shown in Fig. 3.2. The $\neg$ can be evaluated trivially, giving $\langle 0,1\rangle \wedge \langle 1,0\rangle$. At first sight, it may appear that the resulting trace should be $\langle 0,0\rangle$ or just $\langle 0\rangle$. This would be the case if certain constraints on the exact times of the transitions of the $\langle 1,0\rangle$ and $\langle 0,1\rangle$ traces were met, but it is not sufficient to cope with all possibilities. If the $\langle 1,0\rangle$ transition occurs before the $\langle 0,1\rangle$ transition, then the result is indeed $\langle 0\rangle$. Should the transitions occur in the opposite order, the result is $\langle 0,1,0\rangle$. Formally,

$$\{\langle 0,1\rangle\} \wedge \neg\{\langle 0,1\rangle\} = \{\langle 0,1\rangle\} \wedge \{\langle 1,0\rangle\} = \{\langle 0\rangle\} \cup \{\langle 0,1,0\rangle\} = \{\langle 0\rangle, \langle 0,1,0\rangle\}$$

Definition 3.3.2. *Where $\hat{t} \in \wp(\mathbb{T})$ and $\hat{u} \in \wp(\mathbb{T})$, the* nondeterministic choice $\hat{t} \mid \hat{u}$ *is synonymous with $\hat{t} \cup \hat{u}$. For notational compactness, we allow either or both of the arguments of $\mid$ to range over $\mathbb{T}$, e.g., where $t \in \mathbb{T}$, the expression $t \mid \hat{u}$ is equivalent to $\{t\} \mid \hat{u}$.*

The '|' operator allows the above equation to be expressed more compactly as follows:

$$\langle 0,1\rangle \wedge \neg\langle 0,1\rangle = \langle 0,1\rangle \wedge \langle 1,0\rangle = \langle 0\rangle \mid \langle 0,1,0\rangle$$

Using the shorthand notation, this may equivalently be written as:

$$\uparrow_0 \wedge \neg\uparrow_0 = \uparrow_0 \wedge \downarrow_0 = \mathsf{F}_0 \mid \mathsf{F}_1$$

Definition 3.3.3. *Letting X range over $\{\mathsf{T}, \mathsf{F}, \uparrow, \downarrow\}$,*

$$X_{m..n} \stackrel{\text{def}}{=} X_m \mid X_{m+1} \mid \cdots \mid X_n \qquad\qquad X_{a_1|\ldots|a_n} \stackrel{\text{def}}{=} X_{a_1} \mid \ldots \mid X_{a_n}$$

For example, $\mathsf{F}_0 \mid \mathsf{F}_1$ may equivalently be written as $\mathsf{F}_{0|1}$, and rather than fully enumerating a long list of alternate pulse counts of the form $\mathsf{F}_{m|m+1|\ldots|n-1|n}$, the preferred notation $\mathsf{F}_{m..n}$ may be used instead. These notations may be combined, *e.g.,* $\mathsf{F}_{0|3|5..7|10..12}$.

Nondeterministic choice obeys all the laws of set union, *e.g.,*

$$a \mid a = a \qquad\qquad a \mid b = b \mid a \qquad\qquad a \mid (b \mid c) = (a \mid b) \mid c = a \mid b \mid c$$

From this, various subscript laws follow, *e.g.,*

$$X_{a|a} = X_a \quad X_{a..a} = X_a$$

$$X_{a..b} \mid X_{c..d} = \begin{cases} X_{\min(a,c)..\max(b,d)} & \text{if } c \le b \wedge a \le d; \\ X_{a..b|c..d} & \text{otherwise.} \end{cases}$$

[7] We adopt the convention that t and $\hat{t}$ are separate variables that range over $\mathbb{T}$ and $\wp(\mathbb{T})$ respectively.

Definition 3.3.4. *It is convenient to name the following least upper bounds w.r.t.* $\langle \wp(\mathbb{T}), \subseteq \rangle$*:*

$$\mathsf{F}_\bigstar \stackrel{\text{def}}{=} \mathsf{F}_{0..\infty} \quad \mathsf{T}_\bigstar \stackrel{\text{def}}{=} \mathsf{T}_{0..\infty} \quad \uparrow_\bigstar \stackrel{\text{def}}{=} \uparrow_{0..\infty} \quad \downarrow_\bigstar \stackrel{\text{def}}{=} \downarrow_{0..\infty}$$
$$\bigstar \stackrel{\text{def}}{=} \mathsf{F}_\bigstar \cup \mathsf{T}_\bigstar \cup \uparrow_\bigstar \cup \downarrow_\bigstar$$

3.3.3 Galois Connection

In program analysis, it is common practice to relate partially ordered concrete and abstract domains with concretisation γ and abstraction α functions that together form a Galois connection (see also Section 2.2.1.3). In this section, we define α and γ functions that relate the domains defined earlier, then show that they form a valid Galois connection.

Definition 3.3.5. *Given a deterministic concrete signal* $s \in \mathbb{S}$*, the abstraction function* $\beta : \mathbb{S} \to \mathbb{T}$ *returns the corresponding deterministic trace:*

$$\begin{aligned} \beta s &\stackrel{\text{def}}{=} \langle s(-\infty), s(k_1), \ldots, s(k_n) \rangle && \text{where } \{k_1, \ldots, k_n\} = \Psi s \\ &= \langle s(-\infty), \neg s(-\infty), s(-\infty), \neg s(-\infty), \ldots \rangle \end{aligned}$$

Note that βs has exactly $1 + |\Psi s|$ elements.

Definition 3.3.6. *The abstraction function* $\alpha : \wp(\mathbb{S}) \to \wp(\mathbb{T})$ *and concretisation function* $\gamma : \wp(\mathbb{T}) \to \wp(\mathbb{S})$ *are defined as follows:*

$$\alpha \hat{s} \stackrel{\text{def}}{=} \{\beta s \mid s \in \hat{s}\} \qquad \gamma \hat{t} \stackrel{\text{def}}{=} \{s \in \mathbb{S} \mid \beta s \in \hat{t}\}$$

Definition 3.3.7. *Letting* $\sim : \mathbb{S} \times \mathbb{S} \to \mathbb{B}$ *represent the equivalence relation* $s_1 \sim s_2 \Leftrightarrow \beta s_1 = \beta s_2$*, the set* $\mathbb{S}^\sharp \stackrel{\text{def}}{=} \mathbb{S}/\!\sim$ *is the set of equivalence classes in* $\mathbb{S}$ *with respect to* $\sim$*. The set* $[s] \stackrel{\text{def}}{=} \{s' \in \mathbb{S} \mid \beta s = \beta s'\}$ *represents, for any* $s \in \mathbb{S}$*, the equivalence class containing that element.*

Note that $\mathbb{S}^\sharp$ is isomorphic with $\mathbb{T}$.

Theorem 3.3.1. *Together, the functions* $\langle \alpha, \gamma \rangle$ *form a Galois connection between* $\wp(\mathbb{S})$ *and* $\wp(\mathbb{T})$*. Following Cousot & Cousot [43], Theorem 5.3.0.4 and Corollary 5.3.0.5, pp. 273, it is sufficient to show that* $\alpha \circ \gamma(\hat{x}) \sqsubseteq \hat{x}$ *and* $\gamma \circ \alpha(\hat{x}) \sqsupseteq \hat{x}$*. We choose to prove instead the slightly stronger* $\alpha \circ \gamma(\hat{x}) = \hat{x}$*, and since the ordering relations on* $\wp(\mathbb{S})$ *and* $\wp(\mathbb{T})$ *are subset inclusion, we write* $\supseteq$ *rather than* $\sqsupseteq$*. Proof; letting* $\hat{x} = \{x_1, \ldots, x_n\}$

1. $\alpha \circ \gamma(\hat{x}) = \alpha\{s \in \mathbb{S} \mid \beta s \in \hat{x}\} = \{\beta s' \mid s' \in \{s \in \mathbb{S} \mid \beta s \in \hat{x}\}\} = \{\beta s' \mid \beta s' \in \hat{x}\} = \hat{x}.$

2. $\gamma \circ \alpha(\hat{x}) = \gamma \circ \alpha\{x_1, \ldots, x_n\} = \gamma\{\beta x_1, \ldots, \beta x_n\} = \gamma\{\beta x_1\} \cup \cdots \cup \gamma\{\beta x_n\} = \{s \in \mathbb{S} \mid \beta s = \beta\{x_1\}\} \cup \cdots \cup \{s \in \mathbb{S} \mid \beta s = \beta\{x_n\}\} = [x_1] \cup \cdots \cup [x_n] \supseteq \{x_1, \ldots, x_n\} = \hat{x}.$

$\wedge^\sharp$	F_0	F_n	T_0	T_n	$\uparrow_0$	$\uparrow_n$	$\downarrow_0$	$\downarrow_n$
F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0
F_m	F_0	$\mathsf{F}_{0..m+n-1}$	F_m	$\mathsf{F}_{0..m+n}$	$\mathsf{F}_{0..m}$	$\mathsf{F}_{0..m+n}$	$\mathsf{F}_{0..m}$	$\mathsf{F}_{0..m+n}$
T_0	F_0	F_n	T_0	T_n	$\uparrow_0$	$\uparrow_n$	$\downarrow_0$	$\downarrow_n$
T_m	F_0	$\mathsf{F}_{0..m+n}$	T_m	$\mathsf{T}_{1..m+n}$	$\uparrow_{0..m}$	$\uparrow_{0..m+n}$	$\downarrow_m$	$\downarrow_{0..m+n}$
$\uparrow_0$	F_0	$\mathsf{F}_{0..n}$	$\uparrow_0$	$\uparrow_{0..n}$	$\uparrow_0$	$\uparrow_{0..n}$	$\mathsf{F}_{0..1}$	$\mathsf{F}_{0..n+1}$
$\uparrow_m$	F_0	$\mathsf{F}_{0..m+n}$	$\uparrow_m$	$\uparrow_{0..m+n}$	$\uparrow_{0..m}$	$\uparrow_{0..m+n}$	$\mathsf{F}_{0..m+1}$	$\mathsf{F}_{0..m+n+1}$
$\downarrow_0$	F_0	$\mathsf{F}_{0..n}$	$\downarrow_0$	$\downarrow_n$	$\mathsf{F}_{0..1}$	$\mathsf{F}_{0..n+1}$	$\downarrow_0$	$\downarrow_{0..n}$
$\downarrow_m$	F_0	$\mathsf{F}_{0..m+n}$	$\downarrow_m$	$\downarrow_{0..m+n}$	$\mathsf{F}_{0..m+1}$	$\mathsf{F}_{0..m+n+1}$	$\downarrow_{0..m}$	$\downarrow_{0..m+n}$

$\vee^\sharp$	F_0	F_n	T_0	T_n	$\uparrow_0$	$\uparrow_n$	$\downarrow_0$	$\downarrow_n$
F_0	F_0	F_n	T_0	T_n	$\uparrow_0$	$\uparrow_n$	$\downarrow_0$	$\downarrow_n$
F_m	F_m	$\mathsf{F}_{1..m+n}$	T_0	$\mathsf{T}_{0..m+n}$	$\uparrow_{0..m}$	$\uparrow_{0..m+n}$	$\downarrow_{0..m}$	$\downarrow_{0..m+n}$
T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0
T_m	T_m	$\mathsf{T}_{0..m+n}$	T_0	$\mathsf{T}_{0..m+n-1}$	$\mathsf{T}_{0..m}$	$\mathsf{T}_{0..m+n-1}$	$\mathsf{T}_{0..m}$	$\mathsf{T}_{0..m+n-1}$
$\uparrow_0$	$\uparrow_0$	$\uparrow_{0..n}$	T_0	$\mathsf{T}_{0..n}$	$\uparrow_0$	$\uparrow_{0..n}$	$\mathsf{T}_{0..1}$	$\mathsf{T}_{0..n+1}$
$\uparrow_m$	$\uparrow_m$	$\uparrow_{0..m+n}$	T_0	$\mathsf{T}_{0..m+n-1}$	$\uparrow_{0..m}$	$\uparrow_{0..m+n}$	$\mathsf{T}_{0..m+1}$	$\mathsf{T}_{0..m+n+1}$
$\downarrow_0$	$\downarrow_0$	$\downarrow_{0..n}$	T_0	$\mathsf{T}_{0..n}$	$\mathsf{T}_{0..1}$	$\mathsf{T}_{0..n+1}$	$\downarrow_0$	$\downarrow_{0..n}$
$\downarrow_m$	$\downarrow_m$	$\downarrow_{0..m+n}$	T_0	$\mathsf{T}_{0..m+n-1}$	$\mathsf{T}_{0..m+1}$	$\mathsf{T}_{0..m+n+1}$	$\downarrow_{0..m}$	$\downarrow_{0..m+n}$

$\neg^\sharp$	
F_0	T_0
F_n	T_n
T_0	F_0
T_n	F_n
$\uparrow_0$	$\downarrow_0$
$\uparrow_n$	$\downarrow_n$
$\downarrow_0$	$\uparrow_0$
$\downarrow_n$	$\uparrow_n$

$\Delta^\sharp$	
F_0	F_0
F_n	F_n
T_0	T_0
T_n	T_n
$\uparrow_0$	$\uparrow_0$
$\uparrow_n$	$\uparrow_n$
$\downarrow_0$	$\downarrow_0$
$\downarrow_n$	$\downarrow_n$

$\Box^\sharp$	
F_0	F_0
F_n	$\mathsf{F}_{0..n}$
T_0	T_0
T_n	$\mathsf{T}_{0..n}$
$\uparrow_0$	$\uparrow_0$
$\uparrow_n$	$\uparrow_{0..n}$
$\downarrow_0$	$\downarrow_0$
$\downarrow_n$	$\downarrow_{0..n}$

where $m > 0$, $n > 0$.

Figure 3.5: Boolean functions on traces

3.4 Circuits

Definition 3.4.1. *Circuits are modelled by composing four basic operators: zero delay 'and' $\wedge$, zero delay 'not' $\neg$, transmission line delay Δ and inertial delay $\Box$, which are defined on the concrete domain as follows:*

$$\wedge \stackrel{\text{def}}{=} \lambda(\hat{s}_1, \hat{s}_2).\{\lambda\tau.s_1(\tau) \wedge s_2(\tau) \mid s_1 \in \hat{s}_1 \wedge s_2 \in \hat{s}_2\}$$
$$\neg \stackrel{\text{def}}{=} \lambda\hat{s}.\{\lambda\tau.\neg s(\tau) \mid s \in \hat{s}\}$$
$$\Delta \stackrel{\text{def}}{=} \gamma \circ \alpha$$
$$\Box \stackrel{\text{def}}{=} \gamma \circ \Box^\sharp \circ \alpha$$

Their abstract counterparts are defined as follows:

$$\wedge^\sharp \stackrel{\text{def}}{=} \alpha \circ \wedge \circ \langle\gamma, \gamma\rangle$$
$$\neg^\sharp \stackrel{\text{def}}{=} \alpha \circ \neg \circ \gamma$$
$$\Delta^\sharp \stackrel{\text{def}}{=} \lambda x.x$$
$$\Box^\sharp \stackrel{\text{def}}{=} \lambda\hat{t}.\{t \in \mathbb{T} \mid \exists t' \in \hat{t}.\mathit{Val}(t) = \mathit{Val}(t') \wedge \mathit{Subs}(t) \leq \mathit{Subs}(t')\}$$

where $\mathit{Val} : \mathbb{T} \to \{\mathsf{F}, \mathsf{T}, \uparrow, \downarrow\}$ *and* $\mathit{Subs} : \mathbb{T} \to \mathbb{N}$ *are defined as follows:*

$$\mathit{Val}(X_n) \stackrel{\text{def}}{=} X \qquad\qquad \mathit{Subs}(X_n) \stackrel{\text{def}}{=} n$$

Note that defining $\Box$ in terms of α, γ and $\Box^\sharp$ is unusual, though convenient.

And. The function $\wedge : \wp(\mathbb{S}) \times \wp(\mathbb{S}) \to \wp(\mathbb{S})$ represents a perfect zero-delay AND gate. Its abstract counterpart, $\wedge^\sharp : \wp(\mathbb{T}) \times \wp(\mathbb{T}) \to \wp(\mathbb{T})$, is defined in terms of $\wedge$ by composition with α and γ; note that our achronous semantics is based upon an independent attribute model [73].

Or. The function $\vee : \wp(\mathbb{S}) \times \wp(\mathbb{S}) \to \wp(\mathbb{S})$ represents a perfect zero-delay OR gate. Since it can be defined in terms of $\wedge$, $\neg$ and de Morgan's law, *i.e.,* $a \vee b \stackrel{\text{def}}{=} \neg(\neg a \wedge \neg b)$, for the purposes of this chapter we do not consider $\vee$ as a basic operator. Where space allows, $\vee$ is tabulated fully, though it is not otherwise discussed.

Not. The bijective function $\neg : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ represents a perfect zero delay NOT gate. As with $\wedge$, we define $\neg^\sharp : \wp(\mathbb{T}) \to \wp(\mathbb{T})$ by composition of the concrete operator $\neg$ with α and γ. When tabulated, $\wedge^\sharp$ and $\neg^\sharp$ behave as shown in Fig. 3.5.

Transmission line (non-inertial) delay. Our definition of transmission line delay is essentially a superset of all possible delay functions that preserve the underlying trace structure of the signal. The definition, $\gamma \circ \alpha$, captures this behaviour straightforwardly; the α function abstracts away all details of time, though preserves transitions and the

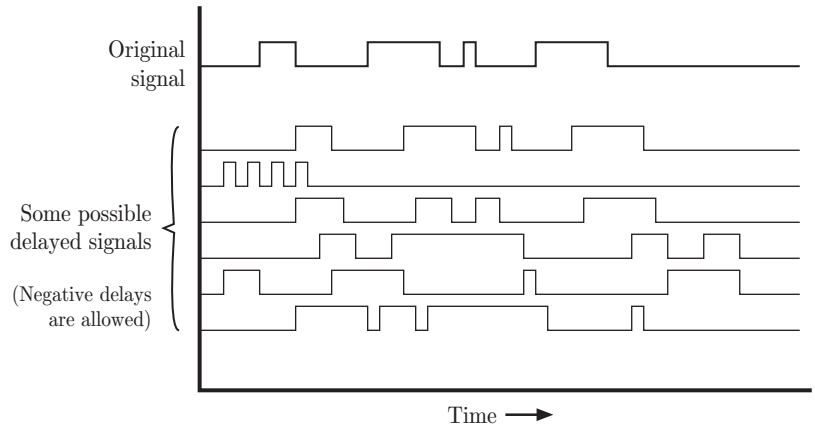


Figure 3.6: Transmission line delay

values at the beginning and end of time, then γ concretises this, resulting in the set of all possible traces with similar structure. This definition is more permissive than more typical notions of delay in that it includes negative as well as positive time shifts as well as transformations that can stretch or compress (though not remove or reorder) pulses.

Inertial delay. Inertial delay is broadly similar to transmission line delay, in that, as well as changing the time at which transitions may occur, one or more complete pulses (*i.e.,* pairs of adjacent transitions) may be removed. This models a common property of some physical components, whereby very short pulses are 'soaked up' by internal capacitance and/or inductance and thereby not passed on. We model inertial delay in the abstract domain – in effect, nondeterministic traces are mapped to convex hulls of the form $\mathsf{F}_{0..a} \mid \mathsf{T}_{0..b} \mid \uparrow_{0..c} \mid \downarrow_{0..d}$. The concrete inertial delay operator $\Box$ is defined in terms of $\Box^\sharp$ by composition with γ and α, so as with transmission line delay, it encompasses all possible inertial delay functions. It can be noted that, for all $\hat{s} \in \wp(\mathbb{S})$, $\Delta\hat{s} \subseteq \Box\hat{s}$.

Circuit Symbols As shown in Fig. 3.3, we adopt standard electronic engineering notation for the perfect gates $\wedge$, $\vee$ and $\neg$. Note that we adopt slight variations on the usual symbol for delay in order to distinguish inertial delay $\Box$ from transmission line delay Δ.

3.4.1 Correctness and Completeness

An abstract function $f^\sharp$ may be described as *correct* with respect to a concrete function f if all behaviours exhibited by f are within the set of possible behaviours predicted by $f^\sharp$. Where these sets are identical (*i.e.,* where $f^\sharp$ predicts all possible behaviours of f), *completeness* holds [58, 59, 60, 56, 101], two forms of which are defined below.

Definition 3.4.2. *Given a concrete domain D and an abstract domain $D^\sharp$, related by functions $\langle \alpha, \gamma \rangle$ that form a Galois connection (i.e., $\alpha \circ \gamma(x) \sqsubseteq x$ and $\gamma \circ \alpha(x) \sqsupseteq x$), a pair of functions $f : D \to D$ and $f^\sharp : D^\sharp \to D^\sharp$ may be said to be* correct *iff the following*

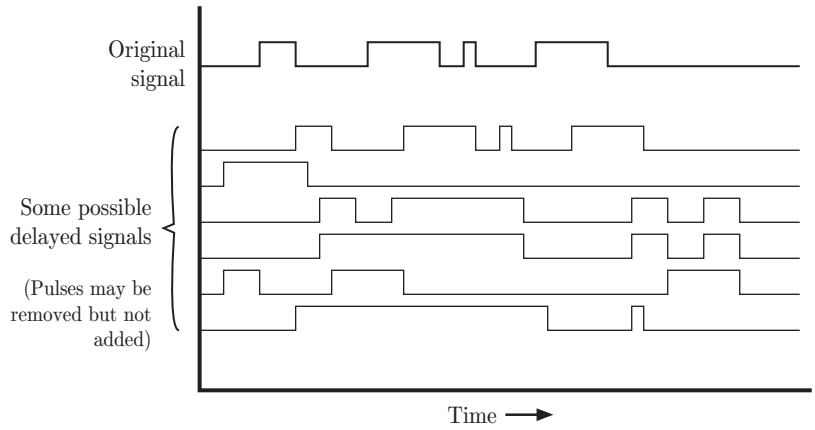


Figure 3.7: Inertial delay

(equivalent) relations hold:

$$\alpha \circ f \sqsubseteq f^\sharp \circ \alpha \qquad\qquad f \circ \gamma \sqsubseteq \gamma \circ f^\sharp$$

Definition 3.4.3. *Let* $f^\sharp_{best} \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma$.

Recalling the definitions of α- and γ-completeness from Section 2.2.1.5,

Definition 3.4.4. *When* $f^\sharp = f^\sharp_{best}$ *and* $f \circ \gamma = \gamma \circ f^\sharp$*, the property* γ-completeness *holds.*

Definition 3.4.5. *When* $f^\sharp = f^\sharp_{best}$ *and* $\alpha \circ f = f^\sharp \circ \alpha$*, the property* α-completeness *holds.*

Note that α-completeness and γ-completeness are orthogonal properties; neither implies the other, though if either or both kinds of completeness hold, correctness must also hold.

Theorem 3.4.1. *The transmission line delay operator* $(\Delta, \Delta^\sharp)$ *is correct,* α*-complete and* γ*-complete. Proof:*

1. $\Delta^\sharp_{best} = \alpha \circ \Delta \circ \gamma = \alpha \circ \gamma \circ \alpha \circ \gamma = \alpha \circ \gamma = (\lambda x.x) = \Delta^\sharp$.
2. α*-completeness:* $\alpha \circ \Delta = \alpha \circ \gamma \circ \alpha = \alpha = (\lambda x.x) \circ \alpha = \Delta^\sharp \circ \alpha$.
3. γ*-completeness:* $\Delta \circ \gamma = \gamma \circ \alpha \circ \gamma = \gamma = \gamma \circ (\lambda x.x) = \gamma \circ \Delta^\sharp$.

Theorem 3.4.2. *The inertial delay operator* $(\Box, \Box^\sharp)$ *is correct,* α*-complete and* γ*-complete. Proof:*

1. $\Box^\sharp_{best} = \alpha \circ \Box \circ \gamma = \alpha \circ \gamma \circ \Box^\sharp \circ \alpha \circ \gamma = \Box^\sharp$.
2. α*-completeness:* $\alpha \circ \Box = \alpha \circ \gamma \circ \Box^\sharp \circ \alpha = \Box^\sharp \circ \alpha$.
3. γ*-completeness:* $\Box \circ \gamma = \gamma \circ \Box^\sharp \circ \alpha \circ \gamma = \gamma \circ \Box^\sharp$.

Theorem 3.4.3. *The perfect NOT operator* $(\neg, \neg^\sharp)$ *is correct,* α*-complete and* γ*-complete. Proof:*

1. $\neg^\sharp_{best} = \alpha \circ \neg \circ \gamma = \neg^\sharp$.
2. *Since* $\neg$ *is a bijection,* $\gamma \circ \alpha \circ \neg = \neg \circ \gamma \circ \alpha$.
3. α*-completeness:* $\alpha \circ \neg = \alpha \circ \gamma \circ \alpha \circ \neg = \alpha \circ \neg \circ \gamma \circ \alpha = \neg^\sharp \circ \alpha$.
4. γ*-completeness:* $\neg \circ \gamma = \neg \circ \gamma \circ \alpha \circ \gamma = \gamma \circ \alpha \circ \neg \circ \gamma = \gamma \circ \neg^\sharp$.

Theorem 3.4.4. *The perfect AND operator* $(\wedge, \wedge^\sharp)$ *is correct*[8]. *Proof:*

1. $\wedge \circ \langle \gamma, \gamma \rangle \subseteq \gamma \circ \wedge^\sharp = \gamma \circ \alpha \circ \wedge \circ \langle \gamma, \gamma \rangle$.

Note that whilst perfect, zero delay AND is correct but not complete, a composite speed-insensitive AND ($\wedge_{SI} \stackrel{\text{def}}{=} \Delta \circ \wedge, \wedge^\sharp_{SI} \stackrel{\text{def}}{=} \Delta^\sharp \circ \wedge^\sharp$) can be straightforwardly be shown to be γ-complete, but not α-complete. Dually, delay-independent AND ($\wedge_{DI} \stackrel{\text{def}}{=} \wedge \circ \langle \Delta, \Delta \rangle, \wedge^\sharp_{DI} \stackrel{\text{def}}{=} \wedge^\sharp \circ \langle \Delta^\sharp, \Delta^\sharp \rangle$) is α- but not γ-complete. We find, however, that ($\wedge_{complete} \stackrel{\text{def}}{=} \Delta \circ \wedge \circ \langle \Delta, \Delta \rangle, \wedge^\sharp_{complete} \stackrel{\text{def}}{=} \Delta^\sharp \circ \wedge^\sharp \circ \langle \Delta^\sharp, \Delta^\sharp \rangle$) is both α- and γ-complete.

3.5 Finite Versions of the Abstract Domain

The abstract domain defined in Section 3.3 allows arbitrary asynchronous combinational circuits to be reasoned about. In this section we present a number of simplifications of this basic model which allow accuracy to be traded off against levels of abstraction. The model presented in Section 3.3 is useful in identifying possible glitches within circuits, though in this case generally one is interested in whether a particular signal *can* glitch, rather than the *number* of possible glitches – this requires less information than that captured by our original abstraction. It follows that further abstraction should be possible, which is indeed the case.

3.5.1 Collapsing Non-Zero Subscripts: the 256-value Transitional Logic $\mathbb{T}_{256}$

Mapping all non-zero subscript traces $t \in X_{1..\infty}$ to the single abstract value X_+, for X ranging over $\{\mathsf{F}, \mathsf{T}, \uparrow, \downarrow\}$, makes it possible to define a finite abstract domain with a Galois connection to $\mathbb{T}$. This domain has the desirable property of abstracting away details of 'how glitchy' a trace may be, whilst retaining the ability to distinguish *clean* traces from *dirty* traces.

Definition 3.5.1. *The abstract domain of* subscript-collapsed deterministic traces *is the set* $\mathbb{T}_\mathsf{c} \stackrel{\text{def}}{=} \{\mathsf{F}_0, \mathsf{F}_+, \mathsf{T}_0, \mathsf{T}_+, \uparrow_0, \uparrow_+, \downarrow_0, \downarrow_+\}$. *Following the usual convention, the corresponding abstract domain of* subscript-collapsed nondeterministic traces *is the set* $\mathbb{T}_{256} \stackrel{\text{def}}{=} \wp(\mathbb{T}_\mathsf{c})$.

Note that unlike $\mathbb{T}$ and $\wp(\mathbb{T})$, both $\mathbb{T}_\mathsf{c}$ and $\wp(\mathbb{T}_\mathsf{c})$ are finite sets, with 8 and 256 members respectively.

[8]Note that we adopt an independent attribute model when considering the dyadic nature of AND.

$\wedge_c$	F_0	F_+	T_0	T_+	$\uparrow_0$	$\uparrow_+$	$\downarrow_0$	$\downarrow_+$
F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0
F_+	F_0	$\mathsf{F}_?$	F_+	$\mathsf{F}_?$	$\mathsf{F}_?$	$\mathsf{F}_?$	$\mathsf{F}_?$	$\mathsf{F}_?$
T_0	F_0	F_+	T_0	T_+	$\uparrow_0$	$\uparrow_+$	$\downarrow_0$	$\downarrow_+$
T_+	F_0	$\mathsf{F}_?$	T_+	T_+	$\mathsf{F}_?$	$\uparrow_?$	$\downarrow_+$	$\downarrow_?$
$\uparrow_0$	F_0	$\mathsf{F}_?$	$\uparrow_0$	$\uparrow_?$	$\uparrow_0$	$\uparrow_?$	$\mathsf{F}_?$	$\mathsf{F}_?$
$\uparrow_+$	F_0	$\mathsf{F}_?$	$\uparrow_+$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\mathsf{F}_?$	$\mathsf{F}_?$
$\downarrow_0$	F_0	$\mathsf{F}_?$	$\downarrow_0$	$\downarrow_+$	$\mathsf{F}_?$	$\mathsf{F}_?$	$\downarrow_0$	$\downarrow_?$
$\downarrow_+$	F_0	$\mathsf{F}_?$	$\downarrow_+$	$\downarrow_?$	$\mathsf{F}_?$	$\mathsf{F}_?$	$\downarrow_?$	$\downarrow_?$

$\vee_c$	F_0	F_+	T_0	T_+	$\uparrow_0$	$\uparrow_+$	$\downarrow_0$	$\downarrow_+$
F_0	F_0	F_+	T_0	T_+	$\uparrow_0$	$\uparrow_+$	$\downarrow_0$	$\downarrow_+$
F_+	F_+	F_+	T_0	$\mathsf{T}_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$
T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0
T_+	T_+	$\mathsf{T}_?$	T_0	$\mathsf{T}_?$	$\mathsf{T}_?$	$\mathsf{T}_?$	$\mathsf{T}_?$	$\mathsf{T}_?$
$\uparrow_0$	$\uparrow_0$	$\uparrow_?$	T_0	$\mathsf{T}_?$	$\uparrow_0$	$\uparrow_?$	$\mathsf{T}_?$	$\mathsf{T}_?$
$\uparrow_+$	$\uparrow_+$	$\uparrow_?$	T_0	$\mathsf{T}_?$	$\uparrow_?$	$\uparrow_?$	$\mathsf{T}_?$	$\mathsf{T}_?$
$\downarrow_0$	$\downarrow_0$	$\downarrow_?$	T_0	$\mathsf{T}_?$	$\mathsf{T}_?$	$\mathsf{T}_?$	$\downarrow_0$	$\downarrow_?$
$\downarrow_+$	$\downarrow_+$	$\downarrow_?$	T_0	$\mathsf{T}_?$	$\mathsf{T}_?$	$\mathsf{T}_?$	$\downarrow_?$	$\downarrow_?$

$\neg_c$	
F_0	T_0
F_+	T_+
T_0	F_0
T_+	F_+
$\uparrow_0$	$\downarrow_0$
$\uparrow_+$	$\downarrow_+$
$\downarrow_0$	$\uparrow_0$
$\downarrow_+$	$\uparrow_+$

Δ_c	
F_0	F_0
F_+	F_+
T_0	T_0
T_+	T_+
$\uparrow_0$	$\uparrow_0$
$\uparrow_+$	$\uparrow_+$
$\downarrow_0$	$\downarrow_0$
$\downarrow_+$	$\downarrow_+$

$\Box_c$	
F_0	F_0
F_+	$\mathsf{F}_?$
T_0	T_0
T_+	$\mathsf{T}_?$
$\uparrow_0$	$\uparrow_0$
$\uparrow_+$	$\uparrow_?$
$\downarrow_0$	$\downarrow_0$
$\downarrow_+$	$\downarrow_?$

where $\mathsf{F}_? \stackrel{\text{def}}{=} \mathsf{F}_0 \mid \mathsf{F}_+$, $\mathsf{T}_? \stackrel{\text{def}}{=} \mathsf{T}_0 \mid \mathsf{T}_+$, $\downarrow_? \stackrel{\text{def}}{=} \downarrow_0 \mid \downarrow_+$, $\uparrow_? \stackrel{\text{def}}{=} \uparrow_0 \mid \uparrow_+$

Figure 3.8: Operators on $\mathbb{T}_c$

Definition 3.5.2. *The Galois connection $\alpha_c : \wp(\mathbb{T}) \to \wp(\mathbb{T}_c)$, $\gamma_c : \wp(\mathbb{T}_c) \to \wp(\mathbb{T})$ is defined as follows:*

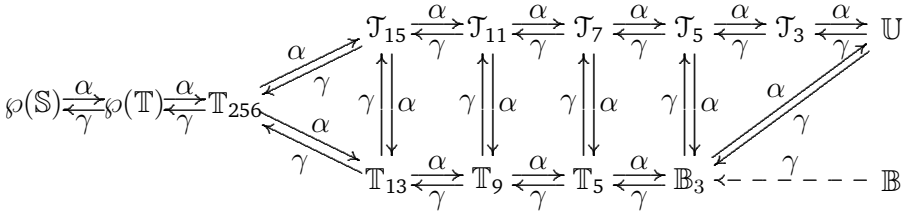
$$\beta_c\, X_n \stackrel{\text{def}}{=} \begin{cases} X_0 & \textit{iff } n = 0\textit{;} \\ X_+ & \textit{otherwise.} \end{cases}$$

$$\alpha_c \hat{t} \stackrel{\text{def}}{=} \{\beta_c t \mid t \in \hat{t}\} \quad \gamma_c \hat{t} \stackrel{\text{def}}{=} \{t \in \mathbb{T} \mid \beta_c t \in \hat{t}\}$$

It is possible to tabulate 256×256 truth tables that fully enumerate all members of $\mathbb{T}_{256}$ along their edges, but they are too large to reproduce here in full. For brevity, Fig. 3.8 defines the operators $\neg_c : \mathbb{T}_c \to \wp(\mathbb{T}_c)$, $\Delta_c : \mathbb{T}_c \to \wp(\mathbb{T}_c)$, $\Box_c : \mathbb{T}_c \to \wp(\mathbb{T}_c)$ and $\wedge_c : \mathbb{T}_c \times \mathbb{T}_c \to \wp(\mathbb{T}_c)$ on $\mathbb{T}_c$. Their fully nondeterministic versions, defined on $\wp(\mathbb{T}_c)$, are as follows:

$$\neg_c \hat{t} \stackrel{\text{def}}{=} \bigcup_{t \in \hat{t}} \{\neg_c t\} \qquad \Delta_c \hat{t} \stackrel{\text{def}}{=} \bigcup_{t \in \hat{t}} \{\Delta_c t\} \qquad \Box_c \hat{t} \stackrel{\text{def}}{=} \bigcup_{t \in \hat{t}} \{\Box_c t\} \qquad \hat{t} \wedge_c \hat{u} \stackrel{\text{def}}{=} \bigcup_{\substack{t \in \hat{t} \\ u \in \hat{u}}} \{t \wedge_c u\}$$

Note that, as with $\Delta^\sharp$, the Δ_c operator is merely an identity function.



where $\mathbb{U} \stackrel{\text{def}}{=} \{\bigstar\}$, $\mathbb{B} \stackrel{\text{def}}{=} \{\mathsf{F}, \mathsf{T}\}$, $\mathbb{B}_3 \stackrel{\text{def}}{=} \mathbb{B} \cup \{\bigstar\} = \{\mathsf{F}, \mathsf{T}, \bigstar\}$.

Note that α and γ are different functions in all cases.

Figure 3.9: Hierarchy of domains

3.6 Further Simplification of the Abstract Domain

A fully tabulated version of the $\neg_c$, Δ_c, $\Box_c$ and $\wedge_c$ operators defined in Section 3.5.1 can be regarded as a 256-value *transitional logic,* where the values are the members of $\wp(\mathbb{T}_c)$. Such an approach still captures more nondeterminism than is useful for many applications. It is possible to further reduce the abstract domain, replacing some nondeterministic choices with appropriate least upper bound elements with respect to $\langle \wp(\mathbb{T}_c), \subseteq \rangle$. The hierarchy of domains that results is shown in Fig. 3.9 – the relationship to 2-value Boolean logic $\mathbb{B}$ and 3-value ternary logic $\mathbb{B}_3$ is shown[9] (see also Appendix A.3). Note that since $\mathbb{B}$ lacks an upper bound that corresponds with $\bigstar$, it is not possible to define $\alpha : \mathbb{B}_3 \to \mathbb{B}$ (though $\gamma : \mathbb{B} \to \mathbb{B}_3$ can be trivially defined), so a Galois connection does not exist in that particular case. Following Cousot & Cousot [42, 43], the domain $\mathbb{U}$, *useless logic,* containing only $\bigstar$, completes the lattice.

Finding the smallest lattice including $\{\mathsf{F}_0, \mathsf{F}_+, \mathsf{T}_0, \mathsf{T}_+, \uparrow_0, \uparrow_+, \downarrow_0, \downarrow_+\}$ that is closed under $\wedge_c$, and $\neg_c$ results in the 13-value transitional logic,

$$\mathbb{T}_{13} \stackrel{\text{def}}{=} \{\mathsf{F}_0, \mathsf{F}_+, \mathsf{F}_?, \mathsf{T}_0, \mathsf{T}_+, \mathsf{T}_?, \uparrow_0, \uparrow_+, \uparrow_?, \downarrow_0, \downarrow_+, \downarrow_?, \bigstar\}$$

Though much smaller than $\wp(\mathbb{T}_c)$, this logic is equivalently useful for most purposes – note that a special element needs to be explicitly included, $\bigstar$, representing the least upper bound (top element) of the lattice.

In cases where it is important to know that a trace is definitely clean, but where it is not necessary to distinguish between 'definitely dirty' and 'possibly dirty', further reducing the domain by folding F_+, T_+, $\uparrow_+$ and $\downarrow_+$ into their respective least upper bounds $\mathsf{F}_?$, $\mathsf{T}_?$, $\uparrow_?$ and $\downarrow_?$ results in a 9-value transitional logic, $\mathbb{T}_9 \stackrel{\text{def}}{=} \{\mathsf{F}_0, \mathsf{F}_?, \mathsf{T}_0, \mathsf{T}_?, \uparrow_0, \uparrow_?, \downarrow_0, \downarrow_?, \bigstar\}$. An even simpler 5-value transitional logic $\mathbb{T}_5 \stackrel{\text{def}}{=} \{\mathsf{F}, \mathsf{T}, \uparrow, \downarrow, \bigstar\}$ results from folding all remaining nondeterminism into $\bigstar$. $\mathbb{T}_{13}$ and $\mathbb{T}_9$ are well suited to logic simulation, refinement and model checking, whereas $\mathbb{T}_5$ is only recommended for glitch checking. The truth tables for $\mathbb{T}_{13}$, $\mathbb{T}_9$ and $\mathbb{T}_5$ are shown in Appendix A.1.

[9] As with our other logics, we assume that $\mathsf{F} \subseteq \bigstar$ and $\mathsf{T} \subseteq \bigstar$ – some ternary logics in the literature (notably Kleene's) lack this formal requirement.

3.6.1 Static-Clean Logics

The $\mathbb{T}_{13}$, $\mathbb{T}_9$ and $\mathbb{T}_5$ logics can be usefully extended by introducing two extra upper bounds: S, the least upper bound of traces whose values are fixed for all time, and C, the least upper bound of traces that may transition, but that *never* glitch. We adopt the notation $\mathcal{T}_n$ to represent a static-clean logic with n values.

Definition 3.6.1. *With respect to $\wp(\mathbb{T}_c)$, the* least upper bounds S, C *and* $\bigstar$ *are as follows:*

$$\mathsf{S} \stackrel{\text{def}}{=} \{\mathsf{F}_0, \mathsf{T}_0\} \qquad \mathsf{C} \stackrel{\text{def}}{=} \{\mathsf{F}_0, \mathsf{T}_0, \uparrow_0, \downarrow_0\}$$
$$\bigstar \stackrel{\text{def}}{=} \{\mathsf{F}_0, \mathsf{F}_+, \mathsf{T}_0, \mathsf{T}_+, \uparrow_0, \uparrow_+, \downarrow_0, \downarrow_+\}$$

The resulting *static-clean* transitional logics $\mathcal{T}_{15} \stackrel{\text{def}}{=} \mathbb{T}_{13} \cup \{\mathsf{S}, \mathsf{C}\}$, $\mathcal{T}_{11} \stackrel{\text{def}}{=} \mathbb{T}_9 \cup \{\mathsf{S}, \mathsf{C}\}$ and $\mathcal{T}_7 \stackrel{\text{def}}{=} \mathbb{T}_5 \cup \{\mathsf{S}, \mathsf{C}\}$ have applications in the design rule checking of 'impure' synchronous circuits. For example, in order to ensure that the clock input of a D-type flip flop can never glitch, a signal generated by the circuit $\mathsf{S} \wedge \mathsf{C} = \mathsf{C}$ might be accepted by a model checker, but $\mathsf{C} \wedge \mathsf{C} = \bigstar$ would not.

Removing $\uparrow$ and $\downarrow$ from $\mathcal{T}_7$ results in a 5-value static-clean logic $\mathcal{T}_5 \stackrel{\text{def}}{=} \{\mathsf{F}, \mathsf{T}, \mathsf{S}, \mathsf{C}, \bigstar\}$ capable of reasoning about gated clock synchronous circuits; an even simpler (though less accurate) 3-value static-clean logic $\mathcal{T}_3 \stackrel{\text{def}}{=} \{\mathsf{S}, \mathsf{C}, \bigstar\}$ results from also removing F and T.

The truth tables for these logics are shown in Appendix A.2.

3.7 Refinement and Equivalence in Transitional Logics

Hardware engineers are frequently concerned with modification and optimisation of existing circuits, so it is appropriate to support this by defining equivalence and refinement with respect to our abstract domains. Refinement relationships between circuits are analogous to concepts of refinement in process calculi, and may similarly be used to aid provably correct design. For example, the Boolean equivalence $a \wedge \neg a = \mathsf{F}$ is not a strong equivalence in many of our models, nor is it a weak equivalence – it actually turns out to be a (left-to-right) refinement, *i.e.,* $a \wedge \neg a \succcurlyeq \mathsf{F}_0$, reflecting the 'engineer's intuition' that it is safe to replace $a \wedge \neg a$ with F_0, but that the converse could damage the functionality of the circuit by introducing new glitch states that were not present in the original design. Such refinement rules are also known as *hazard non-increasing transformations* in the asynchronous design literature [82].

Informally, if the deterministic trace $u \in \mathbb{T}$ *refines* (*i.e.,* retains the steady state behaviour of, but is no more glitchy than) trace $t \in \mathbb{T}$, this may be denoted $t \succcurlyeq u$.

Definition 3.7.1. *Given a pair of traces $t \in \mathbb{T}$ and $u \in \mathbb{T}$,*

$$t \succcurlyeq u \stackrel{\text{def}}{=} \mathit{Val}(t) = \mathit{Val}(u) \wedge \mathit{Subs}(t) \geq \mathit{Subs}(u)$$

For example, $\mathsf{F}_1 \succcurlyeq \mathsf{F}_0$, $\mathsf{T}_3 \succcurlyeq \mathsf{T}_2$, $\uparrow_5 \succcurlyeq \uparrow_5$, but $\downarrow_0$ and $\uparrow_1$ are incomparable. Where $t \in \mathbb{T}$ and $u \in \mathbb{T}$, if $t \succcurlyeq u$ and $u \succcurlyeq t$, it follows that $t = u$.

Refinement and equivalence for nondeterministic traces is slightly less straightforward, in that it is necessary to handle cases like $\downarrow_{1|3|5} \succcurlyeq \downarrow_{0|2|4}$. To make these comparable, we construct *convex hulls* of the form $X_{0..n}$ enclosing the nondeterministic choices, so the above case becomes equivalent to $\downarrow_{0..5} \succcurlyeq \downarrow_{0..4}$. In effect, this approach compares worst-case behaviour, disregarding finer detail; in practice, since $\wedge$, Δ, $\Box$ and $\neg$ typically return results of the general form $X_{0..n}$ anyway, this tends not to cause any practical difficulties. Less permissive definitions of refinement, *e.g.,* $\hat{t} \succcurlyeq_{strict} \hat{u} \equiv \forall t \in \hat{t} \;.\; \forall u \in \hat{u} \;.\; t \succcurlyeq u$, often disallow too many possible optimisations that in practice are quite acceptable – our model better reflects the engineer's intuition that 'less glitchy is better,' but that very detailed information about the structure of possible glitches is generally not important.

Definition 3.7.2. *Given $\hat{t} \in \wp(\mathbb{T})$ and $\hat{u} \in \wp(\mathbb{T})$,*

$$\hat{t} \succcurlyeq \hat{u} \stackrel{\text{def}}{\equiv} (\forall t \in \hat{t}, u \in \hat{u} \;.\; \mathit{Val}(t) = \mathit{Val}(u)) \wedge \mathit{MaxSubs}(\hat{t}\,) \geq \mathit{MaxSubs}(\hat{u})$$

where $\mathit{MaxSubs}(\hat{t}) \stackrel{\text{def}}{=} \max_{t \in \hat{t}} \mathit{Subs}(t)$ is a function returning the largest subscript of a nondeterministic trace.

3.7.1 Equivalence of Nondeterministic Traces.

Given $\hat{t} \in \wp(\mathbb{T})$ and $\hat{u} \in \wp(\mathbb{T})$, if $\hat{t} = \hat{u}$ then the traces are *strongly equivalent, i.e.,* they represent exactly the same sets of nondeterministic choices. If the convex hulls surrounding $\hat{t}$ and $\hat{u}$ are identical, as is the case when $\hat{t} \succcurlyeq \hat{u} \wedge \hat{u} \succcurlyeq \hat{t}$, the traces may be said to be *weakly equivalent,* denoted $\hat{t} \eqsim \hat{u}$. Where $\hat{t} \succcurlyeq \hat{u} \vee \hat{u} \succcurlyeq \hat{t}$, the traces are *comparable,* denoted $\hat{t} \precsim \hat{u}$.

3.7.2 Finite Abstract Domains

Refinement and equivalence can also be defined for the finite abstract domain $\mathbb{T}_{256}$ and some of its simplified forms. Since $\mathbb{T}_{256}$ is implicitly nondeterministic, we do not need to consider the deterministic case.

Definition 3.7.3. *Given traces $t \in \mathbb{T}_{256}$ and $u \in \mathbb{T}_{256}$, the above definitions simplify to*

$$t \succcurlyeq u \equiv \mathit{Val}(t) = \mathit{Val}(u) \wedge (\mathit{Subs}(t) = \mathit{Subs}(u) \vee \mathit{Subs}(u) = 0)$$
$$t \eqsim u \equiv t \succcurlyeq u \wedge u \succcurlyeq t \equiv t = u$$
$$t \precsim u \equiv t \succcurlyeq u \vee u \succcurlyeq t \equiv \mathit{Val}(t) = \mathit{Val}(u)$$

3.8 Algebraic Properties of $\wp(\mathbb{T})$

In Section 3.7, refinement and equivalence were defined for the hierarchy of transitional logics. In this section, we extend this by reexamining the familiar logical identities from classical Boolean logic in order to verify which also apply to $\wp(\mathbb{T})$, and to what extent.

Fig. 3.10 summarises the identites that have been identified for $\wp(\mathbb{T})$.The definitions for $\wedge$, $\neg$ and $\vee$ are taken from Fig 3.5 unless otherwise specified (we omit the $\sharp$ decoration here for readability).

	Boolean Identity	$\wp(\mathbb{T})$ **Identity**
Idempotence	$a \wedge a = a$	$a \wedge a \succcurlyeq a$
	$a \vee a = a$	$a \vee a \preccurlyeq a$
Consistency	$a \wedge \neg a = \mathsf{F}$	$a \wedge \neg a \succcurlyeq \mathsf{F}_0$
Law of the Excluded Middle	$a \vee \neg a = \mathsf{T}$	$a \vee \neg a \succcurlyeq \mathsf{T}_0$
Associativity	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$
	$a \vee (b \vee c) = (a \vee b) \vee c$	$a \vee (b \vee c) = (a \vee b) \vee c$
Commutativity	$a \wedge b = b \wedge a$	$a \wedge b = b \wedge a$
	$a \vee b = b \vee a$	$a \vee b = b \vee a$
de Morgan's Laws	$\neg(a \wedge b) = \neg a \vee \neg b$	$\neg(a \wedge b) = \neg a \vee \neg b$
	$\neg(a \vee b) = \neg a \wedge \neg b$	$\neg(a \vee b) = \neg a \wedge \neg b$
Distributivity	$(a \wedge b) \vee (a \wedge c) = a \wedge (b \vee c)$	$(a \wedge b) \vee (a \wedge c) \succcurlyeq a \wedge (b \vee c)$
	$(a \vee b) \wedge (a \vee c) = a \vee (b \wedge c)$	$(a \vee b) \wedge (a \vee c) \succcurlyeq a \vee (b \wedge c)$
Double Negation	$\neg\neg a = a$	$\neg\neg a = a$
Contrapositive Law	$\neg a \vee b = \neg\neg b \vee \neg a$	$\neg a \vee b = \neg\neg b \vee \neg a$
Properties of T	$a \vee \mathsf{T} = \mathsf{T}$	$a \vee \mathsf{T}_0 = \mathsf{T}_0$
	$a \wedge \mathsf{T} = a$	$a \wedge \mathsf{T} = a$
Properties of F	$a \wedge \mathsf{F} = \mathsf{F}$	$a \wedge \mathsf{F}_0 = \mathsf{F}_0$
	$a \vee \mathsf{F} = a$	$a \vee \mathsf{F}_0 = a$
Absorption Laws	$a \wedge (a \vee b) = a$	$a \wedge (a \vee b) \succcurlyeq a$
	$a \vee (a \wedge b) = a$	$a \vee (a \wedge b) \succcurlyeq a$

Figure 3.10: Identities of Boolean logic and the transitional logic $\wp(\mathbb{T})$

3.9 Related Work

There seems to be relatively little work reported in the literature regarding the application of modern program analysis techniques to hardware. Indeed hardware analyses seem to have been developed the same one-at-a-time manner as software analyses before the advent of unifying frameworks such as Monotone Data Flow Frameworks [76], Kildall's 'Unified Approach to Global Program Optimization' [78] and Abstract Interpretation [42] which significantly enhanced the development of software analyses.

3.9.1 Achronous Analyses

The works considered in this section all make the achrony assumption (which we make in considering signals on wires to be values in $\wp(\mathbb{T})$) and hence re-appear as instances of our framework.

Janusz Brzozowski's *algebra of transients* [28, 29] has many similarities to our transitional logic $\wp(\mathbb{T})$. Values in the algebra of transients are analogous to convex hulls of the form $X_{0..n}$ in our notation. Similar correctness results to our own are reported, achieved

instead through different mathematical techniques. Interestingly, some reduced forms of the algebra of transients turn out to be identical to some of our reduced forms – however, our logics including 'definitely dirty' (X_+) values and/or S and C values do not appear to have equivalent representations, perhaps due to our finer-grained model of nondeterminism.

David S. Kung defines a hazard-non-increasing gate-level optimisation algorithm [82] based partly upon a multi-value logic that closely resembles our transitional logic $\mathbb{T}_9$, though his theoretical justification appears to be somewhat inconsistent. His 9 values, $1, 0, \uparrow, \downarrow, \mathit{S0}, \mathit{S1}, D+, D-$ and $*$ are defined equivalently to our $\mathsf{T}, \mathsf{F}, \uparrow_0, \downarrow_0, \mathsf{F}_+, \mathsf{T}_+, \uparrow_+, \downarrow_+$ and $\bigstar$, though more accurately they should be seen as equivalent to $\mathsf{T}, \mathsf{F}, \uparrow_0, \downarrow_0, \mathsf{F}_?, \mathsf{T}_?, \uparrow_?, \downarrow_?$ and $\bigstar$. The 9 values are claimed to partition possible waveforms into disjoint equivalence classes[10], and a separate $<$ operator is given that defines a Hasse ordering over the values. Though Kung's justification appears to have some problems, his results as regards *hazard-non-increasing extensions* are likely to be correct as a consequence of the similarity of his logic to $\mathbb{T}_9$.

Don Gaubatz [55] proposes a 4-value 'quaternary' logic (see Appendix A.3) that, extended slightly to allow operators to be represented as total functions, is equivalent to our $\mathbb{T}_5$ (see Appendix A.1).

3.9.2 Non-Achronous Analyses

Paul Cunningham [44] extends Gaubatz's work in many respects, though his formalism is based on a conventional 2-value logic with transitions handled explicitly as events rather than as values in an extended logic.

Jerry R. Burch's *binary chaos delay model* [30] underlies a method for verifying speed-dependent asynchronous circuits. Though aimed at a different design paradigm (we primarily consider speed- and delay-independent circuits), his technique's adoption of an underlying dense time model presents an interesting contrast to our approach, particularly in that it allows absolute timing information to be exploited. Burch's model is more abstract than our concrete domain $\wp(\mathbb{S})$ and more concrete (as a consequence of taking into account absolute time) than our most accurate abstract domain $\wp(\mathbb{T})$, though the approaches are sufficiently different that neither subsumes the other.

3.9.3 Synchronous Analyses

Most existing work in hardware analysis is aimed at synchronous circuits and as-such is beyond the scope of this chapter, since we mainly consider asynchronous circuits. It is, however, noteworthy that Charles Hymans [65] uses abstract interpretation to present a safety property checking technique based upon abstract interpretation of (synchronous) behavioural VHDL specifications.

[10] Kung's definition is inconsistent – since the values $1, 0, \uparrow, \downarrow, \mathit{S0}, \mathit{S1}, D+, D-$ cover all possible waveforms, $*$ must be an empty set in order for the logic's 9 values to be disjoint, though informally it is stated to mean 'any value at all'. This problem can be avoided (as in our definition) by abandoning a requirement for disjointness and defining the Hasse ordering in terms of subset inclusion, *i.e.,* $0 \subseteq \mathit{S0} \subseteq *, 1 \subseteq \mathit{S1} \subseteq *, \uparrow \subseteq (D+) \subseteq *$ and $\downarrow \subseteq (D-) \subseteq *$.

Thomas Jensen [71] applied abstract interpretation to the analysis of multiple independent synchronous clock domains within the Lustre language [31]. Lustre began as an academic project in the early 1980s, then in 1993 was commercially adopted by Esterel Technologies, under the name Esterel [51, 18, 19, 20] as the core of the SCADE (Safety Critical Application Development Environment) system. Current implementations of Esterel support multiple asynchronous clock domains, primarily in support of GALS system design, though the language's primary focus remains on synchronous design. Abstract interpretation has been used extensively to analyse systems implemented in Esterel – in particular, the flight control system of the A380 Airbus was implemented entirely in Esterel and verified using the abstract-interpretation-based ASTRÉE static analyser [23].

3.10 Discussion

In this chapter, we have presented a technique based upon the solid foundation of abstract interpretation [42, 43] that allows properties of a wide class of digital circuits to be reasoned about. We describe what is essentially a first attempt at applying abstract interpretation to asynchronous hardware.

Transitional logics are essentially the result of exploring the limits of achronous analysis. The $\wp(\mathbb{T})$ abstraction is effectively the most precise achronous analysis possible, which therefore causes all other achronous analyses to be subsumed. It is also the most accurate analysis that can be achieved assuming an independent attribute model in relation to time. Therefore, all analyses offering greater accuracy must of-necessity adopt a relational attribute model. Taken as-is, transitional logics are probably insufficiently accurate to be directly usable for analysis purposes in an EDA design flow if they are applied to time windows that encompass many transitions – too many false positives are likely to be generated. However, when signals are windowed such that most signals are effectively static or making clean transitions, this greatly increases accuracy, reducing potential false positives to a minimum. For example, an advanced logic simulator might model signals as streams of values in $\wp(\mathbb{T})$, which would guarantee that arbitrarily narrow glitches could not be missed; even though the simulation has finite timing resolution, the dense continuous time model underlying transitional logics guarantees that any possible glitches between simulator events will be identified. This idea was examined further in the author's 2004 workshop paper [131].

It is possibly the case that transitional logics may be more useful as a mathematical framework than as a practical analysis. In Chapter 7, transitional logic is used to help prove that it is impossible for any possible delay-insensitive circuit constructed from gates to guarantee immunity to single-event transients. Though this result could no-doubt have been achieved through alternative mathematical means, the correctness and completeness relationship between $\wp(\mathbb{S})$ and $\wp(\mathbb{T})$, along with some of the algebraic properties of $\wp(\mathbb{T})$, greatly simplifies the proof.

Identities and refinements in the transitional logic (*e.g.,* those described in Section 3.8) are extremely strong, because by definition they hold for all possible combinations of delays and all possible signals. For example, the refinement $a \wedge a \succcurlyeq a$ states that the circuit $a \wedge a$ may be replaced with a without introducing extra hazards, but replacing a with $a \wedge a$ is potentially an unsafe transformation. However, $a \vee (b \vee c)$ is provably

safely interchangeable with $(a \vee b) \vee c$. A gate-level optimiser whose transformations are proven correct on this basis can be safely applied to a delay insensitive asynchronous circuit without risk of inadvertent introduction of unwanted hazards. This approach was introduced by Kung [82], though his formalism is subsumed by ours.

Chapter 4

Bit-Level Partial Evaluation of Synchronous Circuits

The work described in this chapter significantly reworks and extends some ideas from the author's M.Sc thesis [129], the majority of which was undertaken in Cambridge during late 2003 and early 2004. Some extensions were made whilst at the NASA Ames Research Center during Summer 2004 in collaboration with Guillaume Brat and Arnaud Venet.

The work was published in [134].

4.1 Introduction

Partial evaluation [72, 88, 87] is a long-established technique that, when applied to software, is known to be very powerful; apart from its usefulness in automatically creating (usually faster) specialised versions of generic programs, its ability to transform interpreters into compilers is particularly noteworthy.

In this chapter, we present a partial evaluation framework for synchronous digital circuits that, whilst supporting specialisation, also supports the first Futamura projection [54]:

$$PE[\![interpreter, program]\!] = compiler[\![program]\!].$$

In hardware terms, this is equivalent to taking the circuit for a processor and a program ROM image, then compiling this into hardware that represents the program only. As with software partial evaluation, the processor itself is optimised away, leaving only the functionality of the program expressed directly in hardware.

Note that in this chapter, we consider only the partial evaluation of purely synchronous circuits, *i.e.,* circuits consisting only of acyclic networks of gates, with feedback occurring only via D-type latches whose clock inputs are all driven by a single global clock net. Generalisation to the asynchronous case is discussed briefly in Section 9.2.

$$
\begin{array}{rr}
a \wedge a \rightarrow a & a \vee a \rightarrow a \\
a \wedge \mathit{false} \rightarrow \mathit{false} & a \wedge \mathit{true} \rightarrow a \\
a \vee \mathit{false} \rightarrow a & a \vee \mathit{true} \rightarrow \mathit{true} \\
\neg \mathit{false} \rightarrow \mathit{true} & \neg \mathit{true} \rightarrow \mathit{false}
\end{array}
$$

Table 4.1: Rewrite rules for combinational PE

4.2 PE of Combinational Circuits

The simplest form of hardware partial evaluation is already well known to hardware engineers, though under a different name: Boolean optimisation. For example, the combinational circuit represented by the expression

$$a \wedge (b \vee c)$$

where a, b and c are inputs, may be specialised for the case where c is known to be true as follows:

$$a \wedge (b \vee \mathit{true}) = a \wedge \mathit{true} = a$$

Boolean optimisation is well-studied, with many approaches documented in the literature. Some techniques are sub-optimal but can be applied to any circuit, whereas others (*e.g.,* OBDDs, flattening to CNF or DNF) can yield optimal results[1] but suffer exponential size blowup when confronted with some circuits, particularly multipliers. Any of these techniques are potentially capable of PE of combinational circuits, though for clarity and generality, we have opted for a simple approach based upon term rewriting. Table 4.1 shows a simple set of rewrite rules that are sufficient to implement (sub-optimal) combinational PE in time and space that is linear with respect to the original circuit.

The software analogue of a combinational circuit, from the point of view of PE, would be a program consisting only of assignment statements, *if-then* and *if-then-else* constructs, but strictly no loops.

4.3 PE of Synchronous Circuits

A slight variation[2] on the general form of any synchronous circuit, generally referred to as a Mealy machine [94], is shown in Fig. 4.1. In software terms, such circuits resemble a program of the form

[1] In hardware terms, 'optimal' is not easily defined. In some cases, circuits are optimised for minimum gate count, though more commonly they are optimised for speed or power consumption – only rarely will a circuit be optimal with respect to more than one of these considerations.

[2] Conventionally, f would be drawn on the left of the state flip-flops. The (equivalent) form used within this chapter allows unrolling to be described more conveniently, however.

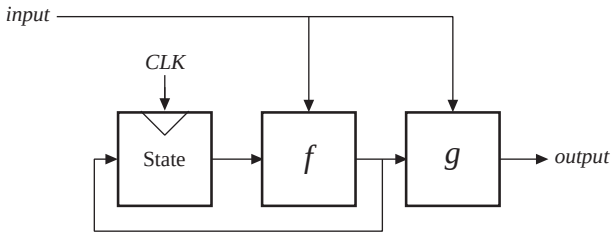


Figure 4.1: General form of synchronous circuits

```
while true
    state_{k+1} := f(state_k, input)
    output := g(state_{k+1}, input)
endwhile
```

Each iteration of the loop represents exactly one clock cycle. The internal state of the circuit, represented by $state_k$, may change only once per clock cycle and is determined only by the result of the combinational Boolean function $f : (\mathbb{B} \times \cdots \times \mathbb{B}) \times (\mathbb{B} \times \cdots \times \mathbb{B}) \rightarrow (\mathbb{B} \times \cdots \times \mathbb{B})$; the subscript k has no effect on execution, and is purely a naming convention that is convenient when describing unrolling. The $input$ is assumed to change synchronously just after the clock, and the $output$ is determined by the combinational Boolean function $g : (\mathbb{B} \times \cdots \times \mathbb{B}) \times (\mathbb{B} \times \cdots \times \mathbb{B}) \rightarrow (\mathbb{B} \times \cdots \times \mathbb{B})$. Note that the internal state of the circuit, represented by $state$, is observable *only* through g.

Partial evaluation of this kind of circuit typically requires specialisation of f and g, but may also involve either partly or completely unrolling the `while` loop. State minimisation is not performed[3]. A single unrolling yields the program

```
while true
    state_{k+1} := f(state_k, input_1)
    output_1 := g(state_{k+1}, input_1)
    state_{k+2} := f(state_{k+1}, input_2)
    output_2 := g(state_{k+2}, input_2)
endwhile
```

which can be equivalently expressed as

```
while true
    state_{k+2} := f(f(state_k, input_1), input_2)
    output_1 := g(f(state_k, input_1), input_1)
    output_2 := g(f(f(state_k, input_1), input_2), input_2)
endwhile
```

corresponding to the circuit shown in Fig. 4.2. After unrolling, since $state_{k+1}$ is not externally observable, it need not be explicitly computed. In one clock cycle, this new

[3]In synchronous circuits, state minimisation is often undesirable – though it reduces flip flop count, the extra state decoding logic required often adversely affects maximum clock rates. As an extreme example, the performance advantages of a carefully designed one-hot encoded state machine might be lost entirely if state minimisation was to be naïvely attempted.

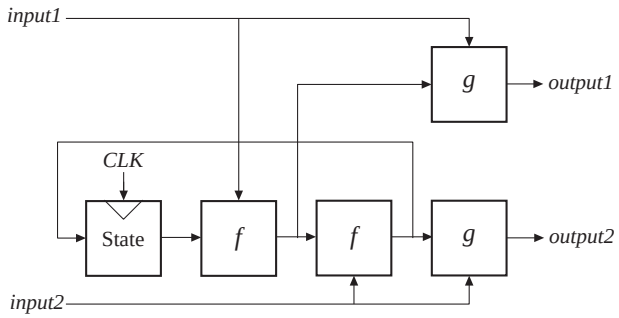


Figure 4.2: Synchronous circuit after one unrolling

circuit performs the same computations that the original circuit performed in two cycles, though possibly with a slower maximum clock rate due to longer worst-case paths.

In the general case, since the $input_i$ and $output_j$ may change at every cycle, they may need to be separately accessible in the unrolled circuit. Often, though, $input$ may be known to remain unchanged for many iterations, or for all time. It is common, also, for outputs other than those resulting from the final state to be unimportant; in combination, this allows loop unrolling to generate much more efficient hardware.

In the rest of this chapter, we make the assumption that $input$ may change only synchronously with the clock of the unrolled hardware, and that $output$ reflects the final state of the unrolled loop body at the end of each clock cycle.

4.3.1 Multiple Unrollings

Loops may be unrolled an arbitrary number of times by the following method:

```
while true
    state_{k+1} := f(state_k, input)
    state_{k+2} := f(state_{k+1}, input)
    ...
    state_{k+n} := f(state_{k+n-1}, input)
    output := g(state_{k+n}, input)
endwhile
```

or, equivalently:

```
while true
    state_{k+n} := f(f(...f(state_k, input), ...input)
    output := g(state_{k+n}, input)
endwhile
```

Since the repetition of $input$ gives potential for common subexpression elimination, and g needs to be evaluated exactly once regardless of the number of unrollings (see Fig. 4.3), the gate count of any resulting circuit is typically (and often substantially) less than $n \times |f| + |g|$, where $|f|$ is the gate count of the original f, $|g|$ is the gate count of the original g, and n is the number of unrollings. In some cases, the final gate count may even be less than $|f| + |g|$ (see Section 4.4).

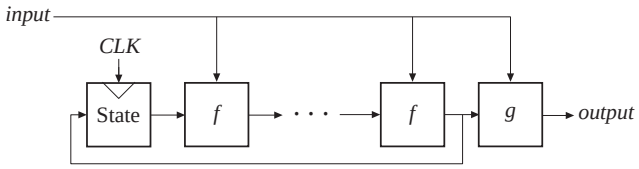


Figure 4.3: Synchronous circuit after n unrollings

4.3.2 Reset Logic

Most synchronous circuits require some form of reset capability, corresponding to the following program:

$state_0 := initialstate$
`while` $true$
    $state_{k+1} := f(state_k, input)$
    $output := g(state_{k+1}, input)$
`endwhile`

In pure-synchronous hardware terms, since $state$ may only change at a clock edge, it is not possible to have code execute outside the loop, so practical implementations usually resemble the following:

`while` $true$
    `if` $reset = true$
        $state_k := initialstate$
    `endif`
    $state_{k+1} := f(state_k, input)$
    $output := g(state_{k+1}, input)$
`endwhile`

Here, $reset$ is a special synchronous input that causes $state_k$ to be reset to $initialstate$ if it is held $true$ for one or more cycles. Note that, in this form, the circuit is just a special case of the general definition given in the introduction to Section 4.3.

4.3.3 Full Unrolling

Where repeated application of f reaches a fixed point, *i.e.,* where

$$state_0 = initialstate$$
$$state_{k+1} = f(state_k, input)$$

and there exists an n such that for all values of $input$, $state_{n+1} = state_n$, it is possible to fully unroll (and therefore eliminate) the `while` loop:

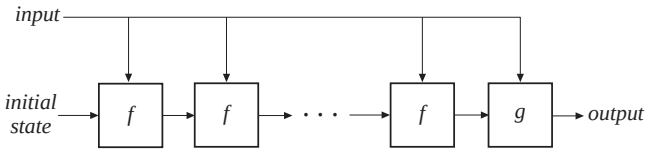


Figure 4.4: Synchronous circuit after full unrolling

$state_0 := initialstate$
$state_1 := f(state_0, input)$
$state_2 := f(state_1, input)$
...
$state_n := f(state_{n-1}, input)$
$output := g(state_n, input)$

Any resulting circuit will be purely combinational (see Fig. 4.4); all D-type flip flops will have been eliminated.

4.4 The HarPE Language

HarPE (pronounced 'harpie') is a simple hardware description language created specifically to aid experimentation in partial evaluation [127]. As is becoming increasingly common [22], HarPE is an *embedded* language, existing within a larger, more sophisticated general purpose programming language, in this case C++.

The HarPE language currently exists as an ISO C++ template library, taking advantage of template metaprogramming techniques [140, 141]. Compiling and then executing a C++ source file incorporating HarPE code causes a hardware netlist to be generated. The current compiler generates gate-level Verilog for further processing by a conventional tool chain.

4.4.1 Semantics

HarPE source code has a standard, imperative semantics, with the characteristic that a whole program defines *exactly one* machine cycle. An implicit outer *while* loop, executing once per clock cycle, is assumed, where one execution of the loop body corresponds to exactly one clock cycle. Partial evaluation is carried out aggressively as compilation proceeds. As a simple example, the following program

```
Bit reset("reset");
IntReg<8> a;
a = a + 1;
If(reset);
    a = 0;
EndIf();
Output("a", a);
```

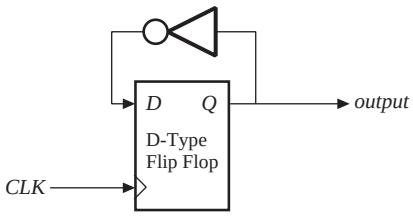


Figure 4.5: A 1-bit 'counter'

implements an 8-bit up counter with a synchronous reset input. The generated circuit outputs one count per clock cycle; though the source code has an imperative semantics, sequential composition does *not* mean that one or more clock cycles must take place – rather, in all cases, sequential composition requires *exactly zero* clock cycles.

4.4.2 Types

4.4.2.1 `Bit`

The fundamental type within HarPE is `Bit`, representing a single bit. Declarations follow C++ syntax:

`Bit` a, b;

C++ operator overloading [122] is used to implement the logical operators representing and, or and not:

`Bit` a, b, c, d, e;
$c = a$ & !b;
$d = b \mid c$;
$e = a \mid d$;

By default, `Bit` variables are initialised to $false$, so

`Bit` a;
`Output`("a", a);

will result in an output, labelled "a", that is connected directly to $false$ (ground).

4.4.2.2 `BitReg`

D-type flip flops are represented by the `BitReg` type. `BitReg` behaves almost identically to `Bit`, with the exception that variables are initialised to reference a D-type flip flop. Any modifications to the value of a `BitReg` variable are incorporated into the feedback loop of the flip flop.

The circuit shown in Fig. 4.5 results from the following code:

`BitReg` a;
$a =$!a;

4.4.2.3 `Int`$\langle n \rangle$

A variable of type `Int`$\langle n \rangle$ represents a n-bit wide unsigned integer, implemented as an array of `Bit`s, with operator overloads supporting the usual arithmetic operators. Since `Bit`'s functionality is inherited, `Int`s are initialised to 0.

A number of alternative constructors are supported, including numeric constants, though they must be explicitly introduced (see the example in Section 4.4.1). For example, the following code generates an 8-bit multiplier:

`Int`$\langle 8 \rangle$ a, b, c;
$c = a * b$;

Individual bits within an `Int` may be addressed through the standard C++ array notation:

`Int`$\langle 8 \rangle$ a;
`Bit` b;
$b = a[3]$;
$a[4] = a[1]$;

If the subscript is a compile-time constant, HarPE simply provides access to the relevant underlying `Bit`. Where the subscript is itself an `Int`-valued expression, HarPE generates an appropriate multiplexer circuit.

4.4.2.4 `IntReg`$\langle n \rangle$

`IntReg` is to `Int` what `BitReg` is to `Bit`; it allows multi-bit registers (normally representing unsigned integers) to be defined straightforwardly. As with `Int`, overloaded numeric operators support the usual arithmetic functions.

4.4.3 External Inputs

Inputs are introduced by passing a string parameter representing the input name to the constructor of `Bit` or `Int`:

`Bit` x("x"), y("y"), z;
$z = x \mid y$;

In this example, z represents the output of an or gate whose inputs are the external inputs "x" and "y". [4]

Similar functionality is provided by `Int`:

`Int`$\langle 8 \rangle$ a("a"), b("b"), c;
$c = a + b$;

In this case, a pair of 8-bit input ports (named $a[0..7]$ and $b[0..7]$ in the netlist) are declared, with c representing the output of an 8-bit adder whose inputs are a and b.

[4]Note the distinction between variable names and input labels – though it is common for these to be named identically, they exist in separate name spaces so there is no formal requirement for this. The declaration `Bit`*foo*("bar") introduces an input labelled "bar" in the netlist which is named "foo" in the source code.

4.4.4 Outputs

All outputs must be declared through the overloaded function `Output`(*"name"*, $expression$), which can accept variables or expressions of type `Bit`, `BitReg`, `Int` or `IntReg`:

`Bit` x("x"), y("y"), z;
`Int`$\langle 8 \rangle$ a("a"), b("b"), c;
$z = x \mid y$;
$c = a + b$;
`Output`("z", z);
`Output`("c", c);
`Output`("q", x & y);

4.4.5 Compilation of Control Flow Constructs

The HarPE compiler flattens all control flow, so programs that do not require D-type flip flops (*i.e.,* those programs that do not use variables of type `BitReg` or `IntReg`) always generate purely combinational hardware – such programs, in effect, execute in exactly zero clock cycles. When D-type flip flops are used, HarPE programs define what happens during exactly one clock cycle of the generated hardware. In this section, we describe how this is achieved.

4.4.5.1 Guarded Assignment

During compilation, HarPE maintains at all times a *guard expression*, Γ, that represents whether or not assignment statements should take place. At the start of compilation, $\Gamma = true$, so all assignments take place. Control flow statements 'and' extra terms into Γ. The HarPE compiler maintains a guard stack, allowing block structured code with arbitrary nesting depth to be handled.

All assignment statements in HarPE, e.g.

$var = newvalue$;

are transformed internally to the following form:

$$var' = \begin{cases} var & \text{iff } \Gamma = false, \\ newvalue & \text{iff } \Gamma = true \end{cases}$$

All subsequent references to var in the program are renamed to var'. At bit level, this is equivalent to a simple multiplexer:

$$var' = (\Gamma \wedge newvalue) \vee (\neg\Gamma \wedge var).$$

Where $\Gamma = true$, this simplifies to $var' = newvalue$. If $\Gamma = false$, the assignment simplifies to $var' = var$, *i.e.,* the assignment has no effect. As a consequence, multiplexers are only generated when they are actually necessary.

Guarded assignment has close parallels with existing work on *static single assignment* (SSA) form [45], though since control flow is fully incorporated into assignments, there is no equivalent of SSA's Φ-functions (control flow merge points).

4.4.5.2 `If..Else..EndIf`

The `If..Else..EndIf` control structure introduces the result of a conditional expression to the guard of all statements within its scope, *e.g.,*

$[\![\Gamma]\!]$
`If`$(cond_1)$;
    $[\![\Gamma \wedge cond_1]\!]$
    `If`$(cond_2)$;
        $[\![\Gamma \wedge cond_1 \wedge cond_2]\!]$
    `Else`();
        $[\![\Gamma \wedge cond_1 \wedge \neg cond_2]\!]$
    `EndIf`();
    $[\![\Gamma \wedge cond_1]\!]$
`EndIf`();
$[\![\Gamma]\!]$

Note that, following the usual convention, the `Else` clause may be omitted.

In the following example, an `If` construct implements a reset circuit for a 3 element 'one hot' encoded shift register:

`BitReg` $a1, a2, a3$;
`Bit` rst("rst"), x;
`If`(rst);
    $a1 = 1$;
    $a2 = a3 = 0$;
`EndIf`();
$x = a3; a3 = a2; a2 = a1; a1 = x$;

HarPE flattens this into the equivalent of the following:

`BitReg` $a1, a2, a3$;
`Bit` rst("rst"), x;
$a1 = (rst \ \& \ 1) \mid (\neg rst \ \& \ a1)$;
$a2 = (rst \ \& \ 0) \mid (\neg rst \ \& \ a2)$;
$a3 = (rst \ \& \ 0) \mid (\neg rst \ \& \ a3)$;
$x = a3; a3 = a2; a2 = a1; a1 = x$;

4.4.5.3 `While..EndWhile`

The HarPE `While..EndWhile` construct provides support for loop unrolling where the number of times the loop should execute may only be determined at run time, though a constant upper bound is necessary in order for compilation to terminate. The unrolling algorithm proceeds by rewriting the `While` statement, unrolling the loop body one iteration at a time (see Section 2.2.2.1), until the conditional can be determined to be false. Note that this requires all `While` loops to have upper bounds that can be determined at compile time in order for the algorithm to terminate.

The code sequence

```
Int⟨3⟩ a(1), b(0), c("stop");
While(b < 3 & !c[b]);
    a = a * a
    b = b + 1
EndWhile();
```

loops through the bits of c, squaring the value of a each time as a side-effect, stopping either when the relevant bit of c is $true$ or when the upper bound, 3, is reached. HarPE unrolls the loop equivalently to the following series of nested `If` statements:

```
Int⟨3⟩ a(1), b(0), c("stop");
If(b < 3 & !c[b]);
    a = a * a
    b = b + 1
    If(b < 3 & !c[b]);
        a = a * a
        b = b + 1
        If(b < 3 & !c[b]);
            a = a * a
            b = b + 1
        EndIf();
    EndIf();
EndIf();
```

Unrolling terminates when the condition of the `While`..`EndWhile` loop can be determined to be $false$ by combinational rewriting.

4.5 HarPE Internals

HarPE exists as a single C++ header file (`harpe.h`) that implements an embedded language within C++. It consists of a number of class definitions and inline functions that, together, allow compilable C++ code to express hardware constructs directly. When this code is compiled, HarPE template definitions and inline functions are expanded (by any ISO C++ compiler) into code that generates and manipulates a directed graph representation of the target circuit. Finally, one of several 'code generators,' may be called that traverse the directed graph either to generate Verilog or SAT problems in a variety of formats.

4.5.1 Syntactic Sugar

The HarPE language, as described in Section 4.4, is a thin layer of syntactic sugar over the internals of the library. All of the user-accessible classes and functions are on the most part wrappers that provide a syntactically pleasing interface to the underlying functionality.

For example, the `Bit` class's implementation of the logical AND operator (actually an inline function) is as follows:

```
inline Bit operator&(Bit a, Bit b)
{
    return hwGate::And(a, b);
}
```

The `Bit` class's overloaded `&` operator causes an AND gate (or equivalent thereof) to be generated, without requiring the user to become involved with the internal implementation details of the library.

Most of HarPE's actual functionality is implemented in or around the `hwGate` class, which has an interface that is oriented towards efficiency rather than syntactic elegance, so is not well suited to being used directly. The various wrappers implement a friendly interface to `hwGate` – by keeping the syntactic sugar separate from the internals, the task of achieving a simple, intuitive syntax need not get in the way of internal sophistication, or vice-versa.

4.5.2 The `hwGate` Class

HarPE represents target circuits as directed graphs, and it is the `hwGate` class that has this responsibility in the code. A single `hwGate` object (instance of the `hwGate` C++ class) represents exactly one of the following circuit elements:

1. 2-input AND gate.
2. 2-input OR gate.
3. NOT gate.
4. D-type flip-flop.
5. Connection to power (logic *true*).
6. Connection to ground (logic *false*).
7. Connection to a named external input.

If higher-level code wishes to create an instance of a `hwGate`, this is never carried out by simply allocating a new instance of the relevant object. Rather, the existing directed graph is searched in order to find an equivalent, already-extant gate, or a signal that is known to be identical. A 'constructor function' is called[5] given the inputs of the gate, if any, and the function returns a pointer to a (possibly pre-existing) `hwGate` object that implements the necessary functionality. Wherever possible, needless duplication is avoided, so the directed graph grows as slowly as possible – this significantly reduces HarPE's memory footprint in comparison with approaches that implement optimisation as a separate pass.

For example, the HarPE code:

[5]Not actually a C++ constructor in the usual sense of the word, rather a static function that implements the necessary optimisations.

```
Bit a("apple"), b("banana"), c, d;

c = a & b;
d = a & b;
```

proceeds by first allocating `Bit` wrappers `a`, `b`, `c` and `d`, which are initialised such that `a` and `b` point respectively to `hwGate` objects representing external inputs `apple` and `banana`, and `c` and `d` point initially to a single global connection to ground. The assignment `c = a & b` calls `hwGate::And()`, which (assuming a previously empty circuit) will cause a single AND gate to be added to the directed graph, whose inputs are connected to the `hwGate` objects representing the external inputs. Note that `hwGate` objects only ever point to other `hwGate` objects, and never to wrapper objects like `Bit` or `BitReg`. When the second assignment, `d = a & b`, is compiled, the `hwGate::And()` constructor first checks the directed graph, and on finding a previously generated AND gate with the necessary inputs, simply returns a pointer to the original AND object. Where possible, the optimisations try to arrange that separate Boolean subexpressions with identical values are represented by the same nodes in the directed graph. It is therefore possible, when performing optimisations, to compare passed-in pointer addresses – if the addresses are the same, then it is guaranteed that both inputs represent the same value. If the addresses are different, the values are assumed also to be different. Of course, since HarPE's optimisation is suboptimal[6] it may miss some cases, resulting in multiple nodes reprenting identical subexpressions. Nevertheless, this approach is safe – equality is only assumed in cases where it is definitely known, and cases missed by the optimiser result only in a larger gate count, never incorrect functionality.

Simple Boolean optimisations are carried out by the `hwGate` constructors directly. The code:

```
Bit a("apple"), b;

b = a & !a;
```

will first cause a NOT gate to be generated, whose input is connected to `apple`. When the wrapper requests `hwGate` to generate an AND gate, the optimisation notices the Boolean identity and returns instead a pointer to the single global-scope `hwGate` object that represents a connection to *ground*.

4.5.3 Assignment and `If()...EndIf()`

In the examples of the previous section, assignment was used without further explanation. In those simple cases, intuitive assumptions about assignment hold. However, the actual mechanism used to implement assignment statements in HarPE is complicated by the need to implement guarded assignments, as described in Section 4.4.5.1.

[6] This was a deliberate design decision – optimal optimisation strategies all tend to have space or time (or both) blow-up problems with important classes of circuit, particularly multipliers. HarPE's optimisations tend towards time roughly linear in the size of the original circuit and space normally smaller than the original circuit.

HarPE implements a guard stack (see Section 4.4.5.1). The `If()` function pushes a new guard onto the stack, and `EndIf()` pops the top value off the stack. Guards are hierarchical – an empty guard stack implies a *current guard* value of *true*. Calling `If()` with a conditional expression places that expression on the guard stack, and the current guard then becomes the value on the top of the stack. Further calls to `If()` conjoin further expressions to the current guard.

The following (simplified) code fragment illustrates HarPE's assignment implementation quite succinctly:

```
Bit operator=(const Bit &b)
{
    g = hwGate::Or(hwGate::And(grd, b.g),
                   hwGate::And(hwGate::Not(grd), g));

    return *this;
}
```

Note that `g` is the member variable of `Bit` that represents the value (signal) that the wrapper currently refers to, and `grd` is the current guard. In effect, this code generates a 2-input multiplexer, whereby the new value for `g` is defined such that when `grd` is false, the previous value is retained, and when `g` is true, the new value `b` is taken instead. The optimisations built into `hwGate::Or()`, `hwGate::And()` and `hwGate::Not()` guarantee that actual hardware is only ever generated when it is really necessary[7].

4.5.4 Loop Unrolling with `While()` ... `EndWhile()`

The `While()` ... `EndWhile()` construct is implemented in HarPE using the following C-style macro expansion:

```
#define While(X) _BeginWhile(); while((X).mayBeTrue()) { _If(X);
#define EndWhile(X) _EndIf(); } _EndWhile();
```

The effect of this code is not immediately obvious from the macro definitions, but it can be illustrated by working through the expansion of a small example:

```
While(a);
    b = c & d;
EndWhile();
```

Manually applying the macro expansions and reformatting for readability gives:

```
_BeginWhile();
while(a.mayBeTrue())
{
    _If(a);
        b = c & d;
```

[7]The actual implementation of the `Bit::operator=()` function performs much more optimisation than this, for obvious reasons, though this has been omitted here for clarity.

```
    _EndIf();
}
_EndWhile();
```

The `_BeginWhile()` and `_EndWhile()` functions perform internal housekeeping and won't be described further. The `_If()` and `_EndIf()` functions are slight variations on the usual `If()` and `EndIf()` that implement the necessary nesting of guards as the loop is unrolled[8]. The tt Bit::mayBeTrue() helper function returns *false* if the underlying `hwGate` is a connection to ground (logical *false*), and returns *true* in all other cases – the C++ `while()` loop therefore proceeds until the conditional becomes 'definitely false.'

Note that, in many cases, it is also possible to use the standard C++ conditional and looping facilities in HarPE code, though it is necessary to bear in mind that these constructs do not affect the guard stack and therefore they should be used with caution.

4.5.5 Handling D-type Flip-Flops

D-type flip flops are supported by the `BitReg` wrapper class and by the underlying `hwGate` functionality. A `Bit` wrapper is initialised to point to logic *false,* and takes on new values when its value is reassigned. The `BitReg` wrapper is extremely similar, except that it allocates a special `hwGate` object representing a D-type flip flop. Initially, the flip flop has no input, and its output is the wrapper variable's initial value. Reassignment of the tt BitReg wrapper is handled in exactly the same way as reassignment of an ordinary `Bit` variable – in fact, in the current implementation, this is carried out by the same code in both cases. Eventually[9], the final value of the *BitReg* variable is connected back to the flip flop's input.

4.5.6 Bit Vectors and Integers

The HarPE library provides several classes that encapsulate multiple `Bit` or `BitReg` variables in larger structures. The `Int` and `IntReg` classes are built on top of `Bit` and `BitReg` respectively, and may be used either as a vector of bits or as a two's compliment integer. Width is specified by a template parameter, and has no inherent granularity – it is equally acceptable to specify a 37-bit register as it is a 16-bit register, for example.

The code:

```
Int<9> a("apples"), b(23), c;

c = a + b;
```

declares `a` to be a 9-bit input, whose individual wires are named `apples_0` ... `apples_8`. The 9-bit variable `b` is initialised to 23 decimal, and `c` is initialised by default to 0 (*i.e.,* all bits *false*). The reassignment `c = a + b` generates a bit-level adder as expected. The optimisations inherent in the underlying `hwGate` implementation offer a significant gate

[8]Note that this example code is not likely to be compilable, since `a` is not altered within the body of the loop. Since loops are unrolled fully, unless `a` was initially *false,* HarPE would fail to terminate in this case.

[9]*i.e.,* after compilation has completed, but before code generation.

count reduction over an unspecialised adder of similar width – see Section 4.6.2.1 for a relevant worked example.

The C++ `[]` operator, normally used for array referencing, is overloaded in the wrapper class in such a way that the usual C++ array notation may be used to access individual bits. When the array index is determined by the C++ compiler to be a constant, no specialised hardware is generated, and the statement resolves to a reference to the underlying `Bit` or `BitReg` object as necessary. If the index is itself an `Int` or `IntReg`, a multiplexer is generated automatically. Note that multiplexers tend to optimise very well when partial information about the index is known. For example,

```
Int<16> x("databus");
Int<4> addr("addr");
Bit p;

p = x[addr & 0x07];
```

will generate an 8-way multiplexer, rather than a 16-way as might be expected. This optimisation results from the usual PE collapse due to partially known values, rather than from any explicitly coded optimisation in HarPE itself.

4.5.7 Generating Gate-Level Verilog

Once a directed graph structure has been created, generating gate-level Verilog is very straightforward. The `Output()` function adds a specified named output to a list, and would normally be called multiple times in order to expose all of a circuit's outputs. Verilog generation proceeds by a simple recursive tree walking algorithm, seeded from each output node in turn:

1. If the current node has already been emitted, emit a reference to the existing `wire` and then return immediately.
2. If the current node is an AND, OR, NOT or D-type, recursively emit code for its inputs first before proceeding.
3. Emit a new wire declaration. Depending on the current node:
   (a) If the node is an AND, OR or NOT object, emit an `assign` statement that implements the necessary gate.
   (b) If the node is a power or ground object, emit an `assign` statement that fixes the wire's value to `1'b1` or `1'b0` respectively. Note that at most one definition for power and one definition for ground will ever be emitted.
   (c) If the node is a D-type flip flop, emit a `reg` declaration and a suitable `always @ (posedge CLK)` block. The code is emitted such that on each clock, the new value of the register is set to the value of the wire representing the input of the D-type flip flop in the directed graph, and its output is wrapped back as described in Section 4.5.5.
   (d) If the node is an external input, emit a suitable `input` statement.

The generated Verilog is purely bit level. For this reason, HarPE should not be thought of as targeting RTL in the normally accepted sense of the term, though its output (always in the form of a single Verilog module) can be integrated with higher-level human-written Verilog and passed to a conventional tool chain for further optimisation and synthesis.

4.5.8 SAT solver interface

The SAT solver interface in HarPE is currently somewhat experimental, though it already provides facilities for extracting SAT problems in a number of formats. HarPE can (with a compile-time `#define` statement) be constrained to build its internal directed graph in CNF or NNF – in both cases, this is achieved by selectively enabling extra 'optimisation' rewrite rules that have the effect of flattening the circuit. Alternative code generators are included that can generate output for ZCHAFF or in the form of Standard ML source code. A more fully developed interface to our own NNF-WALKSAT solver is included that calls the solver library directly as the graph is walked, thereby removing the need to generate a description of the SAT problem in textual format and then have the solver's front end parse it.

The NNF-WALKSAT interface can optionally either expand or retain sharing. Earlier versions of the solver required that input must be structured as a tree, though it later became apparent that the same algorithm (with minor modifications) would also work on a directed graph. In one test case, a complete definition of the MD5 hashing algorithm was implemented in HarPE – without sharing, it was not feasible to apply NNF-WALKSAT as a consequence of the exponential blow-up that undoing MD5's particularly pathological sharing would require. However, with sharing enabled, it was possible to rapidly generate a usable SAT problem, without any requirement to introduce temporary variables.

4.6 Experimental Results

4.6.1 Test Environment and Experimental Procedures

For all of these experiments, code was compiled by HarPE, generating gate-level Verilog, which was then passed to Altera's Quartus II tool chain [8]. Each resulting circuit was compiled for an Altera Excalibur EPXA1F484C1 FPGA [7], then examined using the simulation tools within Quartus. Selected designs were uploaded to an Altera EPXA1 development board (see Fig. 4.6), though as this has a fixed 25MHz clock, timing information quoted below was calculated by post-layout timing simulation by the tool chain for designs that required a substantially different rate. The pure-combinational circuits were not characterised for timing.

4.6.1.1 Empty Circuit

To ensure that the gate count and other similar statistics were not skewed by something similar to the overhead of library code familiar in the software world, the following code

```
Int⟨7⟩ c(0);
Output("R1", c);
```

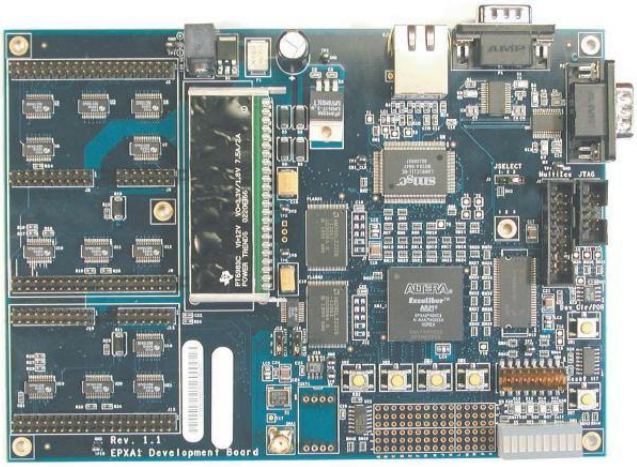


Figure 4.6: Altera EPXA1 development board

was compiled and passed through the Quartus II tool chain. The test confirmed that, as expected, zero gates and zero flip flops were emitted by HarPE, resulting in a test FPGA which used zero logic elements (LEs).

4.6.2 Combinational PE

4.6.2.1 Specialising an Adder

The (unspecialised) program

$\texttt{Int}\langle 7\rangle\ a(\text{“a”}), b(\text{“b”});$
$\texttt{Int}\langle 7\rangle\ c;$
$c = a + b;$
$\texttt{Output}(\text{“R1”}, c);$

causes HarPE to emit 91 gates. Specialising b to the numeric value 1, *e.g.,*

$\texttt{Int}\langle 7\rangle\ a(\text{“a”}), b(1);$

reduces the gate count to 36. In a cases where both a and b are specialised, *e.g.,*

$\texttt{Int}\langle 7\rangle\ a(25), b(9);$

exactly zero gates are generated. Note that, as in all of these tests, HarPE performs this partial evaluation *only* at bit level – it has no higher level rules dealing with integers or any other more complex data types.

Tying both inputs of the adder together:

$c = a + a;$

also results in a zero gate count, generating only wiring that performs a ‘shift left’ operation. Again, this results directly from bit level PE without higher level rules being necessary.

Loops per Cycle	Gates[a]	DFFs[b]	LEs[c]	Max Clk[d]
1	35	7	16	257MHz
2	69	7	13	257MHz
3	103	7	22	183.35Mhz
50	1701	7	18	257Mhz

[a]Total count of AND, OR and NOT gates emitted by HarPE
[b]D-type Flip Flop
[c]FPGA Logic Elements, each of which typically comprises a small lookup table capable of implementing an arbitrary Boolean expression with a (small) fixed number of inpute, and one or more flip flops
[d]Maximum Clock Rate

Table 4.2: Loop unrolling a 7-bit up counter

4.6.2.2 Specialising a Multiplier

Replacing $c = a + b$ in the test case shown in Section 4.6.2.1 with

$$c = a * b;$$

generates a 7bit $\times$ 7bit multiplier, emitting 443 gates. Specialising b to take the value 5 reduces this to just 58 gates.

The code

$$c = a * a;$$

generates a 'squarer' by tying the multiplier's inputs together. In this case the resulting gate count is 432, somewhat improved on the unspecialised version, though not so spectacularly as for addition.

4.6.3 Synchronous PE

4.6.3.1 Loop Unrolling of a Simple Counter

A simple, 7-bit up counter may be implemented as follows[10]:

$\texttt{IntReg}\langle 7\rangle\ reg;$
$reg = reg + 1;$
$\texttt{Output}(\text{“out”}, reg);$

This circuit is particularly amenable to loop unrolling – see Table 4.2 for timing and gate count results. The disparity between the number of gates emitted by HarPE and the number of LEs generated by the Quartus II tool chain is indicative that the latter's more sophisticated combinational optimisation is successfully collapsing multiple increments into a single constant addition. Since HarPE emits purely bit-level Verilog, this optimisation must again be entirely bit-level in nature.

4.6.3.2 Loop Unrolling a Fibonacci Series Counter

The code

```
IntReg⟨7⟩ a, b;
Int⟨7⟩ temp;
Bit reset("rst");
If(reset);
    a = 1;
    b = 0;
EndIf();
temp = a + b;
b = a;
a = temp;
Output("out", a);
```

implements a specialised counter that outputs the Fibonacci series $(1, 2, 3, 5, 8, 13, 21, 34, \dots)$. These test cases, and those of Section 4.6.3.3, are loosely based on an example due to Page & Luk [106]. Test results are shown in Table 4.3. This time, maximum clock rate falls off as the number of unrollings increases – this is an expected (if not entirely welcome) feature of PE, and is caused by increasing propagation delays due to longer, more complex data paths.

4.6.3.3 Partial Evaluation of a Small Processor

Loosely following [106] we define a small, 7-bit microprocessor with one 7-bit general purpose register, 8 bytes of RAM and 8 bytes of ROM. Both the RAM and ROM are mapped into a single 16 byte address space, with address $0..7$ being RAM and $8..15$ being ROM. The contents of address 13 (labelled R_1 in the assembler source below) are externally visible as a 7 bit output port for simulation and verification purposes.

Instructions are all single byte, with the 3 most significant bits representing an opcode and the 4 least significant bits representing a single operand. The supported instruction set is shown in Table 4.5.

In all tests shown here, the ROM contains the following program:

[10] Note that there is an implicit outer *while* loop – see also Section 4.4.1.

Loops per Cycle	Gates[a]	DFFs[b]	LEs[c]	Max Clk[d]
1	107	14	38	163.03MHz
2	191	14	51	120.5MHz
3	275	14	52	96.83MHz
5	443	14	83	73.16MHz

Table 4.3: Loop unrolling a Fibonacci counter

	Gates	DFFs	LEs	Max Clk	Run Time
Unmodified, 2 cycles per instruction	2029	75	646	27.48MHz	5.6μS (at 25MHz)
Merged fetch/execute, 1 instruction per cycle	1810	67	588	28.3MHz	2.8μS (at 25MHz)
2 $\times$ unrolled, 2 instructions per cycle	3883	67	1426	16.05MHz	2.37μS (at 15MHz)
4 $\times$ unrolled, 4 instructions per cycle	8029	67	2776	8.72MHz	2.1μS (at 8.33MHz)
Fully unrolled, 1 loop iteration per cycle	107	14	36	153.92MHz	70nS (at 150MHz)

Table 4.4: Experimental results for partial evaluation of a small processor

$R_1 = 13$
$R_2 = 14$
$X = 15$

$start:$	LDA	R_2
	ADDA	R_1
	STA	X
	LDA	R_2
	STA	R_1
	LDA	X
	STA	R_2
	JMP	$start$

A hardware reset circuit preinitialises R_2 with the value 1. All other locations are initialised to 0. Since the program loops forever unless externally terminated, run times were measured by layout aware timing simulation in Quartus II, measuring from the falling edge of the reset pulse to the time that R_1 reaches the arbitrarily chosen value 34 decimal (0100010 binary).

The basic processor was implemented in HarPE and instrumented to allow various levels of loop unrolling to be applied. Test results are shown in Table 4.4. Without unrolling, the processor requires 2029 gates (646 LEs), and executes one instruction every 2 clock cycles due to an explicit two phase fetch/execute cycle. Flattening this to one cycle, somewhat surprisingly, *reduces* the gate count and maintains a roughly similar maximum clock rate, halving the run time of the program[11]. Further unrolling generated

[11]Though the reason for this reduction is unclear, it seems likely to be an artefact of our very simple

Opcode	Mnemonic	Description
000	SKIP	Do nothing
001	LDC	$acc := operand$
010	LDA	$acc := mem[operand]$
011	STA	$mem[operand] := acc$
100	ADDA	$acc := acc + mem[operand]$
101	JMP	$ip := operand$
110	STOP	Halt

Table 4.5: Instruction set

versions of the processor that executed 2 and 4 instructions per clock cycle – simulation showed that these versions worked correctly, but increasing worst case propagation delays appeared to restrict the practical speedups that could be achieved.

Fully unrolling the loop so that the entire loop executes one iteration per clock cycle causes a dramatic reduction in gate count along with a large increase in speed. The resulting circuit compares well with the simple Fibonacci counter described in Section 4.6.3.2 – partial evaluation apparently optimises away the processor, leaving behind only the hardware necessary to implement the ROM program.

4.6.4 Computational Cost

In all examples described in this chapter, partial evaluation run times were insignificant in comparison with the run time of the FPGA tool chain, under 10 seconds running on a 1GHz Pentium III under Linux. In general, when combinational PE is implemented by the term rewriting rules described in Table 4.1, complexity is approximately $O(M \times N)$, where M is the size of the original circuit and N is the number of loop unrollings carried out.

4.7 Related Work

The author's 1991 M.Sc thesis [129] described a hardware compiler based upon partial evaluation, that compiled a dialect of C to a gate-level netlist – this chapter significantly extends that work and places it in a modern context. In other work [136], HarPE has been used to flatten circuits (in this case, small areas of an FPGA) to a combinational form suitable for analysis by a SAT solver (see also Chapter 5 and Section 4.5.8).

4.7.1 Dynamic Synthesis of Correct Hardware

The *Dynamic Synthesis of Correct Hardware* project [92, 91] at the University of Glasgow, which ran from May 1997 to May 1999, reported encouraging results from bit-level combinational PE, though did not address loop unrolling. Generic circuits were compiled to layouts for Xilinx XC6200 FPGAs, then specialised at the level of the layout, rather than at the level of the specification as implemented HarPE and by typical software partial evaluators. Significant speedups were achieved, though the reported results are perhaps most notable for the relatively lightweight and fast partial evaluator, which generated chip layouts directly via the Xilinx `jBits` interface without any requirement for a back-end tool chain. Layout-specific PE does not lend itself easily to loop unrolling, however, and this was not attempted.

4.7.2 SystemC

SystemC [4, 105] is a sophisticated C++ class library that was initially aimed at high-level hardware modeling, but over time has grown to encompass the entire hardware

fetch/execute implementation and is unlikely to be exhibited when specialising more complex processors.

concept-to-implementation lifecycle. The Open SystemC Initiative (OSCI) reference implementation is a freely downloadable implementation of the SystemC standard. As in the HarPE library, C++ operator overloading is used extensively in order to allow SystemC to function as an embedded language. The reference implementation is essentially a hardware simulator – it does not perform synthesis. Like HarPE, the SystemC standard specifies a *shell,* providing an outer, user-level interface, and a *kernel,* housing the internals of the library. Designs coded in SystemC are typically higher level (system-level, hence the name) than those coded in more traditional hardware description languages. Multiple concurrently communicating subsystems may be modeled at the transaction level – the OSCI reference implementation simulates this by leveraging the coroutine support within the underlying operating system.

Control flow in SystemC code is performed by the usual C++ language constructs. This is illustrated by the following code fragment, taken from one of the sample designs provided with the OSCI reference implementation:

```
void pic::entry(){
    if (ireq0.read() == true) {
        intreq.write(true);
        vectno.write(0);
    } else if (ireq1.read() == true) {
        intreq.write(true);
        vectno.write(1);
    } else if (ireq2.read() == true) {
        intreq.write(true);
        vectno.write(2);
    } else if (ireq3.read() == true) {
        intreq.write(true);
        vectno.write(2);
    } else {
    }
    if ((intack_cpu.read() == true)
        && (cs.read() == true)) {
        intreq.write(false);
    }
}
```

This shows a substantial departure from HarPE's approach to control flow (see Sections 4.4.5 and 4.5.3), and probably makes it infeasible to implement a standards compliant SystemC library implementation that performs synthesis directly without specialised preprocessing upstream of the C++ compiler.

4.7.3 Cynthesizer

Forte Design Systems' [52, 110] *Cynthesizer* product is essentially a compiler that transforms transaction-level SystemC programs to low-level RTL. It is claimed that simulation and synthesis can be carried out from the same source code, though the company's publicly available information does not specify how large a subset of the SystemC standard

is synthesizable directly by their tool. Intellectual property (IP) libraries are provided that implement hardware versions of typical transaction-level constructs, which allows designs to be synthesized whilst retaining strict compliance with the semantics of the high-level design.

4.7.4 Synopsys Behavioural Compiler

The Synopsys Behavioural Compiler [80] (referred to here as BC for brevity) is no longer a current product, but was influential in being one of the first widely available behavioural synthesis tools. BC supports the synthesizable subsets (with some extensions) of both Verilog and VHDL, and generates conventional RTL for synthesis by an existing back-end tool chain (normally the Synopsys Design Compiler, see Section 4.7.5). A typical design consists of Verilog or VHDL code with one or more *processes* embedded within it, each of which takes the form of a Verilog `always` or `forever` block or a VHDL `process`. The Behavioural Compiler ignores any non-behavioural code, and operates only on processes. Processes normally specify behaviour spanning multiple clock cycles – a typical code fragment is as follows:

```
forever begin
    @(posedge clk);
    out = 2'b00;
    @(posedge clk);
    out = 2'b01;
    @(posedge clk);
    out = 2'b10;
end
```

which implements a counter that repeats the sequence 00, 01, 10, 00, 01, 10 . . . indefinitely. Though the syntax would appear to support multiple asynchronous clocks, BC mandates that a single, global synchronous clock is used. All processes are considered (and therefore scheduled) separately; BC does not attempt to enforce protocols between state machines. Loops that have constant upper bounds are unrolled automatically by the compiler, though this can be overridden by a pragma. Infinite loops, and loops that have upper bounds that are computed at run time cannot be unrolled. Though there is no direct equivalent of HarPE's support for loops with a compile-time upper bound and a run-time limit, this can be simulated by coding the upper bound as a loop and embedding a suitable conditional inside it.

Processes are each compiled to a data path that is controled by a single synchronous state machine. BC explores the scheduling solution space in order to find schedules that meet the user's constraints. It is therefore possible, with minor changes to the source code, to explore many potential implementations rapidly. This aspect, along with the code size reduction that comes from behavioural Verilog/VHDL's more concise representation, is often quoted as providing a very significant reduction in design time.

HarPE is actually closer to more conventional, non-behavioural, Verilog compilers than it is to BC. HarPE source code strictly describes single-cycle behaviour, whereas BC's processes strictly describe multiple cycle behaviour. Both tools perform loop unrolling;

in HarPE's case this is mandatory, though since BC is free to schedule loops to execute across multiple cycles, loops can be unrolled or not as necessary.

4.7.5 Synopsys Design Compiler

The Synopsys Design Compiler [124] (referred to here as DC for brevity) is the company's flagship RTL compiler product. Source code is industry-standard RTL, and output is gate-level netlists targetted at any of a range of implementation technologies inlcluding FPGAs and ASICs from a variety of foundries. DC supports constraint-based optimisations, as well as sophisticated technology mapping. The compiler is further supported by a wide selection of reusable IP cores, including support for image processing, communications and a range of hardware interface standards, as well as a wide range of related tools.

4.7.6 Synopsys System Studio

The Synopsys System Studio product [125], originally developed by CoCentric, is a commercial system-level modeling tool that supports SystemC. It has a particular bias toward System-on-Chip (SoC) applications; a variety of additional IP modules may be purchased that provide accurate cycle-level models of a wide range of common interface, algorithm and protocol standards.

4.7.7 Bluespec

The Bluespec hardware compiler [14] performs partial evaluation at the register transfer level[12], (*i.e.,* not at bit-level), though details of this remain unpublished. Bluespec is somewhat more sophisticated than the Synopsys Behavioural Compiler, in that it is capable of automatically enforcing synchronisation protocols between subsystems. Bluespec's native source language is a dialect of Haskell [74] with hardware-specific extensions. A SystemVerilog [5, 70] front-end is also provided – though it is less expressive than the Haskell-based alternative, it is more familiar to engineers that have come from a Verilog [69] background. Recently, SystemC support [24] has been added, which at the time of writing allows Bluespec designs to be simulated with exising SystemC simulators, particularly the OSCI reference implementation. The company has announced an intention to support SystemC synthesis in late 2006, though more specific detail is not currently available.

4.8 Discussion

The experimental results shown in Section 4.6 clearly show that partial evaluation of synchronous hardware is feasible. Partial loop unrolling offers designers an ability to specify circuits relatively simply, then transform them into faster (though possibly more

[12]Personal conversation between the author and Joe Stoy at the APPSEM II workshop in Frauenchiemsee, Germany, 2005.

complex) circuits purely by transformation. Full unrolling goes further, making it possible (as demonstrated in Section 4.6.3.3) to transform a processor and a ROM image into equivalent, low-level dedicated hardware – potentially, any synchronous soft core processor, in conjunction with a suitable partial evaluator, can be used as a hardware compiler for the machine language interpreted by the soft core itself.

In all of our tests, partial evaluation gave a net speed gain in comparison with the original circuit. In some cases, gate count was also reduced. Full unrolling gave the most extreme results, with a 2 orders of magnitude speed up and 1 order of magnitude reduction in gate count.

4.8.0.1 Limitations

Partial evaluation may, in principle, be applied to any circuit that has been designed assuming a strictly synchronous paradigm. However, it is not envisaged that it would normally be applied as an automatic transformation; rather, it is expected to extend the palette of design approaches available to hardware engineers.

Part III

Applications

Chapter 5

Repairing Cosmic Ray Damage in FPGAs with Non-Clausal SAT Solvers

The work described in this chapter was carried out at the NASA Ames Research Center during Summer 2004, in collaboration with Guillaume Brat and Arnaud Venet, and was published in [136].

5.1 Introduction

FPGAs are finding an increasing number of applications within NASA in deep space probes, planetary rovers and manned vehicles. Like other silicon devices, FPGAs can be damaged by high energy cosmic ray impacts, resulting in permanent latch-up conditions that manifest as 'stuck-at' faults. Traditionally, multiple redundancy and voting logic have been employed as a work-around, particularly for high reliability, extreme environment applications. However, reconfigurable FPGAs are becoming increasingly common in flight systems, offering a potentially valuable possibility for improved levels of fault recovery – after a fault is detected and localised within an FPGA, it is feasible to reprogram the device, in flight, with an alternative, equivalent, circuit that does not depend upon the damaged portion of the chip.

Designing such alternative chip layouts by hand is a valid option, though costly in terms of the man-hours of effort required; a fully automated alternative would be far preferable. In this chapter, a technique is presented that allows the automatic generation of FPGA configurations for fault recovery purposes by means of non-clausal SAT solver technology.

Designing hardware capable of reliable operation in deep space is far from trivial. The familiar, tried and trusted design techniques employed by engineers working on conventional, ground-based electronics are not sufficient to ensure reliability in the extreme environment of deep space. Radiation, extreme temperatures, hard vacuum and many other challenges must be addressed whilst accommodating a requirement for extremely high reliability – deep space probes typically must operate for decades, with no possibility of servicing by astronauts if anything goes badly awry.

Inherently radiation hard semiconductor devices do exist, though they carry a very significant cost penalty, as well as generally requiring more power in return for less performance in comparison with commercially available off-the-shelf (COTS) devices. A common radiation hardening design approach involves taking an existing COTS standard cell design, then synthesising a new version where some or all of the original gates and flip flops are replaced with more complex, internally redundant equivalents. The widely used RAD6000 processor was created by replacing the standard cells of the original IBM RS/6000 design with hardened versions, resulting in a processor with greatly improved radiation hardness with respect to the original. Such chips are more radiation resistant than the COTS equivalent, but are slower, require more power and are typically extremely costly ($100k per device is not unusual) due to the need to amortise foundry set-up costs over a relatively small number of saleable devices. Designers operating within contemporary budgetary constraints often therefore prefer to use COTS devices where possible, reserving extremely expensive radiation hard components for critical subsystems only. For example, a mission critical guidance system might be implemented with radiation hardened chips, but a less critical instrument package might use COTS components instead, achieving a significant cost, mass and power saving as well as allowing higher clock rates.

5.1.1 FPGAs in Space

The Apollo programme at it height consumed more than half the world's entire chip manufacturing capacity, comprising many custom-built ASICs. Modern spacecraft, however, are designed within budgetary constrains that mean that full custom ASICs are far too expensive to be considered. Nevertheless, mass limitations[1] still mean that custom chips are necessary. Field programmable gate arrays (FPGAs) offer a good compromise; though less efficient than full-custom ASICs in terms of density and power consumption, they nevertheless offer a means by which custom chips can be incorporated into designs without incurring the huge (approximately US$2 million per iteration) fabrication costs of full-custom devices. FPGAs typically contain a large array of general purpose logic that only 'becomes' the target circuit after an appropriate configuration bit stream is uploaded. In some FPGA families, particularly those manufactured by Actel, programming is carried out once only, after manufacture but typically before the chip is incorporated into a board-level system. Other families, particularly those manufactured by Xilinx and Altera, hold their bit stream in static RAM, thereby making it possible to reconfigure such FPGAs dynamically.

As with any other semiconductor device, FPGAs are susceptible to radiation effects including single-event upsets (SEUs) and permanent latch-up faults. Radiation hard FPGAs are commercially available [9], though they tend to have lower density, lower performance and significantly higher cost than commercial grade devices. At the time of writing, both approaches are in use in ongoing missions – the Galileo/Huygens spacecraft incorporates a number of Actel radiation hardened FPGAs, whereas the Mars Exploration Rover mission's twin rovers, Spirit and Opportunity, depend on COTS devices sourced from Xilinx.

[1] Largely due to launch costs of the order of approximately $30,000 per kg to low earth orbit.

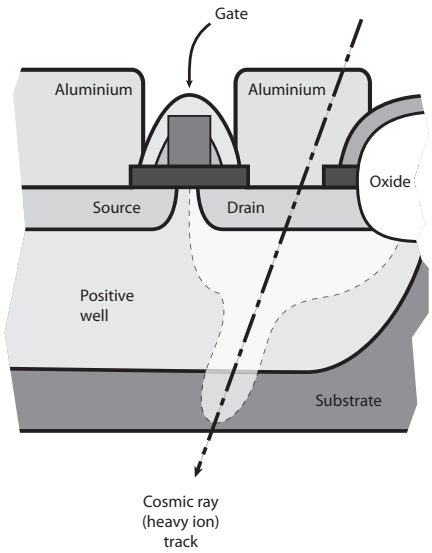


Figure 5.1: SEE triggered by a cosmic ray impact

5.1.2 Radiation Damage

As described in Section 2.1, radiation levels in space vary widely; in low earth orbit, levels can be sufficiently low that conventional electronics can be used unmodified[2]. As spacecraft venture outside the protective effects of the Earth's magnetic field, radiation levels increase both in terms of the frequency and energy of particle impacts.

Fig. 5.1 shows the effect of a heavy ion moving at a relativistic velocity (cosmic ray) passing through the gate of a field effect transistor in a typical gate. The ion leaves a trail of charge that transiently affects the operation of the transistor, which may manifest as an unwanted voltage spike in the circuit. In many cases, such spikes are benign and do not cause circuit behaviour to deviate from specification. Often, however, such a spike, typically referred to as a Single Event Transient (SET), may cause a circuit to enter an invalid state. Normally, such conditions are detected by watchdog circuits and are cleared by simply resetting the malfunctioning subsystem.

Sufficiently high energy particle impacts can cause permanent damage. Often referred to as permanent latch-up, such damage manifests as signals ceasing to function correctly and appearing to be stuck permanently at logic *true* or *false*. Such damage cannot be cleared by a reset, so some form of redundancy is required in order for the subsystem to continue to function.

5.1.3 Modular Redundancy

Traditionally, modular redundancy has been the standard approach toward mitigating the effects of permanent latch-up. In this approach, majority voting logic [143] allows the incorrect output of one or more faulty subsystems to be ignored. In 3-way modular redundancy (see Figs. 5.2 and 5.3), any two subsystems can override the output of the

[2]On the International Space Station (ISS), many computing tasks are carried out by COTS laptop PCs.

third, allowing one subsystem to fail completely without affecting system level behaviour. 5-way modular redundancy, as employed by the Shuttle main computers, allows up to two subsystems to fail without affecting functionality.

Modular redundancy is certainly effective, but its requirement for duplication of subsystems carries a significant mass and power consumption penalty. Whilst it is likely to remain a requirement for critical subsystems, cost precludes its universal applicability.

5.1.4 Exploiting Redundancy within FPGAs

For practical reasons, most FPGA layouts are typically restricted to using no more than approximately 60 – 80% of the chip's theoretical optimal capacity. FPGA layout is thought

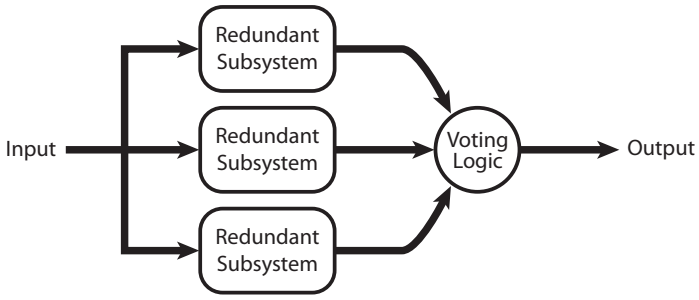


Figure 5.2: Modular redundancy

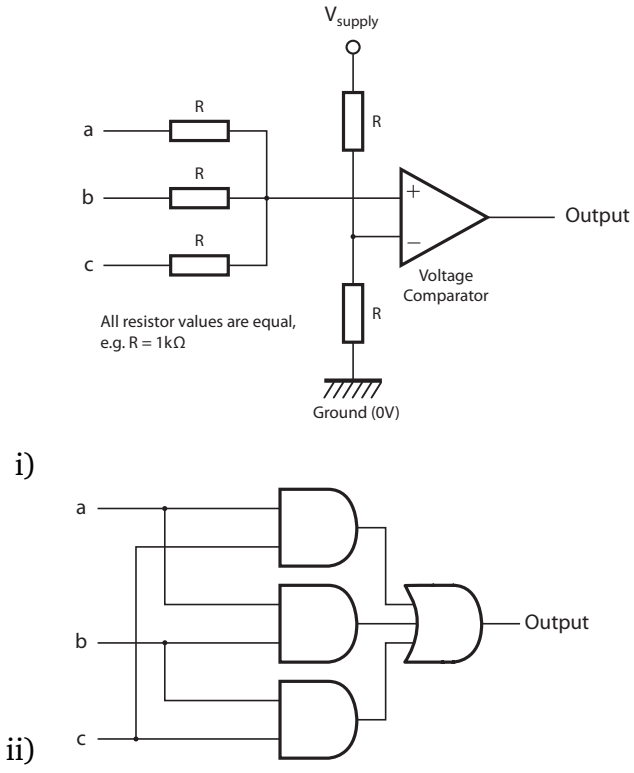


Figure 5.3: Typical majority voting logic implementations: i. Analogue, ii. Digital

to be an NP-complete problem, though good heuristics exist that can do a reasonable job of automatically mapping designs to configuration bit streams. These algorithms tend to reach a solution much faster when the design can be mapped to a relatively small proportion of the chip's resources, and can fail to generate a layout completely in cases where the proportion is close to 100%. As a consequence of this, almost all practical FPGA layouts contain a significant amount of unused resources – though FPGA circuits are not usually in and of themselves redundant, spare FPGA logic capacity unused by the circuit can nevertheless be exploited in order to improve reliability.

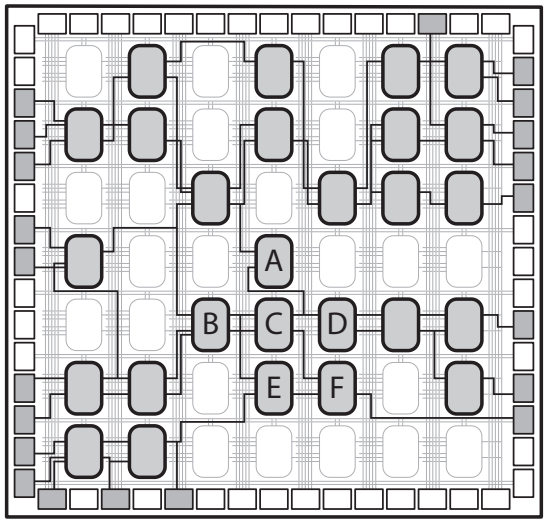
A tempting possibility would be resynthesising logic locally within small areas of the chip, adding redundancy to the circuit until the chip is completely full. This approach, however, would incur a power and performance penalty, whilst adding redundancy to circuits in an unpredictable way, without any guarantee that the resulting layout would in practice survive any particular fault.

A more practical approach is to lay out the FPGA conventionally, then *locally resynthesise* logic around faults as and when they are detected (see Section 5.2.4). Having spare capacity in terms of unused logic blocks and wiring resources spread across the chip layout makes it feasible to consider only a small area near the fault, avoiding the need to generate a complete new layout from scratch. In outline, this approach may be summarised as follows (see also Fig. 5.4):

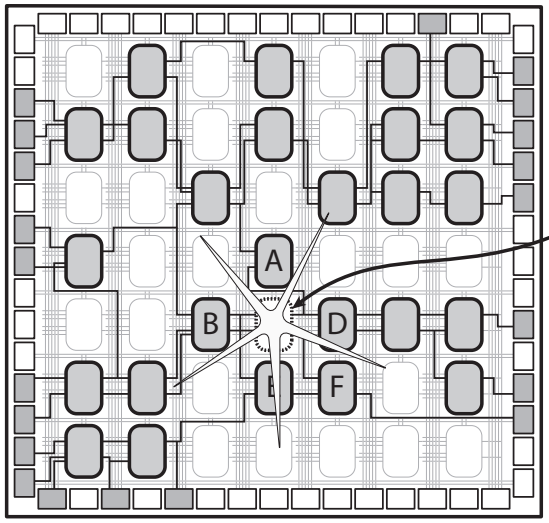
1. FPGA running normally (Fig. 5.4.i)
2. Fault detected (Fig. 5.4.ii)
3. Take FPGA off line and put it through a test procedure in order to localise the fault or faults
4. Locally resynthesise logic around each fault, resulting in a working, work-around layout
5. Upload new configuration bit stream to FPGA
6. Put chip back on line (Fig. 5.4.iii)

Several alternatives are possible as regards the implementation of local resynthesis. Most obvious is perhaps re-running the software responsible for the original FPGA layout again with appropriate constraints preventing it from using damaged parts of the chip – whilst technically feasible, this approach is not well suited to automated in-flight use, since the software required typically assumes a powerful workstation class computer, often with some human intervention.

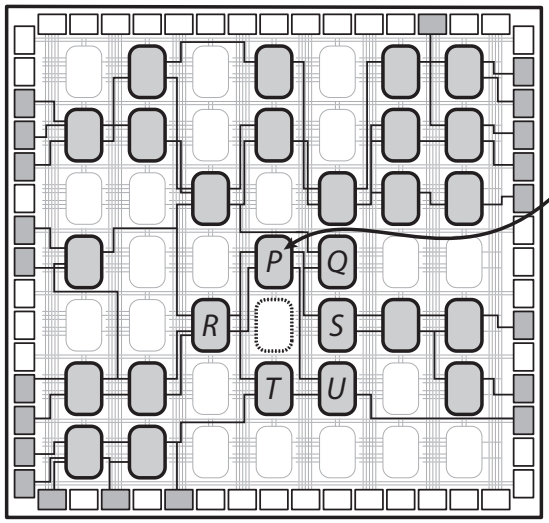
Jason Lohn's group at the NASA Ames Research Centre [86, 85] have experimented with automatically generating FPGA layouts with genetic algorithms. A population of random FPGA bit streams are tested, with their behaviour compared with ideal test traces derived from the original circuit. Over many generations, functionality tends to converge on the desired circuit, even though no formal link other than observed behavior exists between the original design and the generated design. Good results have been achieved on a number of test circuits, but the difficulty of proving that a generated circuit that includes flip flops really does implement the intended behaviour (as opposed to just



i) Undamaged FPGA



ii) Cosmic ray (heavy ion) impact permanently damages logic resource



iii) Functionality of damaged resource implemented in redundant, previously unused logic by local resynthesis

Note that the resulting look up tables need not resemble the original versions – local resynthesis does not just naïvely move blocks and reroute wires

Figure 5.4: Using available FPGA resources to work around permanent latch-up damage

happening to respond correctly to a non-exhaustive set of tests) is likely to limit the technique's applicability.

5.1.5 Availability

An FPGA undergoing repair will, of necessity, not be able to continue performing its intended function during the repair process. As a consequence, our technique will not be suitable for applications requiring high availability unless the FPGA is itself part of a modular redundant subsystem. In such situations, the ability to repair faulty subsystems is still a significant advantage, because it allows redundancy to be maintained over far longer periods.

5.1.6 Local resynthesis as a SAT problem

In this chapter, we describe a technique that can automatically perform local resynthesis whilst retaining functionality that is formally identical to that of the original circuit. In essence, formally correct local resynthesis requires an alternative, work-around bit stream to be determined such that for all possible inputs and/or internal states, the outputs and next internal state of the work-around circuit matches exactly that of the original circuit. Finding such configurations is computationally hard, perhaps prompting the adoption by Lohn's group of heuristic search algorithms that do not attempt to ensure formal correctness.

In the remainder of this chapter, we demonstrate how local resynthesis can be transformed into a equivalent SAT problem [47, 46, 40], thereby demonstrating that local resynthesis is no harder than NP-complete[3]. The resulting SAT problems are suitable for attack by SAT solvers, with solutions guaranteed to preserve correctness with respect to the original circuit.

5.2 Defining the SAT problem

Given an original, correct, bit stream b along with a model of a correct FPGA f, a work-around bit stream b' for a faulty FPGA f' must possess the following property:

$$\forall i \; . \; f(b, i) \Leftrightarrow f'(b', i)$$

Informally, this states that for all possible inputs i, the bit stream b' causes the damaged FPGA to behave exactly identically to the original FPGA and bit stream (see also Fig. 5.5). Letting b' represent any potential work-around bit stream, this expression will evaluate to *true* if and only if correct functionality is preserved – in effect, the expression embodies formal verification of a work-around bit stream with respect to an original bit stream[4].

[3]We conjecture (assuming $P \neq NP$) that no complete P space/time algorithm exists, though such speculation is beyond the scope of this chapter.

[4]Note that this approach may be used to verify the correctness of *any* work around bit stream, including those generated by genetic algorithms or by other means.

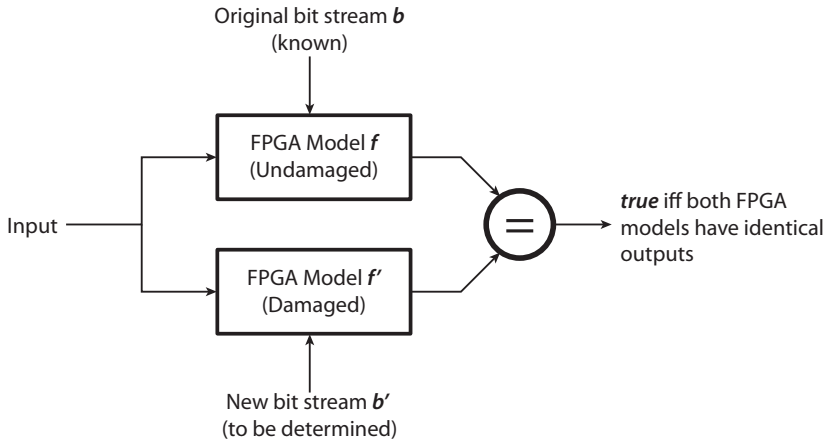


Figure 5.5: FPGA repair as a SAT problem

Alternatively, the expression may be thought of as defining a Boolean satisfiability problem whose solutions represent all possible work-around bit streams – solving such a SAT problem is therefore equivalent to the local resynthesis problem.

After constant propagation, quantifier elimination and (if necessary) transformation to CNF or NNF form, feeding the resulting expression to a SAT solver allows b' to be calculated.

It is noteworthy that no inherently complex conventional chip layout, placement or routing algorithms are required, suggesting that this functionality might be implemented within embedded systems carried on the spacecraft itself. Typical SAT solver memory requirements are generally not particularly severe for the kinds of problem we consider, requiring approximately 3MB for the circuit shown in Fig. 5.6. Run times are of the order of tens of seconds on contemporary CPUs (see Section 5.3, Fig. 5.7).

The SAT problems that result from this process are typically quite *hard,* in the sense that standard SAT solvers do not typically find solutions very quickly. Empirically, *non-clausal* SAT solvers (*i.e.,* those that do not require their formulas to be converted to CNF form) appear to be most effective, possibly because they allow circuits to be modeled in a form that is closer to their original structure.

5.2.1 Quantifier Elimination

SAT solvers typically do not directly support quantifiers, so the first step involves eliminating them from the expression. Removing the universal quantifier $\forall i$ is therefore essential. Since i may consist of several Boolean variables, it is helpful to (equivalently) express the problem as

$$\forall i_1 \; \forall i_2 \; \ldots \forall i_n \; . \; f(b, i) \Leftrightarrow f'(b', i)$$

We can now eliminate these quantifiers one by one by applying the rewrite rule

$$\forall a \; . \; F(a) \quad \longrightarrow \quad F(1) \wedge F(0)$$

repeatedly until none remain. Since this operation has an expression size and space upper bound of 2^N, this restricts our technique's applicability to fairly small sub-circuits, though this is less significant when slicing techniques are adopted (see Section 5.2.2).

After constant folding and common subexpression elimination, the resulting expression is a directed acyclic graph, with exactly one 'output' node representing the result of the expression, and one 'input' node for each bit in b'. The variables i and b are no longer externally exposed, with the resulting expression depending only upon b'. At this point, the expression may be passed to a suitable SAT solver, *e.g.,* NNF-WALKSAT, as described in Appendix B.1.

5.2.2 Slicing

Attempting to resynthesise a complete FPGA is infeasible with our method due to the tendency for the size of the SAT problem to be proportional to 2^N, where N is the total number of inputs and flip flop outputs (see Section 5.2.3). It is therefore necessary to work on a small *slice* of the chip. The rationale behind this approach is that, whilst a cosmic ray impact might render the original circuit useless, many possible work-around bit streams with low Hamming distance from the original bit stream typically exist, differing only near the damage site. Several variant approaches are feasible:

1. *Slicing by Coordinate.* In this case, a slice is chosen such that inclusion is based on physical distance (in terms of the 2D chip layout) from the damage site.

2. *Slicing by Connectivity.* Such a slice might be generated by beginning at the damage site and including all bit stream bits that are electrically reachable through a predetermined number of logic blocks.

3. *Slicing by Heuristic.* In this case, a slice might be generated by some device-specific algorithm capable of exploiting aspects of its design in order to create a more effective slice than either of the above simpler approaches.

It is possible that, in some cases, no local solution may exist, but solutions that differ more significantly may still be possible. The probability that this might occur can be reduced by arranging the original design such that used resources are spread evenly across the chip rather than clustered together, but in extreme cases the fall back option still exists of creating an alternative layout manually (*e.g.,* remotely on Earth). Our experimental results suggest that local solutions are possible in most cases, however.

5.2.3 Handling flip flops

The technique presented here essentially considers combinational circuits; clocked synchronous circuits may be accommodated by a small modification:

1. If a working flip-flop necessary to implement the original circuit falls within the slice under repair, treat its output as if it was an external *input* of the subcircuit. Similarly, treat its input as an *output* of the subcircuit.

2. If a damaged flip-flop necessary to implement the original circuit falls within the slice, exclude its connections from the slice and substitute an alternative, working flip flop. Local resynthesis will take advantage of the alternative flip-flop and avoid the damaged original.

5.2.4 Detection and Localisation of Faults

It is envisaged that faults will initially be detected as a consequence of observably incorrect behaviour of a subsystem implemented on an FPGA. Well known techniques already exist, such as watchdog circuits, suicide/fratricide circuits, etc. In a practical implementation, when incorrect behaviour is detected, an embedded processor[5] will be triggered to begin a repair cycle.

Initially, the fault will only be known to exist somewhere within a particular chip, but gate-level fault information is required in order to allow a work around bit stream to be generated. Most FPGAs support in-circuit testing via the industry standard JTAG interface – this typically allows all flip flops to be temporarily reconnected as a single shift register, allowing the internal state of the chip to be uploaded or downloaded. Assuming that the chip is not so badly damaged that its JTAG interface no longer functions, uploading a series of test vectors and examining their results potentially allows faults to be localised with considerable accuracy. Such testing procedures are ubiquitously employed by automated test equipment during chip manufacture, so this requirement is unlikely to be prohibitive.

5.3 Experimental Results

As a proof-of-concept, a small, FPGA-like circuit was modeled (see Fig. 5.6). Eight inputs, split into two groups of four, feed the inputs of four 16-bit look-up tables, whose outputs feed a fifth 16-bit look up table. The model was configured by randomly generated 'bit streams', each 80 bits long, mapping to the configuring bits of the look up tables. Stuck-at faults were simulated by fixing the values of one or more bits at 0 or 1. For simplicity, fixed wiring was assumed. A non-clausal variant of the WALKSAT algorithm [112], NNF-WALKSAT, was used to solve the resulting SAT problems (see Appendix B.1).

In our experiments, the SAT problems were generated by a modified version of the HarPE hardware partial evaluator [134]. HarPE is a C++ template library [140] that allows circuits to be described in a high-level language, then manipulated by partial evaluation [72, 88, 87]. HarPE represents circuits internally as a directed graph data structure, which is normally emitted as gate-level Verilog. The library was extended slightly to allow output to be emitted in CNF form suitable for contemporary clausal SAT solvers, or alternatively as NNF represented either as a tree or as a directed graph for use with our own NNF-WALKSAT non-clausal SAT solver (see also Appendix B). Note that as HarPE exists purely as a simple C++ header file, it has a far smaller footprint than any typical commercial hardware description language, which lends itself well to embedded applications.

[5]This could either be an on-chip CPU or an external, possibly radiation hardened, general purpose processor.

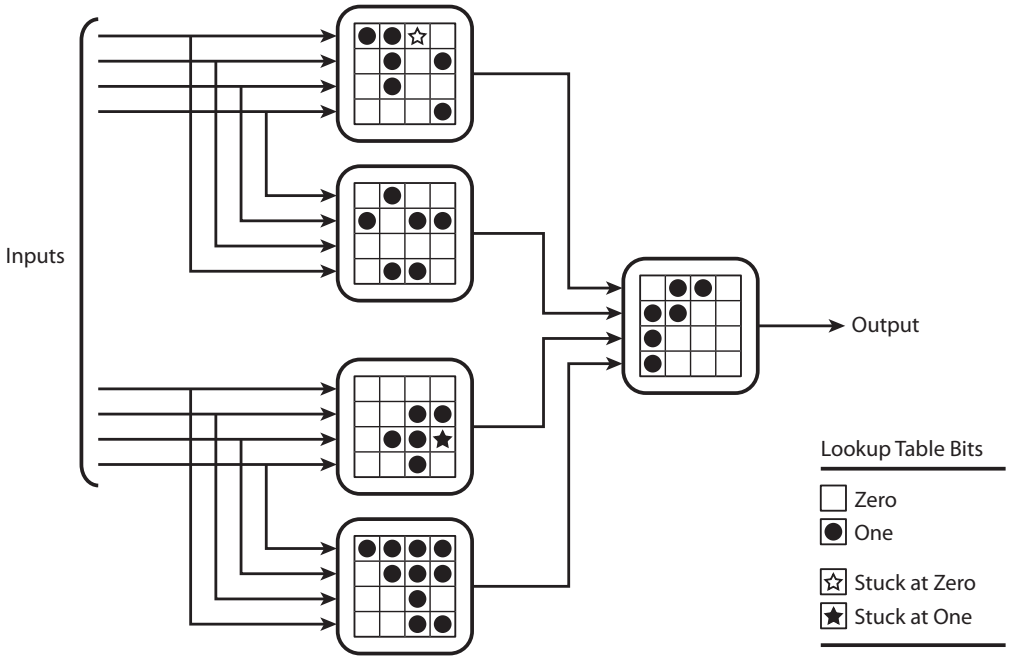


Figure 5.6: Example test circuit model

Test runs were repeated with between 1 and 6 simulated faults. Run times (C++, `gcc -O3`, running on a 1.6GHz Pentium III) and success rate are shown in Fig. 5.7, where 'success' was defined empirically as the SAT solver finding a solution within 20 minutes[6].

5.4 Related Work

The original concept of generating FPGA bit streams with SAT solvers is due to David Greaves at the Computer Laboratory, University of Cambridge [62].

The Dynamic Evolution for Fault Tolerance (ITSR/ES) project headed by Jason Lohn at the NASA Ames Research Centre is applying genetic algorithms to FPGA repair [86, 85, 83]. This approach has been shown to work, but suffers from the problem that its generated circuits are not guaranteed to be formally equivalent to the original.

Adrian Stoica's group at JPL is working on the synthesis and repair of analogue field programmable transistor array (FPTA) devices with genetic algorithms [120].

Toby Walsh's group at the School of Computer Science and Engineering, University of New South Wales, Australia are working on non-clausal SAT solvers, one of which, NOCLAUSE, is due to be released into the public domain shortly [126].

There is a huge amount of literature on the subject of SAT, particularly with regard to resolution of Boolean expressions in CNF form. The web site `http://www.satlive.org/` is a widely-used and very useful resource for information about SAT/QBF solvers.

[6]Note that no attempt was made to verify whether the generated problems were actually soluble – this corresponds well to reality, in that some damage sites in critical positions may not allow any possible work-around configuration to be determined.

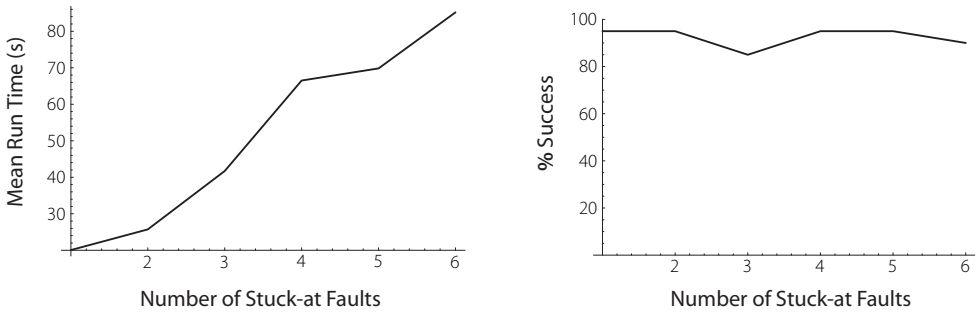


Figure 5.7: Test results

Chapter 6

Reconfigurable Manifolds

Work described in this chapter was carried out in conjunction with the US Air Force Office of Scientific Research, Space Vehicles Directorate, and was previously published in [135].

6.1 Introduction

Spacecraft design is, without doubt, one of the most challenging areas of modern engineering. In order to be viable, spacecraft must mass relatively little, whilst being capable of surviving the considerable G-forces and vibration of launch. In space, they must withstand extreme temperatures, hard vacuum and high levels of radiation, for several years without maintenance.

Conventionally, spacecraft wiring harnesses are built with architectures that are fixed at the time of manufacture. They must therefore be designed to endure the lifetime of the mission with a very high probability, though the conventionally necessary redundant duplication of signals has significant implications for mass. Given that launch costs are typically in excess of $30,000 per kg, reducing the mass of a spacecraft's wiring harness, without compromising reliability, is highly desirable. As a motivating example, the network cabling in the International Space Station (ISS) is known to mass more than 10 metric tonnes.

Recent advances in MEMS-based switching [90] have made it possible to consider the construction of *reconfigurable manifolds* – essentially, wiring harnesses that behave like macroscopic FPGA routing networks. Redundant wiring can be shared between many signals, thereby significantly reducing the total amount of cable required. Reconfigurability has a significant further benefit, in that it also allows adaptation to mission requirements that change over time, whilst also significantly reducing design time.

In a recent initiative, the US Air Force has been moving toward a *responsive space* paradigm which aims to reduce the time from design concept to launch (currently several years) to less than one week [53]. Such a target is unlikely to be achievable with existing bespoke one-off design techniques; a parts-bin driven, plug-and-play approach to satellite construction will become essential. It must be possible to choose a satellite chassis of a size appropriate to the task in terms of accommodating sufficient manoev-

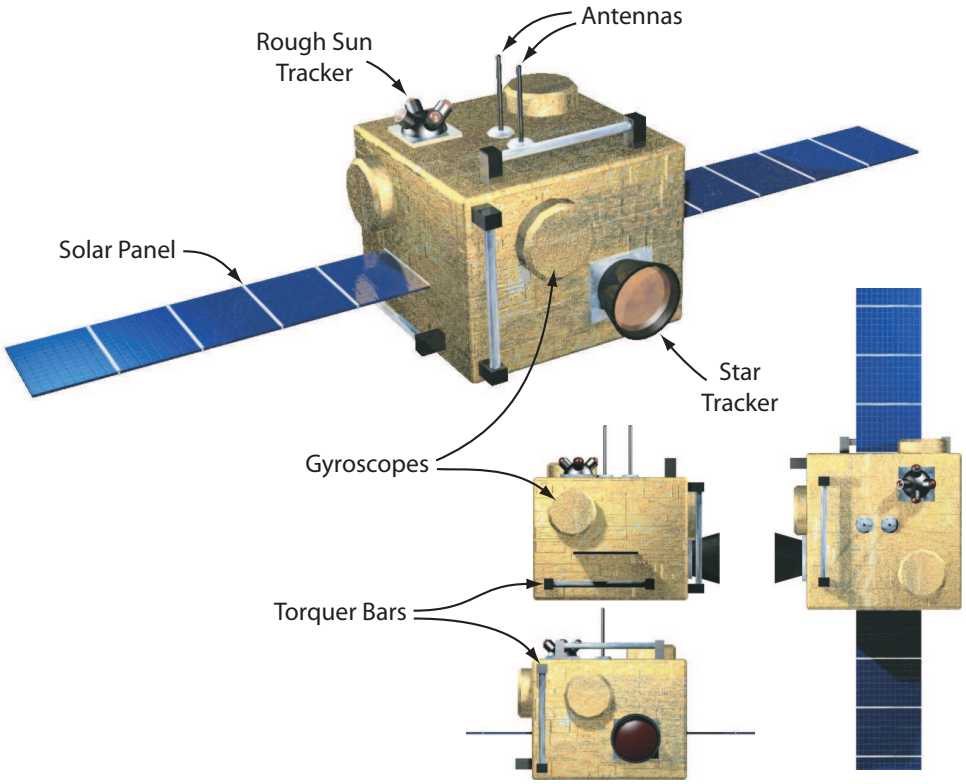


Figure 6.1: A typical near-earth small satellite configuration

ering propellant as well as the necessary instrumentation payload, then bolt everything together and have the resulting satellite 'just work.'

We present an approach that allows such a reconfigurable manifold to be automatically self-configured, then dynamically tested in-situ, such that signals are automatically rerouted around non-functioning wires and switches as soon as faults are detected. Make-before-break switching is used in order to allow wires to that are currently in use to be rerouted transparently from the point of view of subsystems that are interconnected by the manifold, whilst also making it possible to achieve near-100% testability.

6.1.1 Physical satellite wiring architectures

Conventionally, satellites are constructed with fixed wiring architectures. Reliability must therefore be engineered-in through modular redundancy – duplication or triplication (or more) of signal paths is common, which carries with it an attendant mass penalty.

Typically, two kinds of wiring architecture are common:

Card frame with passive backplane Fig. 6.2 shows a typical passive backplane with multiple subsystems, each slotting in to a rack on separate cards[1].

Motherboard/daughterboard Another common approach is shown in Fig. 6.3, where a single motherboard has a number of daughter boards attached to it on standoffs.

[1]Note that the image is representational – actual satellite hardware differs in detail

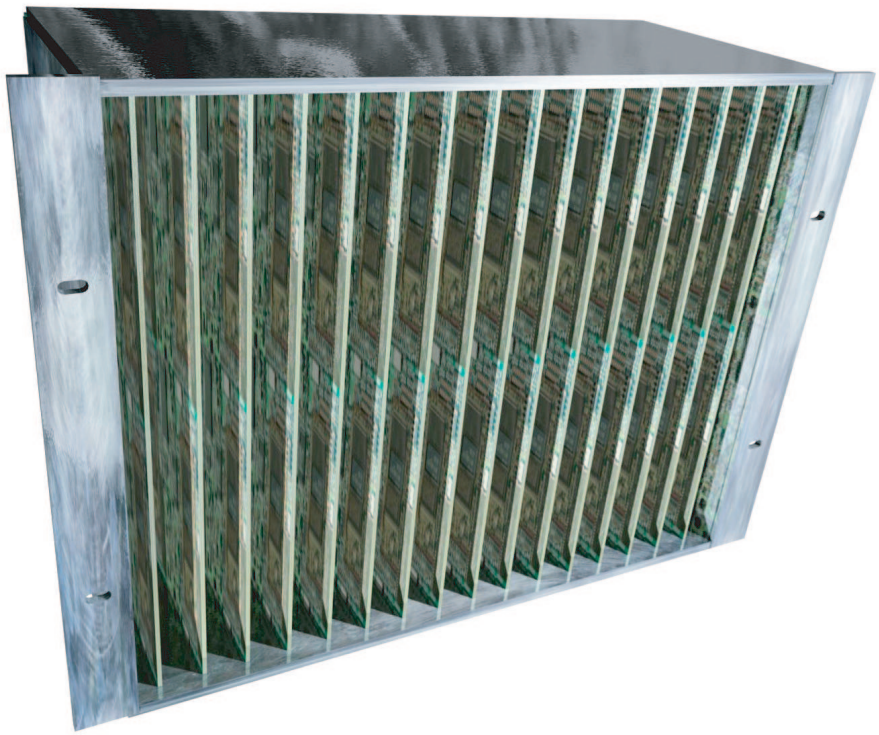


Figure 6.2: Card frame with backplane

Normally (though not visible in the diagram) these daughter boards plug directly into connectors on the motherboard, again avoiding the need for cables.

Wiring harnesses, in the sense that they exist in cars and aircraft as bundles of physical cables, tend to be avoided where possible because of their greater mass and poorer reliability.

Typically, card frames have passive backplanes, which do not normally contain active electronics beyond perhaps some simple power regulation or line termination. Motherboard approaches more commonly include active electronics on the main board itself, though this is not a prerequisite.

6.1.2 Logical satellite wiring architectures

At a logical, block diagram level, fixed architecture satellite wiring harnesses typically follow the structure shown in Fig. 6.4. All of the main subsystems are attached to a motherboard or backplane that provides most of the necessary interconnection infrastructure, with external devices plugging directly into the relevant subsystems. All required redundancy must be in place from the outset. Typically, satellites are one-off designs, so any design changes before launch require physical modifications – of course, such changes *after* launch are typically impossible. As a further consequence of this approach, subsystem re-use is relatively uncommon, requiring considerable effort in terms of design, validation and verification, of the order of several years from concept to launch.

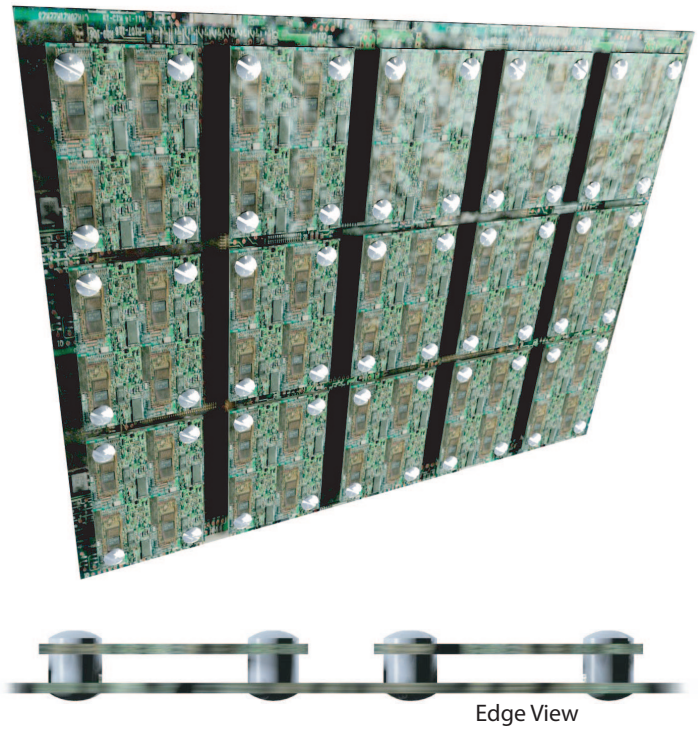


Figure 6.3: Motherboard with attached daughter boards

6.2 Reconfigurable manifolds

The responsive space paradigm [53] implies the requirement to move away from fixed architectures and their consequential design and validation costs toward an autonomous, self-organising approach. In essence, a *reconfigurable manifold* is a self-organising, self-testing, self-repairing replacement for a fixed architecture wiring harness. Ideally, at a system level, a spacecraft adopting this approach should have an architecture similar to that shown in Fig. 6.5.

Ideally, all wiring should be routed by the manifold rather than connected directly to subsystems. From a the point of view of rapid construction, this is ideal – a subsystem such as a gyroscope, star tracker, sun tracker or antenna could be bolted to the spacecraft chassis anywhere that is physically convenient, with all of the necessary wiring being 'discovered' and automatically routed after power-up.

6.2.1 Signal types

Spacecraft wiring harnesses (reconfigurable or otherwise) must be able to carry a wide variety of signals, varying in terms of power, voltage and bandwidth, with similarly variable electrical considerations in terms of impedance, end-to-end resistance, etc. Typical signal types found in satellites, along with example applications are listed as follows:

Power Normally a single $+28$V DC unregulated supply rail powers the entire spacecraft, with local step-down regulators providing lower voltage high quality supply rails to

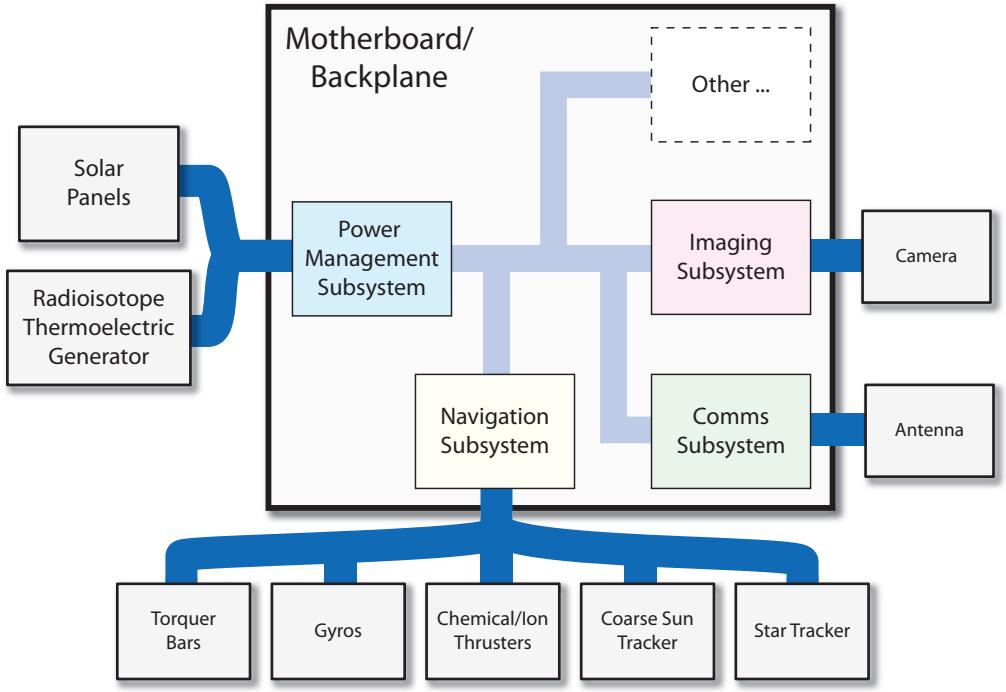


Figure 6.4: Typical non-reconfigurable satellite wiring architecture

each subsystem. Where higher voltages are necessary, e.g. to drive cryocoolers for low background noise imaging sensors, this is normally achieved with local step-up switching DC-DC converters.

Heavy current analogue High current feeds to torquer bars, motor drives, solenoid power, explosive bolts, etc.

Low current, low speed analogue Analogue sensor feeds, thermocouples, rough sun tracker photocells, etc.

Low current, high speed analogue Higher speed sensor wiring, video feeds from cameras and star trackers, etc.

Low speed digital Simple on/off telemetry sensors, e.g. mechanical limit switches.

High speed digital Digital communications between subsystems.

Low power microwave Radio receiver antenna feeds, low power radio transmitter antenna feeds.

High power microwave High power antenna feeds, ion thruster power cabling, etc.

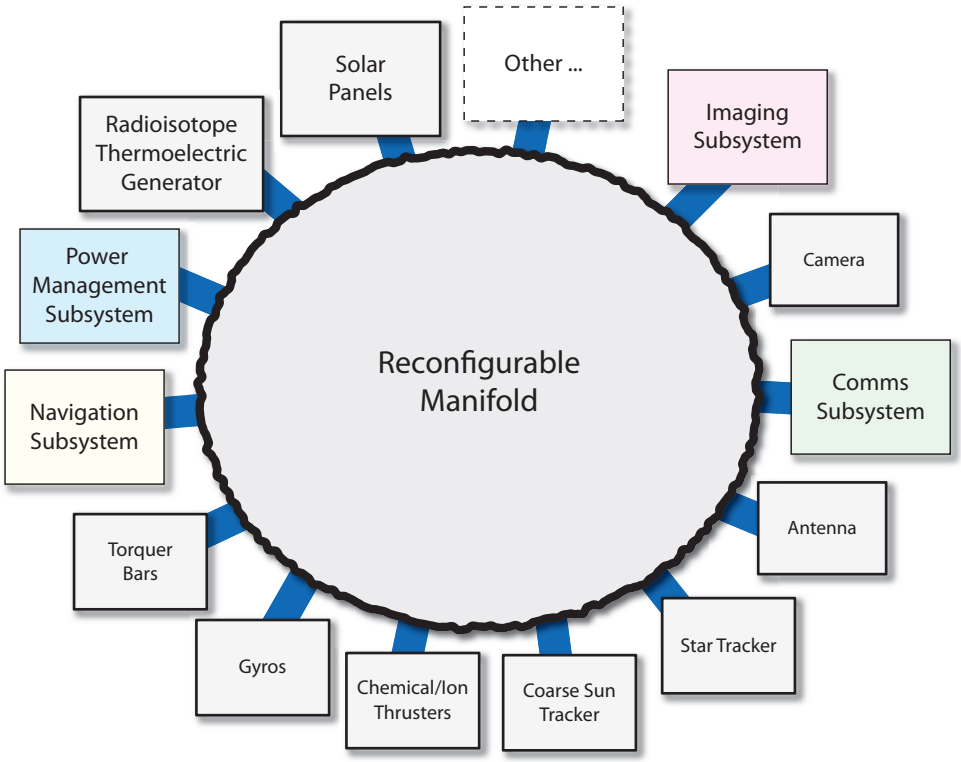


Figure 6.5: Reconfigurable manifold architecture

Optical High speed network connectivity, lower speed sensor applications that require a significant degree of electrical isolation[2].

No single switching architecture, at the time of writing, can accommodate more than a few of the above signal types.

6.2.2 Constructing practical reconfigurable manifolds

A practical reconfigurable manifold must encompass most, if not all, signal types in order to be effective. Since no single switch fabric is suitable, it makes sense to split the manifold into separate sub-manifolds, each of handling a different signal type, as shown in Fig. 6.6.

Some cross-connectivity between the sub-manifolds makes sense, since, for example, several MEMS relays could potentially be connected in parallel in order to switch heavier current, or DC-biased analogue routing with sufficient bandwidth could, in an emergency on orbit, be used to carry digital data.

Fig. 6.7 shows a reconfigurable manifold implemented as a replacement for a passive backplane or passive motherboard. In contrast with Fig. 6.4, external systems connect to the manifold rather than direct to the subsystems themselves. Configuring such a

[2]Optical switching is beyond the scope of this work and will not be discussed further.

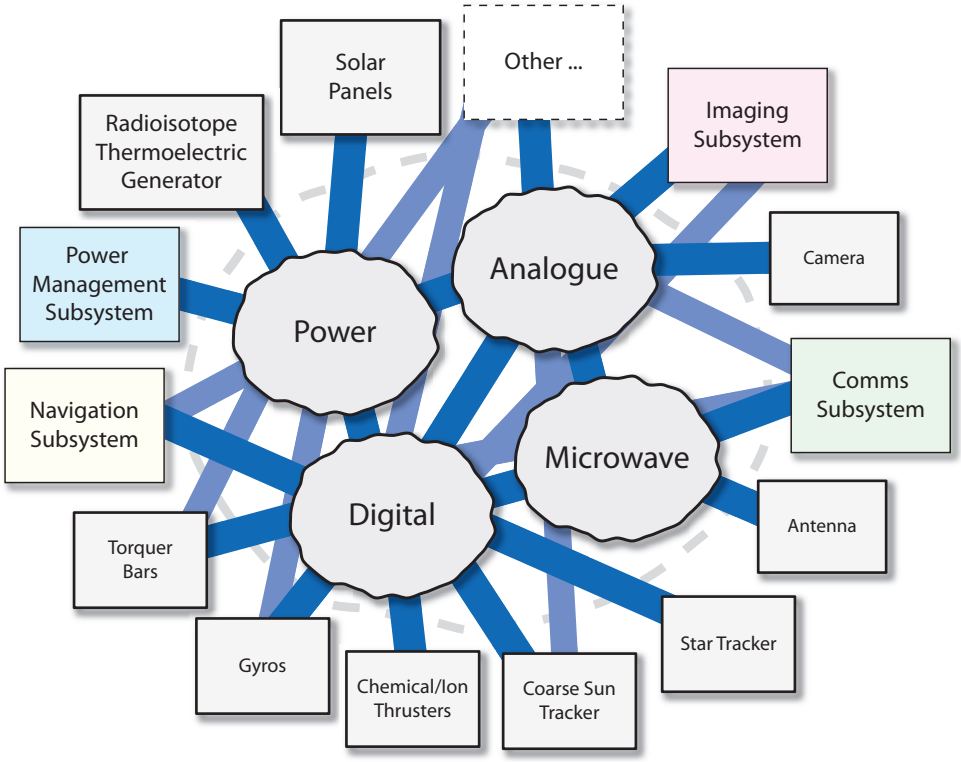


Figure 6.6: Separate routing networks for power, analogue, digital and microwave

satellite might be as simple as installing cards in a backplane or motherboard in any convenient order, then plugging external devices into the manifold. Spare slots could, given sufficient mass budget, be used to provide extra redundancy simply by plugging in extra duplicate cards; appropriate firmware could potentially handle this automatically.

An alternative architecture is shown in Fig. 6.8. Rather than a single manifold routing between devices connected to its periphery, the manifold is itself distributed between the subsystems. Interconnection between subsystems is passive, with the subsystems cooperating to establish longer distance, multi-hop routes.

The single manifold approach is perhaps best suited to small satellites, whereas the (more complex, though more flexible and scalable) distributed approach lends itself to larger spacecraft such as large satellites, manned spacecraft, space stations or indeed also to terrestrial aircraft.

6.2.3 Switching technologies

Many switching technologies exist that differ considerably in capability:

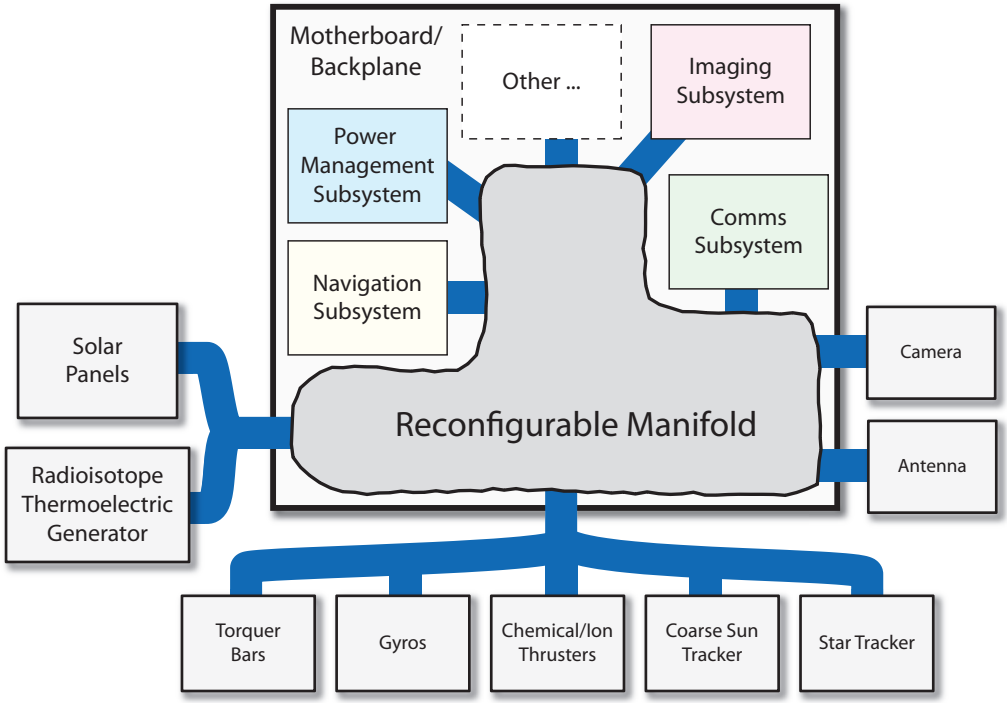


Figure 6.7: Reconfigurable manifold as a motherboard or backplane

FPGAs Field-programmable gate arrays can be used to route digital data, and are also comparatively cheap and readily available.

FPTAs Field-programmable transistor arrays [120] have some similarities to FPGAs, though they are aimed more closely at analogue applications. As with FPGAs, they are not intended from the outset as routing devices for use within a the switch fabric of a reconfigurable manifold, though it would seem feasible to apply them to the switching of low- to medium-speed analogue signals.

Digital Crossbar Switch ASICs A number of commercial, off-the-shelf (COTS) digital crossbar switch chips are available, though this application appears to be becoming dominated by FPGAs as a consequence of the larger FPGA manufacturers getting more directly involved by releasing support for using their devices in this way [11].

Analogue Crossbar Switch ASICs Though not so widely supported as digital crossbar switch devices, analogue crossbar switches are available, mostly aimed at switching analogue video signals [10].

MEMS switches Micron-scale electromechanical switches have been demonstrated to be an effective candidate technology [90]. Though physically far larger than CMOS transistor-based switches, MEMS switches are nevertheless orders of magnitude

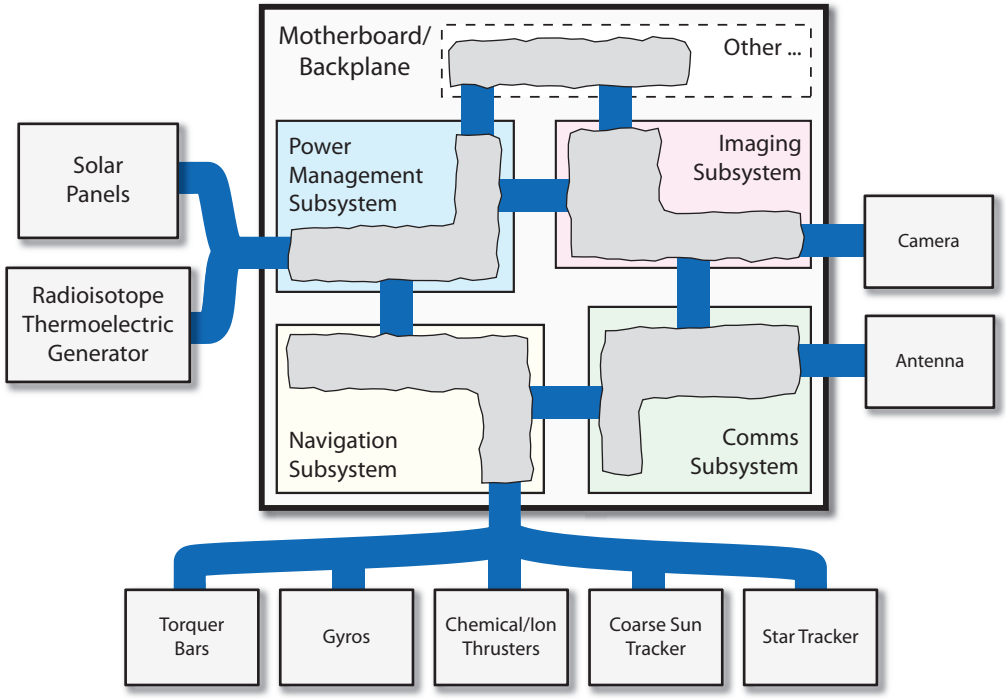


Figure 6.8: Reconfigurable manifold distributed across subsystems

smaller and lighter than full-size mechanical relays, and have excellent electrical characteristics that renders them capable of being applied to almost any low-current switching application, including microwave.

Electromechanical Relays Somewhat old-fashioned, relays are nevertheless capable of switching very heavy currents. They are sufficiently massive, however, that it is difficult to imagine them being used in large numbers in a spacecraft application.

Discrete MOSFET/IGBT Switching Large power transistors, both MOS and bipolar, are commonly used to switch heavy current and moderately high voltage (up to a few hundred volts and/or hundreds of amps) signals, particularly in motor drive applications. They exhibit high reliability and relatively good radiation hardness characteristics due to their very large (in comparison with ASICs) geometries, though their gate drive circuitry can be tricky to engineer. Though physically bulky, they nevertheless remain a useful possibility for constructing heavy current and/or power switching networks.

Table 6.1 shows compatibility between switch technologies and signal types. The notation '?,' denoting 'possibly compatible,' indicates that, under normal operational circumstances, an automated routing algorithm would not attempt to make a connection of this type, though in an emergency such connections might be made in the absence of

more appropriate infrastructure. Normally, signals would be prioritised, so critical signals would almost certainly be routed, but less important connections may be degraded or even omitted. For example, a non-critical redundant temperature sensor might be disconnected in favour of keeping an instrument package in operation.

	FPGA	FPTA	Digital X-bar	Analogue X-bar	MEMS	Relays	MOSFET/IGBT
Power	×	×	×	×	?	√	√
Heavy current analogue	×	×	×	×	?	√	√
Low current, low speed analogue	×	√	×	√	√	√	?
Low current, high speed analogue	×	√	×	√	√	?	?
Low speed digital	√	√	√	√	√	√	√
High speed digital	√	?	√	?	√	?	×
Low power microwave	×	×	×	×	√	×	×
High power microwave	×	×	×	×	?	×	×

× – Not compatible ? – Possibly compatible √ – Compatible

Table 6.1: Compatibility between switch technologies and signal types

6.2.4 Routing architectures

The major alternative switching architectures that may be considered when designing a reconfigurable manifold are as follows:

Crossbar Switch An $M \times N$ grid of switches configured to provide a M-input, N-output routing network.

Permutation Network A *permutation network* performs an arbitrary permutation on N inputs, such that any possible reordering of the inputs is supported.

Ad-Hoc and Hybrid Approaches Practical considerations make it appropriate to consider the possibility of leveraging existing COTS technologies, possibly in combination, to create reconfigurable manifolds. Though the result network topology and routing algorithms may be technically inferior to a purer design, economic considerations are nevertheless important for practical designs.

Embedding into Networks of Arbitrary Topology Given a sufficiently large and complex graph, with nodes representing switches and edges representing wires, it is possible to compute a switch configuration that implements an arbitrary circuit.

Each approach is described in detail below.

6.2.4.1 Crossbar switches

Crossbar switches have a long history, having originally been introduced as a means of routing telephone calls through electromechanical telephone exchanges. Conceptually extremely simple, a crossbar switch is constructed from two sets of orthogonal wires

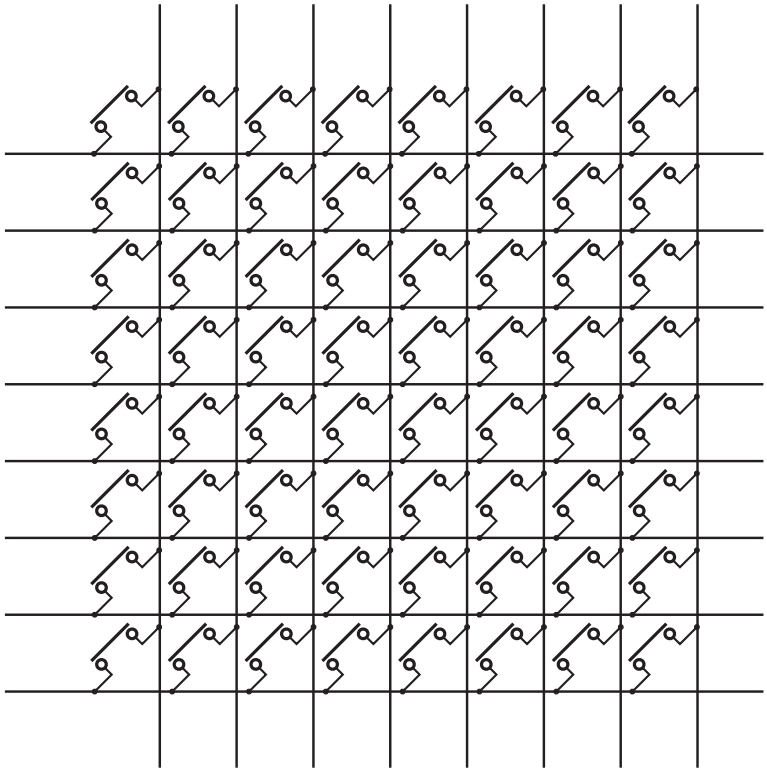


Figure 6.9: Crossbar switch

(bus bars in telecommunications nomenclature), such that each crossing can be bridged by a switch. Fig. 6.9 depicts the circuit of a small 8×8 crossbar switch.

To route a particular input to a given output, all that is necessary is for the switch corresponding to that input and output to be closed. Crossbar switches are somewhat inefficient in terms of hardware requirements, and also in terms of providing more routing capability than is strictly necessary in many cases – it is possible, for example, to route a single input to any number of outputs, or to common inputs together. Achieving reliability is relatively straightforward, however – replacing each non-redundant switch (Fig. 6.10) with a partially- or fully-redundant alternative (Fig. 6.11 or Fig. 6.12 respectively) allows single point failures to be recovered. A fully-redundant switch configuration allows any of its four component switches to fail-open or fail-closed without affecting functionality. The partially redundant version only requires half as many switches, but is only safe against fail-closed faults – however, given one or more spare bus bars on each axis, fail-open faults can easily be patched around and are therefore still recoverable. In cost terms, building a fully-redundant $M \times N$ switch requires $4 \times M \times N$ switches, whereas the partially redundant approach requires $2 \times (M+1) \times (N+1)$ switches, though clearly the larger circuit is more fault-tolerant. Though both circuits can accommodate at least one fail-closed fault per cross point, the smaller circuit is limited to only one fail-open fault across the entire switch for each additional pair of redundant bus bars.

A related architecture, once commonly used in circuit-switched telephone exchanges prior to the widespread introduction of digital technology, was the *Clos network* [38], which was normally constructed from three layers of smaller crossbar switches. This ap-

Figure 6.10: Non-redundant switch

Figure 6.11: Partially redundant switch configuration

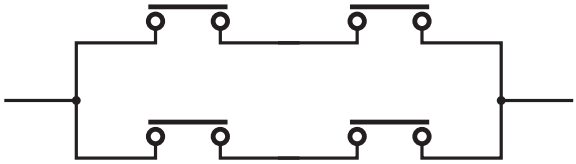


Figure 6.12: Fully-redundant switch configuration

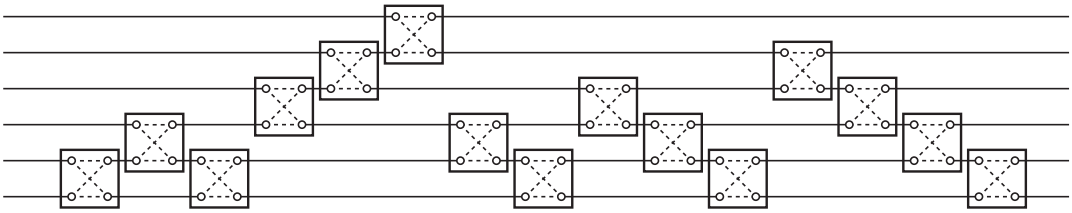


Figure 6.13: 6-way permutation network

proach may or may not support all possible permutations depending upon the details of its construction – non-blocking Clos networks may provide an efficient means of building large manifolds from small crossbar ASICs, though from-scratch designs based on permutation networks (see below) are still likely to require fewer switches.

6.2.4.2 Permutation networks

Permutation networks are an alternative approach to routing that, in many cases, require substantially fewer switches for a given number of inputs – rather than $O(N^2)$, they tend toward $O(N \log N)$, which can be a very significant advantage when the number of inputs is large. Fig. 6.13 illustrates the concept with a 6-way permutation network. Its 15 swap nodes, each of which typically constructed from four switches (see Fig. 6.14), can each be in either of two states: pass the inputs left to right unchanged, or swap them. For 6 inputs, a crossbar switch is likely to be cheaper, in that it is likely to require only 36 switches, in comparison with 60 for the permutation network shown in Fig. 6.13. However, for 1000 inputs, assuming $N \log_2 N$, approximately $40,000$ switches are required, whereas a 1000×1000 crossbar switch would require 1 million switches.

Designing a permutation network can be somewhat baroque, though a useful relationship with *sorting networks* can be exploited. A sorting network [13, 16, 35, 41, 81] is architecturally similar to a permutation network, with the exception that the swap/don't

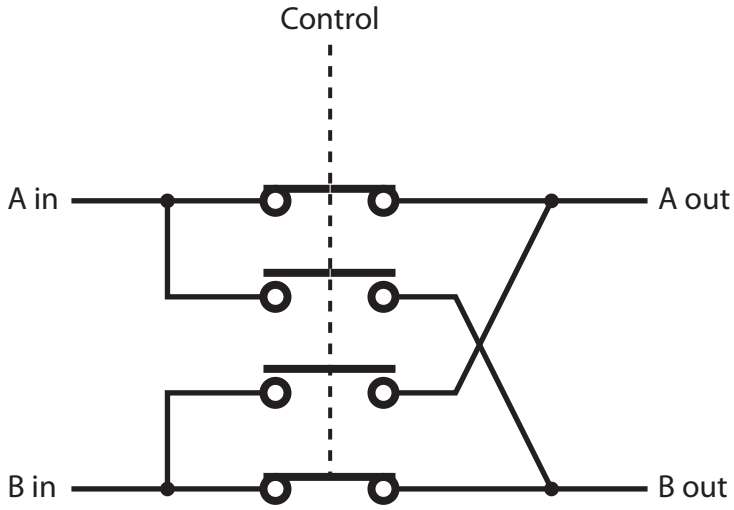


Figure 6.14: Swap node circuit

swap decision at each node is made by comparing its inputs, such that its outputs are constrained always to respect a given ordering relation. Many well-known sort algorithms, e.g. merge sort, bubble sort, transposition sort, bitonic sort or shell sort, can be constructed as sort networks. Since a sort may also be seen as just a particular kind of permutation, sort networks – by definition – must be capable of performing permutations. Furthermore, since the data to be sorted might initially be in any order, a sort network must be capable of supporting *all* possible permutations – therefore, if a sort algorithm can be adapted to create a sort network of arbitrary dimension, it follows that an equivalently structured permutation network would also be capable of any possible permutation. Usefully, the underlying sort algorithm can be leveraged to efficiently generate switch configurations, as follows:

1. Let $\langle W, < \rangle$ be a totally ordered set such that $|W|$ is the number of wires in the switch network, and each $w \in W$ represents exactly one input and one output.

2. Let the total bijection $P : W \to W$ represent the desired permutation to be implemented by the switch network.

3. Sort P with the underlying sort network, such that for each $(a, b) \in P$, a represents the input, and b represents the output. This can be achieved trivially by feeding tuples into the network ordered on a, then having the network sort these tuples ordered on b.

4. Note whether each swap node passed its data through unchanged, or whether it performed a swap. This gives the switch configuration for an isomorphic permutation network that performs an equivalent permutation.

Since suitable sort algorithms exist that have $O(N \log N)$ time complexity, computing a switch plan is therefore also an $O(N \log N)$ operation.

Permutation networks are nevertheless not always a better solution than crossbar switches, particularly when constructed as ASICs – their complex wiring reduces the

effective advantage of their reduced switch count, particularly when considering that regular grids (crossbar switches being a particularly ideal example) are cheap and easy to lay out in comparison with the more spaghetti-like nature of large permutation networks, though Claessen *et al* [35] have shown promising results by adopting a *layout combinator* approach. Limitations on chip packaging limit the number of wires that can be physically connected to a single chip, which places hard limits on the impact of the $O(N^2)$ complexity problem with crossbar switches. However, when switches are large and/or expensive, as is the case with MEMS relays or any discrete component approach (e.g. full-size relays, MOSFETs, IGBTs), the reduction in component count could prove important.

6.2.4.3 Shuffle networks

Shuffle networks are essentially degenerate, incomplete permutation networks that do not support all possible permutations. Shuffle networks implement a *perfect shuffle,* [121], which typically allow any input to be routed to any output through $\log_2 N$ layers of switches. They are perhaps best known in the parallel computing world, where they are commonly used as high speed inter-processor interconnect architectures. Omega networks [84], a commonly used shuffle network architecture, typically require some kind of blocking or queueing hardware at each swap node so that collisions can be arbitrated.

In general, the incompleteness inherent in a single shuffle network is not tolerable for our application – it was, however, conjectured by Beneš in 1975 [17] and again more recently by Mary Sheeran [116] that exactly two shuffle networks in series can implement any possible permutation. The conjecture was recently proven by Çam [32], which means that this approach may lead to a means of designing compact, geometrically regular permutation networks that preserve $O(N \log N)$ complexity.

6.2.4.4 Ad-hoc COTS approaches

In some cases, COTS devices may be used to implement routing fabric. FPGAs, in particular, are ubiquitous, low cost and can be used (with appropriate considerations) in high radiation environments. There are a number of potential approaches:

1. Implement a general purpose crosspoint switch or permutation network as a HDL model, then synthesise it.

2. Generate HDL that routes the FPGA's inputs and outputs according to the desired switching plan, then synthesise the design.

The first option clearly limits the size of switch that can be implemented in a particular FPGA, though is inherently general purpose and can be reconfigured very rapidly. The second option is probably infeasible for embedded use at the time of writing due to the requirement for a complete tool chain in order to perform reconfiguration, though this situation may improve as technology supporting dynamic reconfiguration matures. In particular, the Xilinx jBits library [63, 117] allows FPGA configuration bitstreams to be generated on-the-fly from Java code, though it is currently unclear whether it can be feasibly implemented on the kinds of low-performance radiation-hard CPUs that are typically used for spacecraft applications.

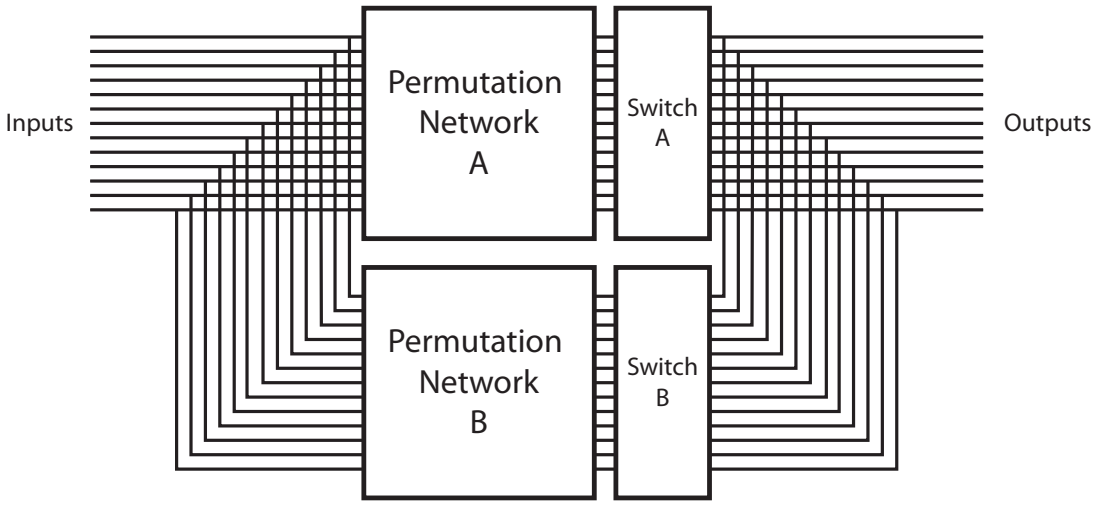


Figure 6.15: Work-around for make-before-break using permutation networks

6.2.4.5 Embedding into networks of arbitrary topology

A reconfigurable manifold of arbitrary topology may be represented by a graph whose nodes represent switches and whose edges represent wires. Embedding a desired circuit into such a network is essentially equivalent to computing a switch configuration. For the general case, this is a difficult computational problem that seems almost certainly to be in NP, with complexity rising exponentially with the number of switches in the network. Though this approach ultimately encompasses all others, in that both crossbar switches and permutation networks may be seen as special cases, the difficulty of computing switching plans makes it unlikely that this approach could be feasible in practice.

6.2.5 Make-before-break switching

At the device level, make-before-break switching requires the capability to establish a new connection, in parallel, before an old connection is disconnected. Where a reconfigurable manifold is routing signals that should not be temporarily interrupted, make-before-break switching allows a connection to be moved to an alternative route transparently to the signal's endpoints. In a reconfigurable manifold that does not alter its wiring plan after it has been initially configured, support for make-before-break switching is unnecessary – however, such a capability is essential in order to support continuous automated testing and fault recovery (see Section 6.4).

Power, heavy current analogue, low-speed digital and low-speed analogue signals are all well suited to make-before-break switching, in that they are not particularly sensitive to minor changes in end-to-end resistance or discontinuities in impedance. However, high-speed digital, high-speed analogue, or (particularly) microwave signals need more careful consideration – in such cases, it may be necessary for the subsystems concerned to become involved in the routing process, at least from the point of view of being able to request that the manifold should not re-route particular signals during critical periods.

Crossbar switches support make-before-break switching by default: it is just neces-

sary to turn on the switch for the new connection, waiting long enough (if necessary) for the switch to close fully and stop bouncing, then turn off the switch for the old connection. Implementing make-before-break switching in a permutation network is not feasible in general – making a change to a single route often requires several signals to be rerouted at once. A work-around solution is shown in Fig 6.15, where a pair of identical permutation networks is connected in parallel and are switched as follows:

1. Initially, Switch A is off and switch B is on.

   Permutation Network A is carrying all signals and Permutation Network B is not connected.

2. A new switch configuration is computed, and used to initialise Permutation Network B

3. Turn on Switch B.

4. Turn off Switch A.

   At this point, Permutation network B is now carrying all signals, and Permutation Network A is not connected.

For the next cycle of reconfiguration, the procedure continues with A and B swapped. This approach avoids switching glitches during reconfiguration of the permutation networks because whenever reconfiguration occurs, the permutation network in question is disconnected – actual switching of live signals only occurs during steps 3 and 4, which can trivially be arranged to be guaranteed clean.

Though this work-around implies slightly more than a doubling of hardware requirements, it nevertheless maintains $O(N \log N)$ complexity. Adding a third, redundant, permutation network as a hot spare allows modular redundancy to be implemented with a 3 times multiplier on hardware requirements, which compares well with the 4 times multiplier that would result from replacing each component switch with a redundant series-parallel switch network (see Fig. 6.12).

6.2.6 Grounding

Grounding of electronic systems within satellites is broadly similar to the grounding of Earth-based electronics; as-such, the same techniques and best practice applies in both cases. In satellites, grounding is particularly important because of the *charging effect,* whereby charged particles impacting the spacecraft impart a (potentially large) electric charge – careful grounding all conductive parts typically reduces or eliminates any consequential problems.

It is normal practice for a spacecraft to implement a ground network with a star topology – a single central grounding point is connected radially to the grounds on all subsystems. Cycles in the ground network are avoided, because they can form unwanted single-turn secondaries that may pick up hum or other unwanted noise from any heavy current subsystems in the vicinity.

Normally, grounds should not need to be switched by a reconfigurable manifold – a conventional, fixed, star ground topology should be sufficient for nearly all cases. Signals

that are routed along shielded paths may require switchable ground connections[3] at one or both ends in order to avoid ground loops, though careful consideration of possible ground routing requirements may avoid this.

6.3 Self-organisation

In some circumstances, it is undesirable or even impossible to precalculate routing for a reconfigurable manifold. The responsive space paradigm requires that disparate subsystems should be able to be plugged together in any convenient manner, at which point they should self-organise and work together without human intervention. Achieving concept-to-launch times of the order of one week does not leave much time for anything other than physical assembly of the spacecraft, so the electronic subsystems must, of absolute necessity, not require a lengthy design process.

Self-organisation, at a fundamental level, requires subsystems to be able to discover each other, negotiate and configure any necessary wiring, and also to cooperate in maintaining the long-term reliability of the connectivity. These issues are discussed in detail in the remainder of this section.

6.3.1 'Space Velcro'

Some technologies absolutely require self-organisation in order to function at all. Fig. 6.16 is an electron micrograph of Joshi *et al*'s *Microcilia* concept [75, 123, 25]. MEMS technology is used to construct micron scale, articulated 'cilia' that are capable of manipulating small objects and of allowing the docking of small microsatellites. Assuming that electrical connections between the mated surfaces can be achieved, a self-organising, reconfigurable manifold based satellite could automatically configure any necessary connections during docking, then automatically recover the routing resources once the microsatellite has undocked.

Brei *et al* have investigated a passive interconnect architecture known as *Active Velcro* [37, 27, 26]. Fig. 6.17 illustrates the concept[4]. Mating, Velcro-like surfaces also contain a (possibly large) number of connectors, a proportion of which happen to make valid connections. Discovering these connections, then routing them via a reconfigurable manifold, potentially allows extremely straightforward ad-hoc construction. In manned spaceflight applications, an astronaut could connect or disconnect a piece of equipment simply by sticking or unsticking it to a Velcro-like pad[5]. In satellite applications, assuming that launch G force and vibration constraints are met, the same approach could allow extremely rapid construction and deployment.

[3] Also known as *ground lifts* to electrical engineers.

[4] Note that this is the author's rendering, and is intended to be representational of the connectivity approach rather than an accurate physical description.

[5] It has long been standard practice to use Velcro to prevent small objects from floating around the cabins of manned spacecraft.

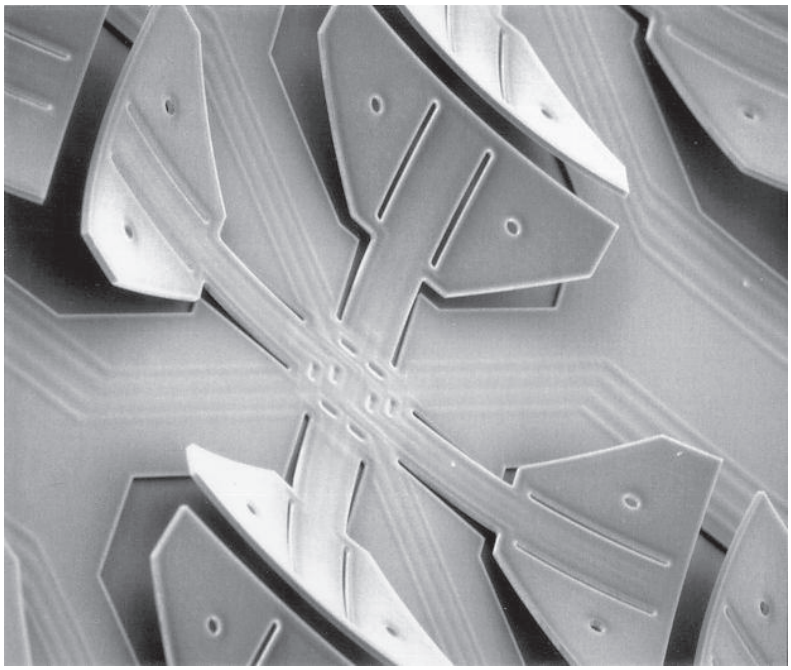


Photo: John Suh, University of Washington

Figure 6.16: Microcilia cell

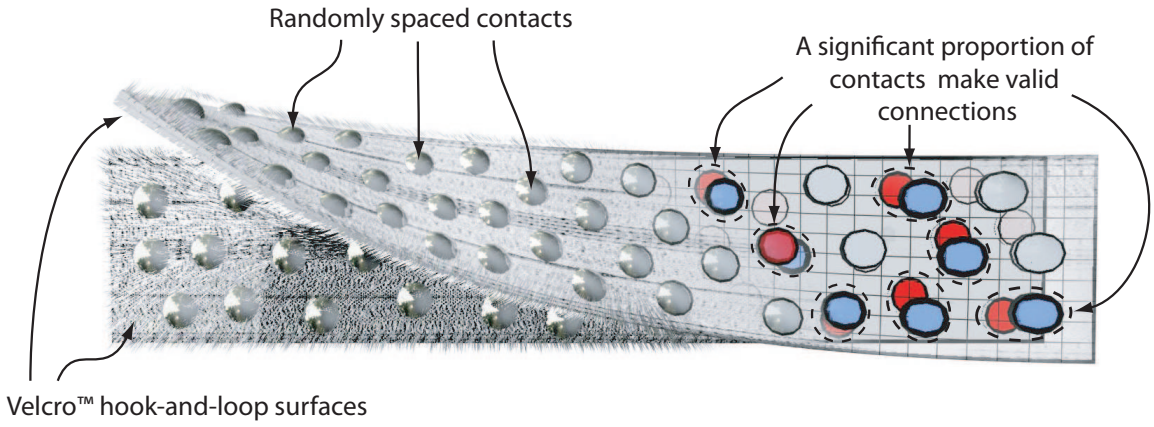


Figure 6.17: Active Velcro

6.3.2 Local routing

In a very small satellite, or within a single subsystem of a more complex satellite, routing may be exclusively *local*, i.e. switched only by a single level of switch networks. All connections in such a case would occur only to the edge of a single manifold, or cluster of sub-manifolds configured to act logically as a single manifold, with the consequence that the routing of all signals is equivalent only to routing across the manifold itself. Computing switch assignments for such an architecture is relatively trivial, with complexity of the order of $O(N^2)$ for a crossbar architecture or $O(N \log N)$ for a permutation network.

6.3.3 System level routing

Purely local routing requires a strict star architecture, with the manifold at the hub. This physical geometry does not suit all applications – in many cases, particularly in larger spacecraft, it is likely to be more appropriate to distribute the switching around the craft. Though it is theoretically possible to construct a large crossbar switch by ganging together smaller switches, this would be an expensive approach since the amount of inter-switch cabling would rise in proportion to the square of the number of switches. A more sensible and practical approach would be to construct a manifold-of-manifolds with an architecture resembling that of a circuit-switched telephone network – a number of manifolds handle primarily local connections internally, whilst handing off longer-distance connections via multicore trunk connections to other manifolds.

Computationally, the system level routing problem tends towards NP in the worst case (e.g. a manifold-of-manifolds where each manifold consists of exactly one switch and connectivity between manifolds is arbitrary is essentially the same problem that is discussed in Section 6.2.4.5), though the relatively small number of manifolds and relatively large amount of connectivity within each manifold is likely to minimise the consequences of this.

As with circuit-switched telephone networks, in general the manifold-of-manifolds approach would not support all possible permutations, which suggests that in responsive space applications, it makes sense either to adopt a local-routing-only approach, or to deliberately overspecify the amount of manifold-to-manifold interconnection resources.

6.3.4 Dynamic discovery

The *dynamic discovery* of connections is something that is becoming increasingly common in general-purpose computing. The USB standard, for example, allows devices to be discovered and configured automatically without significant human intervention. From the point of view of reconfigurable manifolds, the dynamic discovery problem is somewhat trickier, in that it is necessary to first power up any neighbouring subsystems, establish contact with them (potentially with zero prior knowledge of their wiring configuration), negotiate any required connections, then route the necessary signals. As a second requirement, it is then necessary to continuously re-test the existing connectivity in order that faults can be corrected and that subsystems coming on line or going off line can be connected and disconnected correctly.

In this section, the requirements for achieving reliable dynamic discovery, continuous testing and fault recovery are discussed.

6.3.4.1 The 'chicken-and-egg' initial power-up problem

It is a truism that any automatic discovery algorithm can only possibly run on hardware that is itself powered up. However, if a subsystem's power connections have not yet been discovered and configured, it will not (yet) be powered up – hence there is a chicken-and-egg problem. Though no longer in common use, a well-known solution already exists. For many years, the most commonly used PC peripheral interface standards, RS232 and Centronics, both suffered from a design oversight – no power supply

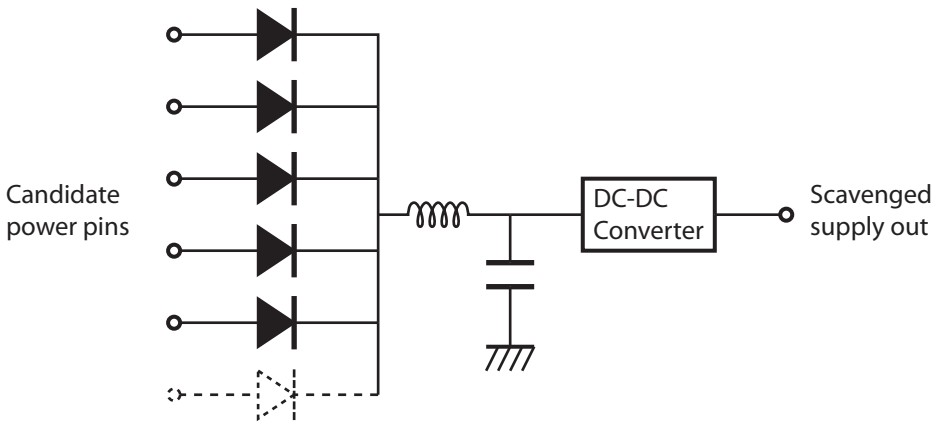


Figure 6.18: Power scavenging circuit

pins – that proved maddening for any hardware engineer attempting to design small peripherals without separate mains power supply connections. Designers nevertheless succeeded in working around the limitation by including circuits that scavenged power from the I/O pins themselves. The technique is illustrated in Fig. 6.18 – a diode network, effectively a large-scale generalisation of a full-wave rectifier circuit, synthesises power rails effectively by implementing a minimum/maximum function on the voltages that are present. The clamping, smoothing and DC-DC converter circuitry takes the potentially rather unpredictable raw output from the diode network and turns it into clean power that can be safely used to power up discovery circuitry prior to permanent routes being put in place.

Given suitable power scavenging circuits, a feasible power-up procedure for a large, manifold-of-manifolds architecture might be follows:

1. Power is applied to the first manifold through an arbitrary power pin.
2. The power scavenger circuit synthesises a suitable voltage rail for the embedded processor and discovery hardware responsible for the manifold.
3. All switches within the manifold are initialised to open circuit.
4. The power connection is detected, then connected via the manifold, thereby disabling the diode network. This step avoids the inherent voltage drop across the diode network, whilst also reducing power consumption and heat dissipation slightly.
5. The manifold starts to listen for connection requests from other subsystems (see Section 6.3.4.3).
6. Power is temporarily routed to arbitrary pins on neighbouring subsystems that currently do not appear to be active, giving them the chance to power up and begin their own discovery process. They may request that power is supplied through a different pin, if necessary, or request that the existing pin should remain connected indefinitely[6].

[6]Though it may be subject to change as part of the self-test algorithm.

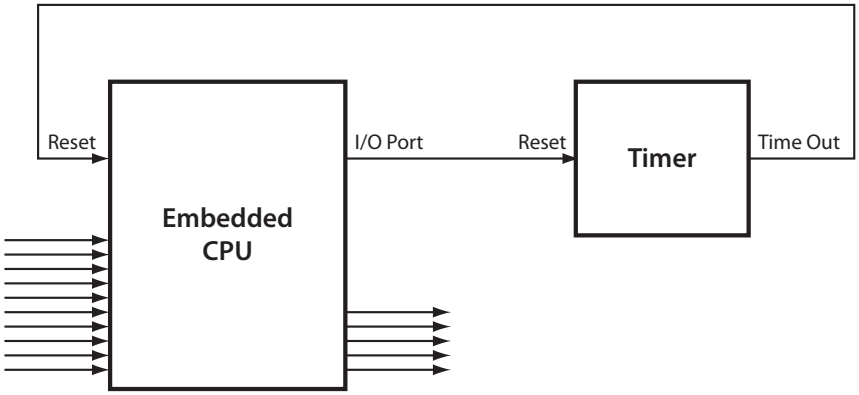


Figure 6.19: Typical watchdog circuit

Eventually, all subsystems will be powered up, with the discovery process continuing to bring online all other necessary connections.

6.3.4.2 Watchdogs

It is standard practice for embedded processors in high reliability, mission critical and safety critical systems to be equipped with *watchdog circuits,* see Fig. 6.19.

A watchdog circuit is essentially a simple timer that is periodically reset by the host processor in such a way that, if the host processor happens to fail to reset it within a predetermined interval, the watchdog timer performs a hard reset on the host processor. Generally, this is integrated into a critical loop within the embedded software, so that if the program crashes the timer will fail to be reset, causing an automatic restart of the processor.

At a simplistic level, there is no reason why such a restart should cause problems for a manifold-of-manifolds architecture, though careful attention must be given to the following issues:

1. In the event of a watchdog reset, all external connections must be torn down, just in case the crash was itself caused by a faulty connection or, for example, by a single-event effect affecting the manifold itself.

2. Any negotiation protocol must be able to cope, e.g. by implementing timeouts, with connections going down without any corresponding explicit notification.

6.3.4.3 Discovery probe circuits

Connection discovery depends upon an ability to safely probe connections to find out what neighbouring subsystem they are connected to. The outline circuit shown in Fig. 6.20 shows how a suitable 'discovery probe' might be implemented. The circuit shows a UART (bidirectional serial interface) connected to a host processor, whose serial I/O ports (marked TxD and RxD) assume good quality, logic-level signals. On the transmit

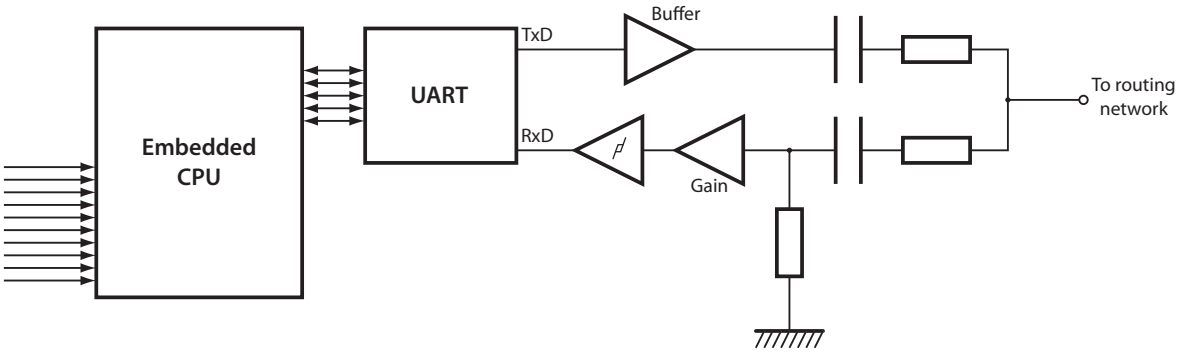


Figure 6.20: A possible discovery probe circuit

side, the signal is first buffered in order to protect the UART[7], then high pass filtered to achieve AC coupling and connected to the probe output via a resistor, whose value should be carefully selected in order to limit worst case current in the event of an accidental connection to a power or high current analogue signal to a level that cannot cause damage. On the receive side, a similar current limiting resistor and high-pass network protects the active components from direct connection to otherwise potentially damaging signals. A DC-coupled linear amplifier boosts the signal, then a Schmitt trigger [111] (comparator with hysteresis) squares up the signal and raises it to logic levels suitable for the RxD input of the UART. Current limiting resistors should be chosen with values that are not too overspecified, since lower values are likely to result in better noise performance and higher achievable data rates.

In essence, the probe circuit is a simplified, extremely robust variation of a shared bus CSMA/CD network interface, in the style of 10Base2 Ethernet. AC coupling and a relatively high series resistance minimises the chance of damage due to accidental connection to higher voltage signals, whilst the ability to send and receive digital data without needing to switch between transmit and receive modes makes implementing higher level protocols relatively straightforward.

Sending NRZ (non-return to zero) serial data across AC coupled connections requires careful design of the low-level line protocol. Sending, for example, a long string of ones will cause the voltage to decay back to a centre value over a period of time that is determined by the time constant of the high-pass filter. Similarly, a data packet that consists predominantly of ones (or zeros) will tend to shift away from the most common value, causing an unwanted DC bias and consequential reduction in noise margins. Typically this is addressed by arranging for the data encoding to implicitly retain an equal number of 0s and 1s – a trivial, though inefficient, approach is to spread an 8 bit byte across 16 bits, where each input bit corresponds to an inverted and a non-inverted copy in the output word. More efficient encodings exist that spread 2 bytes across 24 bits.

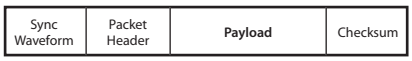


Figure 6.21: Typical packet format

6.3.4.4 Line protocol

The main function of a suitable line protocol is to allow the discovery of connections, then to allow routing negotiation for signals. Probe circuits will typically alternate between sending packets that announce the identity of the relevant wire and listening for incoming packets that identify the other side of the connection. A suitable packet format is likely to follow the pattern shown in Fig. 6.21. Initially, a synchronisation waveform begins the transmission, whose purpose is to overcome any DC bias, whilst allowing the receiving UART time to lock on to the data. A packet header follows, identifying the kind of packet that is being sent, followed by the packet payload and finally a checksum.

6.3.4.5 Connection establishment

Connections are established as follows (assuming a single manifold):

1. Both endpoints announce their identity, and announce the identifier of the signal that they wish to connect to.

2. Manifold detects the announcements.

3. Manifold replies to both end points to say that the connection is being established, then ceases to probe either connection.

4. Manifold establishes the connection, within a predetermined maximum time interval.

5. Both endpoints are now free to use the connection.

More complex manifold-of-manifolds architectures will require more complex negotiation and routing, though the necessary protocols are likely to remain similar. A typical connection establishment protocol across a manifold-of-manifolds architecture would follow the following pattern (assuming that the endpoints are on different manifolds):

1. Both endpoints announce their identity to their local manifolds, along with a globally unique signal identifier.

2. Each manifold announces the signal's availability to neighbouring manifolds, along with a distance measure. This takes place separately for each endpoint.

[7] A high current buffer amplifier constructed from relatively large geometry transistors is far less likely to be damaged by a voltage spike than a UART I/O pin.

3. Signal availability information continues to propagate across the manifold-of-manifolds. If a manifold receives connectivity information from more than one neighbouring manifold, this is ranked with the lowest distance measure first. Termination may be guaranteed by ensuring that connectivity information is propagated only when shorter distance measures are found than any previous measure announced on the same connection – the algorithm will therefore be guaranteed to reach a fixed point after at most a number of steps bounded by the number of manifolds in the system.

4. After a delay to allow propagation to complete, the endpoint manifolds now may use the routing information that has been collected in order to establish a shortest route across the manifold-of-manifolds.

This algorithm is essentially a distributed variation of Dijkstra's algorithm [50]. Since each manifold (graph node) has its own processor, time complexity is effectively $O(N)$ rather than the more usual $O(N^2)$, since processing power scales with N. The approach resembles the OSPF (Open Shortest Path First) routing protocol [100] in some respects.

Though this approach is relatively expensive in terms of the communications bandwidth required for routing, it nevertheless is unaffected by a dynamically changing architecture, or by component manifolds being unavailable, since routes will always be discovered if they exist, however complex they may be. In a fixed architecture satellite, some of this overhead may be avoided by precomputing the routing tables – such an approach would in principle be closer to the approach taken by the BGP (Border Gateway Protocol) [107].

6.3.4.6 Stale connection tear-down

In the event that a subsystem crashes, stale connections should be torn down after a known time-out interval. The discovery probe protocol should also allow a connection to be torn down more rapidly by announcing that a neighbouring connection is no longer in use. Assuming that a dynamic testing and fault recovery process will be continuously applied, there is no requirement for a 'keep alive' protocol to ensure that valid connections stay up (see also Section 6.4).

6.4 Dynamic testing and fault recovery

The same probe architecture necessary for discovery is also well suited to end-to-end testing of connections – if a connection is faulty (e.g. open circuit, shorted to ground or shorted to power), it will not be used, since the discovery process will fail to recognise it. As a consequence of this, at least for a short time after the discovery process has completed, all discovered connections may be regarded as functioning correctly. Over time, there is an increasing probability that, for example, permanent latch-up damage to a digital crossbar switch, may cause one or more connections to fail. This limitation can be avoided by constantly re-testing connections, ideally such that no connection may be established for a period longer than the minimum necessary to achieve the desired level of reliability.

6.4.1 Fault recovery protocol

There is actually no specific requirement to implement a fault recovery protocol as-such; the ability to set up and tear down connections, with make-before-break capabilities, is sufficient. Each end-point manifold should implement the following procedure (discovery and initial establishment of connections is assumed to have happened already):

1. Choose a signal on a round-robin basis.
2. Establish a second route to the same remote end-point through the discovery protocol, which has the side-effect of ensuring that end-to-end connectivity is currently valid.
3. Connect the signal to the newly established route, at both ends, whilst leaving the original connection in place.
4. Tear down the original connection.
5. Repeat.

Note that in larger systems, connections between manifolds must always provide sufficient spare connections to allow the discovery protocol to remain in operation at all times.

The stale connection timeout (see Section 6.3.4.6) should be longer than the worst-case time necessary to cycle through all connections.

When a connection fails, it will be repaired automatically the next time that the fault recovery procedure cycles through the relevant signal, because the failed route will no longer be detected, so it will naturally fall out of the pool of available connections.

6.4.2 Graceful degradation

In a situation where cumulative failures have exceeded the number of available connections, it is sensible to define a graceful degradation strategy in order to maximise the spacecraft's remaining functionality. A simple approach is to rank all signals in order of importance, with signals toward the end of the list simply being disconnected if insufficient connectivity is available, though more sophisticated approaches may allow greater levels of recovery:

Routing signals on a less-ideal sub-manifold Normally, for example, digital data would be routed through dedicated digital switch networks. In the event that insufficient digital switching capacity remains, it is potentially feasible to route signals through spare capacity in other switch networks, e.g. via MEMS switching that would normally be used for microwave signals or via high speed analogue routes.

Multiplexing Manifolds could potentially be equipped with multiplexing hardware, in order that multiple low speed signals could be routed through a single connection. Though this may degrade any signals carried in this way, it may still be preferable to disconnecting signals entirely.

Emergency backup routing As an extension to the multiplexing approach, in an emergency backup routes could be established by non-standard means, such as via low power local digital radio links.

6.5 Discussion

At the time of writing, this technology is at a relatively early stage of development; nevertheless, it is possible to determine the following advantages of reconfigurable manifolds over conventional fixed-architecture spacecraft wiring harnesses:

Cost Reduction Since a reconfigurable manifold does not need to be designed from-scratch for each satellite, considerable cost reductions in terms of initial design, validation and verification are likely.

Reduction in Time To Launch (Responsive Space) Reduced design effort has a direct effect in terms of calendar time, potentially helping reduce a design process that is conventionally measured in years to just weeks or even days.

Possibility for Re-purposing After Launch If a spacecraft is no longer required for its initial purpose, given a modular design, it is quite likely that it could be re-purposed after launch at very low cost. For example, an imaging satellite with excess communications bandwidth could, assuming it has enough fuel, be shifted to another orbit to act as a communications relay.

Disaster Recovery Now legendary, the recovery of Apollo 13 after an explosion that deprived the command module of all three of its fuel cells and its entire oxygen reserve, with all crew alive and unhurt [138], was a direct consequence of heroic efforts to jury-rig the lunar lander's oxygen systems in order to keep the crew alive. A conventional satellite has no astronauts with a kit of spare parts available to make repairs – typically, failures tend to be terminal. A reconfigurable manifold offers great potential for jury-rigging the craft, either from Earth or possibly autonomously, so as to allow it to continue with some or all of its mission.

Mass reduction By sharing redundant wiring capacity across all subsystems, the total amount of copper necessary is reduced considerably in comparison with modular-redundant conventional wiring. At approximately $30,000 per kg to low earth orbit, even small savings can have considerable consequences in terms of cost.

The responsive space paradigm makes it essential for plug-and play concepts that are now ubiquitous in desktop computing (e.g. PCI [2], USB [6] and FireWire/IEEE 1394 [66, 67, 68]) to be migrated to satellite architectures. Though in some cases these technologies may be used directly (USB, in particular, is currently in use in satellites), digital networking alone is insufficient. The reconfigurable manifold approach, however, allows similar results to be achieved for almost all kinds of signal.

Chapter 7

SET Immunity in Delay-Insensitive Circuits

The work described in this chapter was previously published in [132].

7.1 Introduction

Most digital electronics is designed assuming a *synchronous* model; a single global clock signal drives the clock inputs of every flip flop in the system, and unclocked feedback loops are outlawed. Synchronous circuits are relatively easy to reason about, which greatly simplifies the circuit design process. Signals can be thought of as only ever changing in lock-step with each other, and since the clock rate is typically assumed to be slow enough to encompass worst-case propagation delays, this makes that gates can be assumed to behave in exactly the way that Boolean logic predicts.

The real world, however, is fundamentally *asynchronous*. Even something so simple as a push button exhibits the characteristic that it can change state at any time, requiring designers to be very careful when constructing appropriate interface circuits. Intuition based on the synchronous model tends to break down due to the need to consider timing information that is continuous in nature. Formal reasoning is generally difficult, so engineers often use inexact discrete time simulations that often miss possible pathological behaviour.

Abstract interpretation [42, 43], introduced in Section2.2.1, provides a sound formal framework that allows abstractions of concrete systems to be reasoned about and, ideally, proven correct. In particular, the technique is very good for enabling continuous, possibly infinitary, behaviour in the concrete world to be modeled finitely. In [130] (see also Chapter 3), we introduced *transitional logic,* an abstract interpretation technique resembling a multi-valued logic that is capable of supporting reasoning about asynchronous behaviour of combinational circuits.

7.1.1 Majority Voting Circuits

Most safety critical and mission critical systems depend upon modular redundancy in order to engineer high reliability. Typically, *majority voting logic* is used to arbitrate

between the outputs of redundant subsystems, as shown in Fig. 7.1.

The original concept of voting logic appears to be due to John von Neumann [143], and has been widely implemented in terms of digital electronics, analogue electronics and also mechanically.

A typical 3-way voting circuit implements the Boolean function shown in Table 7.1.

a	b	c	$\mathcal{V}_3(a,b,c)$
F	F	F	F
F	F	T	F
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	T
T	T	F	T
T	T	T	T

Table 7.1: Truth table for 3-way voting logic

7.1.1.1 Analogue Voting Logic

Fig. 7.2 shows a typical implementation based on analogue electronics. In this circuit, the inputs are assumed to be driven to ground representing F, or driven to the positive rail when representing T. A resistor network 'sums' these voltages, the result of which is then compared with a reference voltage in order to recover a rail-to-rail Boolean output. The voltages at the inputs of the comparator for all combinations of the input are shown as follows:

a	b	c	Sum Voltage	Reference Voltage	Output
F	F	F	Ground	$\frac{1}{2} \times V_{\text{supply}}$	F
F	F	T	$\frac{1}{3} \times V_{\text{supply}}$	$\frac{1}{2} \times V_{\text{supply}}$	F
F	T	F	$\frac{1}{3} \times V_{\text{supply}}$	$\frac{1}{2} \times V_{\text{supply}}$	F
F	T	T	$\frac{2}{3} \times V_{\text{supply}}$	$\frac{1}{2} \times V_{\text{supply}}$	T
T	F	F	$\frac{1}{3} \times V_{\text{supply}}$	$\frac{1}{2} \times V_{\text{supply}}$	F
T	F	T	$\frac{2}{3} \times V_{\text{supply}}$	$\frac{1}{2} \times V_{\text{supply}}$	T
T	T	F	$\frac{2}{3} \times V_{\text{supply}}$	$\frac{1}{2} \times V_{\text{supply}}$	T
T	T	T	V_{supply}	$\frac{1}{2} \times V_{\text{supply}}$	T

Note that the output of the comparator is T iff the sum voltage exceeds the reference voltage, and that this circuit implements exactly the required Boolean function.

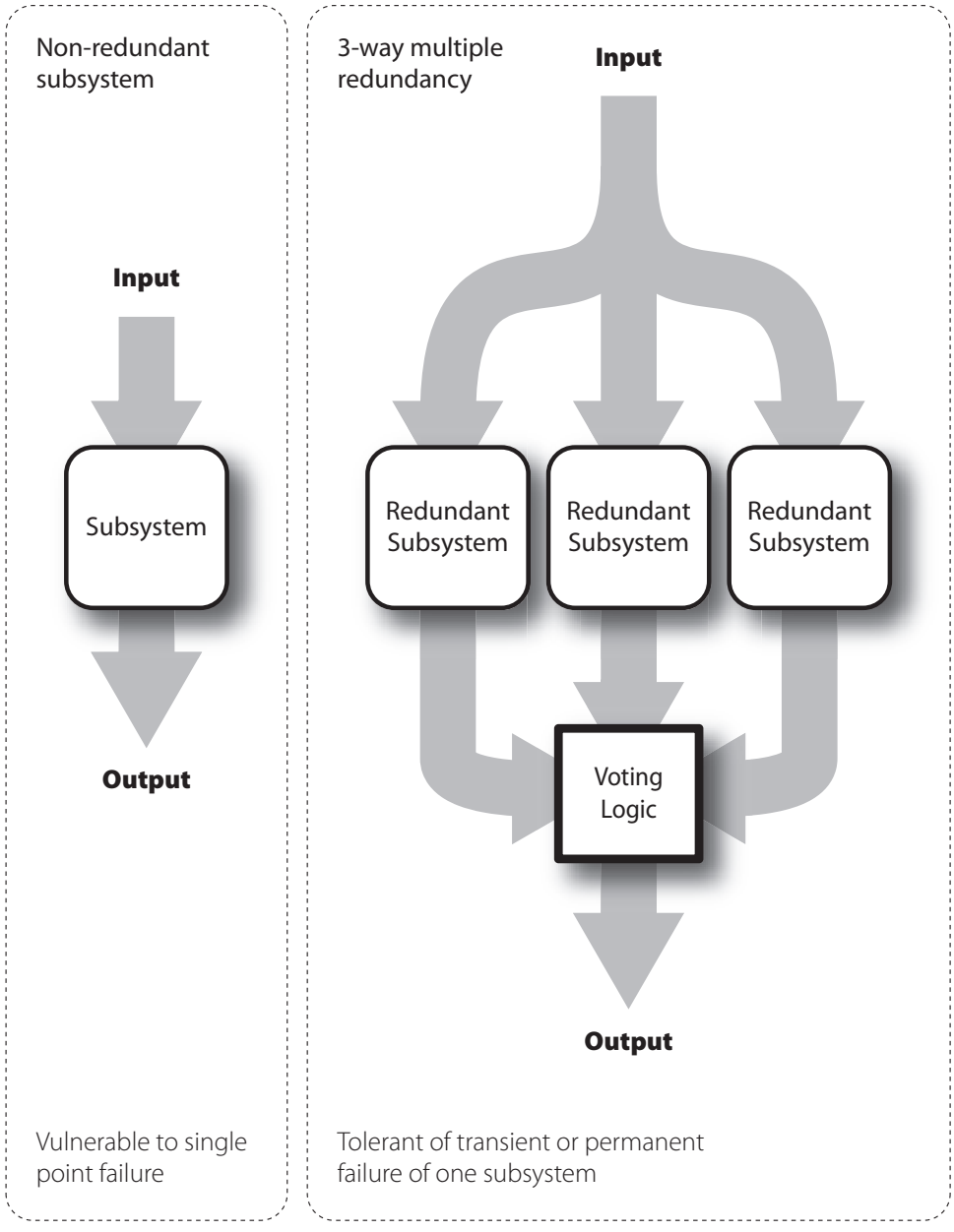


Figure 7.1: Comparison of non-redundant and 3-way redundant subsystems

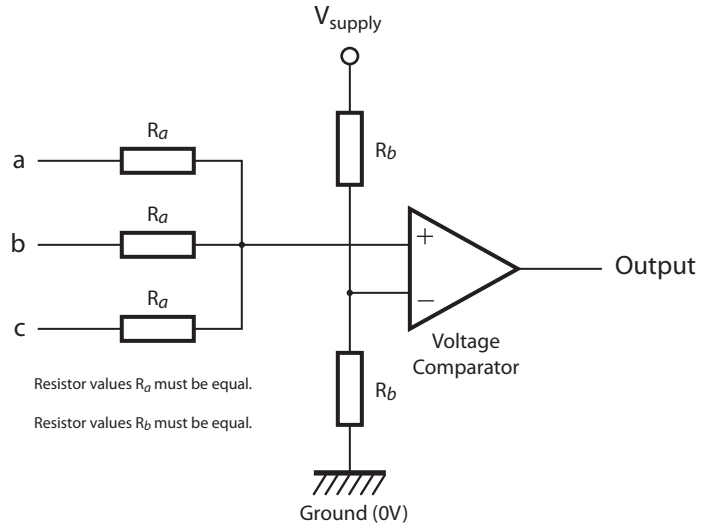


Figure 7.2: Analogue majority voting circuit

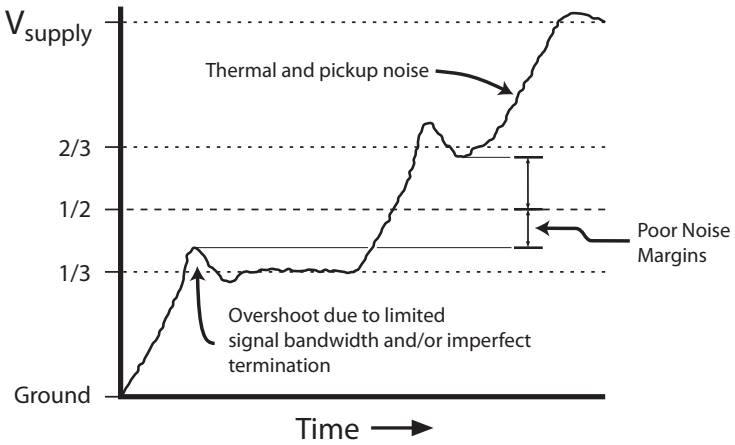


Figure 7.3: Analogue majority voting noise margins

Analogue voting circuits are quite popular due to their simplicity and inherent radiation hardness – resistors are not easily degraded by radiation, and voltage comparators can be built from extremely robust large geometry bipolar transistors. However, as shown in Fig. 7.3, the usually generous noise margins of familiar logic families are eroded significantly: 3-way voting implies a best-case margin of just $\frac{1}{6} \times V_{\text{supply}}$. 5-way voting implies a margin of $\frac{1}{10} \times V_{\text{supply}}$, so overshoot, ringing due incorrect termination, resistor tolerance errors and thermal noise all place limits on effective reliability and create some tricky design challenges. Choosing appropriate resistor values can also be problematic – small values will enable faster switching, but the current paths created between the power rails causes power drain and heating of the resistors themselves. Larger values reduce current requirements, but dynamic performance inevitably suffers.

7.1.1.2 Digital Voting Logic

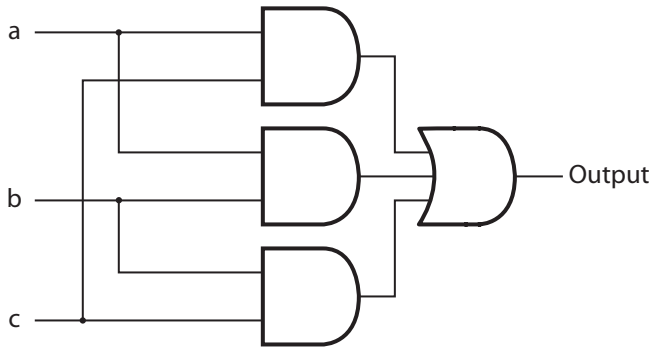


Figure 7.4: Digital majority voting circuit

Fig. 7.4 shows a purely digital implementation that also implements the necessary majority voting function. Building a voting circuit from CMOS gates inherits the basic advantages of the underlying technology; current requirements are essentially proportional to switching rate, with (usually) negligible quiescent leakage current. Switching speed is inherently fast, again due to the properties of the implementation technology, and will typically exceed that of a typical analogue implementation very considerably. However, since the gates themselves are vulnerable to radiation damage and single event effects (SEEs), reliability may suffer.

The circuit of Fig. 7.4 may be equivalently expressed[1] as

$$\mathcal{V}_3(a, b, c) \stackrel{\text{def}}{=} (a \wedge c) \vee (a \wedge b) \vee (b \wedge c).$$

7.1.2 Analysis by the Karnaugh Map Technique

The traditional approach to analysing the dynamic behaviour of such a circuit typically involves drawing a Karnaugh map of the function, and marking it with 'cubes' representing each of the product terms (in this case, $a \wedge c$, $a \wedge b$ and $b \wedge c$). The Karnaugh map for this function is shown in Fig. 7.5.

Any particular assignment of the inputs corresponds to exactly one square of the grid; in Fig. 7.5, the black bars represent the relevant input being T. For example, the top left corner square represents $a = \mathsf{F}, b = \mathsf{T}, c = \mathsf{F}$, and the square representing $a = \mathsf{T}, b = \mathsf{T}, c = \mathsf{T}$ is located second from the left on the bottom row. Changes to the inputs may be visualised as movement within the grid, so with b and c static, a transitioning from F to T represents a move from some square on the top row to the square directly below it on the bottom row. To anyone unfamiliar with Karnaugh maps, the order of the squares in the grid seems unusual at first, but is deliberately arranged

[1] Note that this circuit possesses an unusual symmetry, in that $\wedge$ and $\vee$ may be interchanged, giving $(a \vee c) \wedge (a \vee b) \wedge (b \vee c)$, without affecting functionality in terms of the Boolean function that is implemented. This symmetry turns out not to extend to dynamic behaviour, however – see also Section 7.1.5.

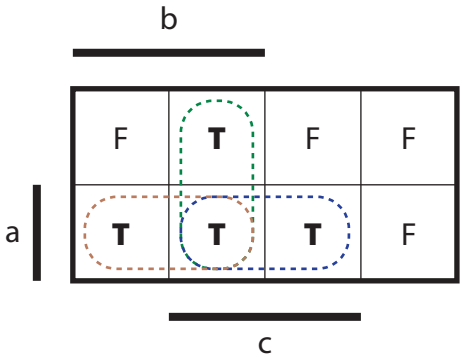


Figure 7.5: Karnaugh map for $(a \wedge c) \vee (a \wedge b) \vee (b \wedge c)$

in order that a change to any single input causes a grid movement of exactly one square left, right, up or down[2].

The utility of Karnaugh maps can be demonstrated by a simple example. The Boolean function described by the truth table shown in Table 7.1 can be equivalently implemented as

$$(a \wedge c) \vee (a \wedge b \wedge \neg c) \vee (b \wedge c).$$

Though this ostensibly implements the same function as $(a \wedge c) \vee (a \wedge b) \vee (b \wedge c)$, the Karnaugh map is different, as shown in Fig. 7.6.

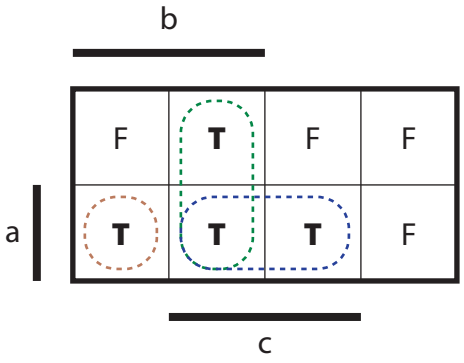


Figure 7.6: Karnaugh map for $(a \wedge c) \vee (a \wedge b \wedge \neg c) \vee (b \wedge c)$

In the original circuit, with $a = \mathsf{T}$ and $b = \mathsf{T}$, switching c from T to F remains within the cube representing the term $a \wedge b$. In the modified circuit, this switching requires jumping between one area, represented by $a \wedge c$ and $b \wedge c$, to the *disjoint* area representing $a \wedge b \wedge \neg c$. Such cases are normally interpreted by electronic designers as a warning of possible glitches (static hazards).

Returning to the original circuit represented by the Karnaugh map shown in Fig. 7.5, it appears that a, b and c should be hazard-free, and from simple analysis of the truth table (or indeed the Karnaugh map), it also seems that errors on any single input will not

[2]Note that the grid is toroidal and wraps around in all axes, so corresponding squares on opposite edges are effectively adjacent.

affect the output. A naïve designer might therefore conclude that such a voting circuit is hazard-free, and will provide immunity to both stuck-at faults and single event transients on the output of any one subsystem.

7.1.3 Analysis by Transitional Logic

In this section, we will analyse the circuit shown in Fig. 7.4 by means of the transitional logic $\wp(\mathbb{T})$ from Chapter 3. The Karnaugh map technique, as described in Section 7.1.2, does not provide specific ways of modeling stuck-at or SET faults. They may, however, be straightforwardly modeled as $\odot_{\mathsf{F}}$, $\odot_{\mathsf{T}}$ and $\circledast$ in $\wp(\mathbb{T})$, as follows:

$$\begin{aligned}
\odot_{\mathsf{F}}(X_n) &\stackrel{\text{def}}{=} \mathsf{F}_0 \\
\odot_{\mathsf{T}}(X_n) &\stackrel{\text{def}}{=} \mathsf{T}_0 \\
\circledast(X_n) &\stackrel{\text{def}}{=} X_{0..n+1} \\
\circledast(\hat{t}\,) &\stackrel{\text{def}}{=} \bigcup_{t \in \hat{t}} \circledast(t)
\end{aligned}$$

The functions $\odot_{\mathsf{F}}$ and $\odot_{\mathsf{T}}$ model permanent failures that manifest as inputs that are permanently stuck at F or T respectively. The $\circledast$ function models the effect of a cosmic ray impact – the signal may be unaffected[3], may gain one extra pulse, or have any number of existing pulses masked by the SET.

It makes sense to begin by modeling a correctly functioning circuit, with no SETs or stuck-at faults, so that its behaviour can be compared with modeled faults. In normal operation, the inputs a, b and c are assumed to track each other's values, possibly with some timing skew. We therefore consider the two steady states, $a = b = c = \mathsf{F}_0$ and $a = b = c = \mathsf{T}_0$, and the clean transitions $a = b = c = \uparrow_0$ and $a = b = c = \downarrow_0$ as representing the behaviour that we are most concerned about. Simple calculation gives the results show in Table 7.2. These results support the engineer's intuition that follows

a	b	c	$\mathcal{V}_3(a,b,c)$
F_0	F_0	F_0	F_0
T_0	T_0	T_0	T_0
$\uparrow_0$	$\uparrow_0$	$\uparrow_0$	$\uparrow_0$
$\downarrow_0$	$\downarrow_0$	$\downarrow_0$	$\downarrow_0$

Table 7.2: Behaviour of correctly functioning circuit

from Karnaugh map analysis. Table 7.3 shows the corresponding result for a stuck-at fault on one input[4] – so far, the simplistic analysis of the previous session remains

[3] *i.e.,* $\forall t \in \mathbb{T} \,.\, t \in \circledast(\{t\})$, which follows trivially since $n \in 0..n+1$ for all $n \geq 0$

[4] Since the circuit is symmetric with respect to all possible permutations of its inputs, we model faults on input a but can be confident that these results also apply to faults on b or c. Sanity check calculations (not shown here) confirm this assumption.

predictive.

a	b	c	$\mathcal{V}_3(\odot_{\mathsf{F}}(a), b, c)$	$\mathcal{V}_3(\odot_{\mathsf{T}}(a), b, c)$
F_0	F_0	F_0	F_0	F_0
T_0	T_0	T_0	T_0	T_0
$\uparrow_0$	$\uparrow_0$	$\uparrow_0$	$\uparrow_0$	$\uparrow_0$
$\downarrow_0$	$\downarrow_0$	$\downarrow_0$	$\downarrow_0$	$\downarrow_0$

Table 7.3: Behaviour of circuit with one stuck-at fault

The results for the SET model, shown in Table 7.4, are unexpected. Where a, b and

a	b	c	$\mathcal{V}_3(\circledast(a), b, c)$
F_0	F_0	F_0	F_0
T_0	T_0	T_0	T_0
$\uparrow_0$	$\uparrow_0$	$\uparrow_0$	$\uparrow_{0..2}$
$\downarrow_0$	$\downarrow_0$	$\downarrow_0$	$\downarrow_{0..2}$

Table 7.4: Behaviour of circuit with one SET

c are steady, the circuit does not pass on SETs, as one might hope. However, during a transition of the inputs, not only can the SET get through, but in worst case, *two* glitch pulses may be introduced – something that very much contradicts the 'obvious' results of the traditional approach. Though the prediction of the transitional logic analysis is not intuitively obvious, the correctness[5] and completeness[6] of its predictions have been proven mathematically (see Section 3.4.1). Nevertheless, a little patience with pencil and paper makes it possible to identify combinations of timing delays that can indeed demonstrate this result in practice.

7.1.3.1 Extension to 5-way Voting Logic

A significant and well-known problem with the Karnaugh map approach is that, when the number of inputs exceeds four, diagrams start to require more than two dimensions. At a pinch, it is possible to represent extra dimensions by drawing multiple tables, but this is difficult and rather error-prone. The transitional logic approach does not suffer from such limitations. Extending the design approach of the circuit shown in Fig. 7.4 to five inputs is relatively straightforward, resulting in the following expression:

$$\begin{aligned}\mathcal{V}_5(a,b,c,d,e) \stackrel{\text{def}}{=} \quad & (a \wedge b \wedge c) \vee (a \wedge b \wedge d) \vee (a \wedge b \wedge e) \vee (a \wedge c \wedge d) \vee (a \wedge c \wedge e) \\ & \vee (a \wedge d \wedge e) \vee (b \wedge c \wedge d) \vee (b \wedge c \wedge e) \vee (b \wedge d \wedge e) \vee (c \wedge d \wedge e)\end{aligned}$$

[5]The prediction includes all possible behaviour, though may include some impossible behaviour, *i.e.,* it may in a formal sense be regarded as an overapproximation.

[6]The prediction includes all possible behavior, and all predicted behaviours can happen.

Stuck-at faults are tolerated as expected, with up to two stuck inputs being tolerated. SET analysis results are shown in Table 7.5. 5-way voting logic, like its 3-way counterpart, is

a,b,c,d,e	$\mathcal{V}_5(a,b,c,d,e)$	$\mathcal{V}_5(\circledast(a),b,c,d,e)$
F_0	F_0	F_0
T_0	T_0	T_0
$\uparrow_0$	$\uparrow_0$	$\uparrow_{0..6}$
$\downarrow_0$	$\downarrow_0$	$\downarrow_{0..6}$

Table 7.5: SET behaviour of 5-value voting logic

susceptible to allowing SETs through during transitions on its inputs. The analysis shows that, in worst-case, a single SET pulse may manifest as up to *six* pulses at the output; though more tolerant of stuck-at faults, it appears that 5-way voting logic may in fact be significantly less tolerant of SETs.

7.1.4 A Possible Solution?

The expression $\mathcal{V}_3(\circledast(\uparrow_0),\uparrow_0,\uparrow_0)$ can be interpreted as modeling a situation where the three inputs of the voting circuit at some unspecified time transition from F to T, with an SET superimposed on one of the inputs at some, again unspecified, time. Recalling that expressions in the transitional logic essentially reflect extremes of behaviour that may be caused under the union of all possible timing relationships, it follows that constraining the timing relationships that are allowed may similarly constrain possible behaviour.

Constraining the inputs to switch one at a time in a predetermined sequence, perhaps as a consequence of an externally imposed communications protocol, can restrict what behaviours are possible. We may therefore view the input protocol as a sequence of well-defined states and transitions[7], as described in Table 7.6.

a	b	c	$\mathcal{V}_3(a,b,c)$	$\mathcal{V}_3(\circledast(a),b,c)$	$\mathcal{V}_3(a,\circledast(b),c)$	$\mathcal{V}_3(a,b,\circledast(c))$
F_0	F_0	F_0	F_0	F_0	F_0	F_0
F_0	F_0	$\uparrow_0$	F_0	$\mathsf{F}_{0..1}$	$\mathsf{F}_{0..1}$	F_0
F_0	F_0	T_0	F_0	$\mathsf{F}_{0..1}$	$\mathsf{F}_{0..1}$	F_0
F_0	$\uparrow_0$	T_0	$\uparrow_0$	$\uparrow_{0..2}$	$\uparrow_{0..1}$	$\uparrow_{0..1}$
F_0	T_0	T_0	T_0	T_0	$\mathsf{T}_{0..1}$	$\mathsf{T}_{0..1}$
$\uparrow_0$	T_0	T_0	T_0	T_0	$\mathsf{T}_{0..2}$	$\mathsf{T}_{0..2}$
T_0	T_0	T_0	T_0	T_0	T_0	T_0

Table 7.6: SET behaviour 3-value voting logic with sequencing

[7]This approach breaks the symmetry between a, b and c, so $\circledast(a)$, $\circledast(b)$ and $\circledast(c)$ must be considered separately

The results can easily be verified to be correct, but are nonetheless disappointing. Extending this approach to the 5-way voting circuit $\mathcal{V}_5$ yields similar results, as shown in Table 7.7.

a	b	c	d	e	$\mathcal{V}_5(a,b,c,d,e)$	$\mathcal{V}_5(\circledast(a),b,c,d,e)$	$\mathcal{V}_5(a,\circledast(b),c,d,e)$	$\mathcal{V}_5(a,b,\circledast(c),d,e)$	$\mathcal{V}_5(a,b,c,\circledast(d),e)$	$\mathcal{V}_5(a,b,c,d,\circledast(e))$
F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0
F_0	F_0	F_0	F_0	$\uparrow_0$	F_0	F_0	F_0	F_0	F_0	F_0
F_0	F_0	F_0	F_0	T_0	F_0	F_0	F_0	F_0	F_0	F_0
F_0	F_0	F_0	$\uparrow_0$	T_0	F_0	$\mathsf{F}_{0..1}$	$\mathsf{F}_{0..1}$	$\mathsf{F}_{0..1}$	F_0	F_0
F_0	F_0	F_0	T_0	T_0	F_0	$\mathsf{F}_{0..1}$	$\mathsf{F}_{0..1}$	$\mathsf{F}_{0..1}$	F_0	F_0
F_0	F_0	$\uparrow_0$	T_0	T_0	$\uparrow_0$	$\uparrow_{0..3}$	$\uparrow_{0..3}$	$\uparrow_{0..1}$	$\uparrow_{0..1}$	$\uparrow_{0..1}$
F_0	F_0	T_0	T_0	T_0	T_0	T_0	T_0	$\mathsf{T}_{0..1}$	$\mathsf{T}_{0..1}$	$\mathsf{T}_{0..1}$
F_0	$\uparrow_0$	T_0	T_0	T_0	T_0	T_0	T_0	$\mathsf{T}_{0..3}$	$\mathsf{T}_{0..3}$	$\mathsf{T}_{0..3}$
F_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0
$\uparrow_0$	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0
T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0

Table 7.7: SET behaviour of 5-value voting logic with sequencing

From examination of Tables 7.6 and 7.7 in comparison with Tables 7.4 and 7.5, it is clear that the 5-way voting circuit has improved worst-case behaviour, but there is no significant advantage in imposing this sequential protocol on a 3-way voting regime.

7.1.5 Duality of $\mathcal{V}_3$ and $\mathcal{V}'_3$

Recalling the definition for $\mathcal{V}_3$,

$$\mathcal{V}_3(a,b,c) \stackrel{\text{def}}{=} (a \wedge c) \vee (a \wedge b) \vee (b \wedge c)$$

an alternative implementation,

$$\mathcal{V}'_3(a,b,c) \stackrel{\text{def}}{=} (a \vee c) \wedge (a \vee b) \wedge (b \vee c)$$

is widely assumed to be equivalent (*dual*), because $\wedge$ and $\vee$ can be interchanged without affecting the Boolean function that the circuit implements. This assumption does *not* hold with regard to dynamic behaviour.

Theorem 7.1.1. *$\mathcal{V}_3$ and $\mathcal{V}'_3$ are not equivalent. Proof by counterexample:*

$$\begin{aligned}
& \mathcal{V}_3(\circledast(\mathsf{F}_0), \mathsf{F}_0, \uparrow_0) \\
= \;& \mathcal{V}'_3(\mathsf{F}_{0..1}, \mathsf{F}_0, \uparrow_0) \\
= \;& (\mathsf{F}_{0..1} \wedge \uparrow_0) \vee (\mathsf{F}_{0..1} \wedge \mathsf{F}_0) \vee (\mathsf{F}_0 \wedge \uparrow_0) \\
= \;& \mathsf{F}_{0..1} \vee \mathsf{F}_0 \vee \mathsf{F}_0 \\
= \;& \mathsf{F}_{0..1}
\end{aligned}$$

whereas

$$\begin{aligned} & \mathcal{V}'_3(\circledast(\mathsf{F}_0), \mathsf{F}_0, \uparrow_0) \\ = \; & \mathcal{V}'_3(\mathsf{F}_{0..1}, \mathsf{F}_0, \uparrow_0) \\ = \; & (\mathsf{F}_{0..1} \vee \uparrow_0) \wedge (\mathsf{F}_{0..1} \vee \mathsf{F}_0) \wedge (\mathsf{F}_0 \vee \uparrow_0) \\ = \; & \uparrow_{0..1} \wedge \mathsf{F}_{0..1} \wedge \uparrow_0 \\ = \; & \mathsf{F}_{0..2}. \end{aligned}$$

Note that inverting all inputs and the output restores duality, *i.e.,*

$$\forall a \in \wp(\mathbb{T}), b \in \wp(\mathbb{T}), c \in \wp(\mathbb{T}) \; . \; \mathcal{V}_3(a, b, c) = \neg \mathcal{V}'_3(\neg a, \neg b, \neg c)$$

but this is just a consequence of the fact that de Morgan's law holds for the transitional logic, rather than any special property of this particular circuit.

7.2 Generalising the Result

It is clear from the preceding sections that 3- and 5-input majority voting circuits are not immune to SETs. In this section, we generalise this result in order to show that SET immunity cannot be guaranteed by any possible delay-insensitive circuit.

Definition 7.2.1. *SET Immunity. A* SET-immune *circuit should be able to function correctly (i.e., not pass on a SET pulse to its output or outputs) when an arbitrarily chosen input is subjected to a SET, which itself may occur at any time and persist for any (finite) duration. More formally,*

$$\begin{aligned} \mathit{Immune}(f^\sharp) \stackrel{\text{def}}{=} & \forall i \in [1, k] \; \forall (x_1, \ldots, x_k) \in (\wp(\mathbb{T}))^k \; . \\ & f^\sharp(x_1, \ldots, x_{i-1}, \circledast(x_i), x_{i+1}, \ldots, x_k) = f^\sharp(x_1, \ldots, x_k) \end{aligned}$$

where $f^\sharp : (\wp(\mathbb{T}))^k \rightarrow \wp(\mathbb{T})$.

Corollary 7.2.1. *SET sensitivity. Since a genuinely SET-immune circuit must be immune for all possible combinations of inputs, it follows that a single counter-example is sufficient to show that a given circuit is not SET immune.*

Definition 7.2.2. *Refinement. Given the traces* $t \in \mathbb{T}$, $u \in \mathbb{T}$, $\hat{t} \in \wp(\mathbb{T})$ *and* $\hat{u} \in \wp(\mathbb{T})$,

$$t \succcurlyeq u \stackrel{\text{def}}{\equiv} \mathit{Val}(t) = \mathit{Val}(u) \wedge \mathit{Subs}(t) \geq \mathit{Subs}(u)$$

$$t \succ u \stackrel{\text{def}}{\equiv} \mathit{Val}(t) = \mathit{Val}(u) \wedge \mathit{Subs}(t) > \mathit{Subs}(u)$$

$$\hat{t} \succcurlyeq \hat{u} \stackrel{\text{def}}{\equiv} (\forall t \in \hat{t}, u \in \hat{u} \; . \; \mathit{Val}(t) = \mathit{Val}(u)) \wedge \mathit{MaxSubs}(\hat{t}) \geq \mathit{MaxSubs}(\hat{u})$$

$$\hat{t} \succ \hat{u} \stackrel{\text{def}}{\equiv} (\forall t \in \hat{t}, u \in \hat{u} \; . \; \mathit{Val}(t) = \mathit{Val}(u)) \wedge \mathit{MaxSubs}(\hat{t}) > \mathit{MaxSubs}(\hat{u})$$

where Val : $\mathbb{T} \to \{\mathsf{F}, \mathsf{T}, \uparrow, \downarrow\}$*, Subs* : $\mathbb{T} \to \mathbb{N}$ *and MaxSubs* : $\wp(\mathbb{T}) \to \mathbb{N}$ *were defined in Section 3.4 as follows:*

$$\mathit{Val}(X_n) \stackrel{\text{def}}{=} X$$
$$\mathit{Subs}(X_n) \stackrel{\text{def}}{=} n$$
$$\mathit{MaxSubs}(\hat{t}) \stackrel{\text{def}}{=} \max_{t \in \hat{t}} (\mathit{Subs}(t))$$

Note that $\hat{t} \succ \hat{u}$ implies $\hat{t} \neq \hat{u}$.

Lemma 7.2.1. *Given any trace* $t \in \mathbb{T}$*,* $\circledast t \succ t$*. Proof:*

1. *Expand* $\circledast t \succ t$ *to* $\circledast X_n = X_{0..n+1} \succ X_n$*.*
2. *Since* $\forall n \in \mathbb{N} \,.\, n + 1 > n$*, the proof follows from Definition 7.2.2.*

Lemma 7.2.2. *The operators* $\circledast$ *and* $\neg$ *commute under composition, i.e.,* $\circledast \circ \neg = \neg \circ \circledast$*. Proof by cases:*

1. $\circledast(\neg \mathsf{F}_n) = \circledast \mathsf{T}_n = \mathsf{T}_{0..n+1} = \neg \mathsf{F}_{0..n+1} = \neg(\circledast \mathsf{F}_n)$.
2. $\circledast(\neg \mathsf{T}_n) = \circledast \mathsf{F}_n = \mathsf{F}_{0..n+1} = \neg \mathsf{T}_{0..n+1} = \neg(\circledast \mathsf{T}_n)$.
3. $\circledast(\neg \uparrow_n) = \circledast \downarrow_n = \downarrow_{0..n+1} = \neg \uparrow_{0..n+1} = \neg(\circledast \uparrow_n)$.
4. $\circledast(\neg \downarrow_n) = \circledast \uparrow_n = \uparrow_{0..n+1} = \neg \downarrow_{0..n+1} = \neg(\circledast \downarrow_n)$.

Corollary 7.2.2. *Negation is linear and strictly increasing, i.e.,* $\neg(\circledast x) \succ \neg x$*. Proof:*

1. *From Lemma 7.2.2,* $\neg(\circledast x) = \circledast(\neg x)$*, so* $\circledast(\neg x) \succ \neg x$*.*
2. *Rewriting* $\neg x$ *as* y *gives* $\circledast y \succ y$*, so the proof follows as a consequence of Lemma 7.2.1.*

Corollary 7.2.3. *SET sensitivity of* $\neg$*. Proof:*

1. *Since* $\neg(\circledast x) \succ \neg x$*, it follows that* $\neg(\circledast x) \neq \neg x$*, therefore* $\neg$ *is SET-sensitive for all* $x \in \mathbb{T}$*.*

Lemma 7.2.3. *Transmission line delay is linear and strictly increasing, i.e.,* $\Delta(\circledast x) \succ \Delta x$*. Proof:*

1. *Since* Δ *is an identity function with respect to* $\wp(\mathbb{T})$*, we can rewrite* $\Delta(\circledast x) \succ \Delta x$ *as* $\circledast x \succ x$*.*
2. *The proof follows from Lemma 7.2.1.*

Corollary 7.2.4. *SET sensitivity of* Δ*. Proof:*

1. *Since* $\Delta(\circledast x) \succ \Delta x$*, it follows that* $\Delta(\circledast x) \neq \Delta x$*, therefore* Δ *is SET-sensitive for all* $x \in \mathbb{T}$*.*

Lemma 7.2.4. *The operators* $\circledast$ *and* $\Box$ *commute under composition, i.e.,* $\circledast \circ \Box = \Box \circ \circledast$*. Proof:*

1. $\Box(\circledast X_n) = \Box X_{0..n+1} = X_{0..n+1} = \circledast X_{0..n} = \circledast(\Box X_n)$.

Corollary 7.2.5. *Inertial delay is linear and strictly increasing, i.e.,* $\Box(\circledast x) \succ \Box x$. *Proof:*

1. *From Lemma 7.2.4,* $\Box(\circledast x) = \circledast(\Box x)$, *so* $\circledast(\Box x) \succ \Box x$.
2. *Rewriting* $\Box x$ *as* y *gives* $\circledast y \succ y$, *so the proof follows as a consequence of Lemma 7.2.1.*

Corollary 7.2.6. *SET sensitivity of* $\Box$. *Proof:*

1. *Since* $\Box(\circledast x) \succ \Box x$, *it follows that* $\Box(\circledast x) \neq \Box x$, *therefore* $\Box$ *is SET-sensitive for all* $x \in \mathbb{T}$.

Lemma 7.2.5. *Linearity of* $\wedge$: $(\circledast x) \wedge y \succcurlyeq x \wedge y$. *Proof by cases*[8]*:*

1. F_0 *is a zero with respect to* $\wedge$, *e.g.,* $x \wedge \mathsf{F}_0 = \mathsf{F}_0 = (\circledast x) \wedge \mathsf{F}_0$.
2. *Where* $n > 0$, $(\circledast x) \wedge \mathsf{F}_n \succ x \wedge \mathsf{F}_n$.
3. *Where* $n \geq 0$ *and* $X \in \{\mathsf{T}, \uparrow, \downarrow\}$, $(\circledast x) \wedge X_n \succ x \wedge X_n$.

Corollary 7.2.7. *Linearity of* $\vee$: $(\circledast x) \vee y \succcurlyeq x \vee y$. *Proof:*

1. T_0 *is a zero with respect to* $\vee$, *e.g.,* $x \vee \mathsf{T}_0 = \mathsf{T}_0 = (\circledast x) \vee \mathsf{T}_0$.
2. *Where* $n > 0$, $(\circledast x) \vee \mathsf{T}_n \succ x \vee \mathsf{T}_n$.
3. *Where* $n \geq 0$ *and* $X \in \{\mathsf{T}, \uparrow, \downarrow\}$, $(\circledast x) \vee X_n \succ x \vee X_n$.

Note that $\wedge$ and $\vee$ are dual and are related by a de Morgan's law, *e.g.,* $a \wedge b = \neg(\neg a \vee \neg b)$.

Corollary 7.2.8. *SET sensitivity of* $\wedge$ *and* $\vee$. *For the specific case where one or both of* a *and* b *in* $a \wedge b$ *or* $a \vee b$ *are zeros, the circuit is SET-insensitive. In all other cases, SET sensitivity follows from strictly-increasing linearity. Since only one counter-example is required to show SET sensitivity, it can be concluded that both* $\wedge$ *and* $\vee$ *are SET sensitive.*

Theorem 7.2.1. *SET immunity cannot be guaranteed for any possible delay-insensitive circuit. Proof:*

1. *From Corollary 7.2.4, we know that* Δ *is SET-sensitive, and from Lemma 7.2.3 we know that it is also linear and strictly increasing with respect to subscripts.*
2. *Corollary 7.2.6 shows that* $\Box$ *is SET-sensitive, and from Corollary 7.2.5 it is linear and strictly increasing.*
3. *From Corollary 7.2.3,* $\neg$ *is SET-sensitive, and from Corollary 7.2.2 it is linear and strictly increasing.*
4. *Lemma 7.2.5 and Corollary 7.2.7 show that* $\wedge$ *and* $\vee$ *are linear, though only strictly increasing for non-zero values.*

[8]For brevity, we do not list all cases in detail

5. *Corollary 7.2.8 shows that $\wedge$ and $\vee$ are SET-sensitive.*

6. *Given an arbitrary expression constructed by composition of $\wedge, \vee, \neg, \Box$ and Δ, linearity follows from structural induction, and given that it is always possible to choose values for variables other than the group-theoretic zeros F_0 and T_0, e.g., by choosing values from $\wp(\mathbb{T}\backslash\{\mathsf{F}_0, \mathsf{T}_0\})$, it follows that all such circuits are SET-sensitive.*

7.3 Related Work

Voting logic was first introduced in 1956 by John von Neumann[143] as a means of constructing reliable circuits ('organisms' in his original terminology) from unreliable components.

NASA's Office of Logic Design maintains an excellent on-line resource [77] that covers many aspects of digital design for aerospace applications.

7.4 Discussion

The work reported in this chapter started with an attempt to create a simple example, whereby transitional logic could be used to prove the correctness of a well-understood circuit that would be familiar to most engineers. Finding that this circuit, unexpectedly, was not in fact able to guarantee to reject single-event transients was surprising, though the result did make sense on closer examination. Our finding that the same limitation extends to all possible delay insensitive circuits has some troubling consequences; a practical SET-immune technology must therefore either be constructed from hardened gates that can never glitch in response to a SET, or must employ some kind of delay-sensitive SET rejection approach. It is interesting to note that both of these approaches are used in aerospace electronics – the widely used RAD6000 and RAD750 radiation hardened PowerPC processors are built using a radiation hardened standard cell technology with internally redundant gates constructed at the transistor level, and it is also common practice to low-pass filter the output of critical voting logic circuits, particularly those used for applications such as triggering of rockets or explosive bolts.

Part IV

Conclusions

Chapter 8

Conclusions

"I'm at the foot of the ladder. The LM footpads are only depressed in the surface about one or two inches. Although the surface appears to be very very fine grained, as you get close to it, it's almost like powder. Now and then it's very fine. I'm going to step off the LM now.

That's one small step for man, one giant leap for mankind."

– Neil Armstrong, 20th July 1969 [138] pp. 261.

This thesis has been primarily concerned with the application of two major techniques from program analysis and transformation, specifically abstract interpretation and partial evaluation, to digital electronics. Abstract interpretation was applied to the analysis of asynchronous digital circuits, and partial evaluation was applied to synchronous circuits. The resulting techniques were then applied to open problems in spacecraft design in collaboration with NASA Ames and the US Air Force Office of Scientific Research, Space Vehicles Directorate.

8.1 Contributions

Our primary contributions are summarised as follows:

Transitional Logics. The work described in Chapter 3 represents a first application of abstract interpretation to the analysis of asynchronous circuits. Our abstract domain resembles a multi-valued logic and supports algebraic reasoning – soundness proofs for the logic's operators guarantee the correctness of any predictions that are made.

Achronous Analysis. As a spin-off from of an attempt to relate our transitional logics to pre-existing related work, we define a class of related *achronous* analyses, which adopt an independent attribute model whilst abstracting away relative timing information.

1st Futamura Projection in Hardware. In Chapter 4, we demonstrate the first example in hardware of a 1st Futamura projection, whereby a small CPU is partially evaluated against a ROM program image, with the effect of compiling the program into low-level hardware.

FPGA repair with SAT solvers. In work carried out in conjunction with NASA Ames, in Chapter 5 we demonstrate the first application of SAT solvers to the generation of work-around FPGA bit streams that can, in principle, allow an FPGA to continue to operate after it has sustained permanent latch-up damage from a cosmic ray impact.

Reconfigurable Manifolds. In work carried out in conjunction with the US Air Force Office of Scientific Research, Space Vehicles Directorate, in Chapter 6 we investigate the feasibility of constructing self-configuring, self-repairing, reconfigurable wiring harnesses for spacecraft.

SET Immunity in Delay-Insensitive Circuits. In Chapter 7, we apply techniques from Chapter 3 to conclude the negative result that, in general, SET immunity is not possible for any delay-insensitive circuit, however it might be constructed.

Some lesser contributions that occurred as indirect consequences of the main body of work are summarised as follows:

HarPE Hardware Description Language. Initially implemented to support the experimental work on hardware partial evaluation that was reported in Chapter 4. HarPE also proved invaluable in allowing the straightforward generation of non-clausal SAT problems. The code is in the public domain [127].

NNF-WALKSAT Non-Clausal SAT Solver. Also placed in the public domain [128], the NNF-WALKSAT solver (see also Appendix B.1) was implemented to support the work described in Chapter 5.

SET Immunity in 3-way and 5-way Voting Circuits. Though later generalised to encompass all possible DI circuits, Chapter 7 formally analyses the behaviour of voting circuits that are subjected to single-event transients and to permanent latch-up failures, concluding that whilst they are (as expected) immune to permanent latch-up faults, they are not immune to SETs except at times well-removed from transients on any input.

Non-equivalence of $(a \wedge c) \vee (a \wedge b) \vee (b \wedge c)$ with $(a \vee c) \wedge (a \vee b) \wedge (b \vee c)$. In Section 7.1.5, we prove that this widely-assumed duality does not hold when dynamic behaviour is properly accounted for.

8.2 Conclusion

The work described in this thesis began with an attempt to bring some of the advantages of modern program analysis and transformation techniques to hardware engineering.

Initially, this was largely a theoretical endeavour, resulting in the chapters of Part II of this thesis. An internship with the Robust Software Engineering group at NASA Ames during the summer of 2004 provided a clear direction, which effectively set the framework for the remainder of the work. A later collaboration with AFOSR consolidated this further – what otherwise might have remained an entirely abstract project found a real application. In particular, our SET immunity results were received with some dismay, because the space engineering community had long hoped for a generic technique that might combat radiation hardening problems once-and-for-all, and our work showed that, at least for DI circuits, this is not possible.

There can be little doubt that program analysis and transformation techniques have a lot to offer the hardware world, though the main 'customers' for this, at least in the short- to medium-term, are likely to come from the safety- and mission-critical systems world. Space electronics is a particularly extreme case, so it is perhaps not surprising that we have found the most enthusiastic support for our work from that community. Nevertheless, our techniques are generally applicable, and with the erosion of noise margins that is a natural consequence of Moore's Law's ever-decreasing device geometries, we conjecture that all VLSI devices of significant complexity will eventually require design techniques that are currently only needed for radiation-hardened circuits.

Chapter 9

Future Work and Open Questions

"It may have been small for Neil but it was a big one for a little fella like me."

– Alan Shepard's first words on stepping onto
the lunar surface, 5th February 1971.

Many opportunities for further work became apparent during the course of the work described within this thesis. For clarity, these ideas are summarised here on a per-chapter basis.

9.1 Transitional Logics

Generalisation of Refinement and Equivalence. In Section 3.7, we define refinement and equivalence relations on circuits. It appears to be possible to generalise this definition of refinement and equivalence to any abstract domain that is itself amenable to abstract interpretation.

A Transitional-Logic-Based Simulator. We have already demonstrated that our technique is potentially useful for logic simulation [131] – implementing a simulator whose underlying model is a transitional logic is an appropriate next step. Such a simulator would be capable of detecting possible hazards occurring within time slices that might otherwise be missed by a conventional discrete-time simulator.

Semi-achronous Analysis. In terms of capturing more analyses and their interrelationships within the abstract interpretation framework, it is desirable to be able to express limited timing knowledge à la Burch [30] within a non-achronous abstract model that sits somewhere between our existing $\wp(\mathbb{S})$ and $\wp(\mathbb{T})$.

Gentzen-style Proof System. We experimented with a simple Gentzen-style proof system for the 11-value clean/static transitional logic – it would seem possible to extend this to the more accurate $\wp(\mathbb{T})$, thereby potentially allowing our abstract model to be used by an automated theorem prover.

Investigating the Group-Theoretic Properties of Transitional Logics. If one regards our transitional logics as abstract algebras, they appear to be semirings. It seems that whenever a signal is forked (*i.e.,* whenever a variable appears more than once in an expression), behaviour is in some respects degenerate, though in other cases the familiar Boolean identities still hold for our logics. This has suggestive parallels with linear logic [61], which perhaps deserve closer examination.

9.2 Partial Evaluation of Synchronous Circuits

Automated Retiming/Pipelining. The timing information in Section 4.4 indicates that increasing worst-case path delays place a limit on the level of speedup that can be achieved with loop unrolling. Such circuits would almost certainly gain significantly in performance if they were pipelined and/or retimed [39], so it would be highly desirable to develop a technique that achieves this automatically, perhaps by bit-level transformation of the circuit. This would appear to be relatively straightforward for combinational circuits (and hence also any fully-unrolled synchronous circuit), but pipelining partially unrolled circuits appears to be non-trivial.

Partial Evaluation of Asynchronous Circuits. Performing PE on asynchronous circuits is fundamentally more difficult than the equivalent transformation of synchronous circuits. Rewrite rules that are perfectly safe when applied to synchronous circuits may alter the dynamic behaviour of asynchronous circuits [131, 130], introducing dangerous glitch states that could cause the circuit to function erratically, if at all. Restricting a partial evaluator only to known, safe, rewrite rules is one possible way forward which is likely to be suitable for specialisation, but no straightforward equivalent to loop unrolling appears to exist.

Register Transfer Level Partial Evaluation. Our work has concentrated on gate-level PE, which has significant advantages in terms of theoretical simplicity and generality. In principle, it should also be possible to perform partial evaluation at the register transfer level, or at the level of Verilog or VHDL source code.

Applying Abstract Interpretation to Hardware Partial Evaluation. Abstract interpretation [42, 43] is often used in combination with PE, usually to determine whether or not it is appropriate to unroll loops. Applying similar techniques, such as representing values as convex polyhedra, may make it possible, for example, to optimise a soft core CPU against a particular program *without* performing full loop unrolling – the CPU's architecture could be retained, with hardware required for unused instructions optimised away.

Extending HarPE. The current implementation of HarPE supports experimental work (as in Section 4.6) quite well, but is not yet suitable for production quality hardware design. Extending its capabilities to better match the architecture and capabilities of contemporary target platforms (FPGAs, ASICs, etc.) would be required in order to make it suitable for commercial use.

PE of an Existing Soft Core. In Section 4.6.3.3, partial evaluation of a very simple microprocessor was demonstrated. Attempting a similar experiment based on an existing soft core CPU, rather than a purpose-built example, is a logical next step. Repeating the full unrolling experiment would be of particular interest.

Power Consumption. The effects of partial evaluation on power consumption are currently unknown. Combinational specialisation is likely to result in savings proportional to the proportion of gates that are eliminated, though the effect of loop unrolling on power is harder to predict and may prove to be specific to a particular circuit.

9.3 Repairing Cosmic Ray Damage in FPGAs with Non-clausal SAT solvers

Extending the Approach to Asynchronous Circuits. The approach described in Chapter 5 assumes an underlying clocked synchronous model. We hope to apply similar techniques to the synthesis and manipulation of a wider class of circuits whose dynamic characteristics are critical, *e.g.,* self-timed circuits and globally asynchronous locally synchronous (GALS) circuits. Transitional logics are capable of reasoning about asynchronous circuits, and also about such circuits' behaviour in response to SEUs and permanent latch up faults. An approach, similar to that described in Chapter 5, but using our more accurate logics may make it feasible to automatically repair FPGA-based circuits whose asynchronous behaviour is more critical than those relying upon the synchronous model assumed here.

Evaluating Non-Clausal SAT. Our finding that non-clausal SAT solvers appear to work better for FPGA synthesis has also been noted by Greaves [62]. Finding out exactly why this is the case may be useful both within our own problem domain and also in the wider SAT solver community.

Combining Boolean SAT with Genetic Algorithms. The Boolean SAT expression necessary for local resynthesis can also be used to check the validity of solutions that have been arrived at by other means, including those generated by genetic algorithms, so it is possible that a combined approach may offer further benefits. The architectural similarity between our NNF-WALKSAT algorithm and simulated annealing is suggestive that useful results may be found in this area.

9.4 Reconfigurable Manifolds

Supporting Many-to-Many Connectivity with Permutation Networks. By their nature, permutation networks assume that inputs and outputs will be paired on a $1:1$ basis. However, some kinds of signal, particularly power busses, would benefit from a facility for being connectable to multiple end points. A trivial approach would be to connect two permutation networks in series, with a ladder of switches between adjacent wires of the connection between the networks: the first manifold would group signals together that need to be commoned, and the switch ladder would connect them together. Finally, the second permutation network would rearrange the connections to suit the desired wiring plan. A more elegant solution involving a single permutation network whose swap nodes have an ability to short their outputs together may also be feasible, though a proof that this approach is universally applicable (and, of course, less costly than the dual-network and switch ladder approach) would be beneficial.

Constructing and Trialling a Practical Reconfigurable Manifold. Many, if not all of the prerequisites for the practical construction of satellites based upon reconfigurable manifold technology are well-established, so the problem is primarily one of systems integration rather than difficult original R&D. The next step, given appropriate funding and the necessary political will, is to design and construct a practical implementation and, hopefully, to test it in space.

9.5 SET Immunity in Delay-Insensitive Circuits

A Methodology for Constructing SET-Immune Circuits. From our work, we know that SET immunity can not be achieved by any possible delay-insensitive circuit design approach. Nevertheless, a design methodology that could guarantee SET-immunity can be very realistically described as being the holy grail of the space electronics community[1]. It seems clear that no achronous model is likely to provide a solution, but this does not rule out the possibility that a more accurate model that takes relative timing into account might be an answer. If achievable, the resulting technique would also be immediately applicable to the design of future deep submicron VLSI circuits, which must overcome effects similar to SETs that are purely a consequence of poor noise margins.

[1] Discussions between the author and AFOSR engineers at Kirtland AFB in December 2005 made this abundantly clear.

Part V

Appendices

Appendix A

Extended Logics

A.1 Transitional Logics

Note that for reasons of brevity tables for $\vee$ are omitted for some of the larger logics in this appendix; all however obey de Morgan's law, so $\neg$ and $\wedge$ are sufficient to completely derive $\vee$ in such cases.

5-value ($\mathbb{T}_5$)

$\neg$	
F	T
T	F
$\uparrow$	$\downarrow$
$\downarrow$	$\uparrow$
$\bigstar$	$\bigstar$

$\wedge$	F	T	$\uparrow$	$\downarrow$	$\bigstar$
F	F	F	F	F	F
T	F	T	$\uparrow$	$\downarrow$	$\bigstar$
$\uparrow$	F	$\uparrow$	$\uparrow$	$\bigstar$	$\bigstar$
$\downarrow$	F	$\downarrow$	$\bigstar$	$\downarrow$	$\bigstar$
$\bigstar$	F	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$

$\vee$	F	T	$\uparrow$	$\downarrow$	$\bigstar$
F	F	T	$\uparrow$	$\downarrow$	$\bigstar$
T	T	T	T	T	T
$\uparrow$	$\uparrow$	T	$\uparrow$	$\bigstar$	$\bigstar$
$\downarrow$	$\downarrow$	T	$\bigstar$	$\downarrow$	$\bigstar$
$\bigstar$	$\bigstar$	T	$\bigstar$	$\bigstar$	$\bigstar$

9-value ($\mathbb{T}_9$)

$\neg$	
F_0	T_0
$F_?$	$T_?$
T_0	F_0
$T_?$	$F_?$
$\uparrow_0$	$\downarrow_0$
$\uparrow_?$	$\downarrow_?$
$\downarrow_0$	$\uparrow_0$
$\downarrow_?$	$\uparrow_?$
$\bigstar$	$\bigstar$

$\wedge$	F_0	$F_?$	T_0	$T_?$	$\uparrow_0$	$\uparrow_?$	$\downarrow_0$	$\downarrow_?$	$\bigstar$
F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0
$F_?$	F_0	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$
T_0	F_0	$F_?$	T_0	$T_?$	$\uparrow_0$	$\uparrow_?$	$\downarrow_0$	$\downarrow_?$	$\bigstar$
$T_?$	F_0	$F_?$	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	$\bigstar$
$\uparrow_0$	F_0	$F_?$	$\uparrow_0$	$\uparrow_?$	$\uparrow_0$	$\uparrow_?$	$F_?$	$F_?$	$\bigstar$
$\uparrow_?$	F_0	$F_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$F_?$	$F_?$	$\bigstar$
$\downarrow_0$	F_0	$F_?$	$\downarrow_0$	$\downarrow_?$	$F_?$	$F_?$	$\downarrow_0$	$\downarrow_?$	$\bigstar$
$\downarrow_?$	F_0	$F_?$	$\downarrow_?$	$\downarrow_?$	$F_?$	$F_?$	$\downarrow_?$	$\downarrow_?$	$\bigstar$
$\bigstar$	F_0	$F_?$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$

$\vee$	F_0	$F_?$	T_0	$T_?$	$\uparrow_0$	$\uparrow_?$	$\downarrow_0$	$\downarrow_?$	$\bigstar$
F_0	F_0	$F_?$	T_0	$T_?$	$\uparrow_0$	$\uparrow_?$	$\downarrow_0$	$\downarrow_?$	$\bigstar$
$F_?$	$F_?$	$F_?$	T_0	$T_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	$\bigstar$
T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0
$T_?$	$T_?$	$T_?$	T_0	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$
$\uparrow_0$	$\uparrow_0$	$\uparrow_?$	T_0	$T_?$	$\uparrow_0$	$\uparrow_?$	$T_?$	$T_?$	$\bigstar$
$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	T_0	$T_?$	$\uparrow_?$	$\uparrow_?$	$T_?$	$T_?$	$\bigstar$
$\downarrow_0$	$\downarrow_0$	$\downarrow_?$	T_0	$T_?$	$T_?$	$T_?$	$\downarrow_0$	$\downarrow_?$	$\bigstar$
$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	T_0	$T_?$	$T_?$	$T_?$	$\downarrow_?$	$\downarrow_?$	$\bigstar$
$\bigstar$	$\bigstar$	$\bigstar$	T_0	$T_?$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$

13-value ($\mathbb{T}_{13}$)

$\neg$	
F_0	T_0
F_+	T_+
$F_?$	$T_?$
T_0	F_0
T_+	F_+
$T_?$	$F_?$
$\uparrow_0$	$\downarrow_0$
$\uparrow_+$	$\downarrow_+$
$\uparrow_?$	$\downarrow_?$
$\downarrow_0$	$\uparrow_0$
$\downarrow_+$	$\uparrow_+$
$\downarrow_?$	$\uparrow_?$
$\bigstar$	$\bigstar$

$\wedge$	F_0	F_+	$F_?$	T_0	T_+	$T_?$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	$\downarrow_0$	$\downarrow_+$	$\downarrow_?$	$\bigstar$
F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0
F_+	F_0	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$
$F_?$	F_0	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$
T_0	F_0	$F_?$	$F_?$	T_0	T_+	$T_?$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	$\downarrow_0$	$\downarrow_+$	$\downarrow_?$	$\bigstar$
T_+	F_0	$F_?$	$F_?$	T_+	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	$\bigstar$
$T_?$	F_0	$F_?$	$F_?$	$T_?$	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	$\bigstar$
$\uparrow_0$	F_0	$F_?$	$F_?$	$\uparrow_0$	$\uparrow_?$	$\uparrow_?$	$\uparrow_0$	$\uparrow_?$	$\uparrow_?$	$F_?$	$F_?$	$F_?$	$\bigstar$
$\uparrow_+$	F_0	$F_?$	$F_?$	$\uparrow_+$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$F_?$	$F_?$	$F_?$	$\bigstar$
$\uparrow_?$	F_0	$F_?$	$F_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$F_?$	$F_?$	$F_?$	$\bigstar$
$\downarrow_0$	F_0	$F_?$	$F_?$	$\downarrow_0$	$\downarrow_?$	$\downarrow_?$	$F_?$	$F_?$	$F_?$	$\downarrow_0$	$\downarrow_?$	$\downarrow_?$	$\bigstar$
$\downarrow_+$	F_0	$F_?$	$F_?$	$\downarrow_+$	$\downarrow_?$	$\downarrow_?$	$F_?$	$F_?$	$F_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	$\bigstar$
$\downarrow_?$	F_0	$F_?$	$F_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	$F_?$	$F_?$	$F_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	$\bigstar$
$\bigstar$	F_0	$F_?$	$F_?$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$	$\bigstar$

A.2 Static/Clean Logics

Logics in this section marked * are also transitional, due to their inclusion of ↑ and ↓ states.

3-value ($\mathcal{T}_3$)

$\neg$	
C	C
S	S
★	★

$\wedge$	C	S	★
C	★	C	★
S	C	S	★
★	★	★	★

$\vee$	C	S	★
C	★	C	★
S	C	S	★
★	★	★	★

5-value ($\mathcal{T}_5$)

$\neg$	
F	T
T	F
C	C
S	S
★	★

$\wedge$	F	T	C	S	★
F	F	F	F	F	F
T	F	T	C	S	★
C	F	C	★	C	★
S	F	S	C	S	★
★	F	★	★	★	★

$\vee$	F	T	C	S	★
F	F	T	C	S	★
T	T	T	T	T	T
C	C	T	★	C	★
S	S	T	C	S	★
★	★	T	★	★	★

7-value* ($\mathcal{T}_7$)

$\neg$	
F	T
T	F
↑	↓
↓	↑
C	C
S	S
★	★

$\wedge$	F	T	↑	↓	C	S	★
F	F	F	F	F	F	F	F
T	F	T	↑	↓	C	S	★
↑	F	↑	↑	★	★	★	★
↓	F	↓	★	↓	★	★	★
C	F	C	★	★	C	★	★
S	F	S	★	★	C	S	★
★	F	★	★	★	★	★	★

$\vee$	F	T	↑	↓	C	S	★
F	F	T	↑	↓	C	S	★
T	T	T	T	T	T	T	T
↑	↑	T	↑	★	★	★	★
↓	↓	T	★	↓	★	★	★
C	C	T	★	★	★	C	★
S	S	T	★	★	C	S	★
★	★	T	★	★	★	★	★

11-value* ($\mathcal{T}_{11}$)

$\neg$	
F_0	T_0
$F_?$	$T_?$
T_0	F_0
$T_?$	$F_?$
$\uparrow_0$	$\downarrow_0$
$\uparrow_?$	$\downarrow_?$
$\downarrow_0$	$\uparrow_0$
$\downarrow_?$	$\uparrow_?$
C	C
S	S
★	★

$\wedge$	F_0	$F_?$	T_0	$T_?$	$\uparrow_0$	$\uparrow_?$	$\downarrow_0$	$\downarrow_?$	C	S	★
F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0
$F_?$	F_0	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$
T_0	F_0	$F_?$	T_0	$T_?$	$\uparrow_0$	$\uparrow_?$	$\downarrow_0$	$\downarrow_?$	C	S	★
$T_?$	F_0	$F_?$	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	★	★	★
$\uparrow_0$	F_0	$F_?$	$\uparrow_0$	$\uparrow_?$	$\uparrow_0$	$\uparrow_?$	$F_?$	$F_?$	★	★	★
$\uparrow_?$	F_0	$F_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$F_?$	$F_?$	★	★	★
$\downarrow_0$	F_0	$F_?$	$\downarrow_0$	$\downarrow_?$	$F_?$	$F_?$	$\downarrow_0$	$\downarrow_?$	★	★	★
$\downarrow_?$	F_0	$F_?$	$\downarrow_?$	$\downarrow_?$	$F_?$	$F_?$	$\downarrow_?$	$\downarrow_?$	★	★	★
C	F_0	$F_?$	C	★	★	★	★	★	★	C	★
S	F_0	$F_?$	S	★	★	★	★	★	C	S	★
★	F_0	$F_?$	★	★	★	★	★	★	★	★	★

15-value* ($\mathcal{T}_{15}$)

$\neg$	
F_0	T_0
F_+	T_+
$F_?$	$T_?$
T_0	F_0
T_+	F_+
$T_?$	$F_?$
$\uparrow_0$	$\downarrow_0$
$\uparrow_+$	$\downarrow_+$
$\uparrow_?$	$\downarrow_?$
$\downarrow_0$	$\uparrow_0$
$\downarrow_+$	$\uparrow_+$
$\downarrow_?$	$\uparrow_?$
C	C
S	S
★	★

$\wedge$	F_0	F_+	$F_?$	T_0	T_+	$T_?$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	$\downarrow_0$	$\downarrow_+$	$\downarrow_?$	C	S	★
F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0	F_0
F_+	F_0	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$
$F_?$	F_0	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$	$F_?$
T_0	F_0	$F_?$	$F_?$	T_0	T_+	$T_?$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	$\downarrow_0$	$\downarrow_+$	$\downarrow_?$	C	S	★
T_+	F_0	$F_?$	$F_?$	T_+	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	★	★	★
$T_?$	F_0	$F_?$	$F_?$	$T_?$	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	★	★	★
$\uparrow_0$	F_0	$F_?$	$F_?$	$\uparrow_0$	$\uparrow_?$	$\uparrow_?$	$\uparrow_0$	$\uparrow_?$	$\uparrow_?$	$F_?$	$F_?$	$F_?$	★	★	★
$\uparrow_+$	F_0	$F_?$	$F_?$	$\uparrow_+$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$F_?$	$F_?$	$F_?$	★	★	★
$\uparrow_?$	F_0	$F_?$	$F_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$F_?$	$F_?$	$F_?$	★	★	★
$\downarrow_0$	F_0	$F_?$	$F_?$	$\downarrow_0$	$\downarrow_?$	$\downarrow_?$	$F_?$	$F_?$	$F_?$	$\downarrow_0$	$\downarrow_?$	$\downarrow_?$	★	★	★
$\downarrow_+$	F_0	$F_?$	$F_?$	$\downarrow_+$	$\downarrow_?$	$\downarrow_?$	$F_?$	$F_?$	$F_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	★	★	★
$\downarrow_?$	F_0	$F_?$	$F_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	$F_?$	$F_?$	$F_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	★	★	★
C	F_0	$F_?$	$F_?$	C	★	★	★	★	★	★	★	★	★	C	★
S	F_0	$F_?$	$F_?$	S	★	★	★	★	★	★	★	★	★	C	★
★	F_0	$F_?$	$F_?$	★	★	★	★	★	★	★	★	★	★	★	★

$\vee$	F_0	F_+	$F_?$	T_0	T_+	$T_?$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	$\downarrow_0$	$\downarrow_+$	$\downarrow_?$	C	S	★
F_0	F_0	F_+	$F_?$	T_0	T_+	$T_?$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	$\downarrow_0$	$\downarrow_+$	$\downarrow_?$	C	S	★
F_+	F_+	F_+	$F_?$	T_+	T_+	$T_?$	$\uparrow_+$	$\uparrow_+$	$\uparrow_?$	$\downarrow_+$	$\downarrow_+$	$\downarrow_?$	★	★	★
$F_?$	$F_?$	$F_?$	$F_?$	$T_?$	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	★	★	★
T_0	T_0	T_+	$T_?$	T_0	T_+	$T_?$	T_0	T_+	$T_?$	T_0	T_+	$T_?$	C	S	★
T_+	T_+	T_+	$T_?$	T_+	T_+	$T_?$	T_+	T_+	$T_?$	T_+	T_+	$T_?$	★	★	★
$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	$T_?$	★	★	★
$\uparrow_0$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	T_0	T_+	$T_?$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	★	★	★
$\uparrow_+$	$\uparrow_+$	$\uparrow_+$	$\uparrow_?$	T_+	T_+	$T_?$	$\uparrow_+$	$\uparrow_+$	$\uparrow_?$	$\uparrow_+$	$\uparrow_+$	$\uparrow_?$	★	★	★
$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$T_?$	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	★	★	★
$\downarrow_0$	$\downarrow_0$	$\downarrow_+$	$\downarrow_?$	T_0	T_+	$T_?$	$\uparrow_0$	$\uparrow_+$	$\uparrow_?$	$\downarrow_0$	$\downarrow_+$	$\downarrow_?$	★	★	★
$\downarrow_+$	$\downarrow_+$	$\downarrow_+$	$\downarrow_?$	T_+	T_+	$T_?$	$\uparrow_+$	$\uparrow_+$	$\uparrow_?$	$\downarrow_+$	$\downarrow_+$	$\downarrow_?$	★	★	★
$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	$T_?$	$T_?$	$T_?$	$\uparrow_?$	$\uparrow_?$	$\uparrow_?$	$\downarrow_?$	$\downarrow_?$	$\downarrow_?$	★	★	★
C	C	★	★	C	★	★	★	★	★	★	★	★	★	C	★
S	S	★	★	S	★	★	★	★	★	★	★	★	★	C	★
★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★

A.3 Logics from Related Work

Boolean logic ($\mathbb{B}$)

$\neg$	
F	T
T	F

$\wedge$	F	T
F	F	F
T	F	T

$\vee$	F	T
F	F	T
T	T	T

Ternary logic ($\mathbb{B}_3$)

$\neg$	
F	T
T	F
★	★

$\wedge$	F	T	★
F	F	F	F
T	F	T	★
★	F	★	★

$\vee$	F	T	★
F	F	T	★
T	T	T	T
★	★	T	★

Quaternary logic

$\neg$	
F	T
T	F
$\uparrow$	$\downarrow$
$\downarrow$	$\uparrow$

$\wedge$	F	T	$\uparrow$	$\downarrow$
F	F	F	F	F
T	F	T	$\uparrow$	$\downarrow$
$\uparrow$	F	$\uparrow$	$\uparrow$	★
$\downarrow$	F	$\downarrow$	★	$\downarrow$

$\vee$	F	T	$\uparrow$	$\downarrow$
F	F	T	$\uparrow$	$\downarrow$
T	T	T	T	T
$\uparrow$	$\uparrow$	T	$\uparrow$	★
$\downarrow$	$\downarrow$	T	★	$\downarrow$

Appendix B

Non-Clausal SAT Solvers for Hardware Analysis

The work presented in this appendix was carried out in support of our work on the automated repair of cosmic ray damage in FPGAs that was reported in Chapter 5. It is included in the interests of completeness, and is presented as an appendix in order to avoid distraction from the narrative content of the body of the thesis.

Our non-clausal SAT solver, NNF-WALKSAT, was implemented from scratch because existing clausal SAT libraries were found to be unsatisfactory, and also due to the lack of a publicly available non-clausal alternative. Our approach is not claimed as novel, though its implementation is new – extending WALKSAT to encompass non-clausal problems was previously suggested by Walser in 1997 [144].

B.1 NNF-based Non-Clausal SAT Solvers

An NNF-compliant SAT solver was implemented as a C++ library for the purposes of supporting experimentation on FPGA bit stream synthesis and repair (see also Chapter 5).

NNF-GSAT Initially, the relatively simple GSAT algorithm [114] was adopted. Though originally intended for use with CNF problems, it was relatively straightforward to adapt the algorithm for use with NNF. In outline, the algorithm works as follows:

1. Initialise the variables to random initial values
2. Check to see whether the current variable values satisfy the expression completely. If so, a solution has been found, so the loop terminates.
3. For each variable, flip its state (*i.e.,* change 0 to 1 and vice-versa), then note the number of subexpressions that are *satisfied* (*i.e.,* evaluate to 1) as a consequence. Return each variable to its initial state after each count.
4. Choose the variable that most increases the number of satisfied subexpressions, then flip it permanently.

5. If a predetermined number of attempts has been exceeded, go back to step 1, otherwise go to step 2.

In practice, this algorithm is a little too simplistic, and requires some extra heuristics in order to prevent it from becoming trivially stuck in local minima. Initial results indicated that, though slow, GSAT was actually surprisingly effective, given its extreme simplicity.

NNF-WALKSAT In order to improve upon the performance of NNF-GSAT, the WALKSAT algorithm [112] was similarly adapted for use with NNF. The basic WALKSAT algorithm may be summarised as follows:

1. Choose a clause at random that is currently unsatisfied
2. Depending on whether a random number exceeds the current $temperature$ parameter, either:
   (a) Randomly choose a variable that appears within the clause and flip it, or
   (b) Attempt flipping each variable that appears within the clause in turn, noting the number of unsatisfied clauses that result in each case, then choose the one flip that results in the lowest number of unsatisfied clauses. This is referred to hereinafter as a *greedy flip*.

WALKSAT is superficially similar to GSAT, but due to the need on each iteration only to enumerate the variables within a single clause rather than all unbound variables in the entire expression, it is generally much faster whilst retaining roughly equivalent power. As with GSAT, the basic WALKSAT algorithm is intended for use with expressions in CNF, so it was necessary to extend and modify it to deal with the more general NNF case.

The resulting SAT solver, is able to solve the majority of our test cases rapidly, even where multiple stuck-at faults were simulated. The basic WALKSAT algorithm required some modifications and extra heuristics, due to a bad tendency to get stuck in local minima. The extensions we used are summarised as follows:

Supporting terms as well as clauses In an expression in CNF, one single outer *term* encapsulates possibly many clauses, and clauses may only contain variables or their negations, not terms. NNF relaxes this somewhat, in that terms may contain clauses and vice-versa, with the only significant restriction in comparison with general Boolean expressions being the requirement that negation may only appear adjacent to a variable.

In NNF-WALKSAT, we perform a preprocessing stage, whereupon for each term and each clause, the list of variables contained within them is cached. Variables that appear directly within a term are regarded as equivalent to singleton clauses containing only that variable.

Pre-optimisation of the NNF expression A simple pre-optimisation pass is performed first, such that clauses that are of the form $a \vee \neg a \vee b \vee \ldots$ are replaced with $true$, terms of the form $a \wedge \neg a \wedge b \wedge \ldots$ are replaced with $false$, then any remaining constants are evaluated out and folded into the expression.

Giving clauses close to the root preference When randomly selecting a clause, preference is given to clauses that appear close to the root of the expression tree, on the basis that such variables are more likely to have a wide impact, so it is appropriate to try to make an estimate of their value early.

Super-flips We add a third kind of flip, in addition to random flips and greedy flips. A *super-flip* requires trying all possible combinations of variables, then selecting the combination resulting in the best score. Since this algorithm has a complexity of $O(2^N)$, it makes sense to set a fairly low upper limit on the number of variables to which it can be applied – in our current implementation, super-flips are only attempted for clauses with 8 variables or less.

Super-flips do not appear to make a big difference to many problems, but in some cases they appear to make it possible to find a solution quickly when the standard algorithm gets stuck for a long time, even when the probability of performing a super-flip is very small. A useful heuristic appears to be to have the probability $\frac{p}{2^N}$ of performing a super-flip, where p is an empirically-derived constant[1].

Dynamic control of the temperature parameter The original WALKSAT algorithm suggests choosing between random and greedy flips with a probability of approximately 0.5. Our finding was that this does not work for expressions in NNF – though random flips are essential for avoiding local minima, they often significantly increase the number of unsatisfied clauses in the expression as a whole. We found that a random flip probability in the range $0.01..0.1$ normally works, but found that the ideal value was highly dependent on the expression being solved. If the probability is too low, the solver gets stuck in local minima, but if it is too high, the algorithm does not converge on a solution at all.

Our implementation dynamically varies the temperature in accordance with the following heuristics:

1. If the most recent flip reduced the number of unsatisfied clauses, reduce the temperature exponentially.
2. If the same variable is flipped twice in succession, suggesting that a local minimum has been encountered, increase the temperature by a (fairly large) additive constant.
3. Otherwise, very gradually move the temperature toward a default (small) value (0.001 in our implementation).

This approach works well for most of the SAT problems we have examined – early in the run, the temperature is kept very low by rule 1, which makes it possible to converge quickly on a possible result. In many cases, a solution will fall out of this initial attempt immediately. However, if the SAT solver gets stuck in a local minimum, this frequently results in the same variable being toggled repeatedly – rule 2 picks up on this, increasing the temperature, thereby pushing the variable bindings away from the minimum.

[1] Our implementation uses $p = 1$.

Retries Whenever a set of variable bindings is found that results in an improvement to the number of unsatisfied clauses, a snapshot of these bindings is taken for later use. If no improvement beyond this snapshot is seen for a predetermined number of attempts (1000 in our implementation), the last snapshot reverted to, giving the search procedure another attempt at finding an improved result. In a significant proportion of cases, this leads to a solution being found after a small number of retries.

Restarts If retrying does not succeed after a large number of attempts (5 in our implementation), this generally means that the solver is stuck in a local minimum that it can not climb out of by normal means. In this case, we reset the variable bindings to new, unrelated values then start again. By experimentation, it was found that determining these values according to the following algorithm is beneficial:

1. Initially, set all variables to 0
2. On the first restart, set all variables by counting the number of times that each appears negated and non-negated, choosing a value likely to satisfy the greatest number of clauses in each case
3. On the second restart, set all variables to 1
4. On all subsequent restarts, set all variables randomly

This can be visualised as initially trying one extreme of the problem space, then a case roughly in the middle of the problem space, then the other extreme, and then finally trying cases at random until a solution is found.

This approach works well as a general purpose SAT solver, although in our application we find it beneficial to first attempt an initial variable set initialised to the existing FPGA bit stream – in many cases, this proves to be a considerable speedup, whilst also increasing the percentage of successful runs.

Bibliography

[1] Annual SAT competition. Web site: http://www.satcompetition.org/.

[2] PCI special interest group web site. http://www.pcisig.com/.

[3] SAT Live! Web site: http://satlive.org/.

[4] SystemC web site. http://www.systemc.org/.

[5] SystemVerilog web site. http://www.systemverilog.org/.

[6] Universal serial bus specification. Revision 2.0, http://www.usb.org/, 2000.

[7] *Excalibur Device Overview Data Sheet, V2.0*. Altera, 2002. DS-EXCARM-2.0.

[8] *Quartus II Development Software Handbook, V4.0*. Altera, 2004.

[9] *Actel web site*. Actel, 2006. http://www.actel.com/.

[10] *AD8116 - 200 MHz, 16 $\times$ 16 Buffered Video Crosspoint Switch*. Analog Devices, 2006. http://www.analog.com/en/prod/0,2877,768

[11] *High Performance Crossbar Switch for Virtex-II and Virtex-II Pro FPGAs*. Xilinx, 2006. www.xilinx.com/esp/xbarswitch.htm.

[12] ABRAMOV, S. A., AND GLÜCK, R. Principles of inverse computation and the universal resolving algorithm. In *The Essence of Computation, Complexity, Analysis, Transformation: Essays Dedicated to Neil D. Jones on occasion of his 60th birthday*, T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, Eds., no. 2566 in Lecture Notes in Computer Science. Springer-Verlag, 2002, pp. 269–295.

[13] AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. An O(n log n) sorting network. *Combinatorica 3(1)* (1983), 1–19.

[14] ARVIND. Bluespec: A language for hardware design, simulation, synthesis and verification (Invited Talk),. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03)* (2003), p. 249.

[15] BAKER, D. N., MASON, G. M., FIGUEROA, O., COLON, G., WATZIN, J. G., AND ALEMAN, R. M. An overview of the Solar, Anomalous, and Magnetospheric Particle Explorer (SAMPEX) mission. *IEEE Transactions on Geoscience and Remote Sensing 31* (May 1993), 531–541.

[16] BATCHER, K. E. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference 32* (1968), pp. 307–314.

[17] BENEŠ, V. E. Proving the rearrangeability of connecting networks by group calculations. *Bell System Tech. J.*, 45 (1975), 421–434.

[18] BERRY, G. Esterel on hardware. *Philosophical transactions of the Royal Society of London A 339* (1992), 87–104.

[19] BERRY, G. The constructive semantics of Pure Esterel. Available from http://www.esterel-technologies.com/, 2000.

[20] BERRY, G. The foundations of Esterel. In *Proof, Language and Interaction: Essays in honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 2000.

[21] BIERE, A., CIMATTI, A., CLARKE, E. M., STRICHMAN, O., AND ZHU, Y. Bounded model checking. *Advances in Computers 58* (2003).

[22] BJESSE, P., CLAESSEN, K., SHEERAN, M., AND SINGH, S. Lava: Hardware design in Haskell. In *International Conference on Functional Programming* (1998), ACM Press.

[23] BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. Design and implementation of a special-purpose static program analyser for safety critical real-time embedded systems. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays dedicated to Neil D. Jones, LNCS 2566*, T. Mogensen, D. A. Schmidt, and I. H. Sudborough, Eds. Springer Verlag, 2002.

[24] BLUESPEC. Bluespec ESL synthesis extensions web site. http://www.bluespec.com/products/ESLSynthesisExtensions.htm.

[25] BOHRINGER, K. F. A docking system for microsatellites based on microelectromechanical system actuator arrays. Tech. Rep. AFRL-VS-TR-2000-1099, US Air Force Research Laboratory, Space Vehicles Directorate, September 2000.

[26] BREI, D., AND CLEMENT, J. Proof-of-concept investigation of active velcro for smart attachment mechanisms. Tech. Rep. AFRL-VS-TR-2000-1097, US Air Force Research Laboratory, Space Vehicles Directorate, September 2000.

[27] BREI, D., AND CLEMENT, J. Velcro for smart attachment mechanisms. Tech. Rep. AFRL-VS-TR-2001-1104, US Air Force Research Laboratory, Space Vehicles Directorate, August 2001.

[28] BRZOZOWSKI, J. A., AND ÉSIK, Z. Hazard algebras. *Formal Methods in System Design 23*, 3 (2003), 223–256.

[29] BRZOZOWSKI, J. A., AND GHEORGIU, M. Gate circuits in the algebra of transients. *EDP Sciences* (2004). To appear.

[30] BURCH, J. R. Delay models for verifying speed-dependent asynchronous circuits. In *Proc. ICCD* (1992), IEEE, pp. 270–274.

[31] CASPI, P., PILAUD, D., HALBWACH, N., AND PLAICE, J. LUSTRE: a declaritive language for programming synchronous systems. In *Proc. 14th ACM Conference on Principles of Programming Languages* (1987), ACM Press, pp. 178–188.

[32] ÇAM, H. Rearrangeability of $(2n-1)$-stage shuffle-exchange networks. *SIAM J. Comput. 32*, 3 (2003), 557–585.

[33] CELOXICA. Handel-C language reference manual. Available from http://www.celoxica.com/.

[34] CHAPIRO, D. M. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[35] CLAESSEN, K., SHEERAN, M., AND SINGH, S. The design and verification of a sorter core. In *Proc. CHARME'01, LNCS 2144* (2001), Springer-Verlag.

[36] CLARKE, A. C. Peace time uses for V2. *Letters to the Editor, Wireless World* (February 1945), 58.

[37] CLEMENT, J. W., AND BREI, D. E. Proof-of-concept investigation of Active Velcro for smart attachment mechanisms. In *In Proc. 42nd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference and Exhibit* (2001). AIAA Paper 2001-1503 (AIAA Accession number 25238).

[38] CLOS, C. A study of non-blocking switching networks. *Bell System Technical Journal 32*, 2 (1953), 406–424.

[39] CONG, J., AND WU, C. FPGA synthesis with retiming and pipelining for clock period minimization of sequential circuits. In *Proceedings of the 34th annual conference on Design automation conference* (1997), ACM Press, pp. 644–649.

[40] COOK, S. The complexity of theorem proving procedures. In *Proc. 3rd Annual ACM Symposium on Theory of Computing* (1971), pp. 151–158.

[41] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. Chapter 27: Sorting networks. In *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990, pp. 704–724.

[42] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, California, 1977), ACM Press, New York, NY, pp. 238–252.

[43] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, 1979), ACM Press, New York, NY, pp. 269–282.

[44] CUNNINGHAM, P. A. *Verification of Asynchronous Circuits*. PhD thesis, University of Cambridge, 2002.

[45] CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, F. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (1991), 451–490.

[46] DAVIS, M., LOGEMANN, G., AND LOVELAND, D. A machine program for theorem proving. *Communications of the ACM 5*, 7 (July 1962), 394–397.

[47] DAVIS, M., AND PUTNAM, H. A computing procedure for quantification theory. *Journal of the ACM 7*, 3 (July 1960), 201–215.

[48] DEAN, M., WILLIAMS, T., AND DILL, D. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In *Advanced Research in VLSI* (1991), C. H. Séquin, Ed., MIT Press, pp. 55–70.

[49] DENNARD, R. Field effect transistor memory. US Patent No. 3,387,286, 1968.

[50] DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.

[51] ESTEREL TECHNOLOGIES. The Esterel v7 reference manual – initial ieee standardisation proposal. Available from http://www.esterel-technologies.com/, November 2005.

[52] FORTE DESIGN SYSTEMS. Corporate web site. http://www.forteds.com/.

[53] FOUST, J. Smallsats and standardization. *The Space Review* (2005).

[54] FUTAMURA, Y. Partial evaluation of computation process – an approach to a compiler-compiler. In *Systems, Computers, Control* (1971), vol. 2 issue 5, pp. 45–50.

[55] GAUBATZ, D. A. *Logic Programming Analysis of Asynchronous Digital Circuits*. PhD thesis, University of Cambridge, 1991.

[56] GIACOBAZZI, R., AND MASTROENI, I. Domain compression for complete abstractions. In *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'03)* (2003), vol. 2575 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 146–160.

[57] GIACOBAZZI, R., AND QUINTARELLI, E. Incompleteness, counterexamples and refinements in abstract model-checking. In *Proc. 8th International Static Analysis Symposium (SAS'01)* (2001), vol. 2126 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 356–373.

[58] GIACOBAZZI, R., AND RANZATO, F. Completeness in abstract interpretation: A domain perspective. In *Proc. of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)* (1997), M. Johnson, Ed., vol. 1349 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 231–245.

[59] GIACOBAZZI, R., AND RANZATO, F. Refining and compressing abstract domains. In *Proc. of the 24th International Colloquium on Automata, Languages, and Programming (ICALP'97)* (1997), R. G. P. Degano and A. Marchetti-Spaccamela, Eds., vol. 1256 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 771–781.

[60] GIACOBAZZI, R., RANZATO, F., AND SCOZZARI, F. Making abstract interpretations complete. *Journal of the ACM 47*, 2 (2000), 361–416.

[61] GIRARD, J.-Y. Linear logic. *Theor. Comput. Sci. 50* (1987), 1–102.

[62] GREAVES, D. J. Direct synthesis of logic using a SAT solver. Unpublished research note, available at http://www.cl.cam.ac.uk/users/djg/wwwhpr/dslogic.html, 2004.

[63] GUCCIONE, S. A., LEVI, D., AND SUNDARARAJAN, P. JBits: A Java-based interface for reconfigurable computing. In *Proc. 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)* (1999), NASA, available at http://klabs.org/mapld/index.htm.

[64] GUSSENHOVEN, M., MULLEN, E., AND BRAUTIGAM, D. Improved understanding of the Earth's radiation belts from the CRRES satellite. *IEEE Trans. on Nuclear Science 43*, 2 (April 1996), 353–368.

[65] HYMANS, C. Checking safety properties of behavioral VHDL descriptions by abstract interpretation. In *9th International Static Analysis Symposium (SAS'02)* (2002), vol. 2477 of *Lecture Notes in Computer Science*, Springer, pp. 444–460.

[66] IEEE P1394 WORKING GROUP. *IEEE Std 1394-1995 High Performance Serial Bus*. IEEE, 1995.

[67] IEEE P1394A WORKING GROUP. *IEEE Std 1394a-2000 High Performance Serial Bus – Amendment 1*. IEEE, 2000.

[68] IEEE P1394B WORKING GROUP. *IEEE Std 1394b-2002 High Performance Serial Bus – Amendment 2*. IEEE, 2002.

[69] IEEE P1800 WORKING GROUP. *IEEE Std 1364-2005 Verilog Hardware Description Language*. IEEE, 2005.

[70] IEEE P1800 WORKING GROUP. *IEEE Std 1800-2005 System Verilog: Unified Hardware Design, Specification and Verification Language*. IEEE, 2005.

[71] JENSEN, T. P. Clock analysis of synchronous dataflow. In *Proc. PEPM'95* (1995), ACM Press, pp. 156–167.

[72] JONES, N., GOMARD, C., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ, Prentice Hall, 1993.

[73] JONES, N. D., AND MUCHNICK, S. Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra. In *21st Symposium on Foundations of Computer Science* (1980), IEEE, pp. 185–190.

[74] JONES, S. L. P. Haskell 98 report (Special Issue). *J. Funct. Program. 13*, 1 (2003).

[75] JOSHI, P. B. On-orbit asssembly of a universally interlocking modular spacecraft (7225-020). Tech. Rep. NASA SBIR 2003 Solicitation Proposal 03- II F5.03-8890, NASA, 2003.

[76] KAM, J. B., AND ULLMAN, J. D. Monotone dataflow analysis frameworks. In *Acta Informatica* (September 1977), vol. 7, pp. 305–317.

[77] KATZ, R. A scientific study of the problems of digital engineering for space flight systems, with a view to their practical solution. *http://klabs.org/*.

[78] KILDALL, G. A. A unified approach to global program optimization. In *Proc. POPL* (1973), ACM Press, pp. 194–206.

[79] KLEENE, S. C. *Introduction to metamathematics*. North Holland, Amsterdam, 1962.

[80] KNAPP, D. W. *Behavioural synthesis: digital system design using the Synopsys Behavioral Compiler*. Prentice Hall, 1996.

[81] KNUTH, D. E. Section 5.3.4: Networks for sorting. In *The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition*. Addison-Wesley, 1997, pp. 219–247.

[82] KUNG, D. S. Hazard-non-increasing gate-level optimization algorithms. In *Proc. ICCAD* (1992), IEEE, pp. 631–634.

[83] LARCHEV, G., AND LOHN, J. D. Hardware-in-the-loop evolution of a 3-bit multiplier. In *Proc. 12th Annual IEEE Symposium on Field Programmable Custom Computing Machines, FCCM-2004* (2004), IEEE Computer Society, pp. 277–278.

[84] LAWRIE, D. H. Access and alignment of data in an array processor. *IEEE Transactions on Computers 25* (1976), 1145–1155.

[85] LOHN, J. D., LARCHEV, G., AND DEMARA, R. F. Evolutionary fault recovery in a Virtex FPGA using a representation that incorporates routing. In *Proc. IPDPS 2003* (2003), IEEE Computer Society.

[86] LOHN, J. D., LARCHEV, G., AND DEMARA, R. F. A genetic representation for evolutionary fault recovery in Virtex FPGAs. In *Proc. ICES 2003, LNCS 2606* (2003), Springer-Verlag, pp. 47–56.

[87] LOMBARDI, L. Incremental computation. In *Advances in Computers, vol. 8*, F. Alt and M. Rubinoff, Eds. New York, Academic Press, 1967, pp. 247–333.

[88] LOMBARDI, L., AND RAPHAEL, B. Lisp as the language for an incremental computer. In *The Programming Language Lisp: Its Operation and Applications* (1964), E. Berkeley and D. Bobrow, Eds., Cambridge, MA, MIT Press, pp. 204–219.

[89] LORENTZ, H. A. Electromagnetic phenomena in a system moving with any velocity less than that of light. *Proc. Acad. Science Amsterdam IV* (1904), 669–678.

[90] LYKE, J., WILSON, W., AND CONTINO, P. MEMS-based reconfigurable manifold. In *Proc. MAPLD* (2005), NASA, available at http://klabs.org/mapld/index.htm.

[91] MCKAY, N., MELHAM, T., SUSANTO, K. W., AND SINGH, S. Dynamic specialisation of XC6200 FPGAs by partial evaluation. In *IEEE Symposium on FPGAs for Custom Computing Machines* (1998), K. L. Pocek and J. M. Arnold, Eds., IEEE Computer Society, pp. 308–309.

[92] MCKAY, N., AND SINGH, S. Dynamic specialisation of XC6200 FPGAs by partial evaluation. In *Field-Programmable Logic and Applications: From FPGAs to Computing Paradigm: 8th International Workshop, FPL'98, Estonia, 1998* (1998), R. W. Hartenstein and A. Keevallik, Eds., vol. 1482 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 298–307.

[93] MCPHERSON, D. A., AND SCHOBER, W. R. Spacecraft charging at high altitudes: The SCATHA satellite program. In *Proc. AIAA Symposium on Spacecraft Charging by Magnetospheric Plasmas* (1996), vol. 47 of *Progress in Astronautics and Aeronautics*, MIT Press, pp. 15–30.

[94] MEALY, G. H. A method for synthesizing sequential circuits. In *Bell System Technical Journal* (1955), vol. 34, pp. 1045–1079.

[95] MONNIAUX, D. Abstract interpretation of probabilistic semantics. In *Seventh International Static Analysis Symposium (SAS'00)* (2000), vol. 1824 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 322–339. Extended version on the author's web site, currently http://www.di.ens.fr/ monniaux/.

[96] MONNIAUX, D. An abstract Monte-Carlo method for the analysis of probabilistic programs (extended abstract). In *28th Symposium on Principles of Programming Languages (POPL '01)* (2001), ACM Press, pp. 93–101.

[97] MONNIAUX, D. *Analyse de programmes probabilistes par interprétation abstraite*. Thèse de doctorat, Université Paris IX Dauphine, 2001. Résumé étendu en français. Contents in English.

[98] MOORE, S., MULLINS, R., AND TAYLOR, G. The Springbank test chip. In *Proc. 12th UK Async. Forum* (June 2002).

[99] MORELLI, G. Coralled: Get hold of wire delays. *Electronic Design News, September 25, 2003*, pp. 37–46.

[100] MOY, J. RFC 2328: OSPF Version 2. *IETF* (1998).

[101] MYCROFT, A. Completeness and predicate-based abstract interpretation. In *Proc. ACM conf. on Partial Evaluation and Program Manipulation* (1993), pp. 179–185.

[102] MYCROFT, A., AND JONES, N. D. A relational framework for abstract interpretation. In *Lecture Notes in Computer Science: Proc. Copenhagen workshop on programs as data objects* (1984), vol. 215, Springer-Verlag.

[103] MYCROFT, A., AND SHARP, R. W. Hardware synthesis using SAFL and application to processor design. In *Lecture Notes in Computer Science: Proc. CHARME'01* (2001), vol. 2144, Springer-Verlag.

[104] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of Program Analysis*. Springer-Verlag, 1999.

[105] OPEN SYSTEMC INITIATIVE. *Draft Standard SystemC Language Reference Manual*. OSCI, April 2005. Available from http://www.systemc.org/.

[106] PAGE, I., AND LUK, W. Compiling Occam into FPGAs. In *FPGAs*, W. Moore and W. Luk, Eds. Abingdon EE&CS Books, 1991, pp. 271–283.

[107] REKHTER, Y., LI, T., AND HARES, S. RFC 4271: a Border Gateway Protocol 4 (BGP-4). *IETF* (2006).

[108] RINTANEN, J. Improvements to the evaluation of quantified boolean formulae. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages* (1999), T. Dean, Ed., Morgan Kaufmann, pp. 1192–1197.

[109] RØINE, P. T. Building fast bundled data circuits with a specialized standard cell library. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (November 1994), pp. 134–143.

[110] SANGUINETTI, J., AND PURSLEY, D. High-level modeling and hardware implementation with general purpose languages and high-level synthesis. In *Proc. Ninth IEEE/DATC Electronic Design Processes Workshop* (2002).

[111] SCHMITT, O. H. A thermionic trigger. *Jour. Sci. Instr. 15*, 1 (1938), 24.

[112] SELMAN, B., KAUTZ, H., AND COHEN, B. Noise strategies for improving local search. In *Proc. 12th National Conference on Artificial Intelligence, AAAI'94* (1994), vol. 1, MIT Press, pp. 337–343.

[113] SELMAN, B., KAUTZ, H., AND COHEN, B. Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge* (1996), D. S. Johnson and M. A. Trick, Eds., vol. 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, AMS.

[114] SELMAN, B., LEVESQUE, H. J., AND MITCHELL, D. A new method for solving hard satisfiability problems. In *Proc. 10th National Conference on Artifical Intelligence, AAAI'92* (1992), pp. 440–446.

[115] SHAND, B. N. Trust for resource control: Self-enforcing automatic rational contracts between computers. Tech. Rep. UCAM-CL-TR-600, University of Cambridge, Computer Laboratory, Available via http://www.cl.cam.ac.uk/TechReports/, Aug. 2004.

[116] SHEERAN, M. Puzzling permutations. In *Proc. Glasgow Functional Programming Workshop* (1996).

[117] SINGH, S., AND JAMES-ROXBY, P. Lava and JBits: From HDL to bitstream in seconds. In *Proc. 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)* (2001).

[118] SPARSØ, J., AND FURBER, S., Eds. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Springer-Verlag, 2002.

[119] STEFFEN, B., JAY, C. B., AND MENDLER, M. Compositional characterisation of program properties. *Informatique Théorique et Applications (AFCET) 26*.

[120] STOICA, A., ARSLAN, T., KEYMEULEN, D., DUONG, V., GUO, X., ZEBULUM, R., FERGUSON, I., AND DAUD, T. Evolutionary recovery of electronic circuits from radiation induced faults. In *Proc. IEEE Conference on Evolutionary Computation* (2004), IEEE Computer Society.

[121] STONE, H. S. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers 20*, 6 (1975), 57–65.

[122] STROUSTRUP, B. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, Reading, Massachusetts, USA, 1997.

[123] SUH, J. W., DARLING, R. B., BOHRINGER, K. F., DONALD, B., BALTES, H., AND KOVACS, G. T. A. SMOS integrated ciliary actuator array as a general-purpose micromanipulation tool for small objects. *IEEE Journal of Microelectromechanical Systems* (1999).

[124] SYNOPSYS. Corporate web site. http://www.synopsys.com/.

[125] SYNOPSYS. System Studio web site. http://www.synopsys.com/products/cocentric_studio/.

[126] THIFFAULT, C., BACCHUS, F., AND WALSH, T. Solving non-clausal formulas with DPLL search. In *10th International Conference on Principles and Practice of Constraint Programming (CP-2004)* (2004), vol. 3258 of *Lecture Notes in Computer Science*, Springer-Verlag.

[127] THOMPSON, S. HarPE web site. http://harpe.findatlantis.com/.

[128] THOMPSON, S. NNF-WALKSAT web site. http://nnf-walksat.findatlantis.com/.

[129] THOMPSON, S. Hardware compilation as an alternative computation architecture. Master's thesis, University of Teesside, 1991.

[130] THOMPSON, S., AND MYCROFT, A. Abstract interpretation of combinational asynchronous circuits. In *11th International Static Analysis Symposium (SAS'04)* (2004), R. Giacobazzi, Ed., vol. 3148 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 181–196.

[131] THOMPSON, S., AND MYCROFT, A. Sliding window logic simulation. In *15th UK Asynchronous Forum* (2004), Cambridge. Available from http://findatlantis.com/.

[132] THOMPSON, S., AND MYCROFT, A. Abstract interpretation in space: SET immunity of majority voting logic. In *Proc. APPSEM II Workshop* (September 2005). Available from http//findatlantis.com/.

[133] THOMPSON, S., AND MYCROFT, A. Abstract interpretation of combinational asynchronous circuits (Extended Version). *Science of Computer Programming* (2006). To appear.

[134] THOMPSON, S., AND MYCROFT, A. Bit-level partial evaluation of synchronous circuits. In *Proc. ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)* (January 2006), ACM Press.

[135] THOMPSON, S., AND MYCROFT, A. Self-healing reconfigurable manifolds. In *Proc. Designing Correct Circuits (DCC '06)* (March 2006). Available from http//findatlantis.com/.

[136] THOMPSON, S., MYCROFT, A., BRAT, G., AND VENET, A. Automatic in-flight repair of FPGA cosmic ray damage. In *Proc. 1st Disruption in Space Symposium* (July 2005). Available from http//findatlantis.com/.

[137] TSEITIN, G. On the complexity of proofs in propositional logics. *Automation of reasoning: classical papers in computational logic 1967–1970 2* (1983). Originally published 1970.

[138] TURNILL, R. *The Moonlandings: an eyewitness account*. Cambridge University Press, 2003.

[139] VANDEVOORDE, D., AND JOSUTTIS, N. M. *C++ Templates – The Complete Guide*. Addison-Wesley, 2002.

[140] VELDHUIZEN, T. Using C++ template metaprograms. *C++ Report 7*, 4 (May 1995), 36–43. Reprinted in C++ Gems, ed. Stanley Lippman, Cambridge University Press.

[141] VELDHUIZEN, T. L. C++ templates as partial evaluation. In *Proc. PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio* (Jan. 1999), ACM Press, pp. 13–18.

[142] VIENS, M. J. Outgassing data for selecting spacecraft materials online. Web site, http://outgassing.nasa.gov/.

[143] VON NEUMANN, J. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies* (1956), 43–98.

[144] WALSER, J. P. Solving linear pseudo-Boolean constraint problems with local search. In *Proc. 14th National Conference on Artificial Intelligence* (1997), AAAI Press.

Glossary of Terms and Symbols

$=$ Strong equivalence (between traces), 72

$D/\sim$ The set of equivalence classes in D with respect to the *equivalence relation* $\sim$, 63

$\eqsim$ Comparability (between traces), 72

Γ Guard expression (in HarPE semantics), 85

$\wedge^\sharp$ Abstract AND, 65

$\Box^\sharp$ Inertial delay (abstract), 65

$\neg^\sharp$ Abstract NOT, 65

$\Delta^\sharp$ Transmission line delay (abstract), 65

α Abstraction function, see *Galois connection*, 63

β Deterministic abstraction function (typically used to help define α more conveniently), see *Galois connection*, 63

$\bigstar$ The least upper bound element of a finitary *transitional logic*, 70

$\simeq$ Weak equivalence (between traces), 72

C The value representing 'unknown, but can never glitch' in *static-clean logics*, 71

$\wedge$ Concrete AND, 65

$\wedge_{complete}$ A fully α- and γ-complete version of $\wedge$, constructed by composition with Δ, 68

$\neg$ Concrete NOT, 65

$\mathbb{R}$ see *concrete time*, 59

$\mathcal{V}_3'$ The dual of $\mathcal{V}_3$, constructed by interchanging $\wedge$ and $\vee$, 152

γ Concretisation function, see *Galois connection*, 63

$\Box$ Inertial delay (concrete), 65

$\langle \ldots \rangle$ see *trace notation*, 60

Achronous Analysis an analysis technique for asynchronous circuits that abstracts away notions of absolute time, 56

Achrony assumption A hardware analysis technique in which timing is calculated assuming an *independent attribute model*, 73

Actel Lower volume, niche FPGA manufacturer, specialising in radiation tolerant, one-time programmable FPGAs, 105

Altera Major FPGA manufacturer, main competitor to Xilinx, also specialising in high performance SRAM-based COTS FPGAs, 105

Antifuse FPGA A one-time programmable, non-reconfigurable FPGA architecture pioneered by Actel, 105

Asynchronous Circuit a digital circuit that is not designed to strictly synchronous design rules, see also *synchronous circuit*, 55

Backplane Usually constructed as a *PCB,* a backplane integrates all of the wiring necessary to connect together cards that are inserted into a *card frame.* See also *passive backplane.*, 118

Bit stream The information required to configure an FPGA, 105

CMOS Complementary Metal Oxide Semiconductor, the most common VLSI fabrication technology, 105

CNF Conjunctive Normal Form, 111

COTS Commercial, Off The Shelf , 105

Card frame A frame into which individual cards (PCBs) may be slotted. Normally, a *backplane* at the rear of the card frame is populated with a series of sockets, allowing cards to plug in and function without any need for physical wiring, 118

Coarse Sun Tracker Often quite simplistic, constructed from photocells pointing in four directions, a coarse sun tracker's purpose is to provide a quick-and-dirty estimate of a satellite's orientation. Typically, when a satellite is recovering from a failure that might have caused it to tumble, the coarse sun tracker would be used to help get the satellite pointing in roughly the correct direction, at which point the star tracker would take over and help perform fine positioning, 120

Commercial-grade devices General purpose (non military, non radiation hard) devices sold on the open market, 105

Completeness In *abstract interpretation,* an abstract prediction may be said to be complete if and only if the concrete model is capable of any possible predicted behaviour, 58

Concrete domain From *abstract interpretation,* the domain D that (usually) more closely models reality, see also *Galois connection*, 63

Concrete time is modeled on real time, having properties similar to the real numbers (continuous, linear and dense), 59

Convex hull A arbitrarily dimensioned generalisation of the bounding box concept, often used as *abstract domains* in *abstract interpretation*, 66

Correctness In *abstract interpretation,* a correct abstraction faithfully models all behaviours of the concrete system, 58

Crossbar Switch See Section 6.2.4.1, 123

DC-DC Converter A DC-DC converter takes an incoming DC supply and synthesises from it another, often somewhat different, supply rail. DC-DC converters can often step up voltages as well as step them down. See also *voltage regulator*, 120

DNF Disjunctive Normal Form, 111

Delay-insensitive circuit an asynchronous paradigm whereby both wires and gates are considered to have delays (see also *speed-independent circuit*), 57

Dense continuous time a time model in which time is modeled on the real numbers (*i.e.,* $\forall \tau_1 \in \mathbb{R}, \tau_2 \in \mathbb{R} \,.\, \tau_1 \neq \tau_2 \Rightarrow \exists \tau_3 \in \mathbb{R} \,.\, \tau_1 < \tau_3 < \tau_2$), 55

Deterministic signal a total function from real time $\mathbb{R}$ to the Booleans $\mathbb{B}$, 59

Deterministic trace A deterministic trace characterises a signal (in $\mathbb{S}$) by retaining transitions but abstracting away the times at which they occur, 60

Dynamic hazard A hazard (unintentional short pulse) that may occur at a time that a signal is switching from *true* to *false* or vice-versa, 55

Earth A connection to a metal rod or pipe that is physically buried. Not to be confused with *ground*, 105

Explosive bolt Commonly used between the stages of multi-stage rockets, though occasionally used in satellites, explosive bolts contain a small pyrotechnic device that when electrically triggered shatters the bolt, causing whatever it was previously fastening to part, 120

Extreme environment An environment that is significantly characteristically different from Earth ground level room temperature conditions, *e.g.,* high radiation, very high temperature, very low temperature, high G force, high vibration, hard vacuum, etc., 105

FET Field-Effect Transistor, 105

FPGA Field Programmable Gate Array, 105

FPTA Field Programmable Transistor Array, 115, 123

Fault recovery The process by which a system may recover from a transient or permanent failure, 105

Fibonacci series The numeric series generated by $x_n = x_{n-2} + x_{n-1}$, *i.e.,* 1, 2, 3, 5, 8, 13, 21, 34, . . . , 96

Flight systems Electronic systems in use in an aircraft or spacecraft, 105

GEO Geosynchronous Earth Orbit, a special orbit, quite far out, centred over Earth's equator, designed such that the spacecraft's orbital period is exactly 1 day, causing it to appear to be in a fixed position in the sky. Used by most communications and TV broadcast satellites. Also known as a Clarke orbit, after the science fiction author Arthur C. Clarke who is generally credited as having first proposed the idea, 105

Galois connection Given a pair of partially-ordered domains D and $D^\sharp$, and a pair of functions $\langle \alpha, \gamma \rangle$ with types $\alpha : D \to D^\sharp$ and $\gamma : D^\sharp \to D$ such that $\alpha \circ \gamma(\hat{x}) \sqsubseteq \hat{x}$ and $\gamma \circ \alpha(\hat{x}) \sqsupseteq \hat{x}$, a (monotone) Galois connection may be said to exist, 63

Galois insertion A *Galois connection* where $\alpha \circ \gamma(\hat{x}) = \hat{x}$, 63

GeV/n Giga-electron volts per nucleon (see also *eV/n*), 29

GeV Giga-electron volts (see also *eV*), 29

Glitch a (usually unintentional) short pulse, see also *static hazard* and *dynamic hazard*, 55

Ground The common voltage reference for both digital and analogue circuits, also known as 0V (zero volts) or logic false. Not necessarily (though often) connected to *earth*, 105

HDL see *Hardware Description Language*, 55

HarPE 'Hardware Partial Evaluator,' an experimental *hardware description language* implemented by the author to aid in experimentation with hardware partial evaluation, 82

Hardware Description Language a language, usually closely syntcatically resembling a programming language, that is used to represent the design of digital circuits in order to enable simulation and/or synthesis, 55

IGBT Insulated Gate Bipolar Transistor. Some IGBT devices are capable of switching very heavy loads – the Toyota Prius hybrid electric vehicle is powered via an IGBT-based 50kW inverter, 123

ISS International Space Station, 105, 117

Independent attribute model an analysis technique whereby (usually abstract) values are considered independently from each other (see also *relational attribute model*), 56

Inertial delay a delay function that models the kind of delay exhibited by a typical logic gate – short pulses may be further shortened or lost (see also *transmission line delay*), 57

JPL Jet Propulsion Laboratory, 115

LEO Low Earth Orbit, as used by ISS, Shuttle and most Earth observation satellites, 105

LE FPGA logic element, 95

Leakage current In a CMOS gate, when a transistor is switched off its resistance is finite, so as a consequence of Ohm's law, a leakage current flows across the gate that contributes to the power requirements for the chip as a whole. In the past, leakage current in larger geometry CMOS devices was often negligible, but modern deep submicron devices tend to exhibit worse leakage current characteristics as a consequence of their extremely small geometries, 147

Limit Switch A mechanical system that involves linear motion will often incorporate limit switches that provide a simple on/off indication when the limits of travel are reached, 120

Local Resynthesis Resynthesis of a small area of an FPGA, normally surrounding a fault that is to be worked around, 109

Logic simulation (usually automated) simulation of digital logic circuits, 55

Logic synthesis the synthesis of digital logic (circuits), usually based on a specification written in a hardware description language, 55

MEO Mid Earth Orbit, higher than LEO, but lower than geosynchronous orbit. Used by the Global Positioning System (GPS) satellites, 105

Manifold See Chapter 6, 117

MeV/n Mega-electron volts per nucleon (see also *eV/n*), 29

MeV Mega-electron volts (see also *eV*), 29

Model checking a technique whereby mathematical models are automatically checked to verify whether or not they satisfy particular properties, usually by exhaustive search of the state space, 55

NNF Negation Normal Form, 111

NOT gate a logic gate implementing the logic function $\neg$, 57

Non-achronous Analysis an analysis technique that may take into account absolute time (see also *achronous analysis*), 56

Non-clausal SAT problem A *SAT problem* that has not been reduced to CNF, 111

Non-clausal SAT solver A *SAT solver* that is capable of accepting problems that have not been reduced to CNF, 111

Nondeterministic trace A nondeterministic signal (in $\wp(\mathbb{S})$) may be modeled by a set of deterministic traces (in $\wp(\mathbb{T})$ – see also *deterministic trace* and Section 3.3.2, 62

OR gate a logic gate implementing the logic function $\vee$, 57

Ohm's law In electronics, the relationship $V = I \times R$ between voltage V, current I and resistance R, 147

Omega Network A type of *shuffle network* often used in parallel computing, 126

Ordinary delay see *transmission line delay*, 57

PCB Printed Circuit Board, 118

Partial evaluation See Section 4.1, 77

Passive Backplane A passive *backplane* typically does not include active components, though it may include line termination, power regulation, decoupling or other similar hardware., 118

Perfect gates a gate (AND/OR/NOT) model that assumes zero delay, 57

Permanent latch-up A common failure mode in CMOS circuits, generally caused when a charged particle triggers a brief short-to-ground, thereby burning out one or more FETs, 105

Permutation Network A network, constructed by composing 2-way swap/pass through nodes that is capable of generating any arbitrary permutation of its inputs, 126

QBF solver A solver for expressions in QBF form (see also *SAT solver*), 173

QBF Quantified Boolean Formula, 173

RAD6000 A widely used, commercially available, radiation hard processor based on an early IBM RS/6000 processor design, 105

RAD750 An updated, higher performance alternative to the *RAD6000* that is based on a more recent PowerPC design, 105

Radioisotope Thermoelectric Generator A radioisotope thermoelectric generator exploits the natural tendency for subcritical quantities of certain radioactive isotopes to generate large amounts of heat. A thermal gradient set up between the isotope and a heat sink is typically used to generate power by exploiting the *Seebeck effect*, 120

Reconfigurable FPGA An FPGA that can be dynamically reconfigured at any time by uploading a new *bit stream*, 105

Reconfigurable Manifold See Chapter 6, 117

Refinement A circuit c_1 may be said to *refine* (be a refinement of) circuit c_2 iff c_1 preserves all of the steady state and clean (glitch-free) behaviour of c_2. Sometimes also referred to as a *hazard non-increasing extension* in the asynchronous design literature, 71

Regulator See *voltage regulator*, 120

Relational attribute model an analysis technique whereby values may be related to each other such that they may considered to exist in certain value combinations but not others (see also *independent attribute model*), 56

Responsive Space A collective term for a number of technologies and working practices that, in combination, aim to reduce the time from concept to launch of new satellites from several years to less than one week., 117

SAT problem Given an arbitrary Boolean expression constructed from $\wedge$, $\vee$, $\neg$ and variables (*i.e.,* no quantifiers or functions), the problem of finding an assignment for the variables such that the expression evaluates to true is known as a SAT problem, and is known to be *NP*-complete. The related problem of proving that no such assignment exists is thought to be more difficult, and is in co-*NP*., 111

SAT solver A generic program or library capable of solving *SAT problems*, 111

Seebeck Effect Effectively the better-known Peltier effect in reverse, the Seebeck effect is the direct conversion of temperature differentials to electricity, 120

Self-timed circuit an clockless asynchronous design paradigm whereby computation proceeds at the speed allowed by the arrival of signals, 57

Short to ground A signal path, usually undesired, that causes a large current to flow from power to ground, 105

Shorthand notation A concise alternative to *trace notation* – see Section 3.4, 60

Shuffle Network Essentially a degenerate *permutation network* that is only capable of a proportion of possible permutations, 126

Single-Event Effect (SEE) The consequential effect on a circuit of a single charged particle (cosmic ray) impact, 105

Single-Event Transient (SET) A brief pulse, generally of the order of 0.5nS, caused by a charged particle impact that is not sufficiently energetic to cause permanent damage, 105

Single-Event Upset (SEU) A single bit error in a register, memory location or flip flop caused by a charged particle impact, 105

Singleton trace A trace, denoted $\langle 0 \rangle$ or $\langle 1 \rangle$, representing a signal whose value is static for all time, 60

Solar Panel Satellites operating within the orbit of Mars are typically powered by solar panels, which appear as 'wings' constructed from photovoltaic cells, 120

Solenoid A (usually) open-cored coil that is typically used to apply a force to a magnet or ferrous rod. Solenoids are in common use in engineering when linear actuation is required, but the application is not critical enough to require a more complex solution., 120

Sorting Network A sort algorithm that can be reduced to a statically constructible network of 2-way sort nodes, 126

Soundness (of abstract interpretation) see *correctness*, 58

Speed-independent circuit an asynchronous paradigm whereby gates have delays, but wires are assumed to have zero delay (see also *delay-insensitive circuit*), 57

Star Tracker A navigation device commonly used by satellites and deep space probes to align themselves. A camera images stars within its field of view, and image recognition software uses this to determine the satellite's exact orientation, 120

Static hazard A hazard (unintentional short pulse) that may occur at a time that a signal should maintain the same state, 55

Static-Clean logic A logic (often, but not necessarily, also a *transitional logic*) that incorporates the logic values S and C, 71

Stuck-at Fault A (usually permanent) failure that presents as a signal that is permanently connected to power or ground, see also *permanent latch-up*, 105

Synchronous Circuit a digital circuit timed by a single global clock, with no feedback allowed except via flip-flops, 55

Termination Signals on *transmission lines* will normally reflect back from an open end – it is therefore necessary to construct a termination network that has an impedance matched to that of the transmission line itself. Often this is a simple as a resistor connected to ground, though active circuitry is sometimes used., 119

Thermoelectric Generator See *radioisotope thermoelectric generator*, 120

Torquer bar A metal rod, normally mounted externally on a spacecraft, that generates a magnetic field when current is passed through it. In Earth orbit, this field interacts with the Earth's magnetic field thereby applying a torque to the spacecraft. Often used in conjunction with gyroscopes and reaction thrusters, torquer bars are in common use as part of spacecraft navigation systems, 120

Trace notation Traces (in $\mathbb{T}$) are represented by the notation $\langle \ldots \rangle$, where a list of Boolean values represents the values present within a trace, but not the times at which the transitions occur, 60

Transitional logic a multi-value logic whose values explicitly capture transitions in truth value, 55

Transmission line delay a delay function that may neither remove or re-order pulses, named because of its resemblance to a transmission line (impedance-balanced wire) in analogue electronics (see also *inertial delay*), 57

Transmission line A transmission line is typically a cable or PCB track, where a signal is paired deliberately with a corresponding ground, such that the impedance that is presented at all points along the signal path are, as far as possible, equal. See also *termination.*, 119

UNSAT A SAT problem with no solution, see also *SAT problem*, 40

Unregulated Power An unregulated power rail typically does not feature exact control over its absolute voltage, and in many cases also over noise or ripple characteristics. See also *voltage regulator*, 120

VHDL a commonly used *hardware description language*, 55

VLSI Very Large Scale Integration, 57, 105

Verilog a commonly used *hardware description language*, 55

Voltage Regulator A voltage regulator takes a (usually greater) DC voltage and derives from it a carefully controlled DC voltage output, 120

Xilinx Major FPGA manufacturer, with a specialism in high performance SRAM-based COTS FPGAs, 105

eV/n Electron volts per nucleon: the kinetic energy gained by an atomic nucleus stripped of electrons whilst passing through an electrostatic potential difference of 1 volt in vacuum, 29

eV Electron volt: the kinetic energy gained by an electron whilst passing through an electrostatic potential difference of 1 volt in a vacuum, 29

de Morgan's law *e.g.,* $\neg(a \wedge b) = (\neg a) \vee (\neg b)$, 57

Index