Asynchrobatic logic for low-power VLSI design

David John Willingham

School of Electronics and Computer Science

University of Westminster

ASYNCHROBATIC LOGIC FOR LOW-POWER VLSI DESIGN

David John WILLINGHAM

A thesis submitted in partial fulfillment of requirements of the

University of Westminster for the degree of

Doctor of Philosophy

March 2010

i. Table of Contents

XXXIX

ii. List of Abbreviations

123-DD	123 Decision Diagram
ADL	Adiabatic Dynamic Logic
ANSI	American National Standards Institute
ASWC	Asynchronous Stepwise Charging
BDD	Binary Decision Diagram
CAL	CMOS Adiabatic Logic
CCN	Controlled-Controlled-NOT
CL	Carry Look-ahead
CLA	Carry Look-ahead Adder
CMOS	Complementary Metal Oxide Semiconductor
CPAL	Complementary Pass-transistor Adiabatic Logic
CRL	Charge Recovery Logic
DAG	Directed Acyclic Graph
DC	Direct Current
DCVSL	Differential Cascode Voltage Switch Logic
DeMUX	De-Multiplexer
DPA	Differential Power Analysis
DRC	Design Rule Check
DTMOS	Dynamic Threshold MOS
DUT	Device Under Test
EACRL	Efficient Adiabatic Charge Recovery Logic
ECC	Elliptic Curve Cryptography
ECRL	Efficient Charge Recovery Logic
EEL	Energy Efficient Logic
ERC	Electrical Rule Check
FBDD	Free Binary Decision Diagram
FET	Field Effect Transistor
FSM	Finite State Machine
GCD	Greatest Common Denominator
GF	Galois Field

HDL	Hardware Description Language
HRKA	Higher-Radix Knowles Adder
IECRL	Improved Efficient Adiabatic Charge Recovery Logic
IP	Intellectual Property
LSB	Least Significant Bit
LVS	Layout Versus Schematic
MOS	Metal Oxide Semiconductor
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MSB	Most Significant Bit
MUX	Multiplexer
MVL	Multi-Valued Logic
NMOS	N-Type Metal Oxide Semiconductor
OBDD	Ordered Binary Decision Diagram
OEIS	Online Encyclopædia of Integer Sequences
PAL	Pass-transistor Adiabatic Logic
PFAL	Positive Feedback Adiabatic Logic
PMOS	P-Type Metal Oxide Semiconductor
PVT	Process, Voltage and Temperature
RAM	Random Access Memory
RERL	Reversible Energy Recovery Logic
RF	Radio Frequency
RFBDD	Reduced Free Binary Decision Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
ROR	Rotate Right
SCRL	Split-level Charge Recovery Logic
SIMD	Single Instruction Multiple Data
SOI	Silicon on Insulator
SPICE	Simulation Program with Integrated Circuit Emphasis
SRAM	Static RAM
SWC	Stepwise Charging
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
V_T	Threshold Voltage
XOR	Exclusive-OR
XNOR	Exclusive-NOR

iii. List of Symbols

ΔE	Error between sinusoidal and ideal waveforms
θ	Normalised angle
φ	Clock phase
τ	Time
V_{pc}	Power-clock voltage
V_{dd}	Static power supply voltage
$!_T$	(Binary) tree factorial operator
$!_{T(n)}$	n-ary tree factorial operator
P_D	Dynamic power
f_e	Effective frequency
C_L	Capacitive load
$\overline{P}$	Average power
t_1, t_2	Time 1, Time 2
i_{Vdd}	Current flowing through power supply V_{dd}
i(t)	Instantaneous current
v(t)	Instantaneous voltage
x	8-bit input to q-box
y	8-bit output from q-box
$q_n[m]$	8-bit substitution function of q-box n
a_0, a_1, a_2, a_3, a_4	Internal, intermediate 4-bit functions of q-boxes
b_0, b_1, b_2, b_3, b_4	Internal, intermediate 4-bit functions of q-boxes
$t_n[m]$	Look-up table, giving the m^{th} value of the n^{th} table
A, B	Inputs to adder or subtractor
R	Reverse subtraction selector
Z	Output from adder or subtractor
$F_{(n)}$	The n^{th} Fibonacci number
P	An integer
w	Width of the data-path
$\mathbb{Z}$	The set of integers
$\mathbb{N}$	The set of natural numbers

iv. List of Figures

v. List of Tables

vi. List of Equations

vii. List of Publications

- David J. Willingham and İzzet Kale, “Asynchronous, quasi-adiabatic (Asynchrobatic) logic for low power very wide data width applications”, Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS 2004), volume 2, pages II:257-260, Vancouver, Canada, 23rd-26th May 2004.

- David J. Willingham and İzzet Kale, “An Asynchrobatic, Radix-four, Carry Look-ahead Adder”, Proceedings of Ph.D. Research in Microelectronics and Electronics (PRIME 2008), pages 105-108, İstanbul, Turkey, 22nd-25th June 2008.

- David J. Willingham and İzzet Kale, “Using Positive Feedback Adiabatic Logic to implement Reversible Toffoli Gates”, Proceedings of NORCHIP 2008, pages 5-8, Tallinn, Estonia, 17th-18th November 2008.

- David J. Willingham and İzzet Kale, “A system for calculating the Greatest Common Denominator implemented using Asynchrobatic Logic”, Proceedings of NORCHIP 2008, pages 194-197, Tallinn, Estonia, 17th-18th November 2008.

viii. Acknowledgements

"I find television very educational. The minute somebody turns it on, I go to the library and read a good book." Groucho Marx.

I must thank my parents, Doug and Ann, and my partner Jacqueline for their support. Special thanks go to my father who claims to have ended up looking like Ben Turpin after his assistance proof-reading this thesis.

I would also like to thank my supervision team and other colleagues at the University of Westminster; Prof. İzzet Kale, Prof. Dik Morling, Dr. Ediz Çetin and Alan Wood. Special thanks must go to my Director of Studies, Prof. İzzet Kale, for his support, assistance, and understanding throughout the duration of my research.

Thanks also go to following libraries and their staff:
University of Westminster (New Cavendish Street, London),
British Library (St. Pancras, London),
Institution of Engineering and Technology Library (Savoy Place, London),
Barking Library (London Borough of Barking and Dagenham),
Cherry Hinton Library (Cambridgeshire),
Eastville Library (City and County of Bristol),
Ilford Central Library (London Borough of Redbridge).

And to the online research services of:
The British Computer Society (BCS),
The Institution of Engineering and Technology (IET) and
The Institution of Electrical and Electronic Engineers (IEEE).

Finally, I must acknowledge that this research was partially supported by a Quintin Hogg Research Scholarship from the University of Westminster.

ix. Abstract

In this work, *Asynchrobatic* Logic is presented. It is a novel low-power design style that combines the energy saving benefits of asynchronous logic and adiabatic logic to produce systems whose power dissipation is reduced in several different ways. The term "*Asynchrobatic*" is a new word that can be used to describe these types of systems, and is derived from the concatenation and shortening of **Asynchro**nous, Adia**batic** Logic. This thesis introduces the concept and theory behind *Asynchrobatic* Logic. It first provides an introductory background to both underlying parent technologies (asynchronous logic and adiabatic logic). The background material continues with an explanation of a number of possible methods for designing complex data-path cells used in the adiabatic data-path. *Asynchrobatic* Logic is then introduced as a comparison between asynchronous and *Asynchrobatic* buffer chains, showing that for wide systems, it operates more efficiently. Two more-complex sub-systems are presented, firstly a layout implementation of the substitution boxes from the Twofish encryption algorithm, and secondly a front-end only (without parasitic capacitances, resistances) simulation that demonstrates a functional system capable of calculating the Greatest Common Denominator (GCD) of a pair of 16-bit unsigned integers, which under typical conditions on a 0.35µm process, executed a test vector requiring twenty-four iterations in 2.067µs with a power consumption of 3.257nW. These examples show that the concept of *Asynchrobatic* Logic has the potential to be used in real-world applications, and is not just theory without application. At the time of its first publication in 2004, *Asynchrobatic* Logic was both unique and ground-breaking, as this was the first time that consideration had been given to operating large-scale adiabatic logic in an asynchronous fashion, and the first time that Asynchronous Stepwise Charging (ASWC) had been used to drive an adiabatic data-path.

Chapter 1 Introduction

1.1 Aims and Motivation

Until quite recently, the power consumption of VLSI computation devices had not been the major limiting factor in improvements and advances in microelectronic technology. Computers were static, powered by mains electricity, and even the most power hungry microprocessor of a desktop computer could be cooled using a fan-assisted heat-sink. Current processors can consume 140W, drawing over 100A of current in the process [AMD09]! However, the rise of portable consumer electronics, ubiquitous computing devices [Weis93], and implanted or wearable bio-medical electronics means that power efficient computation has become more important in widely deployed technologies, rather than being confined to niche and specialist areas. Also, as the dimensions of CMOS technologies have shrunk, so that for nanometre technologies the thickness of the gate dielectric is a countable number of atoms [Inte07], traditional power reduction techniques like voltage scaling have ceased to be as effective, and new issues like source-drain leakage and even gate leakage have become significant sources of power dissipation.

Depending upon the application, there are numerous methods that can be used to reduce the power consumption of VLSI circuits, these can range from low-level measures based upon fundamental physics, such as using a lower power supply voltage or using high threshold voltage transistors; to high-level measures such as clock-gating or power-down modes. The two that motivated this investigation were asynchronous logic [Mull59] & [Spar01] and adiabatic logic [Koll92]. These two technologies have been combined to create an Asynchronous, Adiabatic Logic methodology, called *Asynchrobatic* Logic [Will04], which is the subject of this thesis. The name is derived as a concatenation and shortening of **Asynchro**nous, Adia**batic** Logic.

Although known from very early in the history of computation, asynchronous logic [Mull59] has until recently remained more a subject of academic research than commercial interest. However, in applications like RF-powered smart cards, asynchronous implementations of microprocessors normally used in embedded applications have found a commercially useful role [Spar01]. One of the properties of asynchronous systems that make them useful in these applications are that circuits include a built-in insensitivity to variations in power supply voltage, with a lower voltage resulting in slower operation rather than the functional failures that would be seen if traditional synchronous systems were used. Another major advantage is the fact that when an asynchronous system is idle there will be no ticking clock signals, whereas in synchronous systems, these clock signals are propagated throughout the entire system and convert energy to heat, often without performing any useful computations.

Adiabatic logic is a newer area of low-power research [Koll92]. It is focused on issues associated with the thermodynamics of computation. By taking this branch of physics, that usually looks at mechanical engines, and applying it to computing engines, research fields such as reversible computation as well as adiabatic logic have been created. By moving to a computing paradigm that is reversible, energy can be recovered from a computing engine, and reused to perform further calculations. The analogy of regenerative braking is a good example which illustrates this idea in the context of a familiar mechanical system. The low-power benefit of adiabatic logic is that energy can be recycled by being stored and reused, thus reducing the amount of energy drawn directly from the power supply. There are other low-power consequences of using certain realisations of adiabatic logic, but these are implementation specific rather than being directly due to the reversible nature of the logic.

The two research areas described above both had a different set of low-power benefits which they could bring to circuit design, and *Asynchrobatic* Logic was born out of the novel idea to attempt to find a way to unify the low-power benefits from these fields. When written succinctly as "unifying the low-power benefits of asynchronous logic and adiabatic logic", this idea may sound like a simple and innocuously easy task. However, when it is realised that one of the commonly used synonyms for adiabatic logic is "clock-powered logic", and that from its Greek etymology, "asynchronous" has evolved to mean without synchronised clocks, it can be seen that the task of merging anything from these two research areas is not going to be without substantial challenges. How a clock-powered logic can be operated without clocks appears initially to be an impossible and contradictory requirements specification.

1.2 Original contributions

The original contributions that this project has added to the state-of-the-art can be summed-up as follows:

- The novel concept of *Asynchrobatic* Logic. That is, a processing circuit which operates both asynchronously and adiabatically. Prior to this, adiabatic processing had occurred synchronously, any self-timed adiabatic circuits only had applications as drivers, just capable of repeating a signal, but not performing any logical operation upon it, and asynchronous logic had been used with logic incapable of charge recovery or adiabatic operation.
- The application of capacitor-based, Asynchronous Stepwise Charging (ASWC) as a method for driving adiabatic data-path logic.
- The design and implementation of a simple system (of a pipeline of buffers) to act as proof of concept.
- The design and implementation of a complex system (a 16-bit GCD calculator), capable of fulfilling the concepts of *Asynchrobatic* Logic.

- A method for modelling *Asynchrobatic* Logic in Verilog, an industry standard Hardware Description Language (HDL), which in the behavioural paradigm can be easily extended to VHDL, another industry standard HDL.
- Systematic identification of other adiabatic logic families based around cross-coupled pairs of PMOS transistors. However, these did not show any extra low-power benefits over previously disclosed technologies.
- The realisation that the Positive Feedback Adiabatic Logic (PFAL) family can be used to implement complex reversible processing logic. Design and implementation of a Toffoli Gate [Feyn00] to demonstrate this, noting that the design was operated under ideal adiabatic conditions, but with the caveat that nothing would preclude *Asynchrobatic* operation of such a circuit.
- A proposal to extend the family of Knowles Adders from radix-two to higher-radices and the use of this as a solution to overcome the large fan-out or wiring density required for wide, radix-four, *Asynchrobatic* or adiabatic adders.
- A generalisation of results for the rate of growth of the search space for "Free *n*-ary Decision Diagrams". These sequences only appear to have been documented for binary and ternary decision diagrams, but could be usefully extended to Free Quaternary, Quinary or higher-order Decision Diagrams, with possible applications being the design of functions for Multi-Valued Logics (MVL).

1.3 Outline of thesis

Asynchrobatic Logic is the result of a successful experiment that attempted to create a low-power CMOS logic structure that operated both asynchronously and adiabatically. It is therefore predicated upon prior-art from the fields of adiabatic logic and asynchronous logic, as well as more familiar ideas from the electronic engineering and computer science disciplines associated with VLSI design. The first two introductory chapters

provide explanations of both of these logic styles. This thesis has approached this idea from the perspective of implementing an asynchronous controller to drive adiabatic logic, and the majority of the focus was on the adiabatic logic, with the asynchronous controller using previously known and unremarkable implementations. The third introductory chapter examines the design methods for the dual-rail adiabatic logic families. The initial proof of concept that introduced *Asynchrobatic* Logic and demonstrated that it was viable, was an implementation of a chain of buffers [Will04]. These were compared against a similar structure implemented in standard asynchronous logic, and for a suitably wide data-path, with a reasonable switching probability, were shown to have a better power efficiency. Subsequent work demonstrated that more complex arithmetic functions could be implemented [Will08a] and ultimately that a complex system could be implemented. An important discovery, subsidiary to the main thrust of the work, was the realisation that this work had potential for use in fully reversible logic systems [Will08b].

This introduction will be followed by three introductory chapters, one that provides an in-depth introduction and background to adiabatic logic, and another that provides an overview of asynchronous logic, and a third that detail the design methods for dual-rail logic used in adiabatic data-paths. The chapters subsequent to this document the design details and other principles applied during the creation of *Asynchrobatic* Logic. The HDL modelling of *Asynchrobatic* Logic using Verilog, and issues surrounding physical implementation are considered. Finally, an *Asynchrobatic* implementation of Euclid's Greatest Common Denominator (GCD) algorithm is presented [Will08c]. Conclusions are drawn, including comparisons of *Asynchrobatic* Logic against other authors' works; future work and possible commercial applications are proposed and discussed; and bibliographic references are cited.

The appendices contain annotated source code. Appendix A contains Verilog source code for both the single-rail and dual-rail versions of the GCD algorithm. Appendix B contains the C source code for automated Ordered Binary Decision Diagram (OBDD) minimisation Appendix C contains the SPICE sources for the Twofish q-boxes and the GCD algorithm, along with Layout versus Schematic checking summaries for the Twofish q-boxes.

Chapter 2 An introduction to Adiabatic Logic

2.1 Introduction

In this chapter, the basic concepts of Adiabatic logic will be introduced. “Adiabatic” is a term of Greek origin that has spent most of its history associated with classical thermodynamics. It refers to a system in which a transition occurs without energy (usually in the form of heat) being either lost to or gained from the system. In the context of electronic systems, rather than heat, electronic charge is preserved. Thus, an ideal adiabatic circuit would operate without the loss or gain of electronic charge. The first usage of the term “Adiabatic” in this context appears to be traceable back to a paper presented in 1992 at the Second Workshop on Physics and Computation [Koll92]. Although an earlier suggestion of the possibility of energy recovery was made by Bennett where in relation to the energy used to perform computation, he stated “*This energy could in principle be saved and reused*” [Benn82].

2.2 Physics and Computation

The introduction to this section details the etymology of the term “adiabatic logic”. In this section, the underlying physics are considered. Because of the Second Law of Thermodynamics, it is not possible to completely convert energy into useful work. However, the term “Adiabatic Logic” is used to describe logic families that could theoretically operate without losses, and the term “Quasi-Adiabatic Logic” is used to describe logic that operates with a lower power than static CMOS logic, but which still has some theoretical non-adiabatic losses. In both cases, the nomenclature is used to indicate that these systems are capable of operating with substantially less power dissipation than traditional static CMOS circuits, which as is shown in equation (2.1) operates with a Power, P, that is proportional to both the Capacitive load, $C_{L,}$ and the square of the Voltage, V.

$$P \propto C_L V^2 \tag{2.1}$$

The fact that these techniques take the operational power of adiabatic and quasi-adiabatic circuits below this threshold was emphasised by some initial works in this research area, which had titles proclaiming that they operated "*sub-CV²*" [Sven94b] & [Sven96].

The underlying principles of adiabatic logic can be traced back more than two centuries to the industrial revolution when James Clerk Maxwell proposed his paradoxical Dæmon [Maxw71]. This led to the conclusion during the latter part of the twentieth century that Maxwell's Dæmon cannot violate the Second Law of Thermodynamics as the erasure of information causes entropy to increase [Land61], [Benn73] & [Leff03]. This also led to the concept of a system that could be operated arbitrarily slowly such that its dissipation could asymptotically approach zero as its speed was reduced [Youn93].

There are several important principles that are shared by all of these low-power adiabatic systems. These include only turning switches on when there is no potential difference across them, only turning switches off when no current is flowing through them, and using a power supply that is capable of recovering or recycling energy in the form of electric charge.

To achieve this, in general, the power supplies of adiabatic logic circuits have used constant current charging (or an approximation thereto), in contrast to more traditional non-adiabatic systems that have generally used constant voltage charging from a fixed-voltage power supply.

The power supplies of adiabatic logic circuits have also used circuit elements capable of storing energy. This is often done using inductors [Moon96], which store the energy by converting it to magnetic flux, or, as in case of *Asynchrobatic* Logic, by using capacitors, which can directly store electric charge.

There are a number of synonyms that have been used by other authors to refer to adiabatic logic type systems, these include: “Charge recovery logic” [Youn93], “Charge recycling logic” [Kong96a], “Clock-powered logic” [Atha97], “Energy recovery logic” [Hinm93] and “Energy recycling logic” [De96c]. Because of the reversibility requirements for a system to be fully adiabatic, most of these synonyms actually refer to, and can be used interchangeably, to describe quasi-adiabatic systems. These terms are succinct and self-explanatory, so the only term that warrants further explanation is “Clock-Powered Logic”. This has been used because many adiabatic circuits use a combined power supply and clock, or a “power-clock”. This a variable, usually multi-phase, power-supply which controls the operation of the logic by supplying energy to it, and subsequently recovering energy from it.

The possibility of using adiabatic logic in larger systems has been shown to be viable with both clock-powered and adiabatic processors having been successfully implemented [Atha97], [Shin03] & [Shin04].

2.3 A review of adiabatic logic families

Over the last two decades many different adiabatic or quasi-adiabatic logic families have been proposed. A substantial list of these, in approximate chronological order, is shown below in Table 2.1.

List of Adiabatic Logic families in approximate chronological order
Hot clock nMOS [Seit85]
Retractile cascades [Hall92]
Recovered Energy Logic (REL) [Hinm93]
Charge Recovery Logic (CRL) [Youn93]
Split-level Charge Recovery Logic (SCRL) [Youn94a] & [Youn94b]
Pulsed Power Supply CMOS (PPS CMOS) [Gaba94a] & [Gaba94b]
2N-2N2D [Kram94]
Adiabatic Dynamic Logic (ADL) [Dick94] & [Dick95]
Adiabatic Pseudo-Domino Logic (APDL) [Wang95]
Clocked CMOS Adiabatic Logic (CAL) [Maks95], [Maks97a] & [Maks97b]
Efficient Charge Recovery Logic (ECRL) [Moon95] & [Moon96] [1]
2N-2P [Kram95] [1]
2N-2N2P [Denk94] & [Kram95] [2]
Quasi-Adiabatic Ternary CMOS Logic (QAT) [Mate96] & [Mate97]
Positive Feedback Adiabatic Logic (PFAL) [Vetu96] [3]
Transmission gate-interfaced APDL (T-APDL) [Lau96]
Fully Adiabatic MOS Logic (ADMOS) [De96a]
Complementary Adiabatic MOS Logic (CAMOS) [De96a] & [De96c]
Dynamic Adiabatic MOS (DAMOS) [De96b] & [De96c]
Charge Recycling Differential Logic (CRDL) [Kong96a] & [Kong96b]
Half Rail Differential Logic (HRDL) [Choe97]
Energy Efficient Logic (EEL) [Yeh97]
Pass-transistor Adiabatic Logic (PAL) [Oklo97]
Quasi-Static Energy Recovery Logic (QSERL) [Ye97]
Improved Adiabatic Pseudo-Domino Logic (IAPDL) [Lau97]
True Single-Phase CRDL (TCRDL) [Kong97]
Forward body-bias MOS (FBMOS) [Kioi97]
Reversible Energy Recovery Logic (RERL) [Lim98]
Feedback Reversible Energy Recovery Logic (fRERL) [Kwon98]
Adiabatic Differential Cascode Pass-transistor Logic (ADCPL) [Lo98]
PAL-2N [Liu98a] [3]
Improved Efficient Charge Recovery Logic (IECRL) [Liu98b] [2]
Bootstrapped NMOS Charge Recovery Logic (BNCRL) [Yoo98]

True Single-Phase Energy-Recovering Logic (TSEL) [Kim98]
Modified Half Rail Differential Logic (MHRDL) [Won98]
CMOS Pass-gate No-race Charge-recycling Logic (CPNCL) [Yoo99a]
No-race Charge-recycling Differential Logic (NCDL) [Yoo99b]
Source Coupled Adiabatic Logic (SCAL) [Kim99]
Retractile Clock-Powered Logic (RCPL) [Tzar99]
Adiabatic Dynamic CMOS Logic (ADCL) [Taka00]
NMOS Energy Recovery Logic (NERL) [Kim00]
Bootstrapped Charge-Recovery Logic (BCRL) [Li00]
Adiabatic Differential Cascode Voltage Switch Logic (ADCVSL) [Suva00]
Dynamic Threshold MOS (DTMOS) ADCVSL [Lega01] & [Lega03]
High Efficient Energy Recovery Logic (HEERL) [Hong01]
Dual-Swing Charge-Recovery Logic (DSCRL) [Li01]
Efficient Adiabatic Charge Recovery Logic (EACRL) [Varg01a]
Improved Pass-Gate Adiabatic Logic (IPGAL) [Varg01b]
Complementary Pass-transistor Energy Recovery Logic (CPERL) [Chan02]
Complementary Pass-transistor Adiabatic Logic (CPAL) [Hu03] & [Hu04]
Improved Adiabatic Pseudo-Domino Logic 2 (IAPDL-2) [Widj03]
Unnamed modification to HEERL [Song04]
Improved Positive Feedback Adiabatic Logic (IPFAL) [Fisc04] [4]
Energy Recovery Capacitance Coupling Logic (ERCCL) [Qian04]
2N-2N2P2D [He06]
Quasi-Static Single-phase Energy Recovery Logic (QSSERL) [Li07]
Improved Positive Feedback Adiabatic Logic (IPFAL) [Vija07a] [Vija07b] [4]
[1] ECRL and 2N-2P are identical [2] IECRL is similar to the already proposed 2N-2N2P [3] PAL-2N is similar to the already proposed PFAL [4] The IPFAL name is duplicated with different functionality

Table 2.1: A list of adiabatic logic families

The list in Table 2.1 does not claim to be exhaustive, but it does aim to provide a general idea of the progress made in this field, and to show how the design of adiabatic logic families has progressed over time. Critically, it is noteworthy that all of these adiabatic logic families used synchronous clocking. However, because they are early attempts or are some of the more

frequently cited adiabatic families that have been proposed, a number of these adiabatic logic families warrant to be given brief descriptions. For those actually given serious consideration for use in Asynchrobatic Logic, expanded descriptions are provided later in this chapter:

- Split-level Charge Recovery Logic (SCRL) [Youn94a] & [Youn94b],
  - SCRL is a truly adiabatic logic family as it has a feedback path for logical recovery. Its topology resembles static CMOS gates followed by a transmission-gate, but it has both of its power rails replaced by power-clock signals, hence its description as "split-level". It has complex clocking, often with eight phases. It is an extension of a simpler adiabatic logic family called Charge Recovery Logic (CRL) [Youn93].
- 2N-2N2D [Kram94],
  - 2N-2N2D is a diode-based complementary logic family. The diodes mean that it will have non-adiabatic losses and is therefore only quasi-adiabatic. The systematic naming of this logic family indicates that it uses pairs of NMOS devices (designated by "*2N*") and a pair diodes (designated by "*2D*").
- Adiabatic Dynamic Logic (ADL) [Dick94],
  - ADL stages alternate between being constructed of NMOS and PMOS devices. As with 2N-2N2D recovery occurs through diodes, making it only quasi-adiabatic.
- Efficient Charge Recovery Logic (ECRL) [Moon95] & [Moon96],
  - It appears to have been independently discovered by Kramer, who gave it the systematic name "*2N-2P*" [Kram95],
  - ECRL and 2N-2P are identical, and are based upon the standard CMOS family called Differential Cascode Voltage Switch Logic (DCVSL) [Hell84]. This structure uses pairs of pull-down NMOS devices to evaluate functions (designated by the "*2N*") and a pair of cross-coupled PMOS devices (designated by the "*2P*") to hold state. It is frequently cited and has clearly been the inspiration for many

other similar adiabatic logic families. Complete recovery of the power-clock is not possible through the PMOS devices, so it is still only a quasi-adiabatic logic style. ECRL/2N-2P is described in more detail later in this section.

- 2N-2N2P [Denk94] & [Kram95],
  - It was originally simply described by Denker as an "*Adiabatic Logic Gate*", systematically named by Kramer, and later called Improved Efficient Charge Recovery Logic (IECRL) [Liu98b]. IECRL differs very slightly from the previous description by specifically using the a PMOS device to form a pair of recovery diodes. The previous works do not document how the bulk terminals are connected.
  - IECRL is very similar to ECRL, but with the addition of a pair of cross-coupled NMOS devices to give a better connection to ground when inputs have had their charge recovered. IECRL/2N-2N2P is also described in more detail later in this section.
- Clocked CMOS Adiabatic Logic (CAL) [Maks95],
  - CAL is similar to 2N-2N2P, but has clocked NMOS devices between the NMOS decision tree and the outputs. These are driven by a square-wave clock to access to the evaluation logic. This allows it to use fewer clock-phases, but requires extra control signals.
- Positive Feedback Adiabatic Logic (PFAL) [Vetu96],
  - Also later called PAL-2N [Liu98a],
  - PFAL is very similar to IECRL, but has its evaluation tree connected between power-clock and outputs. It can achieve fully adiabatic operation when a recovery path is provided. PFAL is described in more detail later in this section.
- Energy Efficient Logic (EEL) [Yeh97],
  - EEL is an extension of ECRL. It employs a pair of externally controlled, pulsed NMOS devices between power-clock and outputs to allow full charge recovery. However, it is not truly adiabatic because it is lacks logically reversibility.

- Pass-transistor Adiabatic Logic (PAL) [Oklo97],
  - The topology of PAL resembles a PFAL gate without the pair of cross-coupled NMOS devices. The absence of pull-down NMOS devices means that this family lacks a good ground connection.
- Reversible Energy Recovery Logic (RERL) [Lim98],
  - RERL holds its logic state using a pair of cross-coupled NMOS devices, but evaluation and recovery occurs through transmission gates, like SCRL it also requires complex, eight-phase clocking.
- Efficient Adiabatic Charge Recovery Logic (EACRL) [Varg01a],
  - EACRL has a pair of cross-coupled PMOS devices, and duplicate sets of evaluation logic, one set is connected between ground and the outputs, whilst the other is connected (with an opposite assertion level) between the power-clock and the outputs. In the same way as PFAL, it too can be made fully adiabatic when a recovery path is provided. EACRL is the final family described in more detail later in this section.
- Complementary Pass-transistor Adiabatic Logic (CPAL) [Hu03],
  - CPAL uses a PFAL inverter or buffer, with the main part of the evaluation tree constructed using pass-transistors to connect to the gates of the NMOS pull-ups.

Both Starosel'skii [Star02] and Amirante [Amir04] have provided good descriptions and classification in their works, and these works also provide an overview into some other adiabatic circuit styles. The earliest quasi-adiabatic styles such as ADL and 2N-2N2D used diodes, which means that they could not be loss-less. For this reason, no diode-based families were considered for *Asynchrobatic* implementation. Some of the later families were too complex, having a large implementation overhead. This overhead was either related to the number of devices required, which would increase the area requirements, or due to excessively complex constraints on power-clock phasing or additional control signalling. Consequently, these families were not considered for use with *Asynchrobatic* Logic.

The simple adiabatic logic families that are described below all share a common heritage that can be traced back to the static CMOS DCVSL family [Hell84]. For a designer, this is immensely useful, as the DCVSL family has been in use for over twenty years. This means that various design methodologies for more complex logic functions have been thoroughly investigated and published. These include a table-based Quine-McClusky method [Chu86], Ordered Binary Decision Diagram (OBDD) [Brya86] & [Karo95] methods, as well as various extensions thereto, including Free Binary Decision Diagram (FBDD) [Bern93] and 123-Decision Diagram (123-DD) [Arma98] methods. The OBDD of these design methodologies will be detailed in Chapter 4, and PFAL gate implementations resulting from use their are shown in Chapter 7.

ECRL, IECRL and PFAL are three of the simplest adiabatic or quasi-adiabatic logic families that are suitable for use in *Asynchrobatic* Logic. EACRL, although more complex, is also a relatively simple derivative of these, and could be a potential candidate for use in *Asynchrobatic* Logic. Therefore, only these four logic families will be more fully detailed. It should be clear to someone *au fait* with this logic style that other families including EEL could be used, but they will not be described herein. This omission does not mean that these logic styles will not work nor that they cannot be made to work in an *Asynchrobatic* fashion, but merely that the requirements to achieve this are beyond the scope of this thesis. Each potential logic family would need to be evaluated to determine whether the compromise of silicon area for lower power or other benefits is worthwhile. Clearly, this should be done with reference to a base-line, which should be ECRL as this is the simplest of the four-phase DCVSL-based adiabatic logic families. Where the suggested adiabatic family has extra signals, other factors need to be considered. For example, with the EEL family, the question the designer must ask is; would any power-savings in the data-path be counteracted by the extra power and silicon area required to provide a second local signal to each pipeline stage?

2.3.1 Efficient Charge Recovery Logic (ECRL)

ECRL [Moon95] (also known as 2N-2P [Kram95]) is based around a pair of cross-coupled PMOS transistors. Their source terminals are connected to the power-clock, and the gate of each one is connected to the drain of the other. These nodes form the complementary output signals. The function is evaluated by a series of pull-down NMOS devices. In the original description, the connectivity of the PMOS transistors' bulk terminals was not specified. However, experimentation has shown that better power performance can be obtained by connecting the bulk to the power-clock, as the power-clock can be recovered to a lower voltage. This improvement is not without cost, as it introduces layout constraints that require the hot n-wells of each *Asynchrobatic* stage to be kept separated. One disadvantage of ECRL is that once the charge from the previous stage has been recovered from the gate of the NMOS devices, there is no pull-down path to ground. This has implications for noise susceptibility. Figure 2.1 shows an inverter/buffer (in buffer configuration) implemented using the ECRL style. The power-clock drives the terminal labelled “Vpc”. The dual-rail input pair “A” and dual-rail output pair are shown with their high and low assertion levels indicated by the “_H” and “_L” suffixes.

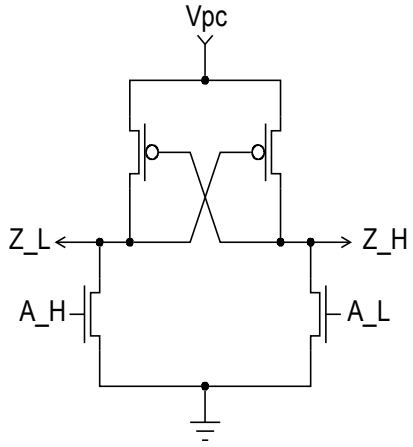

Figure 2.1: An ECRL Buffer [Kram95] & [Moon95]

2.3.2 Improved Efficient Charge Recovery Logic (IECRL)

IECRL [Liu98b] (originally known as 2N-2N2P by Kramer *et al.* in [Kram95], but first described by Denker in [Denk94] improves ECRL with the addition of a pair of cross-coupled NMOS devices. This produces a logic family that is based around a pair of cross-coupled inverters, a structure that is identical to the storage elements in a Static RAM (SRAM). The cross-coupled NMOS devices are an improvement over ECRL because they provide a pull-down path to ground that remains even after the charge is recovered from the gates of the evaluation FETs. However, because of the two extra NMOS devices, it will require a larger area in which to be implemented. Figure 2.2. shows an inverter/buffer (also in buffer configuration) implemented using the IECRL style. This figure uses the same naming and assertion level conventions as the ECRL circuit shown on the previous page.

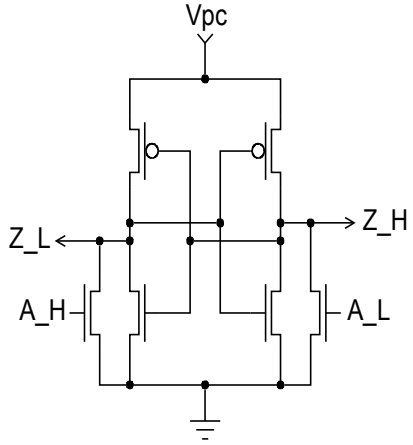

Figure 2.2: An IECRL Buffer [Denk94]

2.3.3 Positive Feedback Adiabatic Logic (PFAL)

PFAL [Vetu96], like IECRL, is also based around a pair of cross-coupled inverters. However, whilst in IECRL the NMOS devices used to evaluate the function are connected between the outputs and ground, in PFAL, these evaluation NMOS devices are connected between the outputs and the power-clock. The similarities between PFAL and IECRL gates are such that IECRL gates can be easily converted into PFAL gates. This is done by re-labelling the outputs so that their assertion levels are swapped, and connecting the NMOS evaluation devices between the power-clock and the outputs rather than between ground and the outputs. This can be made as easy to achieve in layout as it is in abstract representations of the circuit. When the power-clock is in its recovery phase, the NMOS devices between the outputs and the power-clock can allow complete recovery of those outputs. This means that the low-power performance of PFAL can be enhanced by making it fully reversible [Vetu96] & [Will08b]. Figure 2.3 shows an inverter/buffer (again in buffer configuration, with identical signal naming conventions) implemented in the PFAL style.

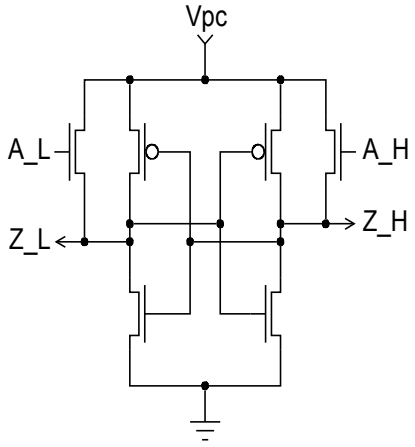

Figure 2.3: A PFAL Buffer [Vetu96]

2.3.4 Efficient Adiabatic Charge Recovery Logic (EACRL)

EACRL [Varg01a] is a mixture of ideas from ECRL and PFAL, as it too is based around a pair of cross-coupled PMOS devices. It uses both pull-up NMOS devices (like PFAL) and pull-down NMOS devices (like ECRL) to evaluate its function. Its main disadvantage is that for multi-input functions, there is a substantial overhead associated with the complete duplication of the evaluation logic. EACRL shares another minor disadvantage with ECRL, because like ECRL, it could also suffer from noise as EACRL does not have a pull-down path to ground after the charge on its inputs had been recovered. EACRL suggested incorporating a recovery path. An inverter/buffer (yet again in buffer configuration and with the same naming convention) implemented using the EACRL style is shown in Figure 2.4.

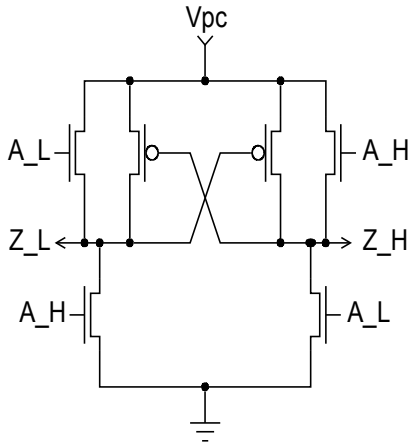

Figure 2.4: An EACRL Buffer [Varg01a]

A possible extension to EACRL is to add a cross-coupled pair of NMOS devices. This is the same design modification that improved ECRL to produce IECRL. This is likely to make the power consumption worse due to increased capacitive load, but in certain situations would increase the circuit's performance with respect to signal integrity in an electrically noisy environment.

Several of the papers proposing quasi-adiabatic logic families have alluded to design improvements that would render the designs either fully adiabatic or fully able to the outputs. These have included PFAL, EACRL and EEL. However, because the circuits used as demonstrators for these families have been either inverters or buffer, the full potential for these designs to be used to implement reversible logic gates appears to have been overlooked. It has now been shown that the PFAL design style can be used to implement fully-reversible logic gates [Will08b]. Although the paper only investigates Toffoli gates [Fred82], it is obvious to anyone appropriately skilled that reversible designs are not limited to these gates. This means that, theoretically, these gates can operate reversibly. Furthermore, it is also an obvious conclusion that this concept for the adiabatic data-path can be extended such that it can function under *Asynchrobatic* operation.

2.4 Systematic search for other potential adiabatic logic families

If a matrix of all potential constructions of simple adiabatic gates based upon a pair of cross-coupled PMOS devices is constructed, then simulations can be performed to determine which style has the lowest operational power. This also allows a more complete design space to be searched to ascertain if any other potential designs have been missed.

In this thesis, a more systematic nomenclature is proposed to classify these families. It is as follows:

$2n_d$: indicates NMOS pull-down evaluation devices.
$2n_u$: indicates NMOS pull-up evaluation devices.
2n: indicates a pair of cross-coupled NMOS devices.
2p: indicates a pair of cross-coupled PMOS devices.

This allows the acronyms of some of the adiabatic families to be replaced with systematic descriptions. Table 2.2 shows the possible

combinations, and identifies that there is one logic style that has not previously been presented.

Original Acronym	Systematic Nomenclature	Notes
ECRL	$2n_d$-2p	
PAL	$2n_u$-2p	No path to ground!
EACRL	$2n_d2n_u$-2p	
IECRL	$2n_d$-2n2p	
PFAL	$2n_u$-2n2p	
Previously unknown	$2n_d2n_u$-2n2p	**New!**

Table 2.2: A systematic nomenclature for basic adiabatic logic families

The contributions of this work beyond the previously known state of the art are to have fully explored the potential design space, and to have proved that arbitrarily complex reversible logic functions can be implemented in both the adiabatic and *Asynchrobatic* Logic styles. This allowed the discovery of the $2n_d2n_u$-2n2p logic family. If the inputs were split so that the $2n_d$ and $2n_u$ functions were the forward and recovery functions respectively, this could have operated at a lower-power than PFAL, with the evaluation happening through the $2n_d$ path and the recovery through the $2n_u$ path. Unfortunately, in simulations, this new logic family did not achieve any better power performance. However, where uniform power consumption, rather than minimum power consumption is a target, the use of some of these less efficient design styles may be of further research interest. The proposed systematic nomenclature could also be extended to cover more recent ideas such as the pass-gate inputs used by CPAL, allowing further methodical exploration of this extended potential adiabatic design space.

2.5 Adiabatic Power Supplies

Unlike static CMOS logic, due to being clock-powered, the adiabatic logic families derived from static DCVSL require a separate power supply. It

is necessary to consider these, and whilst brief details of inductor-based resonant clock-power supplies is necessary for completeness, the majority of this section will concentrate on the capacitor-based Stepwise Charging (SWC) methodology. Just as being able to design increasingly complex adiabatic logic families is futile if the only implementable components are buffers and inverters, it would be equally fruitless to design an adiabatic logic family that requires an excessive number of power-clocks, or has power-clocks which have unrealisable phasing or waveform requirements. The majority of the previously described adiabatic logic families have a requirement for a four-phase power-supply. The phase relationships of these ideal waveforms is shown in Figure 2.5. The four different power-clock phases are labelled from φ0 to φ3, and the time periods are labelled $\tau(0)$ to $\tau(8)$. Each power-clock moves from "*Idle*" to "*Charge*" to "*Hold* to "*Recoup*" and back to "*Idle*". Time periods $\tau(0)$ to $\tau(3)$ show how the power-clocks' phases are related during initial power-up and time periods $\tau(4)$ to $\tau(7)$ show how the power-clocks' phases are related during normal operation. It can be seen that time periods $\tau(4)$ & $\tau(8)$ are identical, and therefore to continue operation can be achieved by repeating the sequence shown in time periods $\tau(4)$ to $\tau(7)$ *ad infinitum*.

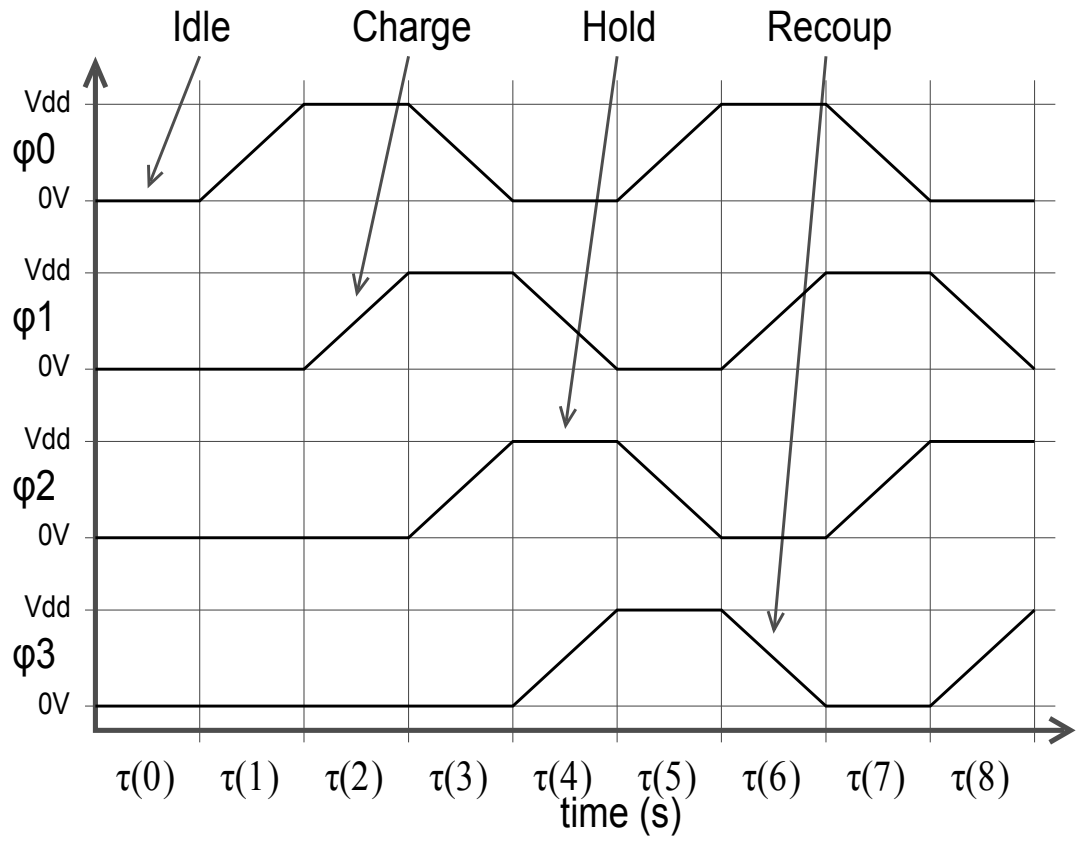

Figure 2.5: Phase relationships of adiabatic power-clocks

The early adiabatic logic families used inductor-based power supplies [Youn93]. These require off-chip inductors that are forced to resonate using relatively high-power MOSFET switches. As can be seen in the micrograph presented by Gabara *et al.* [Gaba95], the physical size of the inductor is similar to that of the silicon die! These were either synchronised by an external clock, or allowed to run freely at their resonant frequency. These generate sinusoidal waveforms that represent a reasonable approximation to the required four-phases. However, it is obvious that the sinusoid deviates from the ideal waveform substantially in the "*Idle*" and "*Hold*" phases. This can be quantified to an error of approximately 14.6% of the peak voltage, and because the ideal waveform is piecewise linear, for the "*Idle*" phase only, this error (ΔE) is calculated from a normalised angle (θ) as shown in Equation (2.2).

$$\Delta E = \frac{1}{2} - \frac{1}{2}\cos(\theta) \qquad (2.2)$$

This is derived by assuming that the normalised ideal waveform varies from 0V when in the "*Idle*" phase to 1V when in the "*Hold*" phase, and that the sinusoidal wave's minimum is at 0º and in the centre of the "Wait" phase. This means that the sinusoidal waveform will be defined by the negative cosine function, but to be normalised to 1V, it will need to be scaled by half and shifted up by half. Thus the maximum deviation of 14.6% from the ideal waveform will occur when ($\theta = 45º$), and although the function may be different, every 90º thereafter. There are four locations, halfway through each phase where the ideal sinusoid and the ideal waveform are identical. Figure 2.6 shows how the sinusoidal waveform compares with the ideal waveform, and also plots the absolute difference between these two waveforms. The error is important because during the "*Idle*" and "*Hold*" phases, the devices' power-clocks will not be driven to full-rail voltage potentially allowing leakage, and other power-consuming conditions to occur.

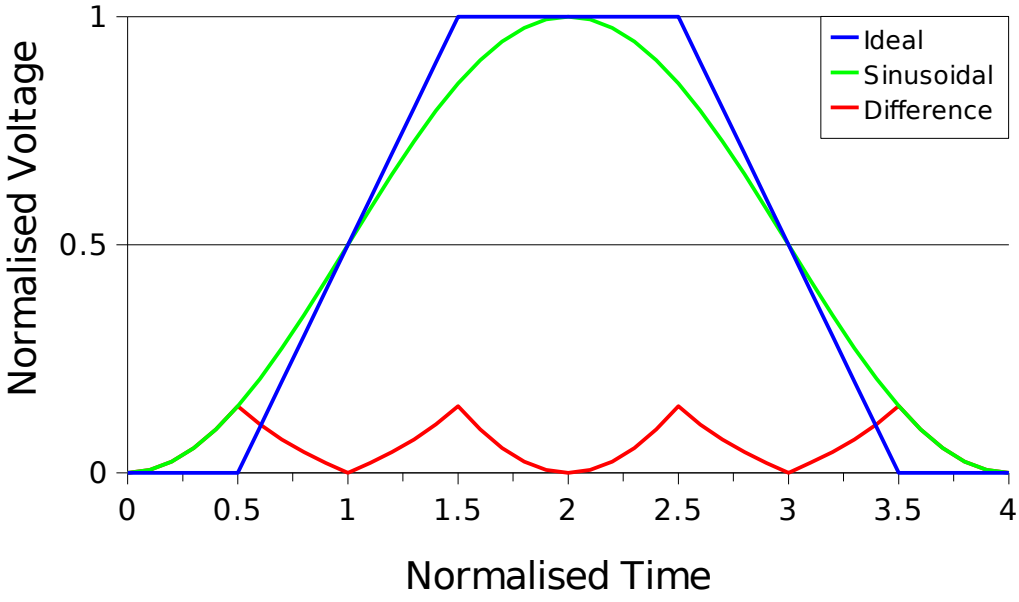

Figure 2.6: Difference between Ideal and Sinusoidal waveforms

There may be potential to achieve less difference if it is possible to slightly overdrive the sinusoidal waveform without exceeding the electrical tolerances of the fabrication process. However, whilst sinusoidal

power-clocks are used in adiabatic logic, such power-clocks do not feature in *Asynchrobatic* Logic, so no further investigations relating to them were performed.

An alternative, capacitor-based method that can be used as a charge recovering power supply is Stepwise Charging (SWC) [Sven94a] & [Sven94b]. It achieves a different approximation to the ideal waveform which, like the ideal waveform, but unlike that derived from a resonant power-clock, is static in its "*Idle*" and "*Hold*" phases. The waveform can be created by successively charging and discharging the capacitive load through various intermediate voltages by means of a series of sequentially switched tank capacitors. It is obvious that as the number of steps approaches infinity, the waveform becomes a progressively better approximation to the ideal waveform.

It has been shown that for a suitably large load, even a power supply with a single intermediate step can improve the efficiency of some systems [Hahm94]. A familiar macroscopic implementation of a system that uses the idea of a stepwise process is a flight of locks on a canal.

Stepwise charging waveforms are shown in Figure 2.7. Figure 2.7(a) shows an idealised stepwise charging waveform with three intermediate stages and Figure 2.7(b) shows the realised stepwise charging waveform. These deviate from the ideal, because each step charges the load capacitance using a constant voltage, resulting in the shown curve.

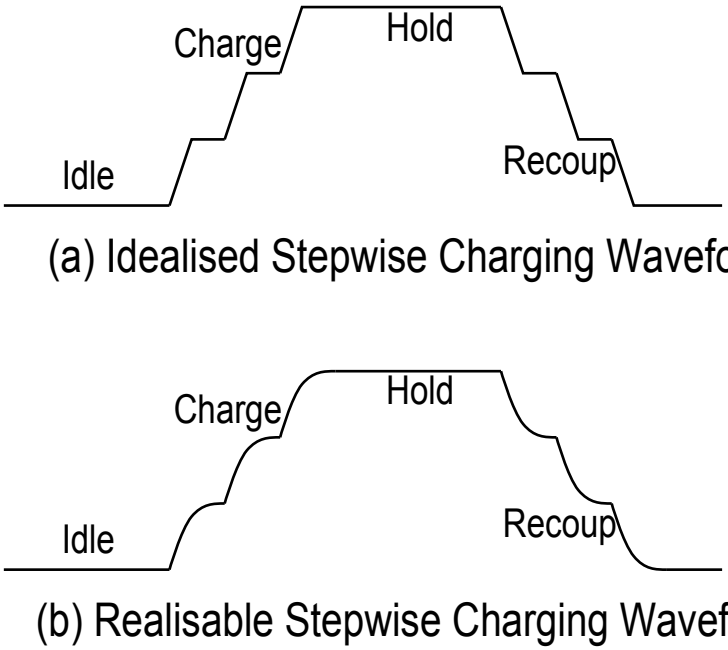

Figure 2.7: Stepwise Charging Waveforms

SWC has a number of properties that make it far more attractive than inductor-based designs for use in *Asynchrobatic* Logic:

- It has been shown to converge at start-up [Dhar96] & [Naka04].
- It has simple next-state logic that can easily be operated asynchronously.
- It can be implemented in a modular fashion allowing design reuse.
- It can be implemented using on-chip capacitors.
- It is static during the "*Idle*" and "*Hold*" phases.
- It does not require off-chip inductors.

2.6 Reversible Computation

In this introduction to adiabatic logic, it is essential to look at the closely related area of reversible computation. This requires that any computation could be performed both in a direction that would be thought of as forwards, processing the inputs to produce outputs **and** in the opposite, reverse direction, where the outputs are "de-processed" to produce the inputs. The

following simple example, which pays homage to Douglas Adams and his science-fiction writings [Adam79], will demonstrate why this seemingly simple concept can be problematic in practice.... If the answer is forty-two, what was the question? Even if it is known that the answer was obtained using six-bit unsigned integer addition, the question could be forty-one plus one or twenty plus twenty-two. This problem arises as a consequence of the destruction of information, and this information loss occurs in the majority of processing. An obvious way to make addition reversible is to preserve one of the inputs. This leads to the requirement for reversible systems to have the same number of inputs as outputs. For logic implementations, it is required that the function being used is invertible. So for any function of three-bits, each one of the eight possible input states would map to one of the eight possible output states.

A familiar application that is invertible, when thought of as a black-box, is a block cipher. Its forward operation is encryption and its operation reverse is decryption. For such a cryptosystem, the Key remains unchanged, but transforms the Plain Text into the Cipher Text or *vice-versa* without the loss of information. An in-depth *exposé* of the on-going theoretical work on Reversible Logic, Reversible Computation and associated subject areas is beyond the scope of this thesis, but interested readers are directed to, for example, the works of Kerntopf [Kern02].

The logical reversibility of an operation is an essential part of a truly adiabatic system, and therefore, the majority of systems that purport to be adiabatic would be better described as only quasi-adiabatic, as whilst charge recovery allows them to operate with more power efficiency than standard static CMOS, they still have not insignificant non-adiabatic losses. Clearly, there are some designs which are reversible where the non-adiabatic losses have been substantially reduced. However, just as friction prevents mechanical systems operating without some dissipation, relegating perpetual

motion machines to the realms of pseudo-science, electrical resistance prevents even these electronic systems operating without some dissipation.

2.7 Summary

In this chapter the concepts of adiabatic logic and quasi-adiabatic logic have been introduced. Details of a large number of adiabatic logic families have been presented. This has included providing an overview of twelve examples that are frequently cited, and expanded descriptions of the four candidate families: ECRL, IECRL, PFAL and EACRL, which are proposed to be used in *Asynchrobatic* Logic. A systematic exploration of the design space was conducted, and although this yielded a previously unknown design, it was not found to have any benefits over already known designs.

The concept of a power-clock has been introduced, detailing an ideal power-clock, and two possible realisable implementations. The generation of power-clocks using inductor-based resonant charging, and stepwise charging was introduced. Finally reversible computation was discussed, as unless a process is reversible, it can not be adiabatic, but only quasi-adiabatic.

Chapter 3 A review of asynchronous logic

3.1 Introduction

In this chapter, the basic concepts of asynchronous logic will be introduced. Since this work was more focused upon adding a basic asynchronous controller to an adiabatic data-path, the depth of coverage of asynchronous logic is lower.

Like "*Adiabatic*", the term "*Asynchronous*" is also of Greek origin. It literally means "without alike time", but in the context of microelectronic design, its meaning has evolved to mean without a global time reference (a clock). Thus asynchronous logic is a logic design style that does not use a clock to synchronise the performance of operations. This is in stark contrast to the majority of digital circuits currently in existence, which have been designed using a synchronous design methodology. Unlike them, asynchronous logic does not use a ticking clock, but instead it uses a handshaking protocol to facilitate inter-stage communication.

This can have several benefits when compared against traditional synchronous designs. For *Asynchrobatic* Logic, the most important of these documented benefits are lower power and potential for design reuse. However, in some of the proposed applications, other benefits like lower electro-magnetic noise would also be of benefit [VBer94].

When compared to a synchronous system both asynchronous and *Asynchrobatic* systems will have the benefits of power-supply voltage tolerance, allowing lower voltage operation than would normally be possible for synchronous systems. This is because the synchronous design methodology requires its design elements, standard cells, to be characterised at fixed Process, Voltage and Temperature (PVT) conditions. The slowest of these is normally slowest process, lowest tolerable voltage and highest

temperature. Outside of the qualified range the circuits may fail due to violating setup-time requirements, with data-arriving too late to be correctly latched by a clock edge. This limits the voltage scaling that may be applied to a synchronous circuit. For both standard synchronous and standard asynchronous logic, the effect of lowering the voltage will reduce the power-consumption according to equation (3.1).

$$P_D = f_e C_L V_{DD}^2 \tag{3.1}$$

in which: P_D is the dynamic power dissipation,
f_e is the *effective* switching frequency,
C_L is the capacitive load, and
V_{DD} is the supply voltage.

Other low-power benefits can be illustrated by looking forward and considering a large *Asynchrobatic* system in comparison to an equivalently large asynchronous system and an equivalently large synchronous system. The lack of a global clock is also a major benefit. In the synchronous system, the ticking global clock would at minimum, reach a clock-gating element at the entry to each stage, and consequentially would waste energy. In the asynchronous and *Asynchrobatic* systems, the inactive states are just that, inactive, and as such, with no switching occurring, only leakage power will be consumed. However, in the *Asynchrobatic* system, there will be, on average, a quarter of the controllers in each of the charging states ("*Idle*", "*Charge*", "*Hold*", "*Recoup*"). The active states ("*Charge*" and "*Recoup*") require several operations, whilst the inactive states ("*Idle*", "*Hold*") require no operations. It should also be noted that for *Asynchrobatic* Logic, for any stage in the "*Idle*" state, the entire data-path has no potential difference across it from the power-supply, meaning that there is not even the possibility of having any source-drain leakage current through it! The only possible leakage current paths are gate-leakage from adjacent stages.

In his introduction Sparsø notes that research into asynchronous logic has been taking place since the 1950's [Spar01], and important theory had

been published by the end of that decade [Mull59]. However, probably the most influential development was Sutherland's invention of Micropipelines [Suth89]. These explain the control components required to implement asynchronous systems, but their hand-shaking protocol does not suit the adiabatic parts of an *Asynchrobatic* system. Asynchronous logic is more technologically mature than adiabatic logic, and as well as having been shown to be suitable for the implementation of complex processors [Pave94], commercial asynchronous processors are now available [Hand04].

3.2 Asynchronous signalling

There are two different types of asynchronous signalling conventions, two-phase and four-phase. Two-phase signalling simply reacts to a change of the signals, whilst four-phase signalling is dependent upon the levels of the signals. There are also two different methods for data transmission; bundled-data and dual-rail. For different reasons, the adiabatic data-path is already dual-rail, but the chosen implementation operates using principles far more akin to those of bundled-data. Whilst a brief background to both of the signalling styles and data transmission methods will be provided, this section will concentrate on, and elaborate more fully, the principles of systems based upon four-phase signalling with bundled-data. The asynchronous communication occurs between the Asynchronous Stepwise Charging controllers, and the bundled-data is held on the adiabatic data-path.

The handshaking protocols in asynchronous logic usually use two signals, a "*request*" from the sender to the receiver, and an "*acknowledge*" from the receiver to the sender.

In dual-rail asynchronous logic, a similar signalling protocol to that described for adiabatic logic is used. These states are used to perform completion detection so that for a dual-rail data-path, the next stage will be activated only when all dual-rail outputs have a valid state. Without error

detection, this requires an OR operation on each pair of bit-lines, and the AND of these results. Whereas the bundled-data method assumes that a delay in the control logic will match the worst delay in the data-path, and uses this to delay the sending or request and acknowledge signals.

Two-phase asynchronous signalling is dependent upon structures that are edge-triggered. An edge (either rising or falling) on the request signal is used to signal that data is available. The receiver responds with an edge (again either rising or falling) on its acknowledge signal. There is therefore no information about the state of the communication channel held in its signals' levels.

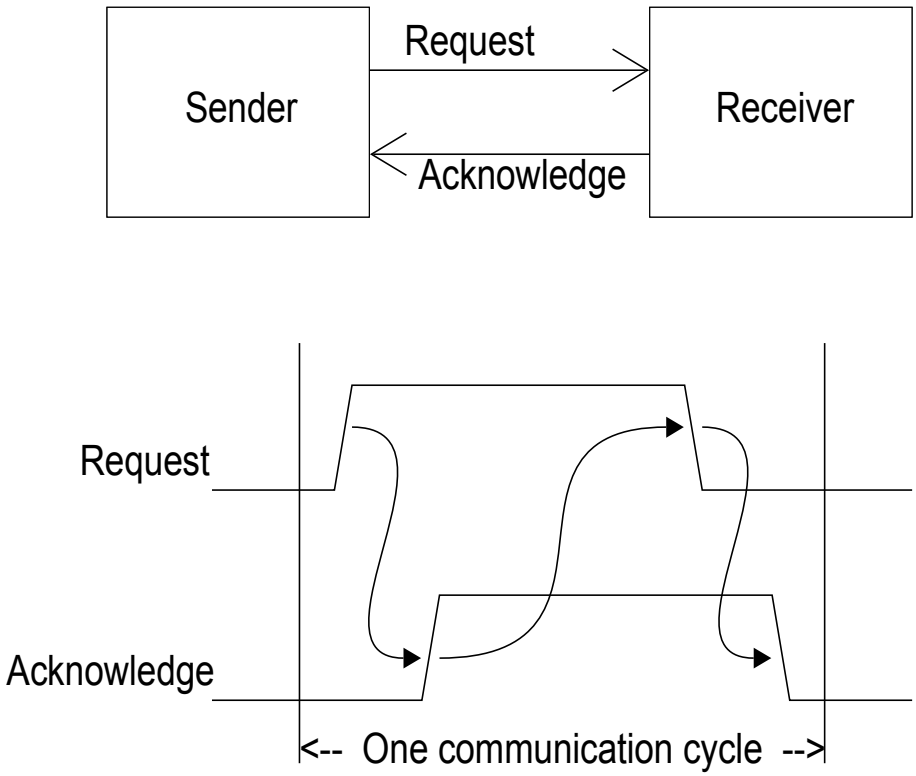

Figure 3.1: Four-phase handshaking protocol [Pave94]

Four-phase signalling fits much better with the adiabatic charging and discharging cycle. Whilst it could be possible to use dual-rail signalling, as this is available in the adiabatic data-path, this would introduce more complications than would appear to be necessary. Figure 3.1 shows the four-

phase request-acknowledge communication protocol. The communication is initiated by the sender asserting its request signal. This signal indicates that any bundled data is valid. The receiver responds to this by asserting its acknowledge signal, showing that it has accepted the valid data. The receiver negating its request signal to indicate that any subsequent data will no longer be valid. Finally, the receiver negates its acknowledge to show that it is ready to accept new valid data. Unlike the two-phase signalling, the state of the communication channel can be determined by checking its signals' levels.

The striking observation here is that the four-phases of the asynchronous communication channel directly correspond to the four states of the power-clock used by the adiabatic data-path.

3.3 The Muller C-Element

Asynchronous Logic relies upon the Muller C-Element as a principle storage element. For brevity, it will be referred to just as a “C-Element”. The C-Element is as important to asynchronous design as the D-type Flip-Flop is to traditional synchronous design. The state-space of a C-Element is shown in Table 3.1, with “X” representing a “don't care” state.

It can be seen from this table that the output of a C-Element only changes when both its inputs have changed.

Input A	Input B	Current Output	Next Output	Notes
0	0	X	0	
0	1	0	0	No change
0	1	1	1	No change
1	0	0	0	No change
1	0	1	1	No change
1	1	X	1	

Table 3.1: State table of a C-Element [Mull59]

A generalised n-input C-Element extends this by requiring all input to be identical before changing its output state. C-Elements with a greater number of inputs can be implemented either directly in logic for C-Elements with up to four inputs, or by cascading several stages of C-Elements. It is also possible to add “Reset” (to low or logic zero) or “Preset” (to high or logic one) inputs to the C-Element. As a critical component in asynchronous design, the function, design, and performance of the C-Element has been investigated in detail by others [Sham96].

Figure 3.2 shows a schematic design for a static C-Element. It is drawn in such a way that it approximately represents a stick diagram. This means that the circuit's topology would be suitable for taking the circuit to layout almost as is. The two inputs are labelled “A” and “B” and the output is labelled “Z”. They all use positive logic assertion levels. The usual circuit symbol for a C-Element is an AND gate with a capital “C” at its centre. This symbol is shown in Figure 3.3.

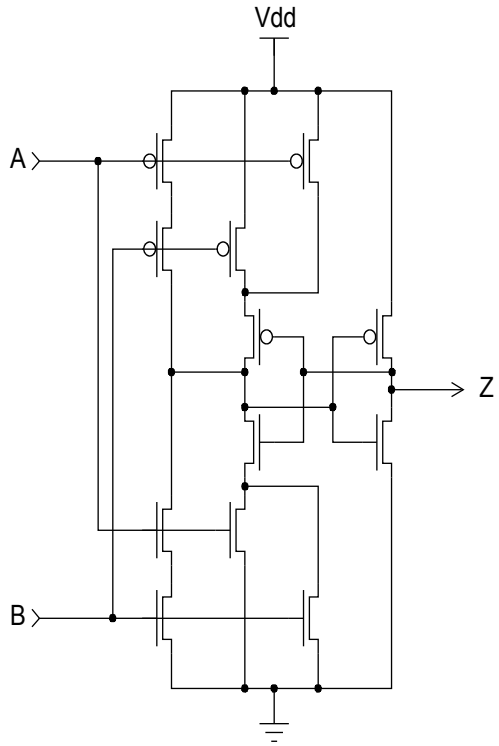

Figure 3.2: Schematic of static C-Element [Spar01]

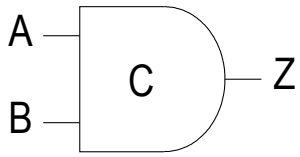

Figure 3.3: C-Element symbol [Spar01]

3.4 Asynchronous Multiplex and Demultiplex

On its own, the C-Element can be used in simple asynchronous pipelines, and can be used to manage simple "*join*" operations, where a C-Element waits for several different input sources to indicate that they have all completed an operation and are presenting valid data. For example, an adder must wait until valid data is present on both inputs (and possibly a carry-in too) before it can perform an addition. However, to manage more complex actions like decisions or loops, functions like multiplexing and demultiplexing are required.

The multiplexing (MUX) operation chooses between a number of input streams depending upon a select signal. The operation requires that both the select control signal and the selected input both have valid data. However, the unselected inputs to the multiplexer are not required to be valid, and must neither be processed nor be sent an acknowledge signal.

The demultiplexing (DeMUX) operation takes input data and a select control input, and must only forward the input data to one of several possible outputs depending upon which output is selected by the control input.

The MUX and DeMUX operations both require complementary, dual-rail signals on their request inputs from the control path. This is important, because, as will be shown in later chapters, it provides a relatively simple interface between the adiabatic data-path and the asynchronous control logic.

Circuit diagrams of these Asynchronous operations as described by Sparsø [Spar01] show possible implementations of MUX (Figure 3.4) and DeMUX (Figure 3.5). In the MUX, one of the two input channels (X and Y) is selected by asserting one of the complementary, dual-rail control signals. Once both the chosen select signal and the channel of the chosen data source are valid, this data passes through the data-path MUXes and appears on the output (Z). The DeMUX performs the opposite operation. The input data (Z) is directed to only one of the outputs (X and Y) depending upon which control signal is driven. The control signals (Ctrl) have a pair of mutually exclusive request signals, and a single acknowledge.

The control logic in both of these diagrams uses positive logic with assertion levels shown on all signals.

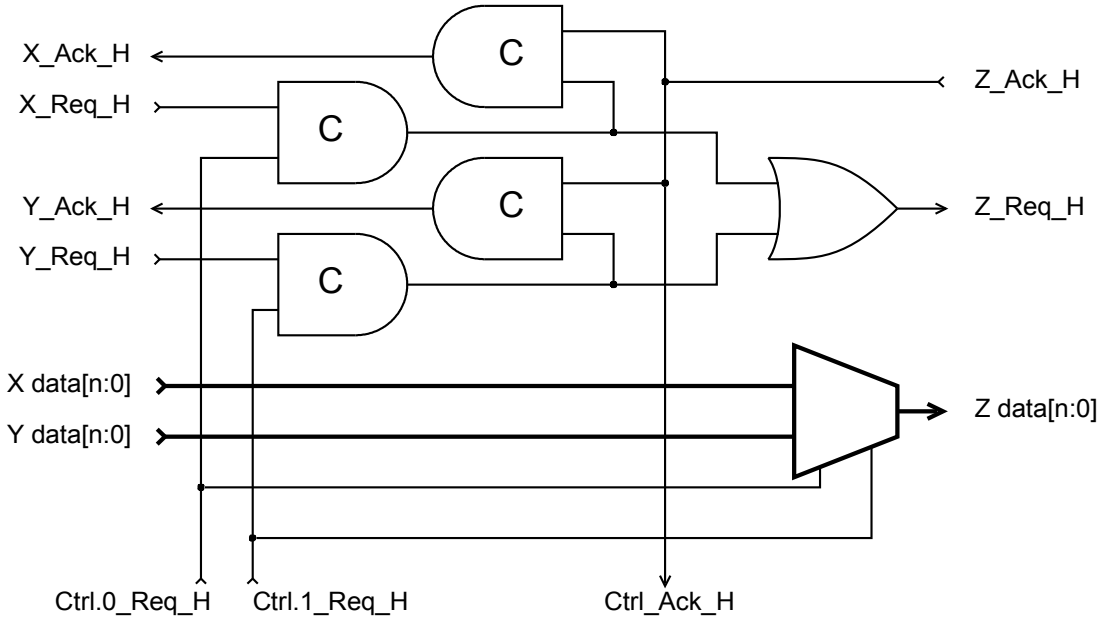

Figure 3.4: Asynchronous MUX for 4-phase bundled data [Spar01]

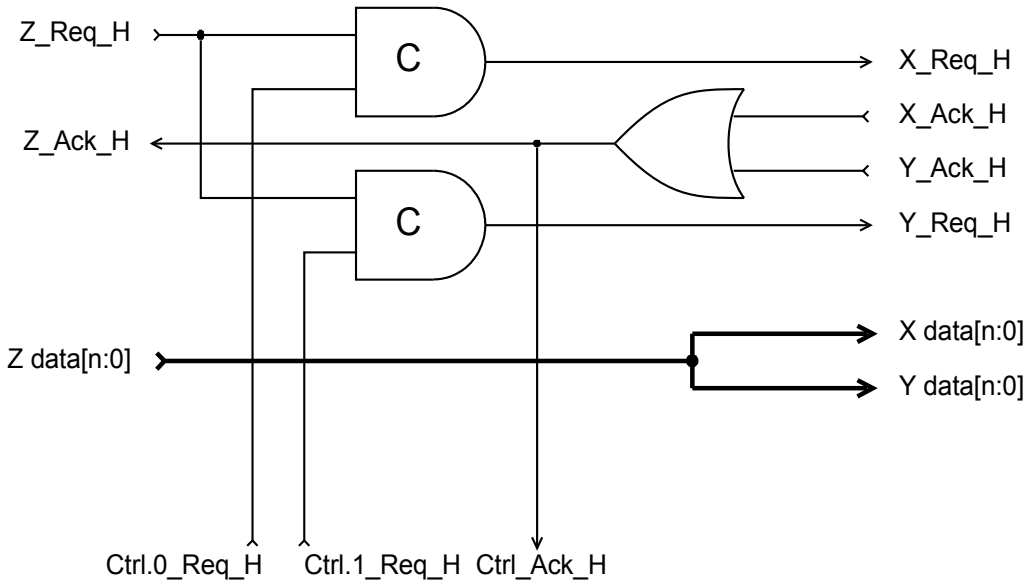

Figure 3.5: Asynchronous DeMUX for 4-phase bundled data [Spar01]

The multiplexer and demultiplexer are the only complex asynchronous decision logic used in current *Asynchrobatic* Logic implementations, but for larger systems, there are others that may be required. For example, access to a shared resource, like a single-port register-file, would need to be controlled with arbitration logic, capable of determining which calling process first requested access. The use of shared resources that can block other parts of the system can cause complexity problems, that will be briefly detailed in the next section.

3.5 Complexity issues in asynchronous systems

As the complexity of an asynchronous system increases, problems like deadlock can manifest in poorly thought-through designs. Deadlock is a failure mode where multiple processes each block each other from finishing, meaning that no process can ever finish! A good example of a system that can fail with deadlock is the “Dining Philosophers” problem, originally introduced in a less visually emotive form of five computers attempting to access five shared tape drives, and solved by Dijkstra in [Dijk65].

Although such issues do not occur in the simple *Asynchrobatic* Logic demonstrations described herein, larger systems would need to be thoroughly designed, tested and validated to ensure that they do not exhibit this failure mode. However, one of the beneficial properties of *Asynchrobatic* Logic that will be described in Chapter eight is the separation of control logic from data-path logic. This means that generally the asynchronous control logic can be verified and validated in a way that is mostly separate from the data-path. The exception to this is where the result of a data-path operation is required in the control logic, but even in this case, it should be possible to perform the necessary tests without implementing the whole data-path.

3.6 Summary

In this chapter, asynchronous logic has been introduced. The concept of using handshaking rather than a global clock was introduced, and the four-phase handshaking protocol was described. Details of the Muller-C Element, which is a vital component of any asynchronous system were provided, these included a state table, its symbol and a schematic of a possible CMOS implementation. Schematics of more complex asynchronous functions that allow multiplexing and demultiplexing to occur were provided, and the problem of deadlock, which can arise in complex asynchronous systems was discussed.

Chapter 4 Design methods for dual-rail data-paths

4.1 Introduction

In this chapter the design methods for adiabatic logic families will be considered. As noted in the introduction, it is all very well being able to implement inverters or buffers, and these are commonly used to demonstrate that circuits can be implemented. However, this is not an adequate test, because inverters and buffers are not able to perform data processing. Unless it is possible to design and viably implement more complex logic functions, a logic family will not have any real-world applications. Fortunately for *Asynchrobatic* Logic, there are various published design methodologies for Differential Cascode Voltage Switch Logic (DCVSL) circuits [Chu86] & [Karo95], and these can be used to efficiently design any of the adiabatic derivatives of that logic family which are used in the data-paths of *Asynchrobatic* Logic systems.

4.2 Adiabatic design methodologies

The easiest design methodology to describe, understand or implement is the one based upon Ordered Binary Decision Diagrams (OBDD). In this method, the logic function to be implemented is described as a tree. Bifurcation occurs in the same order irrespective of which branch is followed, hence the tree being “ordered”. Using OBDD methods is guaranteed to result in a functioning circuit, but this circuit may be sub-optimal. It has been shown that the optimal OBDD solution(s) is dependent upon the order in which the variables are evaluated. However, for four-inputs, it is possible to use an exhaustive, brute-force search to discover the minimum solution(s). This method reveals the already known result [Harr63] that every one of the 65,536 possible functions of four inputs (including degenerate functions where one or more input variable has no effect) can be implemented using only 222 different evaluations structures [Harr63], when their input order transformed by

permutation, and/or negation, and their outputs also transformed by negation. The ability to permute the input order is obvious and is available in single-rail logic. However, because this is a dual-rail logic, negation is also freely available by swapping the asserted high and asserted low inputs or outputs. This result is clearly of importance if it were desired to commercialise this design style as a commodity, as this value leads to a clearly defined limit on the required size of a universal four-input cell library. Although, as will be shown subsequently, it may be desirable to include certain functions of more than four inputs in any commercial offering.

Where the OBDD methodology becomes less useful is with AND-OR structures where the tree depth can be reduced. This can be achieved by using Quine-McClusky methods, or by applying simple transforms to appropriate structures if these are found in the design. The improvements obtained by using this method in specific cases are such that the seven-input AND-OR function, used in look-ahead functions, can be implemented with a maximum depth of four gates as can an eight-way MUX, which has eleven inputs! Free Binary Decision Diagram (FBDD) methods allow more freedom than OBDD methods as the bifurcation may occur in different orders in different branches. However, this increases the design space that must be searched if one wishes to find an optimal solution by brute force. It also makes representing the order in which variables are evaluated more difficult. Whereas the size of an OBDD's search space grows as a factorial of the number of inputs, the FBDD space grows faster.

Number of Inputs (n)	OBDD search space (n!)	FBDD search space ($n!_T$)
1	1	1
2	2	2
3	6	12
4	24	576
5	120	16,558,880

Table 4.1: Growth of search space for OBDD and FBDD optimisers

Although the sequence occurring for the FBDD result is previously known [OEIS02], it does not appear to have been named or provided with an operator symbol. In the absence of any other succinct name, this has been referred to as a "*Tree-factorial*", and designated by using the $!_T$ operator, where the subscript-T refers to a "*Tree*", and can be represented by either the function (4.1) or the recursive definition (4.2).

$$n!_T = \prod_{r=1}^{r=n} r^{2^{(n-r)}} \quad \forall n \in \mathbb{N} \tag{4.1}$$

$$n!_T = \begin{cases} 1 & if\ n=0 \\ n \cdot ((n-1)!_T)^2 & if\ n>0 \end{cases} \quad \forall n \in \mathbb{N} \tag{4.2}$$

This "*Tree factorial*" function's naming is implicitly binary, but in certain circumstances that would need to be specified, as the function can be usefully extended to higher powers. For example, ternary or tri-valued logic could be designed using, a Free Ternary Decision Diagram with trifurcations at each node. It could have its search space represented by a "*Ternary tree factorial*" [OEIS06]. Therefore this result, which does not appear to have been documented or explored for powers higher than three, may be of interest to those researching logic systems that consider logical possibilities other than just "*True*" or "*False*", a research area which is commonly know as Multi-Valued Logic (MVL). It would most likely be of benefit if useful applications are found for Free Quaternary or Quinary Decision Diagrams. The sequences created by applying this function at the fourth and fifth orders have been submitted to, and published as new discoveries by the maintainers

of the Online Encyclopaedia of Integer Sequences (OEIS) [OEIS09a] & [OEIS09b]. The generalised equations for a decision diagram of b[th] order are shown in the function (4.3) or the recursive definition (4.4).

$$n!_{T(b)} = \prod_{r=1}^{r=n} r^{b^{(n-r)}} \quad \forall\, n \in \mathbb{N} \tag{4.3}$$

$$n!_{T(b)} = \begin{Bmatrix} 1 & if\ n=0 \\ n\cdot((n-1)!_{T(b)})^{b} & if\ n>0 \end{Bmatrix} \quad \forall\, n \in \mathbb{N} \tag{4.4}$$

The rapid growth of these search spaces for higher order decision diagrams means that it would rapidly become too time-consuming to perform a brute-force search on structures with more than four inputs, and that heuristic-based methods would be needed to find the optimal variable ordering.

Finally, as well as limitations introduced by computational complexity, there can be practical limitations when it comes to implementing designs created using theoretical FBDD methods. These occur because the possibility of different variable ordering at each bifurcation can introduce twists in the required wiring. Each input variable's wiring is dual-rail, and must cross other input variable's wiring as well as the wiring of the inter-node connexions. These wiring twists would result in designs that can easily be represented as a circuit schematic, but are difficult or inefficient to layout because of the number of layers required.

4.3 The design of logic functions

The design of logic functions for implementation in adiabatic or *Asynchrobatic* Logic required detailing. This is important because if there is not an efficient and effective design methodology for complex logic, then the application is only of theoretical value. To be viable for real-world application, it is necessary to be able to implement arbitrary logic structure. *Asynchrobatic* Logic has this real-world viability because there are numerous published

design methodologies for DCVSL circuits that can be applied to any of the adiabatic families used in the data-path. These will be explained below.

This section of the thesis looks at the methodologies used to construct more complex logic functions. There are substantial differences and a plethora of optimisation opportunities available when using these logic families. The reason that these exist is due to the dual-rail nature of the logic families. Most of the methods described were pioneered in the construction of DCVSL circuits, but they remain applicable to all the adiabatic families that are based upon that static logic family. The design methods for adiabatic families that are not based on DCVSL are not within the scope of this work because only DCVSL-based adiabatic logic families were used to implement *Asynchrobatic* Logic.

The first set of methods is based upon Binary Decision Diagrams (BDDs). They are referred to by mathematicians as Directed Acyclic Graphs (DAG). The application of BDDs to switching circuits was proposed as early as the 1950s [Lee59]. However, they were popularised by Bryant [Brya86], and since then, many other derivatives have been proposed. Some of these will be detailed.

The simplest concept is the Reduced Ordered Binary Decision Diagram (ROBDD). This constructs the function as a tree of nodes. Starting from the root, which is at the bottom of each tree, each node performs a bifurcation determined by the value of one of the input variables, and irrespective of which path is followed, the sequence of the input variables is invariant. The full tree can then be reduced by removing redundant nodes. There are two situations where redundant nodes can occur. The first is if both branches of the node point to the same function. This means that this node plays no part in the decision process, and the node can be removed. The other is where a node points to a function for which evaluation logic already exists. In this case, the duplicated evaluation logic can be removed and the node can be

redirected to point at the pre-existing evaluation logic. An example of these optimisations is shown for the three-input majority function in Figure 4.1. Figure 4.1(a) shows the unoptimised, OBDD solution. It can be seen that node 4 and 7 can be removed as redundant, because irrespective of their input, they both deliver the same result. This minimisation is shown in Figure 4.1(b). It is also evident that nodes 5 and 6 duplicate the same function, meaning that one of them is redundant and can also be removed. This results in Figure 4.1(c) which is the optimal ROBDD implementation of the three-input majority function.

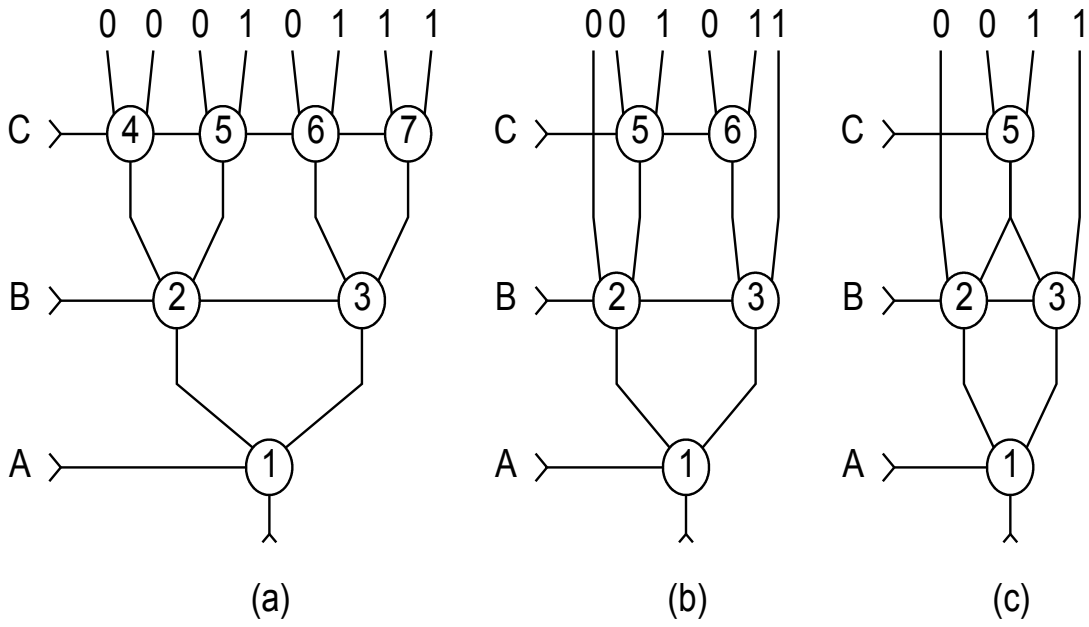

Figure 4.1: An example of OBDD minimisation

There are a number of practical, physical consequences of performing the reductions to achieve a ROBDD solution. The first one, which can be seen in the example, is that even for a symmetrical output function, some inputs may present a higher capacitive load due to containing more decision nodes. The implications of this would need to be considered to avoid problems caused due to excessive fanout.

There is another issue, which cannot be seen in the above example is the sensitivity of the ROBDD methodology to the variable ordering. This can be demonstrated by considering a three-input function that implements a two-

way multiplexer. Table 4.2 shows the different truth tables that are obtained when two possible variable orders are used. The OBDDs generated from these two possible orderings are shown in Figure 4.2. Figure 4.2(a) shows how an optimal variable ordering of {S,A,B} can be reduced, as described above, it results in a ROBDD tree with three nodes, which requires a six NMOS device implementation, with a load of one gate for each of the six dual rail inputs. However, Figure 4.2(b) show how a sub-optimal variable ordering of {A,B,S} reduces less well. The resulting ROBDD tree contains five nodes, requiring an extra four NMOS devices, doubles the load to two gates on both the B and S dual rail inputs, and adds two extra internal source-drain connections.

Optimal ordering (S,A,B)				**Sub-optimal ordering (A,B,S)**			
S	**A**	**B**	**Z**	**A**	**B**	**S**	**Z**
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	1	0	1	0	0
0	1	1	1	0	1	1	1
1	0	0	0	1	0	0	1
1	0	1	1	1	0	1	0
1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1

Table 4.2: Effect of variable ordering in OBDDs

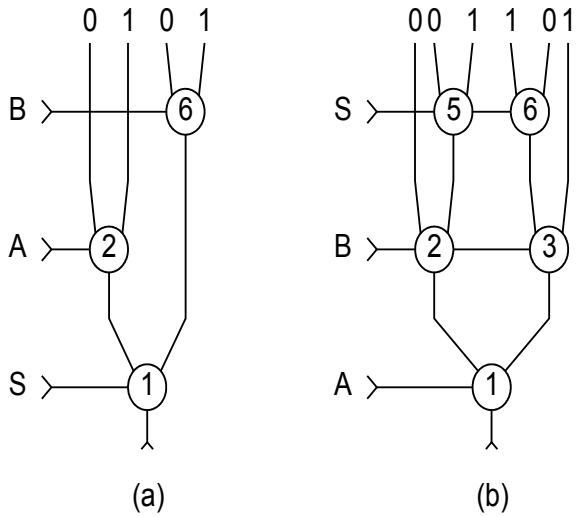

Figure 4.2: Effect of variable ordering in OBDDs

The majority of the mathematical and computer science theory relating to the optimisation of ROBDDs is beyond the scope of this thesis, but it is necessary to provide some details. The optimisation problem of finding the best variable ordering for a ROBDD tree has been shown to be in the hardest class of computational complexity problems, that is, NP-complete [Boll96], but fortunately, with just four input variables, an exhaustive, brute force search (trying every possible combination) does not require excessive computation time. As well as aiming to reduce the number of decision nodes, other metrics may be of value if two different ROBDD structures have the same number of nodes, but different topologies. These could include the number of intermediate nodes, the maximum path length, or the total path length.

Having established the desired function as a ROBDD, it is then necessary to translate this into a circuit capable of being implemented. This is a simple task, as each node can be directly mapped to a five-terminal network consisting of two NMOS devices. Figure 4.3 shows this mapping for the design of pull-up PFAL trees. For pull-down trees, the polarity of either the inputs or the outputs needs to be swapped. However, notwithstanding whether the derived tree uses pull-up or pull-down devices, the topology remains identical, allowing the same design methods to be used for ECRL, PFAL or IECRL gates.

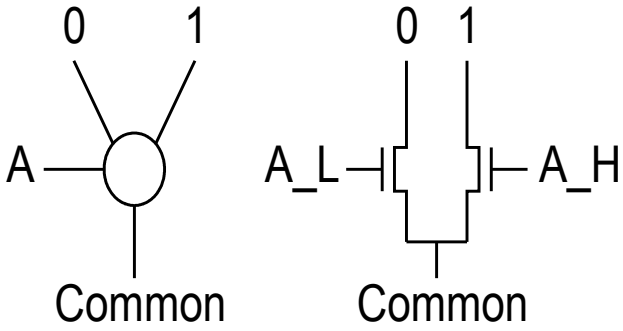

Figure 4.3: Mapping a BDD node to a pair of NMOS devices

The Free Binary Decision Diagram (FBDD) [Bern93] removes the caveat that variables must be evaluated in the same order in every part of the decision tree. This allows variables to be evaluated in different orders in each branch with the obvious requirement that every variable is evaluated at some point between the root and the final branches. If a FBDD is optimised following the same rules as would be used to minimise an OBDD to a ROBDD, then a Reduced Free Binary Decision Diagram (RFBDD) can be obtained. The same translation method can be used to convert a RFBDD into a circuit.

RFBDDs can result in structures with a smaller number of decision nodes than ROBDDs, although for trees with four-inputs or fewer, the benefits are minimal. Unfortunately, as mentioned previously the use of RFBDDs can have the undesirable side-effect of introducing twisted structures that are more difficult to layout. For this reason, FBDD methods were not pursued further.

Another method that could have been used to design functions, was 123-Decisions Diagrams (123-DD) [Jaek97] & [Arma98]. However, this method was not used in the design of structures used in this thesis, but may be of use in future work.

The there are two non-BDD which will be described. The first uses the Quine-McClusky method [Chu86]. It uses prime implicant tables to find minimum variable cover. One advantageous feature of this design style is that the root of the evaluation tree can have multiple connections. This means that more source-drain capacitance is directly connected to the power-clock, rather than being on internal nodes. This means that it can be directly recovered by the power-clock, whereas internal nodes can retain stored charge on the capacitance of the source-drain connections. This cannot be fully recovered.

The other method can only be used to convert certain static CMOS circuits into dual-rail implementations. The two complementary evaluation paths of the static CMOS gate are split. The NMOS evaluation logic has its inputs relabelled with the asserted high inputs and connected between the power-clock and the asserted low output. The PMOS evaluation logic is replaced with NMOS devices, has its inputs relabelled with the asserted low inputs and is then connected between the power-clock and the asserted high output. If the gate is followed by an inverter, its effect can be removed by swapping the assertion levels of the outputs. This method can be visualised as folding the CMOS evaluation path in upon itself. It is this method of direct conversion from static CMOS that allowed the simple implementation of the three-, five- and seven-input AND-OR structures which were used in radix-four carry look-ahead logic.

Comparison of the two different two-input AND gates obtained by different design methodologies leads to the conclusion that certain OBDD structures can be transformed into more optimal ones using a simple substitution. The possible implementations of the pull-up logic for a two-input PFAL AND gate are shown in Figure 4.4. Figure 4.4(a) resembles the complementary NMOS and PMOS stacks that would be found in a static CMOS AND gate. The original unoptimised BDD tree is shown in Figure 4.4(b). This type of transformation can be applied to n-input AND or OR functions, which in the OBDD methodology would be formed by a lopsided

tree with a single path to one evaluation node, and the remainder of paths going to the other evaluation node with branches at every level. By making the evaluation in parallel, it reduces the height of one of the paths. This reduces the internal capacitance within the evaluation tree, which lessens the irreversible, non-adiabatic losses. It also allows certain logic structures with a larger number of inputs to be implemented. Crucially, these high-input functions include the multi-stage AND-OR logic functions used in arithmetic look-ahead structures.

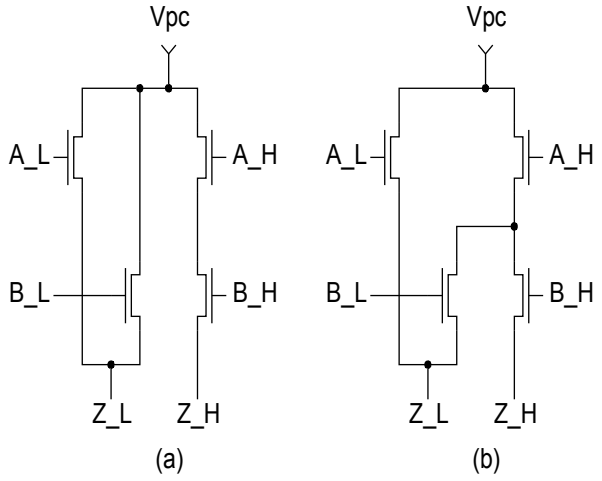

Figure 4.4: Possible implementations of a two-input AND function's pull-up logic

4.4 Summary

In this chapter, some of the design methods for dual-rail logic families have been discussed. These methods include OBDDs, FBDDs and Quine-McClusky. The problems caused by the OBDD design method's sensitivity to variable ordering have been discussed, as has the problem of FBDD methodologies producing designs that are difficult to physically implement. The concept of a "*Tree factorial*" has been introduced with respect to calculating the space that needs to be searched if a design needs to be optimised using a brute-force search of a FBDD's design space. This concept of "*Tree factorials*" was extended to higher orders, and resulted in the discovery of previously unknown, but potentially useful integer sequences. It

has been shown how the nodes in a decision diagram can be converted into physical designs. Finally, it has been shown that the use of Quine-McClusky based methods can be applied to reduce the height of a decision tree such that it can be physically implemented.

Chapter 5 An introduction to *Asynchrobatic* Logic

5.1 Introduction

This chapter describes *Asynchrobatic* Logic both as a novel concept and as a practical implementation. It commences with the thought processes that led to the development of this novel technology. It will be followed by further chapters looking at how it has been refined from initial conception to a series of structures which show it can be viably used to implement complex processing structures.

5.2 The concept of *Asynchrobatic* Logic

The derivation of the term "*Asynchrobatic* Logic" is from a concatenation and shortening of **Asynchro**nous (quasi-) Adia**batic** Logic. When this novel concept was introduced in 2004 [Will04], all adiabatic logic was designed with strict constraints on the power-clock phasing requirements. The removal of this barrier has important ramifications for design reuse, as it removes the need to add buffer stages to the pipeline in order to correctly synchronise the multi-phase power-clocks. This frees designers by allowing them to consider just the data-path interfacing, without having to be concerned about the clock phasing associated with that interface. This is a direct benefit of using an asynchronous design methodology.

The idea underlying the creation of *Asynchrobatic* Logic is to find a viable way to operate a low-power adiabatic (or quasi-adiabatic) data-path asynchronously. This concept is obviously challenging as, until this time, all adiabatic logic families were documented as using multiple, multi-phase power-clocks, which, in general, perform their charge recycling using inductive elements, whereas asynchronous logic is required to operate using handshake signals and to not have signals that would be defined as a global clock. The theoretical benefits of *Asynchrobatic* Logic from a low power point-

of-view should include less switching losses, which is due to switching only occurring when active, charge recovery and recycling, and almost complete leakage reduction in inactive circuitry, due to the absence of any potential difference across inactive data-path elements.

In synchronous adiabatic systems, whether driven by a resonant driver or by synchronous Stepwise Charging, there is a strict phase relationship between the phases of the power-clock. The ideal waveform requires one of each of the four phases to be in each of the possible states, and a 90° phase difference is required between the sinusoidal power-clocks from a resonant driver. *Asynchrobatic* Logic relaxes these relationships. In *Asynchrobatic* Logic, a pipeline stage may be left in its "*Idle*" state indefinitely. From an "*Idle*" state, the next state is the transient "*Charge*" state. This may only be entered if the adjacent previous pipeline stage is in its stable "*Hold*" state, and subsequent adjacent pipeline stages (there may be more than one following a demultiplexer) are in their stable "*Idle*" states. The "*Charge*" state will be followed by the stable "*Hold*" state. From the "*Hold*" state, the next state is the transient "*Recoup*" state. This may only be entered if the adjacent previous pipeline stage is in its stable "*Idle*" state, and subsequent adjacent pipeline stages are in their stable "*Hold*" states". The "*Recoup*" state is followed by the stable "*Idle*" state and completes the cycle. These relationships generate waveforms that closely resemble those detailed earlier in Figure 2.5, but with more fluidity due to overlap of stable states between adjacent pipeline stages. As alluded to, these relationships become a little more complex for the demultiplex operation, as "*Recoup*" can occur when an adjacent subsequent pipeline stage in an unselected path is in its stable "*Idle*" state. When selecting between two different inputs using a multiplex operation, they only hold for the control inputs and active data-path.

5.3 The components of an *Asynchrobatic* Logic implementation

There were several novel and inventive steps required to create *Asynchrobatic* Logic. The initial observation that led to these was that the four-phase handshaking protocol of asynchronous logic could be directly mapped onto the four-phase power-clock of certain adiabatic logic families. It was then realised that each stage of elements in the data-path pipeline can be driven by an individually created local power-clock, and that this local power-clock can be generated using a Stepwise Charging (SWC) methodology. Whilst the first of these realisations is a relatively simple reuse and re-appropriation of ideas from vanilla asynchronous logic, the second realisation was a substantial inventive step. Prior to this, SWC had only been widely documented as being used to drive large capacitive loads, for instance as a low-power pad-driver for off-chip components [Sven94b] & [Sven96]. Furthermore, these SWC systems had been controlled by synchronous Finite State Machines (FSMs) [Sven95]. The first published suggestion of using Asynchronous Stepwise Charging (ASWC) to drive an adiabatic data-path, was made in May 2004 [Will04]. Although the idea of using SWC to drive an adiabatic data-path was independently discovered for this work, the idea of using synchronous SWC like this appears to have been first disclosed to the Japanese Applied Physics press by Nakata in 2000 [Naka00], and eventually reported to the Japanese Electronics press in November 2004 [Naka04]. However, Nakata's papers continued to use synchronous FSMs to drive the adiabatic data-path. Irrespective of this separate and independent reportage, *Asynchrobatic* Logic vastly expands the available complexity of circuitry, removes clock-phase restrictions from the control logic, and as will be shown, allows complex data-processing structures to be implemented in a reusable fashion.

A further realisation was that the required tank capacitors could be shared between all the *Asynchrobatic* pipeline stages. The sharing of these capacitors also leads to a theoretical conclusion that in an average case for a

simple sequential pipeline, one quarter of the data-path stages will be in each of the four possible clock phases. This means that as well as being recovered to the tank capacitors, some energy may be directly recovered from one part of the data-path to another. Furthermore, the fact that these capacitors can be implemented as on-chip devices makes the design viable for real-world implementation. Again, this is an improvement over previous adiabatic logic implementations, because off-chip inductors are required for resonant energy-recovering power supplies. These ideas allow an *Asynchrobatic* system to be split into three reusable sub-systems, an asynchronous controller, a Stepwise Charging circuit and an adiabatic data-path. This separation allows complex designs to be created from smaller, easily verified building blocks, and this potential for design reuse is essential if there were a future desire to produce commercial products based upon *Asynchrobatic* Logic.

The one minor restriction to real-world operation is the power used in generating the charge recovery drivers [Inde94]. It is therefore necessary to amortise the cost, in terms of power consumption, of the Asynchronous Stepwise Charging controller over the width of the data-path. This means that in general, *Asynchrobatic* Logic will be more suitable to higher data-width applications. However, even at modern 64-bit processing widths there is sufficient amortisation of these costs, and applications with 128-bit processing widths or greater can be found, both in general-purpose and niche applications. This means that practical applications can be found in which to use this logic style.

5.4 Design of Control Structures

The control structures need to process handshake inputs from the stages adjacent to the current one. A request handshake from a previous stage will indicate one of two states. When it makes a rising zero-to-one transition, it shows that data is available, allowing the asynchronous controller to initiate the stepwise charging process to latch that data. Conversely, when

it makes a falling one-to-zero transition, it indicates that the charge on the inputs has been recovered, and that the asynchronous controller should initiate the stepwise discharging process to recover the charge. Depending upon the direction of its transition, an acknowledge from the subsequent stage will either indicate that it is in the "*Idle*" state and that its inputs may be changed, or that it has evaluated its inputs and that charge recovery may commence. For simple pipeline stages, this handshaking can be achieved using only a C-Element, as described in Chapter three. For operations more complex than a simple pipeline, the number of components in the asynchronous control logic increases. This allows the creation of MUX and DeMUX operators. The implementations of these closely follow that of Sparsø [Spar01]. The original asynchronous versions were shown earlier in Figures 3.4 and 3.5. The MUX's data-path is driven by a standard ASWC that follows the asynchronous MUX circuit. In the DeMUX, the previous stage drives the inputs of both subsequent stages, but the DeMUX only instructs one to be activated.

5.5 Design of Stepwise Charging Logic

The Asynchronous Stepwise Charging controller logic is the glue that allows *Asynchrobatic* Logic to function. It is the confluence of these novel ideas which differentiates *Asynchrobatic* Logic from pure asynchronous systems and from pure adiabatic systems. The output from the asynchronous controller is used to indicate whether stepwise charging or stepwise discharging is to be performed. When this input changes, it is fed into a series of pulse generators. These pulse generators must generate pulses on each transition of the output from asynchronous controller. They are constructed by performing an exclusive-OR of the input and output of a variable delay element driven by the output from the asynchronous controller. The variable delays were constructed using analogue-controlled current-starved inverters [West94], of the design shown in Figure 5.1. The bias circuit is very simple, and allows the delay to be varied, it would be anticipated that in larger

designs, this circuit would be optimised so that less current is drawn, and that the cost power drawn by its bias circuit would be distributed across a large number of delay elements. Other alternative delay structures that could be considered include shunt capacitor delays [West94] and digital delays using multi-tapped buffer chains. The shunt capacitor delays were not used because of the increase in the capacitive load was likely to be a source of power-consumption, and the multi-tapped buffers were not used because the number of switching events necessary to create the delay was likely to be a source of power-consumption.

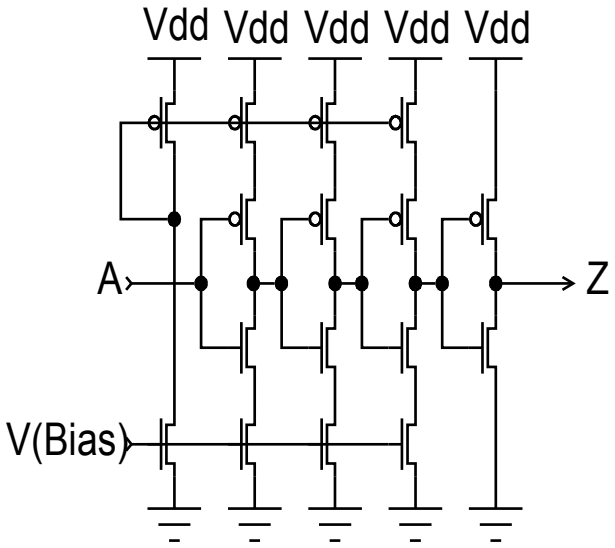

Figure 5.1: Current-starved inverters as variable delay [West94]

The design of the XOR gate is shown in Figure 5.2, although other equally valid designs for XOR gates exist.

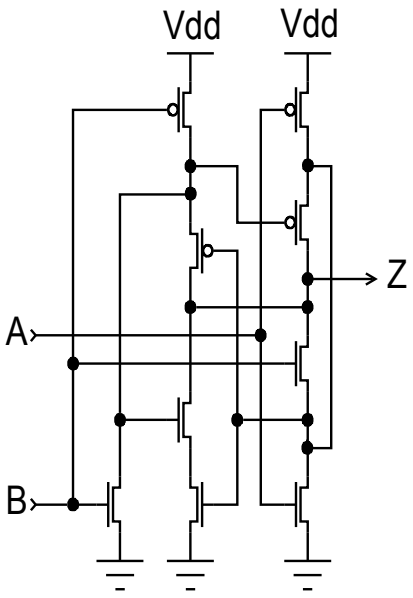

Figure 5.2: Static CMOS 2-input XOR gate [MIT04]

The pulses are generated in the same order irrespective of whether charging or discharging is occurring, but the SWC circuit requires a different, but symmetrical, switching order depending on whether it is performing charging or discharging. However, since the output of the asynchronous controller indicates the direction of the SWC circuit, it can be used as the select input to multiplex the pulses appropriately. Based upon this, the pulses are routed, using pass-gate multiplexers where necessary, to the correct intermediate switches in the stepwise charging power-clock logic. If the driven switch is a PMOS device, then the signal is also inverted. The terminal switches, which supply the power and ground voltages of the stepwise charging power-clock logic are driven by standard logic functions that are conditional on the “first input into” and “last output from” the pulse generator being equal. This relationship is derived as follows: If the signals are not equal, then the pulses are still being generated. If both signals are low, then the switch for the NMOS device to ground (P0_H) should be active. If both signals are high, then the switch for the PMOS to the power-supply (P4_L) should be active. The final output from the pulse generator is also buffered to be used as the handshake signal from the current stage to the adjacent stages. The use of this type of delay makes this part of the internals of the

SWC logic fall into the “self-timed” class of asynchronous circuits. Circuit diagrams for these components are shown in Figures 5.3 and 5.4. In Figure 5.3, it can be seen that the number of pulse signals generated can be varied by changing the number of pulse generating delay and XOR gates, a feature which could allow the design to be made modular.

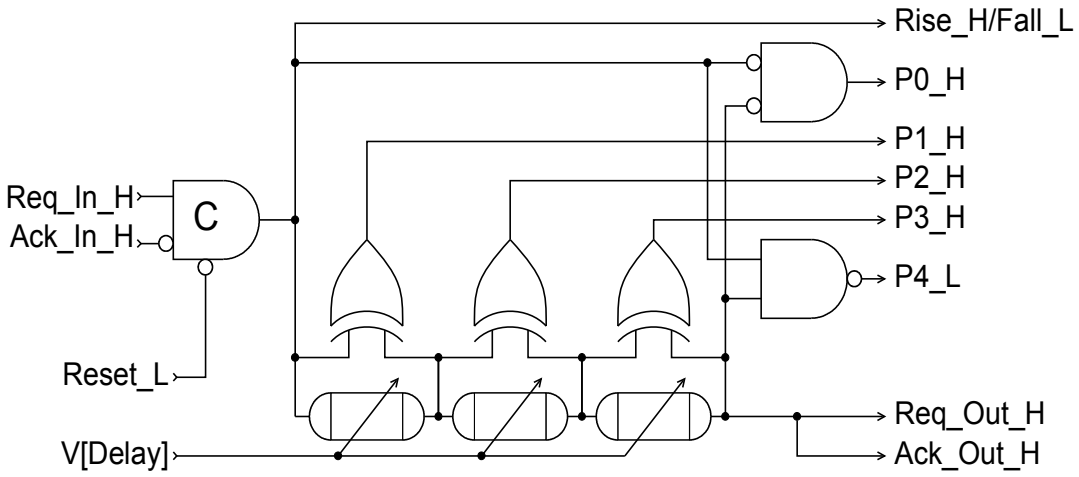

Figure 5.3: An asynchronous stepwise charging controller [Will04]

The circuit shown in Figure 5.4 shows the three possible configurations that can be used for stepwise charging, the transmission-gate version will operate universally, but depending upon the convergence voltage of a particular step, it may be possible to optimise this circuit by removing the NMOS device for a convergence value close to the power-supply voltage, or removing an PMOS device for a convergence value close to ground. This can be done because the threshold voltage of the removed device would mean it carried almost no current. In the case where a NMOS device is removed, this is likely to prolong the time taken for the tank capacitors to converge if they all start from ground.

In contrast to the resonant charging schemes used in traditional adiabatic systems, those based on SWC do not require any extra circuitry to operate correctly upon start-up [Dhar96]. Furthermore, for an ideal load with

ideal pulse widths, it can be demonstrated that the voltages across the tank capacitors will converge [Naka04].

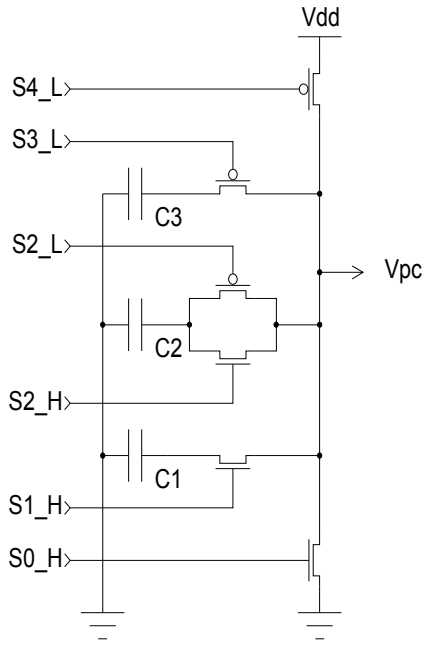

Figure 5.4: A Stepwise Charging circuit [Sven94a]

However, there is a risk that charging the data-path too slowly will cause it to lag behind the asynchronous signalling, and that this could lead to logical evaluation errors. Fortunately, the ASWC serves to create a margin of safety. This is because, even if the data-path is not fully charged, there should be differential between the two inputs. This means that, provided there is sufficient differential, the next stage will behave like a sense-amplifier and still be able to determine the state of its inputs. This is sub-optimal operation, and therefore likely to result in increased power consumption. The greatest risk of this type of failure mode occurring is during start-up when all the tank capacitors are discharged and the only charge is supplied from the main power rail. An obvious and simple solution to this is to pre-charge some of the tank capacitors (those switched later during the charging cycle) during the reset time. An alternative that was considered, but not pursued, was to use the dual-rail nature of the data-path to provide completion-detection [Spar01]. This could be achieved in a similar way to self-timed asynchronous completion-detection, by using an OR gate on the outputs of the data-path element furthest from the SWC circuit. However, because of the analogue

nature of the ramp-like, stepwise staircase waveform, this would result in logically indeterminate voltage values being applied across a digital gate's inputs. This gives rise to the possibility of causing short-circuit currents to flow in these completion-detection gates for an unacceptably large proportion of the cycle time. Not only would this have a negative impact on the circuit by increasing its power consumption, but it could also provoke power-related device failures and life-time reduction due to factors such as electromigration [Blac69].

5.6 Design of data-path logic

The data-path logic for *Asynchrobatic* Logic can be designed in exactly the same way as it would be for adiabatic logic. A designer would be free to choose from a variety of different adiabatic families including, but not limited to; Efficient Charge Recovery Logic (ECRL), Improved Efficient Charge Recovery Logic (IECRL), Positive Feedback Adiabatic Logic (PFAL) or Efficient Adiabatic Charge Recovery Logic (EACRL). The first publication of *Asynchrobatic* Logic used ECRL, but later work was performed using PFAL, as it has been shown to be more power efficient [Blot02] & [Will05], and because of its potential for reversible operations. Because the design is pipelined, making the data-path as short as possible will be beneficial from a speed point of view. This leads to the conclusion that higher-radix arithmetic and computation structures should be considered, and, where possible, stages of logic merged. However, *Asynchrobatic* Logic does simplify some parts of data-path design because in contrast to a normal synchronous adiabatic data-path, other than having a sufficient number of stages to avoid deadlock, *Asynchrobatic* Logic does not impose any phase constraints on re-entrant parts of the design, leaving the join to be correctly managed by the asynchronous control logic. This is in contrast an adiabatic data-path with a resonant power-clock which would require paths to be padded with buffers so that their lengths are a multiple of the phases in its clocking scheme. Even in simple cases, this could add up to three stages of buffers. More complex

synchronisation could require either more buffer stages, or could require the control structures to be made more complex. *Asynchrobatic* Logic makes the data-path completely reusable. This potentially allows optimal versions of frequently used functional blocks to be developed and commercialised as Intellectual Property (IP).

5.7 Implementation of an *Asynchrobatic* pipeline

The proof-of-concept for *Asynchrobatic* Logic was published at the 2004 International Symposium on Circuits and Systems [Will04]. It used the Asynchronous Stepwise Charging control logic previously detailed, but coupled this with an ECRL data-path. However, it correctly observed that other adiabatic families, including PFAL, could be used, and that OBDD design methods could be used to design more complex cells. It compared the new *Asynchrobatic* logic methodology with a standard asynchronous methodology. Some of the results from this paper are detailed below.

5.7.1 Comparison of *Asynchrobatic* and asynchronous buffer chains

The initial proof of concept design was implemented using 16-bit and 32-bit wide 12-stage pipelines of ECRL buffer chains. These were drawn using Chipwise [Kent98] and simulated using SPICE, on a 0.7μm (0.8μm drawn) process in the "typical" process corner. Figure 5.5 shows the floor-plan of the ASWC and Figure 5.6 shows the layout of the ASWC controller circuit as described previously. This layout uses a combination of the schematics shown in Figures 5.3 and 5.4. It was obtained from automatic compaction of a stick diagram. The layers are shown as follows: active in green, polysilicon in red, the first metal in blue and the second metal in cyan. The C-Element is at the bottom of the layout, and the asynchronous power-clock V_{pc} is at the top of the layout. In general, the NMOS devices were made minimum size, and the PMOS devices were double the size of the

corresponding NMOS devices. Although some devices with a larger fan-out were made larger.

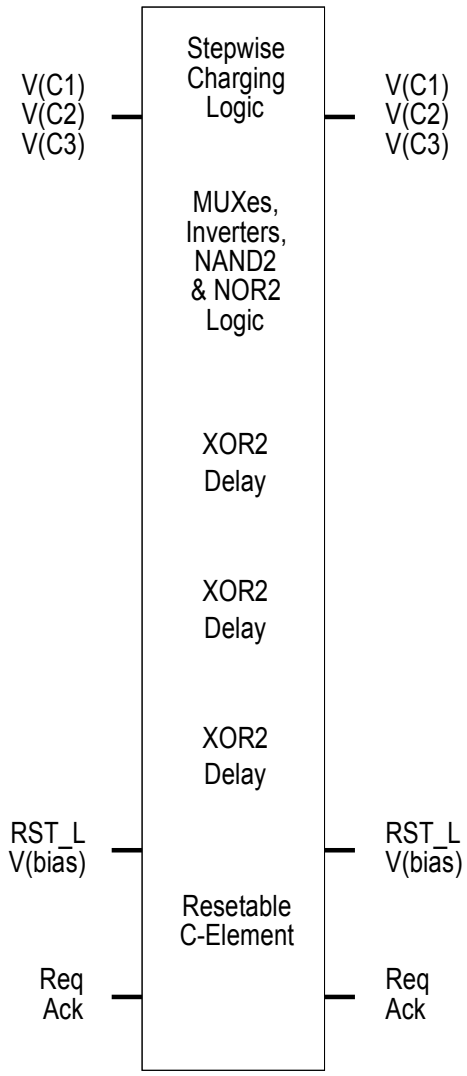

Figure 5.5: Floorplan of an Asynchronous Stepwise Charging Controller

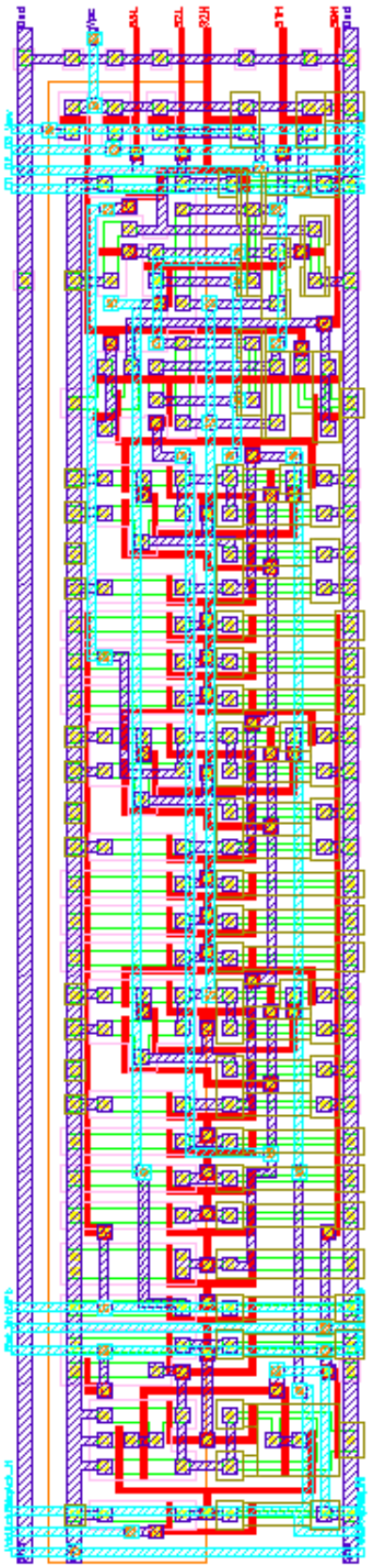

Figure 5.6: Layout of an Asynchronous Stepwise Charging Controller

This layout was extracted, as was a layout (not shown) for a data-path of ECRL buffers. These buffer were driven with three different input switching probabilities, 0% (static, never switches), 50% (switches every other transaction), and 100% (switches every transaction). After a simulation period of 10μs (50 transactions), the cumulative current consumption was measured. This was done in SPICE, and will be detailed in Chapter 6. To demonstrate the efficiency of the data-path, separate measurements were taken for the control logic and the data-path logic. The *Asynchrobatic* Logic versions were compared against an asynchronous T-Latch Micropipeline as detailed by Paver [Pave94].

The worst-case average power consumption for a single transaction of *Asynchrobatic* data-path element was 1.4pW per data-path bit, and the *Asynchrobatic* SWC control logic used 248pW per transaction. This compares with 7pW per data-path bit per transaction for the asynchronous T-Latch, and 124pW per transaction for the asynchronous control logic. This shows that the quasi-adiabatic data-path uses five times less power than the standard data-path, but the ASWC control logic uses twice as much power as the standard asynchronous control logic. Graphs showing the various cumulative current consumption data are shown in Figures 5.7, 5.8 and 5.9, and were first presented in [Will04].

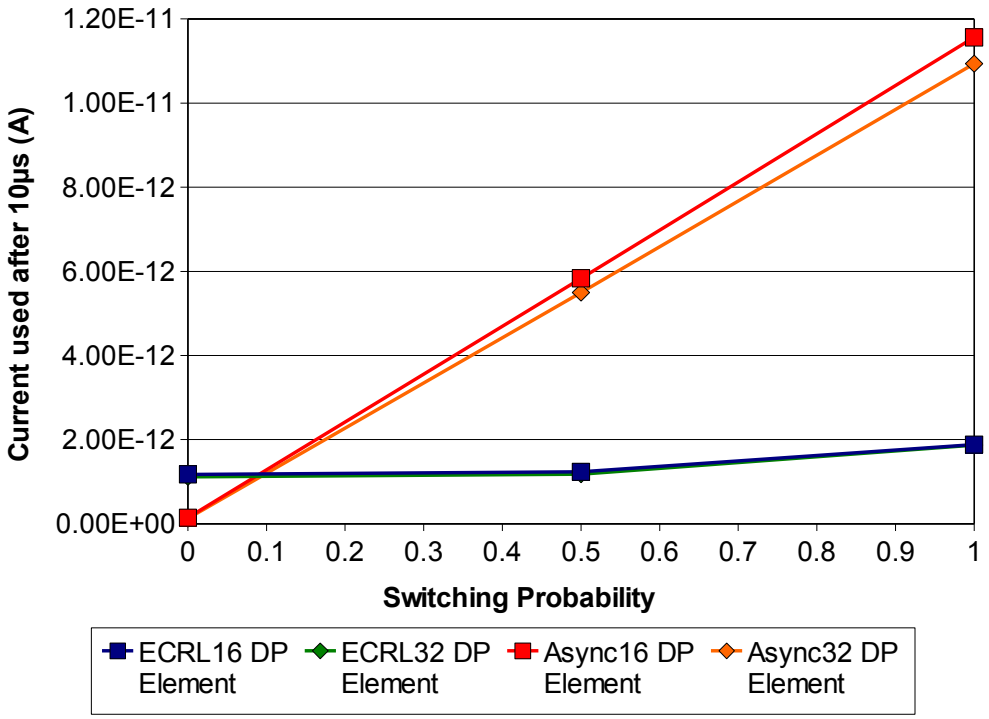

Figure 5.7: Cumulative current consumption for data-path elements

Figure 5.7 shows the cumulative current consumption of a single data-path element. As would be expected, the asynchronous data-path consumed negligible power when static, but as the amount of switching increased, its power consumption increased linearly. It can be seen that the purely asynchronous design's data-path power consumption follows linearly follows Equation (3.1). Conversely, the ECRL data-path showed a more-or-less fixed power consumption irrespective of the amount of switching, although its power consumption did increase slightly as the switching probability increased. This variation can most likely be attributed to the non-linear nature of an ECRL circuit as a load on a SWC circuit causing the convergence voltages of the tank capacitors to vary, thus affecting the amount of recoverable charge.

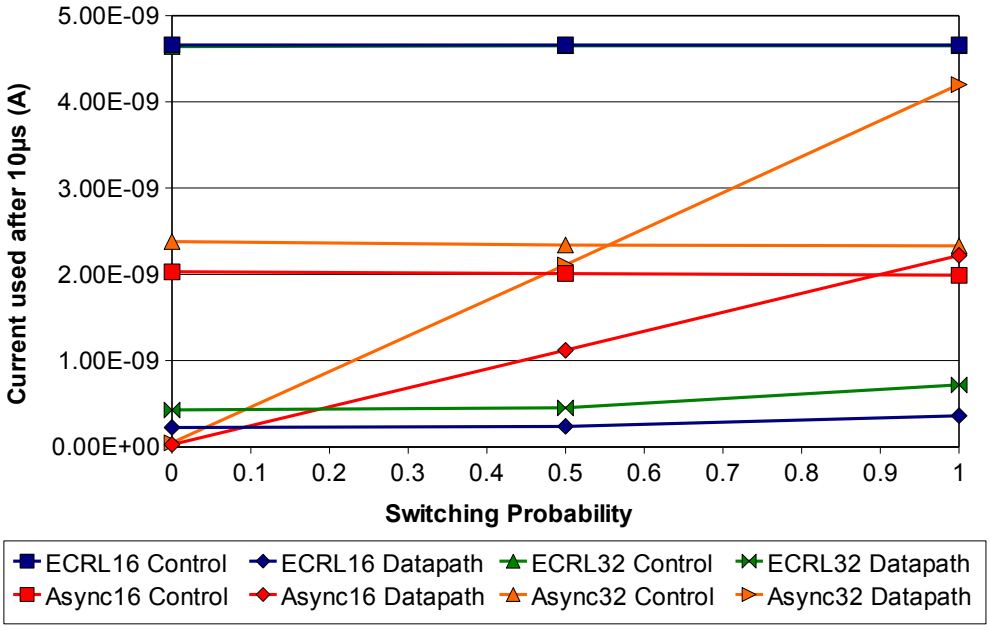

Figure 5.8: Cumulative current for the control and data-path

Figure 5.8 shows the cumulative current for the each sub-system. Because the asynchronous control logic drove its local clock, its power consumption was independent of the switching probability in the data-path, but was affected by the width of the data-path structure it was driving. However, the ASWC control logic had a fixed power consumption that was independent of both the switching probability in the data-path and the width of the data-path structure being driven. The fixed power-consumption of the ASWC control logic means that its power consumption cost can be amortised by increasing the width of the data-path.

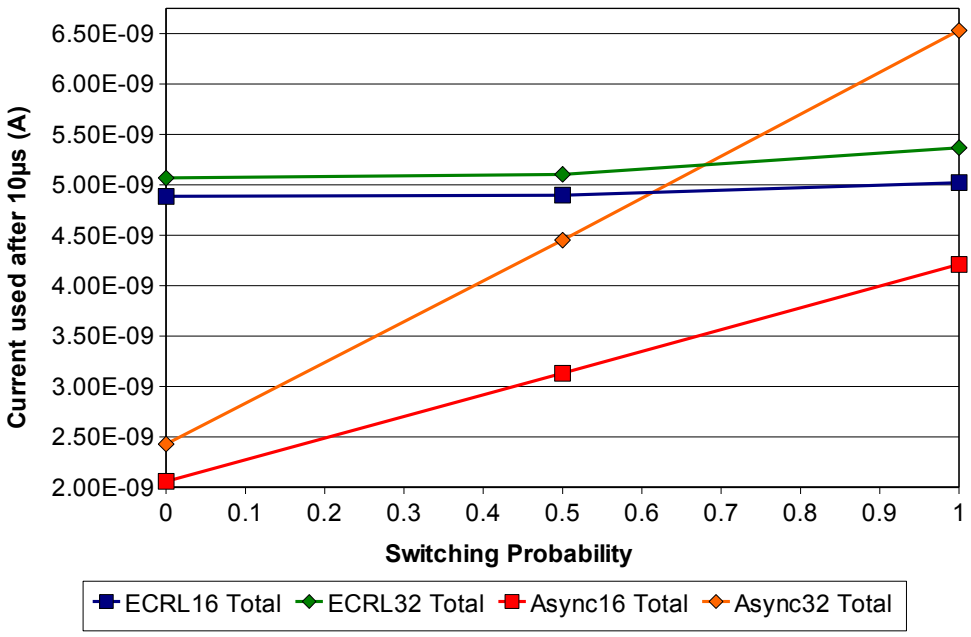

Figure 5.9: Total cumulative current

Figure 5.9 shows the total cumulative current for the complete system. It is clearly evident that a 16-bit, ECRL *Asynchrobatic* Logic pipeline will always be less power efficient than an equivalent asynchronous pipeline. However, for a 32-bit, ECRL *Asynchrobatic* Logic pipeline, it can be seen that if more than 70% of the bits (23 bits) change, then the *Asynchrobatic* Logic implementation will be the more efficient.

Figure 5.10 shows simulation output of an ECRL inverter driven by the Asynchronous Stepwise Charging Logic used in *Asynchrobatic* Logic. Its inputs were operated with a 50% switching probability, that is a waveform that repeats the following four values "0,0,1,1...".

The upper chart shows the asynchronously generated stepwise power-clock. Because the power-clock was sampled from the fourth pipeline stage, it is labelled "V(PC#4)". It is shown along with the voltages across each of the tank capacitors (labelled "V(C1)", "V(C2)" and "V(C3)"). The plot shows the capacitors after they have converged, but the ripples which can be seen are caused by charging and recovery occurring in the other pipeline stages.

The lower graph shows the complementary outputs of an ECRL buffer. It can clearly be seen that ECRL buffers do not perform complete recovery from the asserted output, and that if no input switching occurs, some charge is carried over to the next transaction, but that when input switching does occur any stored charge is non-adiabatically dumped to ground. It can also be seen that there is undershoot on the non-asserted output. This is probably caused by capacitive coupling and the absence of any path to ground once the charge on the inputs has been recovered.

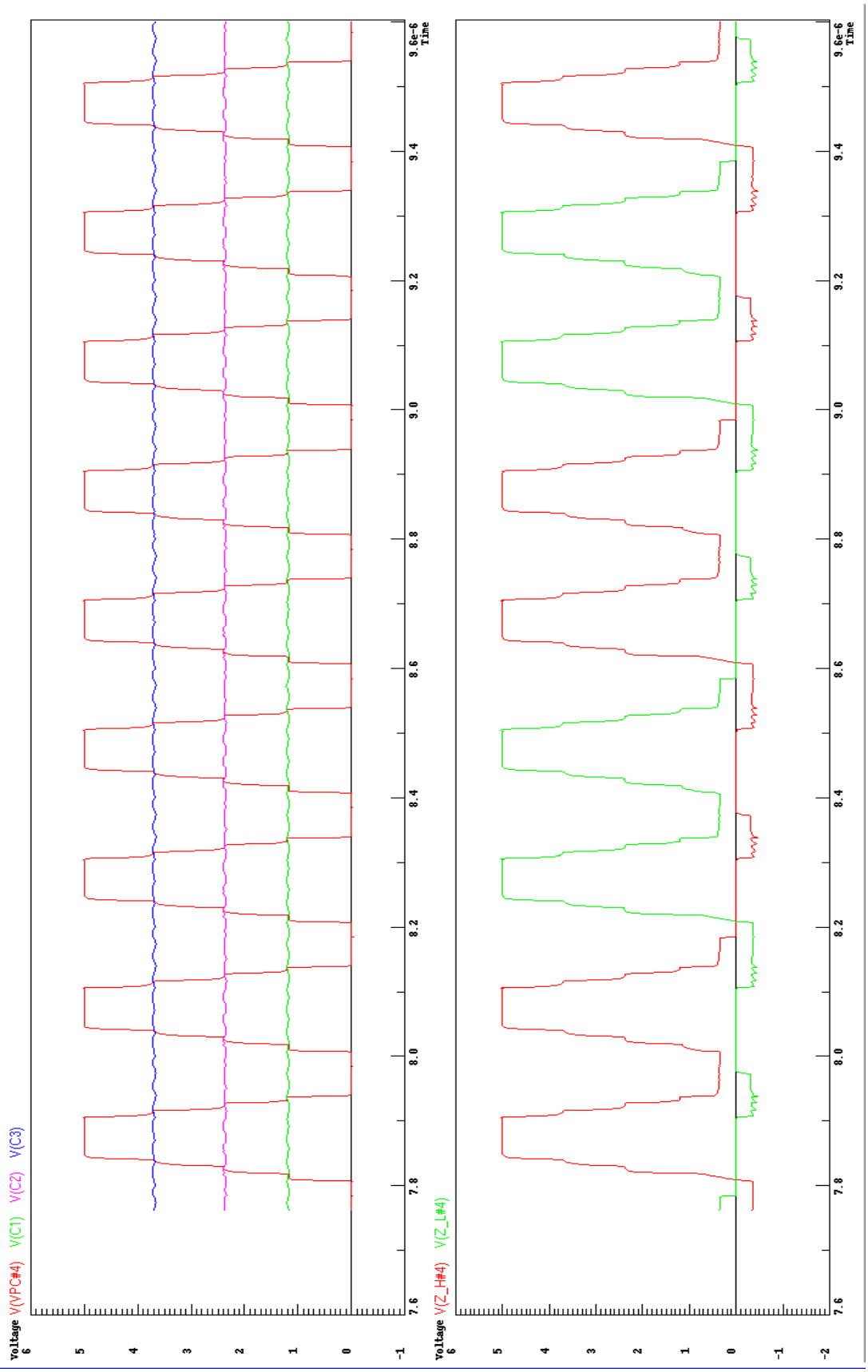

Figure 5.10: Performance of an Asynchrobatic ECRL pipeline

5.8 Potential for fully reversible operations

The majority of “adiabatic” logic families mentioned in Chapter 2 are in fact only quasi-adiabatic. This is because they are not logically reversible, which means that they have some non-adiabatic losses. However, using the concept of Reversible Computation, which was introduced in section 2.6, fully adiabatic, reversible processing gates could be constructed. It has been shown that complex PFAL gates can be constructed so that they are fully-reversible [Will08b]. This means that arbitrary reversible gates with up to four inputs can be constructed. Because it is also possible to drive the asynchronous control logic in the reverse direction by inverting the handshake signals, there is enormous potential for the creation of reversible *Asynchrobatic* logic systems.

The potential for reversibility was alluded to in the initial paper describing PFAL [Vetu96], but in various subsequent works which utilise PFAL [Amir00], [Amir04], [Blot04], [Fisc05], [Fisc06] & [Teic07], this potential does not appear to have been explored any further. The creators of EACRL also considered reversibility [Varg01a]. The experimental implementation of a Toffoli gate [Fred82] using PFAL technology [Will08b] demonstrates that structures from the Reversible Logic paradigm can be viably created in PFAL, and that for an ideal waveform, it reduced power consumption by about two-thirds. Conceptually it is only a small step to move from driving these structures with ideal waveforms, to using SWC circuits from *Asynchrobatic* Logic. Currently unpublished experimental results suggest that *Asynchrobatic* Logic with a fully-reversible PFAL data-path operates with lower power consumption than a PFAL data-path with irreversible losses.

Reversible logic design is a very different paradigm from traditional logic design, as most logic functions like AND and OR are not reversible. The simplest familiar logic function that can easily be made reversible is XOR. By passing one input unchanged through to the output, a reversible system with

two inputs and two outputs is created, this results in a Controlled-NOT, where the one signal is inverted between input and output depending upon the value of the other signal, which passes through always passes through unmodified. Unfortunately, this gate is not universal. However, a universal gate can be obtained by extending this idea to a three-input, three-output gate where the two signals always pass through unmodified, and the other signal is inverted only if both the unmodified signals hold values representing “True”. This is called a Controlled-Controlled-NOT or a Toffoli Gate. Table 5.1 shows the truth-table for a Toffoli Gate, and a Toffoli Gate symbol (after [Feyn00]) is shown in Figure 5.11.

A	B	C		A'	B'	C'
0	0	0		0	0	0
0	0	1		0	0	1
0	1	0		0	1	0
0	1	1		0	1	1
1	0	0		1	0	0
1	0	1		1	0	1
1	1	0		1	1	1
1	1	1		1	1	0

Table 5.1: Truth-table for a Toffoli Gate [Feyn00]

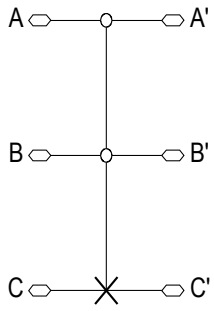

Figure 5.11: Controlled-Controlled-NOT (CCN) or Toffoli Gate Symbol [Feyn00]

Figure 5.12 shows a logic-level schematic of how a Toffoli Gate can be implemented in PFAL. It should be noted that because the PFAL implementation uses a dual-rail logic, a port's labels refer to a pair of wires, one asserted high, the other asserted low, for example the port labelled “A” consists of two signals, “A_H” and “A_L”.

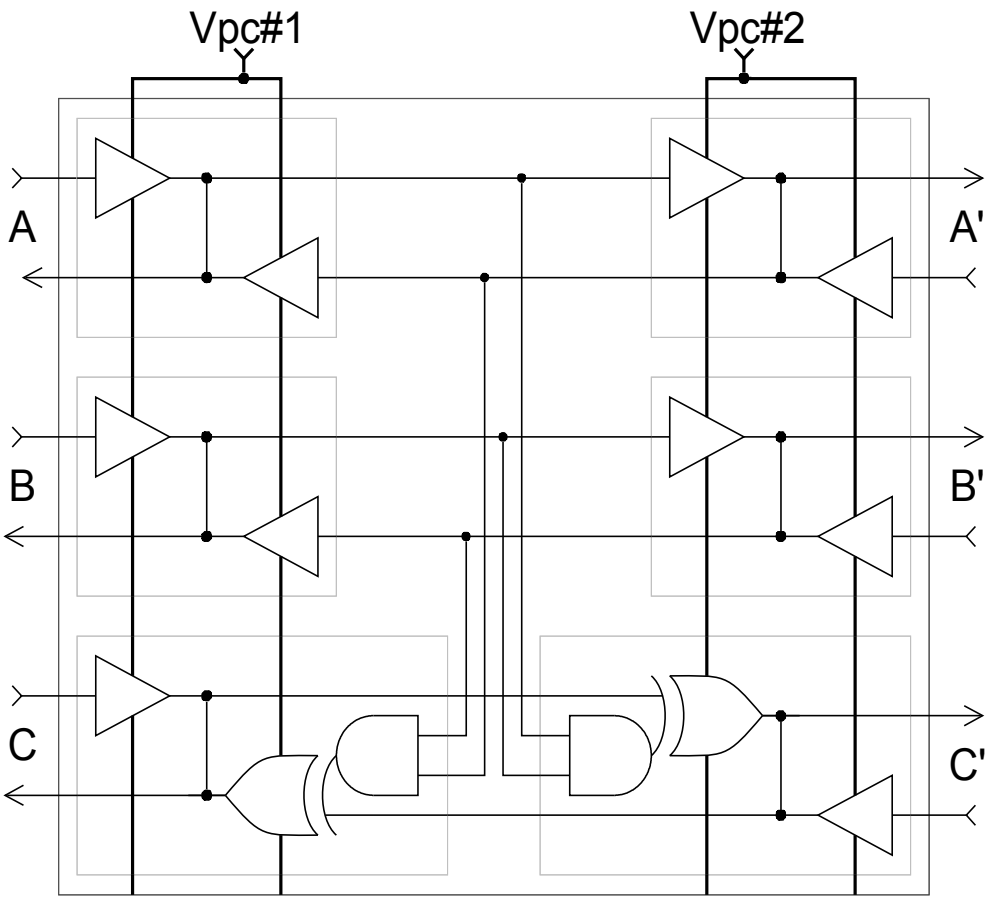

Figure 5.12: Schematic of a PFAL Toffoli Gate [Will08b]

The Toffoli Gate, contained within the outer box, is created from six PFAL gates (one within each of the six inner boxes). Each of these PFAL gates has two separate functions with separate evaluation trees, there is a forward-path function, and a recovery-path function. Ten of these functions are just buffers, and the other two, which form the Controlled-Controlled-NOT (CCN), are implemented using an AND-XOR gate. The NMOS tree for this is shown in Figure 5.13. As is alluded to by the way the schematic is drawn in Figure 5.12, it is conceptually easier to think of the reversible function as

existing over two adjacent pipeline stages, although in certain circumstances, it is possible to put a sequence of functions closer together by using the rather esoteric concept of the reversible function existing in the space between two pipeline stages.

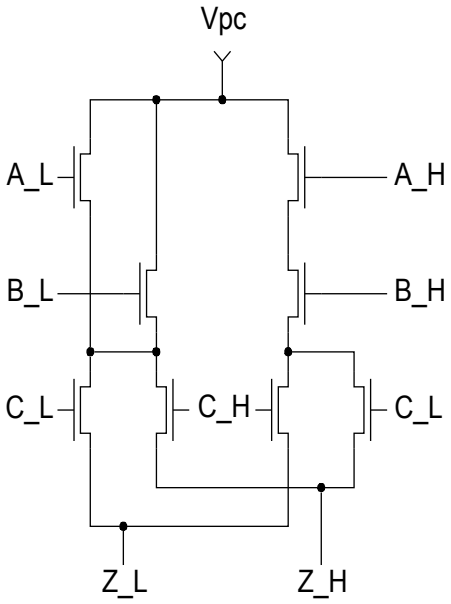

Figure 5.13: Evaluation tree for AND-XOR function [Will08b]

It should be evident from the evaluation tree for the AND-XOR gate, that in any transistor based design, the Toffoli Gates implemented are still not entirely loss-less. This is due to small amounts of charge that will be retained on the three internal source-drain connections within the evaluation or recovery logic.

Moving to reversible logic is not without costs. It adds many extra complications. The implemented functions must be invertible, and to accommodate the recovery path, require feedback from the outputs of the subsequent stage. The Toffoli gate is only a starting point, and any invertible function of four inputs or fewer can be implemented using PFAL gates. This may have applications to cryptographic algorithms that use four-bit substitution boxes, as these could be implemented in fully reversible logic, possibly improving resistance to Differential Power Analysis (DPA).

Figure 5.14 shows how the performance of PFAL gates varies depending upon whether they include a recovery path, which makes them reversible, or whether they are non-reversible, quasi-adiabatic gates with only the forward evaluation path.

The upper graph shows the currents flowing into and out of two adjacent stages. There are a pair of signals labelled “I(VPC5)” and another pair labelled “I(VPC6)”. The red and green signals chart the performance of the reversible circuit, whilst the magenta and blue signals chart the performance of the non-reversible circuit. The current supplied to the reversible circuit can be seen to be greater, which is to be expected because it includes more devices and has has a higher capacitive load, but the current recovered from the reversible circuit is also greater.

The lower graph shows the output voltage on the asserted high “C” output of a Toffoli gate. It can be seen that the output voltages of the non-reversible versions do not track the power-clock voltage all the way to ground, but switch later as the inputs to the evaluation stage change.

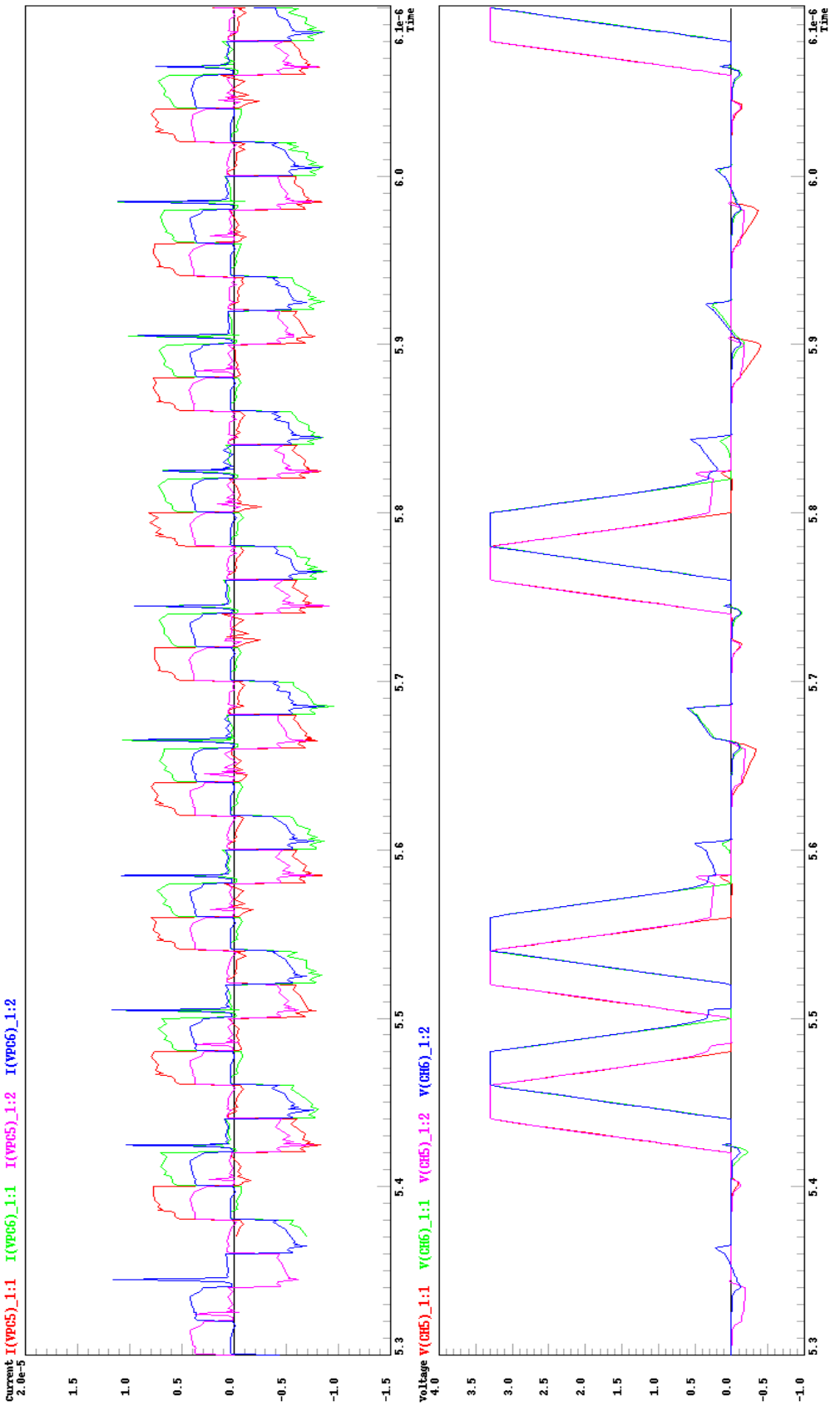

Figure 5.14: Performance of reversible versus non-reversible PFAL gates

5.9 Summary

In this chapter *Asynchrobatic* Logic has been introduced. It has been shown that once the extra cost, in terms of power, has been amortised by the increased power efficiency of the data-path, that it is capable of lower power operation than asynchronous systems. The amortisation of the extra power is performed by having a wide data-path. In the example given, it was shown that a 16-bit ECRL data-path would never make sufficient savings, but that with a 32-bit ECRL data-path, if more that 70% (23-bits) of the data bus changed during each transaction, then *Asynchrobatic* logic would be more efficient. It should be noted that ECRL is a quasi-adiabatic logic family, so by using fully adiabatic, reversible logic structures, it is likely that the data-path's power consumption can be further lowered.

The concept of fully reversible PFAL gates that can perform data processing was also presented. A reversible Toffoli gate was implemented using PFAL, and its performance, using ideal adiabatic charging, was compared against that of a non-reversible AND-XOR gate. The reversible Toffoli gate used about two-thirds less power than the non-reversible AND-XOR gate.

Chapter 6 Modelling and Simulating *Asynchrobatic* Logic

6.1 Introduction

It is important to be able to describe, model and simulate *Asynchrobatic* Logic systems using Hardware Description Languages (HDLs) and circuit simulators like SPICE. HDL simulation is important not only because they are the industry standard, but because it would take too much effort and be too error-prone to attempt to debug a SPICE simulation of a large system. However, unlike the majority of traditional CMOS-based logic systems, *Asynchrobatic* Logic does not perform signalling using a single wire to represent a single bit. A bit is represented on two wires, with three defined states, one invalid state and the possibility of various undefined states at initialisation. The defined states are shown in Table 6.1 below:

State	Asserted Low wire	Asserted High wire
Inactive	Low Voltage	Low Voltage
Logic 0	High Voltage	Low Voltage
Logic 1	Low Voltage	High Voltage
Invalid	High Voltage	High Voltage

Table 6.1: Dual-rail logic states in Asynchrobatic Logic

Because of the redundancy in the signalling, extra logic can be used to detect certain faults. This is because if the power-clock is asserted and the two wires have equal logic values then there is clearly a fault. When simulated in software using HDLs, these states can be detected and reported. In hardware, the same idea can be extended to fault and integrity checking. This would allow test structures constructed with XOR or XNOR gates to be used for manufacturing tests, and possibly for tamper detection or signal integrity checking. However, these structures would need to be able to be disabled to prevent unnecessary power consumption, and care would be needed to avoid short-circuit currents if static CMOS gates were used.

As well as being used for simulation, HDLs can be used for logic synthesis, allowing a design specified using an HDL to be implemented automatically.

Verilog HDL was chosen as the HDL to use to model *Asynchrobatic* Logic. As well as exercising personal choice due to greater familiarity, this was done because it provides a better range of modelling options than VHDL. Specifically, VHDL lacks switch-level models as an integral part of its language specification. Since the choice of HDL can be due to ideological decision-making, both Verilog and VHDL will be discussed.

6.2 Verilog modelling

At its simplest, the Verilog model can be made using a series of standard latches with data-processing inputs. These are clocked by the outputs of modelled asynchronous components. These are used to represent the various local power-clocks. The essential part of modelling *Asynchrobatic* pipelines, and where it differs from the modelling of static CMOS, is to return the output or outputs of the data-path to an invalid state on the falling edge of the simulated power-clock. For the single rail version, the invalid state can be an output of either undefined (`1'bx`) or high-impedance (`1'bz`), or the output could just return to logic zero. The third option is probably the least preferable since it would make it difficult to disambiguate between a zero and an error condition. Code Segment 1 shows a single-rail Verilog implementation of a two-input AND gate. Figure 6.1 shows, for a two-input AND gate, how the single-rail Verilog can be conceptualised as a logic function merged into a resettable D-type Flip-Flop, followed by a tristate output driver, with both controlled by the same simulated power-clock signal. This is the novel contribution because by capturing both the rising and falling edges of the power-clock, it can capture pipeline timing errors, something that would be missed if the design was simulated using standard flip-flops or latches.

```
// Single-rail, functional
// representation of an adiabatic
// two-input AND gate
module Buffer (A,B,Z,PClk,Rst0)
input A, B; // Inputs
input Pclk; // Simulated Power-Clock
input Rst0; // Reset (active low)
output Z;   // Output

// Detect Reset, otherwise
// Simulate Charge & Hold stages
always @(posedge PClk or negedge Rst0)
  if (~Rst0)
    #`RESET_DELAY Z <= 1'bz;
  else
// Define output function
    #`STAGE_DELAY Z <= A & B;

// Simulate Recover & Wait stages
always @(negedge PClk)
  #`STAGE_DELAY Z <= 1'bz;

endmodule
```

Code Segment 1: Single-rail Asynchrobatic two-input AND gate in Verilog

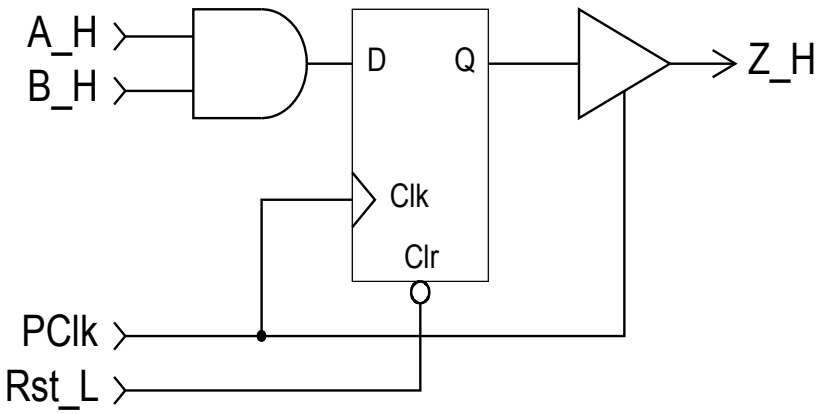

Figure 6.1: Conceptualisation of a two-input AND gate for single-rail Asynchrobatic data-path simulation

The ideas behind the behavioural modelling of the single-rail scheme can be simply extended to dual-rail by the addition of extra wiring, and, in the majority of cases, this can be achieved by the use of regular expression substitutions.

The dual-rail implementation allows state checking can also be added to detect and report invalid circuit operation. However, its effectiveness will depend upon how the complementary outputs are generated. For components like multiplexers, the state checking is more complex, because the unselected input does not require state checking. The dual-rail method can then be replaced by switch-level models. For the dual-rail version, the idle state can be modelled by the pair of complementary outputs both being driven to the same logic value (normally both at zero), although there is nothing to prevent these outputs both returning to either undefined or high-impedance. Code Segment 2 shows a dual-rail buffer, which also includes basic checking that the complementary inputs are not equal on the rising edge of the power-clock.

```
// Dual-rail, functional representation
// of an adiabatic buffer

module Buffer (A_L, A_H, Z_L, Z_H, Pclk, Rst0)
input  A_L, A_H;  // Inputs to be buffered
input  Pclk;      // Simulated Power-Clock
input  Rst0;      // Reset (active low)
output Z_L, Z_H;  // Outputs

// Detect Reset, otherwise
// Simulate Charge & Hold stages
always @(posedge PClk or negedge Rst0)
  if (~Rst0)
    begin
      #`RESET_DELAY Z_L <= 1'bx; Z_H <= 1'bx;
    end
  else
    begin
      if (A_L == A_H) // An invalid state
        $display("Input violation in %m at %t",$time);
        // Define both versions of output function
      #`STAGE_DELAY Z_L <= A_L; Z_H <= A_H;
    end

// Simulate Recover & Wait stages
always @(negedge PClk)
begin
  #`STAGE_DELAY Z_L <= 1'b0; Z_H <= 1'b0;
end

endmodule
```

Code Segment 2: Dual-rail Asynchrobatic buffer in Verilog

As previously noted, the major benefit that Verilog has over VHDL is its switch-level modelling. This would allow the data-path to be modelled using primitive devices that represent the MOS switches, but with a major increase in simulation speed. The primitive Verilog constructs that could be used are `nmos`, `pmos`, `tranif0` and `tranif1`. Due to limitations of the Verilog simulator used (Icarus Verilog [Icar02]), these models were not fully implemented, and were not necessary, as the behavioural models were more than adequate. However, the potential applications where these switch-level models would be useful are HDL modelling of power consumption, and as a

schematic source in applications such as Layout Versus Schematic (LVS) checking.

A minor issue encountered when attempting to model such circuits using HDLs is that static CMOS circuits are required for the control structures and adiabatic circuits are required for the data-path. Since various functions, like, for example, an inverter could exist in both design styles, and are identically named, but not interchangeable, it is important to use a naming system, or programming language concepts like package scope, to ensure that the different logic types are kept separated. This may be an area where the use of VHDL would have advantages over Verilog.

For the automated implementation of more complex functions, it would be possible to use a pre-processor, to take the description of a cell's function, and pass this through a OBDD minimiser and optimiser to determine the minimum tree size required to form that function. If the technology were commercialised, then this step would need to re-order and re-label the inputs and outputs to map the required output function onto an appropriate cell in the library. For example, for a single-input gate, there is only one possible circuit design, but the inverter is a buffer with its outputs' assertion levels exchanged. For a two-input gate, the XNOR function can be obtained from the XOR function by simply exchanging its outputs' assertion levels, and the AND, OR, NAND, NOR, and versions of these with a single inverted input can all be obtained from an AND gate by exchanging the assertion levels of either the inputs, the outputs or both.

6.3 VHDL Modelling

It was noted in the previous section that Verilog was used for all the large scale simulations of *Asynchrobatic* Logic, but that VHDL was an alternative HDL that could be used. Code Segment 3 shows a possible implementation of an *Asynchrobatic* buffer in VHDL. Apart from this single

piece of code, no other work on simulating or modelling *Asynchrobatic* Logic was done in VHDL.

```
-- VHDL model of an Asynchrobatic buffer

library IEEE;
use     IEEE.std_logic_1164.all;

entity buffer is
  port (A    : in  std_logic;
        PCLK : in  std_logic;
        RST0 : in  std_logic;
        Z    : out std_logic);
end buffer;

architecture behavioural of buffer is
begin

  process (PCLK, RST0) is
  begin
    if RST0 = '0' then
      Z <= 'X';
    elsif Rising_edge(PCLK) then
      Z <= A;
    elsif Falling_edge(PCLK) then
      Z <= 'Z';
    end if;
  end process;
end architectures behavioural;
```

Code Segment 3: Single-rail Asynchrobatic buffer in VHDL

6.4 Circuit level simulation

Since SPICE and SPICE-like tools exist to simulate circuits, the use of SPICE presents few major problems. There are minor problems that need to be considered. The primary problem with SPICE is the simulation time. This can be overcome by using table look-up based SPICE simulators, but possibly compromises accuracy for speed. There are other accuracy issues with SPICE which included the parameters and numerical simulation method. Other issues relate to ensuring that the minutiæ of simulation details are correct, and to correctly measuring the simulated power dissipation.

There are a number of extra details that need to be considered when performing SPICE simulations rather than just HDL modelling. These include: device sizing, the connectivity of a device's bulk terminals, and when targeting more advanced processes, which threshold voltage (V_T) of device to use.

The V_T of devices did not need to be considered for either the 0.8µm or 0.35µm processes as only one was available. The device sizing used in the adiabatic data-path was minimum length and minimum width. The connections to the bulk terminals were made as follows. All NMOS devices had their bulk terminal connected to ground. The cross-coupled PMOS devices in the data-path had their bulk terminal connected to the power-clock. This allows them to recover more charge through their internal diode, but for full-layout simulations, it would be necessary to model the reverse-biased n-well to p-substrate diode that will be created.

For circuits that have a physical layout implementation, a full parasitic extraction of at least resistance and capacitance should be performed, and this was done for the initial presentation of *Asynchrobatic* Logic. The subsequent works used front-end SPICE netlists, that is netlists containing only ideal devices without these parasitics, meaning that the quoted performance figures are likely to be optimistic. However, to maintain the integrity of the research this was made clear in the publications.

A major use of SPICE was to measure power consumption, as this cannot be performed using HDL representations. The initial measurements of power consumption were performed by using extra pseudo-circuits to perform these measurements. These pseudo-circuits consisted of a current-controlled voltage source, controlled by the voltage source under observation, driving a capacitor. This performed the integration of the current with respect to time to allow the power to be calculated. This was done following the method in Rabii's tutorial [Rabi03].

However, propriety extensions to SPICE incorporate measurement statements (for example, the .EXTRACT card) that can perform this integration directly, and waveform viewers also include waveform processors that allow waveforms to be processed, including integration. Equations (6.1) and (6.2) show how the average power ($\overline{P}$) can be obtained by performing the integration of the instantaneous current, i(t), and voltage, v(t), for a fixed time period, from t_1 to t_2. Equation (6.2) show how this can be simplified when the power supply, V_{dd}, is at a fixed voltage.

$$\bar{P} = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} v(t)\, i(t)\, dt \tag{6.1}$$

$$\bar{P} = \frac{V_{DD}}{t_2 - t_1} \int_{t_1}^{t_2} i_{V_{DD}}(t)\, dt \tag{6.2}$$

The use of the simplified equation (6.2) should provide sufficiently accurate results, and can be applied to static DC voltage sources. These would include the voltage sources used to model the V_{dd} supplies to the Asynchronous Stepwise Charging controller logic, and the Stepwise Charging logic. However, for experiments using the ideal, piecewise-linear voltage sources, it would be necessary to use the more complex method shown in equation (6.1).

6.5 Summary

In this chapter, the methods of modelling and simulating *Asynchrobatic* Logic have been detailed. It has been shown that *Asynchrobatic* Logic can be adequately, and efficiently modelled using both Verilog and VHDL. The ability to use HDLs is a prerequisite to being able to design large *Asynchrobatic* systems and would be necessary to commercialise the product, as simulating large systems with SPICE-like simulators makes circuits too difficult to debug, and takes too long. It has been shown that behavioural models of

Asynchrobatic Logic circuits can be constructed in both Verilog and VHDL. However, whilst HDL simulation is essential for functional checking, it is not a panacea and the use of SPICE is still required to measure power consumption.

The novel use of both the rising edge and falling edge of the simulated power-clock enhanced the ability to detect design errors that would not have been detected if the design had been modelled using flip-flops.

Chapter 7 Implementing *Asynchrobatic* Logic

7.1 Introduction

The initial demonstration of *Asynchrobatic* Logic was done using a 0.7µm (0.8µm drawn) two-layer metal CMOS process, but it was created using tools no-longer viable for commercialisation, and did not demonstrate sufficient complexity in the data-path. The later work upon the Adder and Greatest Common Denominator (GCD) circuits, which will be described in Chapter 8, was only performed using front-end netlists (without parasitics) and not extracted layout. In this chapter, some layout of complex data-path functions is described. Being able to produce and verify the layout of a design is an important step in showing the viability of the technology, as it demonstrates that it is possible to produce a physical product.

7.2 The Twofish algorithm

The circuit portion chosen to demonstrate viable layout was the substitution boxes, q_0 and q_1, of the Twofish cryptographic algorithm [Schn98]. This algorithm was chosen because cryptography is a potential target use of *Asynchrobatic* Logic, and the substitution boxes are four-bits wide meaning that they can each be implemented using single stage logic functions. These are complex eight-bit functions that are ultimately grouped into a 64-bit wide data-path, to form part of the *F*-function in a Feistel network [Feis73]. For an input value (*x*), the output (*y*) is defined as shown in equations (7.1) to (7.7), with XOR representing a bit-wise exclusive OR operation, ROR_4 representing a logical Rotate Right operation on the specified four-bit nibble, and $t_m[n]$ representing a substitution performed using the look-up tables detailed in Table 7.1. This is exactly as designed and documented by Schneier *et al.* in [Schn98].

$$y = q_n[x] \tag{7.1}$$

$$a_{0,}\, b_0 = [x/16],\ \ x \bmod 16 \tag{7.2}$$

$$a_{1,}\, b_1 = a_0\ XOR\ b_{0,}\ \{a_0\ XOR\ (ROR_4(b_0, 1))\ XOR\ 8a_0\} mod\ 16 \tag{7.3}$$

$$a_{2,}\, b_2 = t_0[a_1],\ \ t_1[b_1] \tag{7.4}$$

$$a_{3,}\, b_3 = a_2\ XOR\ b_{2,}\ \{a_2\ XOR\ (ROR_4(b_2, 1))\ XOR\ 8a_2\} mod\ 16 \tag{7.5}$$

$$a_{4,}\, b_4 = t_2[a_3],\ \ t_3[b_3] \tag{7.6}$$

$$y = 16b_4 + a_4 \tag{7.7}$$

Table	Substitution for q_0	Substitution for q_1
n	0 1 2 3 4 5 6 7 8 9 A B C D E F	0 1 2 3 4 5 6 7 8 9 A B C D E F
t_0[n]	8 1 7 D 6 F 3 2 0 B 5 9 E C A 4	2 8 B D F 7 6 E 3 1 9 4 0 A C 5
t_1[n]	E C B 8 1 2 3 5 F 4 A 6 7 0 9 D	1 E 2 B 4 C 3 7 6 D A 5 F 9 0 8
t_2[n]	B A 5 E 6 D 9 0 C 8 F 3 2 4 7 1	4 C 7 5 1 6 9 A 0 E D 8 2 B 3 F
t_3[n]	D 7 F 4 1 2 6 E 9 B 3 0 8 5 C A	B 9 5 1 C 3 D E 6 4 7 F 2 0 8 A

Table 7.1: Look-up Tables for Twofish substitution q-boxes [Schn98]

It can be seen that the implementation of these two q-functions will more fully use the Ordered Binary Decision Diagram (OBDD) design methods. The procedures used are detailed. Firstly, it is necessary to determine how many pipeline stages are needed. It can be seen that functions (7.1) and (7.6) only serve to split the 8-bit input into two 4-bit sections and concatenate the resultant two 4-bit outputs into a single 8-bit output. Neither of these operations require a pipeline stage. Each of the table look-ups have only four inputs and can therefore be performed in a single pipeline stage, and each of the XOR sections also requires one pipeline stage. The bit-wise rotations within the XOR sections can be simply merged. This means that a single q-box requires four pipeline stages. The 2- and 3-input XOR gates are simple to design, so the focus of the design work will be upon the look-up tables. The first task is to calculate the output function for each bit of each table. This will result in thirty-two different output functions. These then need to be optimised to a minimal Reduced Ordered Binary Decision Diagram (ROBDD) representation. This can be turned into Verilog and more importantly, SPICE.

The layout can be performed, and subsequently compared to the SPICE. The Verilog and/or SPICE can be verified against a known good reference implementation, written in ANSI C, for example, so that when the layout is shown to match the SPICE source, the designer can have every confidence that the design is correct.

7.3 Binary Decision Diagram Optimisers

In order to implement the complex input to output functions of the q-box tables in the Twofish algorithm it was necessary to be able to convert complex functions described as a hexadecimal look-up table into a circuit description. This can be done by using a uniform HDL description for circuits simulated in this way, and an OBDD-based preprocessor to convert the function into an optimal ROBDD circuit. Code Segment 4 shows an example of how an arbitrary function can be described in Verilog. In the event, the look-up table data was already available and was processed standalone, but the principle of this type of preprocessing has clear uses in larger scale designs, where a higher level of automation is desirable.

The C code for the optimiser software is shown in Appendix B. However, its operation can be summarised as follows. Take a representation of a logic function, which is supplied as a hexadecimal output function. Given a variable ordering, create an OBDD representation of that output function. Count the number of nodes n the resulting design, and the total path length. If this is better (less) than the previous best result, replace the best result with the current result. Repeat this for all possible variable orderings. Output the resulting OBDD as a SPICE netlist. Repeat this operation for each of the logic functions.

```
// Single-rail, functional representation
// of an arbitrary adiabatic function
module q1t0b2 (A,Z,PClk,Rst0);
input     [3:0]     A;
input               Pclk;     // Simulated Power-Clock
input               Rst0;     // Reset (active low)
output              Z;        // Output
reg                 Z;
reg       [0:15]    data;     // Look-up table data

// Look-up table data can be defined in each cell
// It can be extracted using regular expressions
initial
     data = 16'h1F13;

// Detect Reset, otherwise
// Simulate Charge & Hold stages
always @(posedge PClk or negedge Rst0)
     if (~Rst0)
          #`RESET_DELAY Z <= 1'bz;
     else
// Perform table look-up
          #`STAGE_DELAY Z <= data[A];

// Simulate Recover & Wait stages
always @(negedge PClk)
     #`STAGE_DELAY Z <= 1'bz;

endmodule
```

Code Segment 4: Single-rail arbitrary adiabatic function in Verilog

7.4 Layout Design

The layout performed for the initial experiments that validated *Asynchrobatic* Logic, were performed using a stick-diagram layout tool, Chipwise. The resulting cells were neither ideal for re-use nor for automated placement.

Amirante [Amir04] has implemented various Positive Feedback Adiabatic Logic (PFAL) cells without recovery devices using standard cell based layout geometries. This would appear to provide the good layout methodology. However, it would be easy to extend this work to the reversible

PFAL circuits described in Chapter 5, by having a standard cell with its core, central elements formed by the pair of cross-coupled inverters. The PFAL evaluation trees would be placed on one side of this, which leaves space on the unused side for the recovery devices. If the central core and evaluation trees were modularised, then it would probably be a relatively simple matter to construct closely interlaced reversible logic data-paths. There are two factors that complicate mixing adiabatic logic cells with standard CMOS. The first is the hot n-wells that must be kept isolated from each other. This is not a new problem for static CMOS, as similar isolation issues can occur if it is desired to apply back-bias to devices. The implication is that only n-wells sharing the same power-clock can be abutted. The second issue related to the obstructed metal layers. In general, standard CMOS only uses layers up to and including the first metal layer for routing within a standard cell. Because of the number of dual-rail signals that must cross each other, it is necessary to have access to the second metal layer when constructing complex adiabatic logic gates. Clearly, these layers can be marked as obstructed for automated tools, but it would make adiabatic logic less suitable for older CMOS processes with only two metal layers.

The Asynchronous Stepwise Charging (ASWC) controller can clearly be modularised, with the C-element, the pulse generators, and the intermediate switching stages all being separate components. This would allow the construction of ASWC circuits with an arbitrary number of steps.

The more complex asynchronous control structures can be broken down into modular units, most of which are components usually found in standard CMOS cell libraries, so it should not be too difficult to go to layout for these.

To show that complex circuits could be produced and validated, an LVS and DRC correct implementation of the q_0 and q_1 permutations for the Twofish cryptographic algorithm [Schn98] were created. This was done using an

AMIS 0.35um five-layer metal, two-layer poly process. This is a good example, as these permutations are constructed both from common logic functions (XOR gates) and eight 4-bit wide look-up tables, four in each permutation. For a non-reversible implementation, this leads to thirty-two functions, which can be referenced according to which permutation they are used in (q_0 or q_1), their table number within that permutation (t_0, t_1, t_2 or t_3) and which output bit of that permutation's table they generate (from 0 to 3).

This allows the look-up table functions to be named systematically from “q0t0b0” to “q1t3b3”, and also allows the tables and permutations to be constructed hierarchically, with the look-up tables being named from “q0t0” to “q1t3”, and the permutations simply being “q0” and “q1”. The OBDD-based layout structure meant that these cells could be further modularised into a standard core element which contains the cross-coupled PMOS and NMOS devices, and a standard element containing two NMOS devices in an arrangement that represents a OBDD decision node. Within the layout layout design, it is also very easy to modify PFAL-based designs into Efficient Charge Recovery Logic (ECRL) or Improved Efficient Charge Recovery Logic (IECRL). The conversion to IECRL is performed by disconnecting the root of the decision tree from the power-clock, and reconnecting it to ground, as well as swapping the assertion levels of the outputs. ECRL can be obtained from IECRL by removing the cross-coupled NMOS devices, and leaving only a pair of PMOS cross-coupled devices.

Each of these thirty-two functions was presented to be analysed using an OBDD reducer written in C, and capable of generating Verilog or SPICE output. This analysis shows how minimisation can be performed for more complex circuits. Performing this also showed that in some cases, at the bit-level of the table, some of the functions were isomorphic to each other, meaning that the function need only be drawn once, and could then have different labels applied to its ports. This result might also be of minor interest to cryptanalysts investigating the Twofish algorithm.

The layouts of these substitution boxes were drawn using Cadence's Virtuoso tool with a five-layer metal, two-layer polysilicon Alcatel 0.35μm CMOS process. The floor-plans of both layouts are identical and this floor-plan is shown in Figure 7.1. This shows the placement and orientation of the cells, the location of inputs, outputs and power-clock signals, and where the routing channels. The complete layouts for the q_0 and q_1 q-boxes, which were drawn hierarchically using Cadence Virtuoso, are shown in Figures 7.2 and 7.3 respectively. Both of these q-boxes are 96.8μm × 144.0μm. In both of these layouts, the eight pairs of dual-rail inputs are on the left, the eight pairs of dual-rail outputs are on the right, and the ground and power-clock signals run from top to bottom. Looking from left to right, the designs can clearly be seen to alternate between routing channels and columns of data-path cells, with the data-path columns alternating between sparsely and densely packed cells. The sparsely packed data-path cells, which occupy the first and third columns, are the XOR gates, as defined in equations (7.3) and (7.5), and the densely packed cells, which occupy the second and fourth columns, are the look-up tables, as defined in equations (7.4) and (7.6) and Table 7.1. The hierarchical SPICE for these blocks appears in Appendix C, and is followed by truncated outputs from Mentor Graphics's Calibre tool which was used to perform LVS checking on these designs. This confirms that all the devices in the layouts match the hierarchical SPICE.

Since Figures 7.2 and 7.3 show complete blocks, the necessary magnification makes it difficult to comprehend exactly how the internals of an individual cell have been drawn. To address this, Figure 7.4 shows a close up example layout of a cell (q1t0b2). It can be seen from the layout of the single cell, that converting a PFAL gate to either an ECRL or an IECRL gate can be done with minimal layout effort. The circuit diagram for this cell (q1t0b2) is shown in Figure 7.5, and has been arranged in such a way as to mimic both the layout and the ROBDD structure used in its construction. The minimised SPICE for this cell (q1t0b2) is shown in Code Segment 5, this is a good example of how using variable re-ordering can improve a design. The

reordering mapping shown in Figure 7.6 visually validates the operation of the variable reordering. This optimisation has allowed the function to be implemented using only four nodes (obviously the minimum for any four-input, non-degenerate function), rather than the six that would have been required if the function had not been reordered. Figure 7.7 shows the non-reordered OBDD and its resulting ROBDD, and Figure 7.8 shows an optimally reordered OBDD and its resulting ROBDD. This optimisation therefore reduces the input capacitance on two of the inputs, and also lowers the number of inter-node connections that could store charge on their parasitic capacitances. Another beneficial consequence of the reordering is that the maximum path length through the evaluation tree reduces from four to three, which will reduce any power-losses due to gate resistances. Finally, the reordering reduces the total number of paths through the evaluation tree from nine to six, and this could make testing quicker.

One issue which is obvious upon visual inspection of the layouts is the amount of space required for the dual-rail interconnect. As the number of metal layers increases in modern deep sub-micron and nanometre processes, this issue should not be insurmountable. The layout design also suggests that the design may have crosstalk issues. However, because the data-path in an *Asynchrobatic* system has a different *modus operandi* from that of static CMOS, it is less likely that crosstalk will cause major problems, but any risk of this could be minimised by precluding the use of adiabatic logic families without cross-coupled NMOS devices (like ECRL), and by ensuring that the parasitic extraction of layouts is performed to include inter-net, cross-coupled capacitances, and not just capacitance lumped to ground.

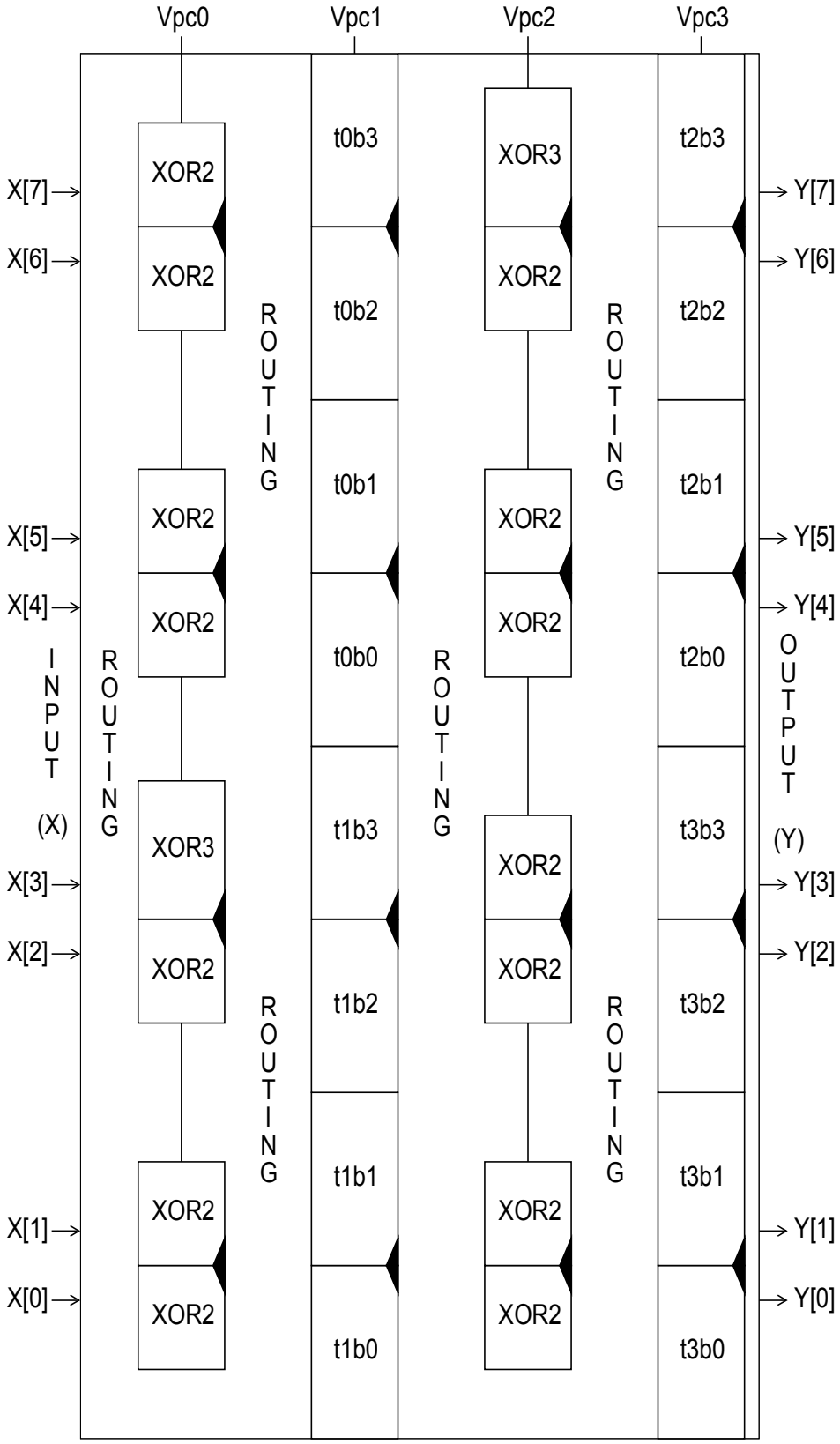

Figure 7.1: Floor-plan of Twofish q-boxes

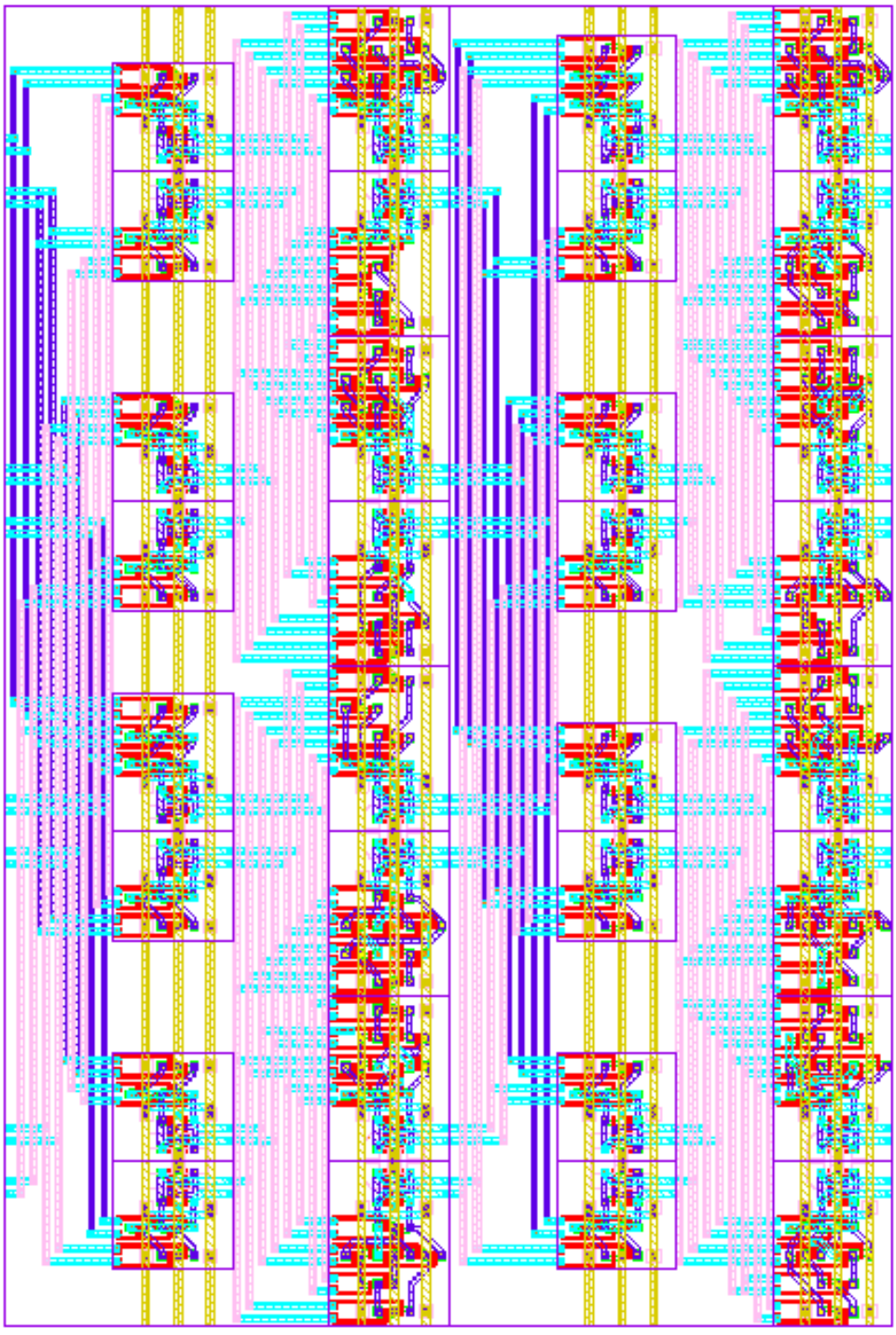

Figure 7.2: Layout of Twofish q0 substitution box

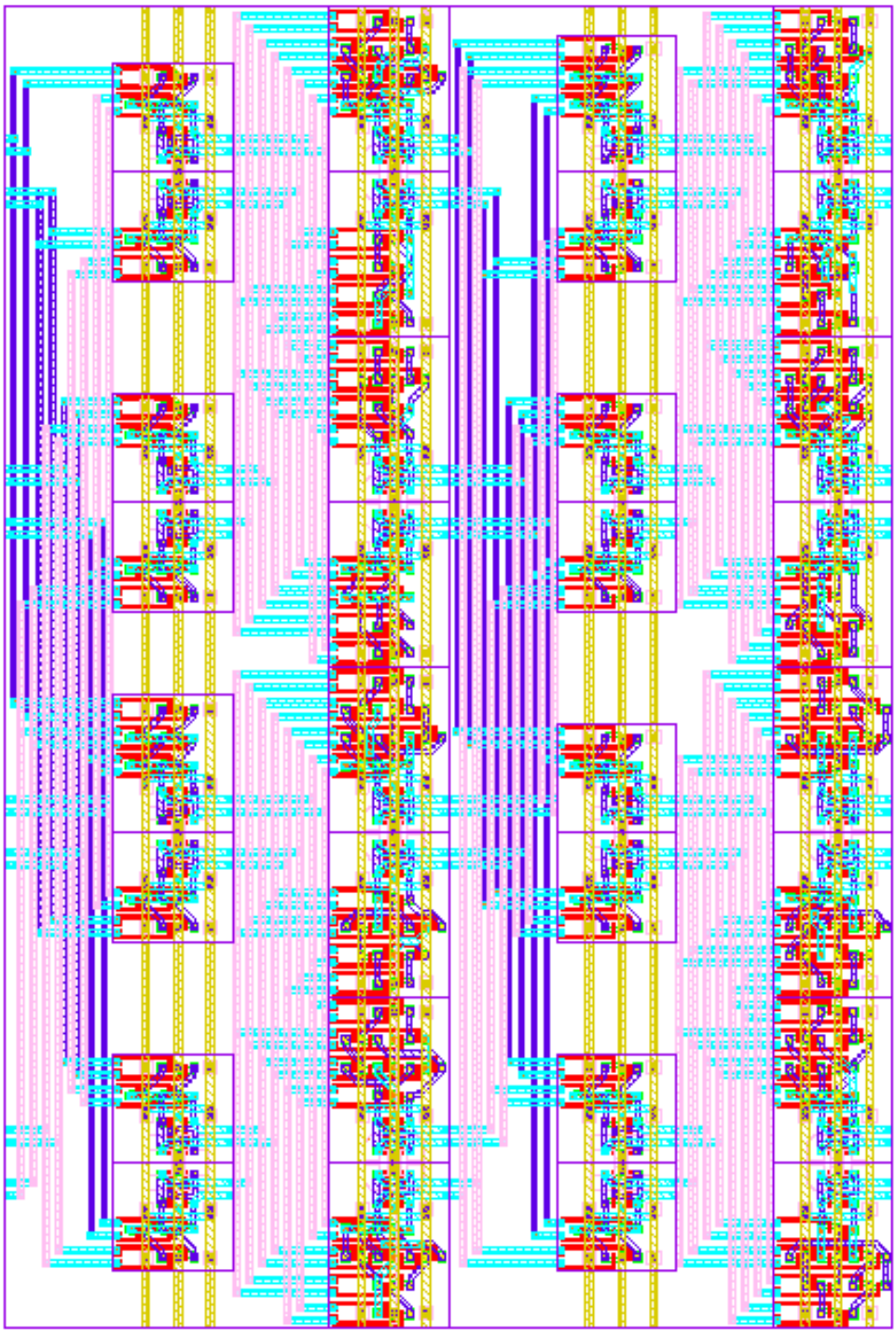

Figure 7.3: Layout of Twofish q1 substitution box

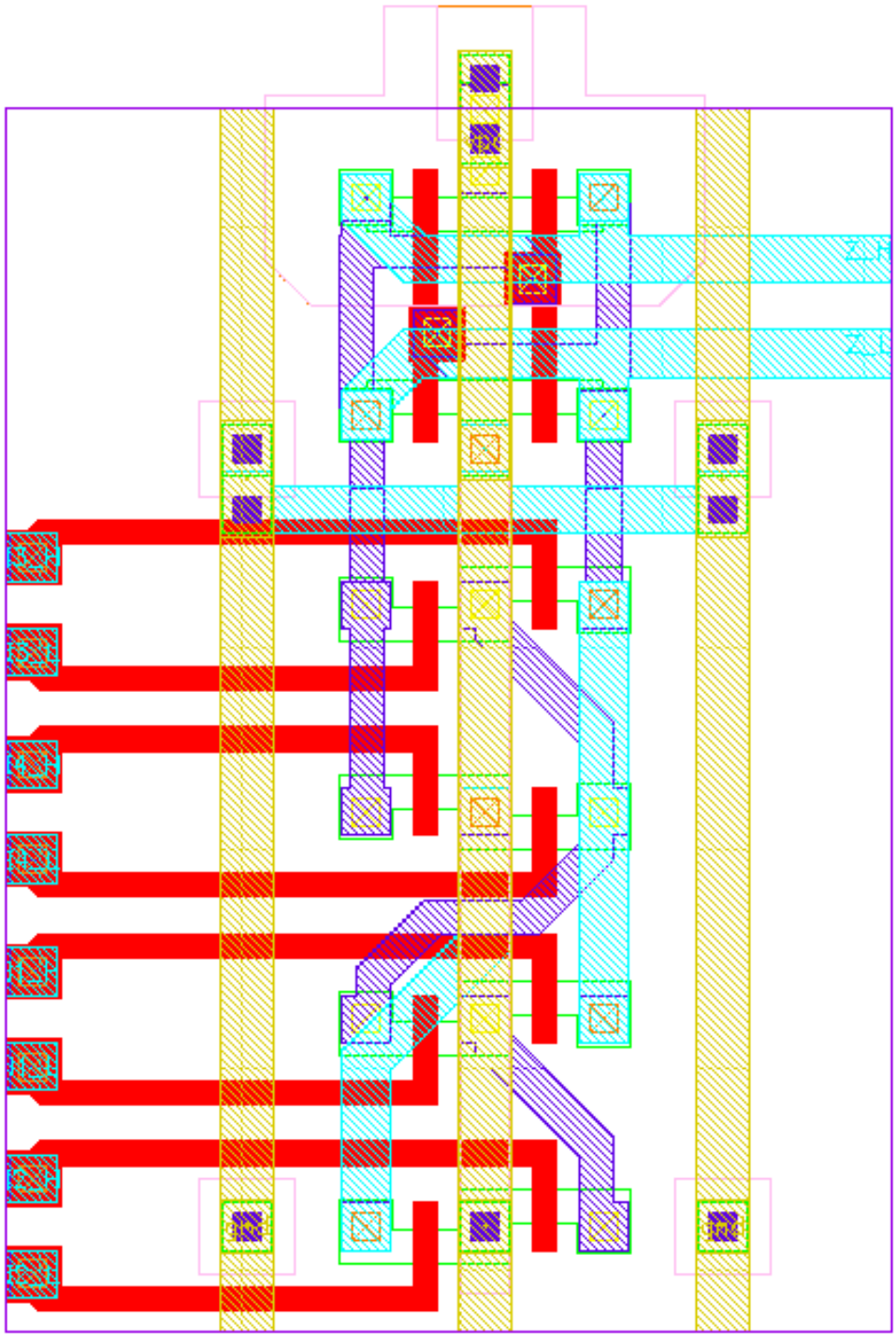

Figure 7.4: Layout of q1t0b2 cell

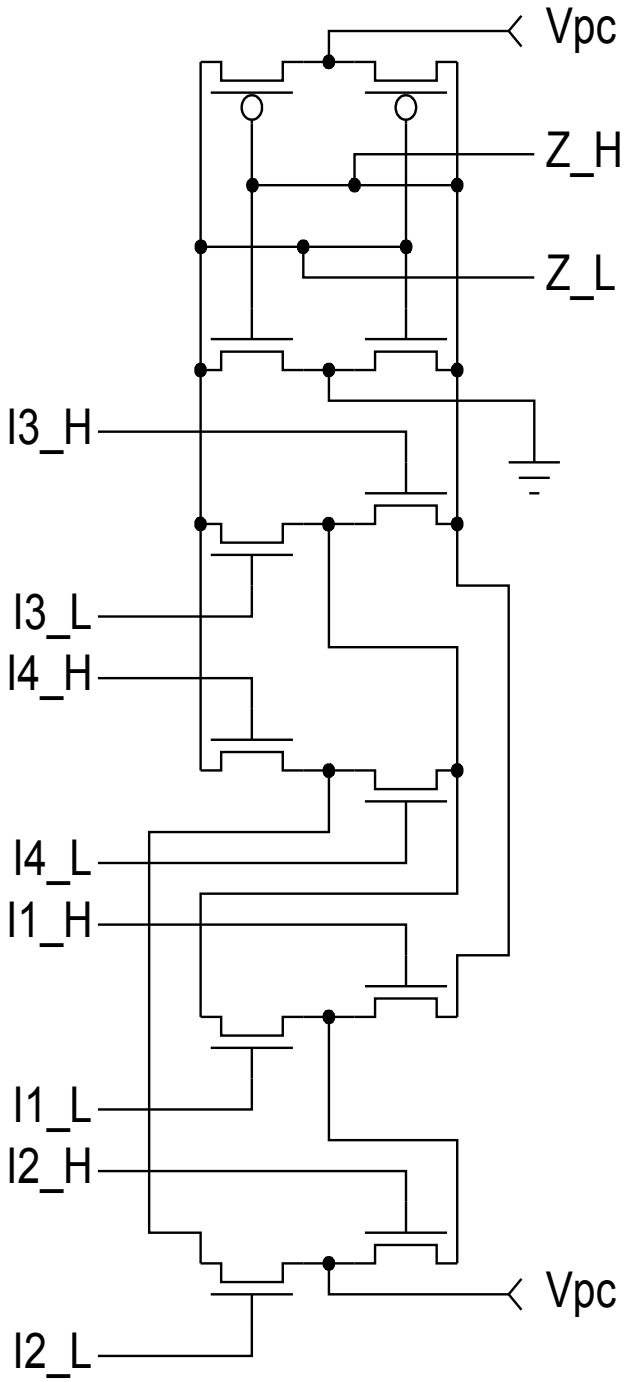

Figure 7.5: q1t0b2 circuit diagram

```
* PFAL circuit for q1t0b2
.SUBCKT X4X1F13
+I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L
+Z_H Z_L vpc gnd
* Cross-coupled PMOS devices
MP0  Z_L Z_H  vpc vpc PMOS L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc PMOS L=0.35u W=0.50u
* Cross-coupled NMOS devices
MN0  Z_L Z_H  gnd gnd NMOS L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd NMOS L=0.35u W=0.50u
* Evaluation logic
MNDL vpc I2_L B   gnd NMOS L=0.35u W=0.50u
MNDH vpc I2_H C   gnd NMOS L=0.35u W=0.50u
MNCL C   I1_L A   gnd NMOS L=0.35u W=0.50u
MNCH C   I1_H Z_L gnd NMOS L=0.35u W=0.50u
MNBL B   I4_L A   gnd NMOS L=0.35u W=0.50u
MNBH B   I4_H Z_H gnd NMOS L=0.35u W=0.50u
MNAL A   I3_L Z_H gnd NMOS L=0.35u W=0.50u
MNAH A   I3_H Z_L gnd NMOS L=0.35u W=0.50u
.ENDS
```

Code Segment 5: Minimised SPICE circuit of function for q1t0b2

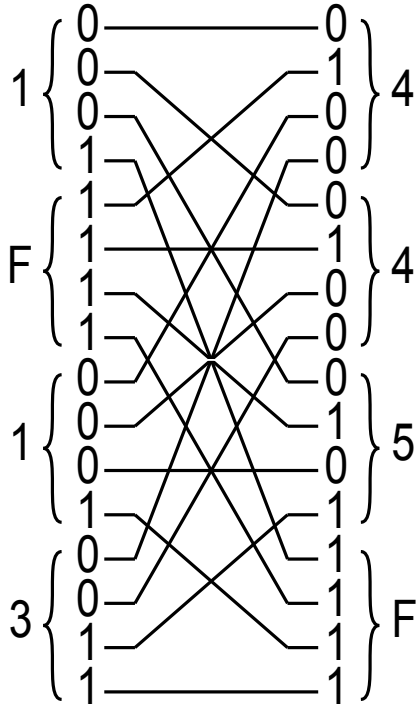

Figure 7.6: A visual validation of variable reordering

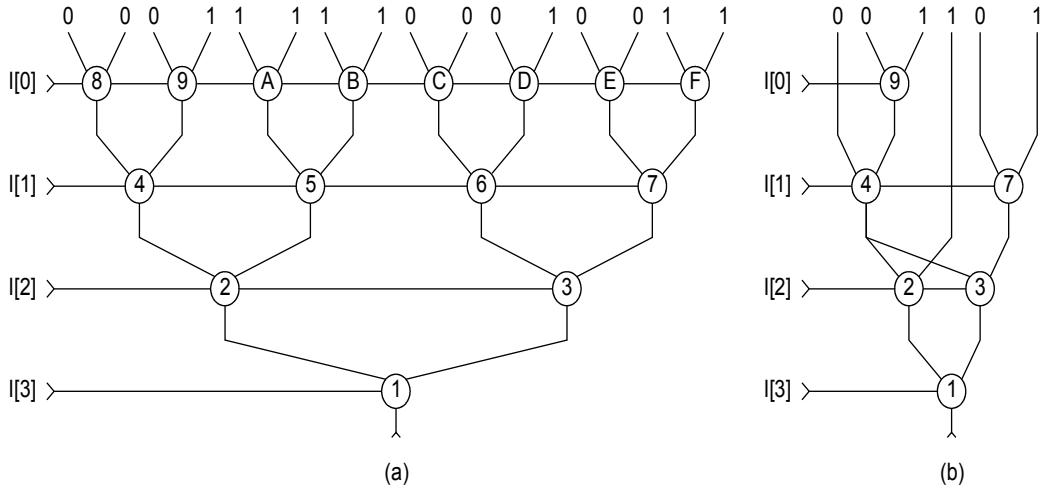

Figure 7.7: A non-reordered, sub-optimal ROBDD implementation of q1t0b2

There are nine paths through the non-reordered, sub-optimal, six-node implementation of the q1t0b2 function. These are tabulated below in Table 7.2. The numbers are the node numbers from the Figure 7.7, and T_0 and T_1 are the "0" and "1" Terminals respectively.

Path	Path Length
1, 2, 4, T_0	3
1, 2, 4, 9, T_0	4
1, 2, 4, 9, T_1	4
1, 2, T_1	2
1, 3, 4, T_0	3
1, 3, 4, 9, T_0	4
1, 3, 4, 9, T_1	4
1, 3, 7, T_0	3
1, 3, 7, T_1	3

Table 7.2: Paths and Path Lengths for a sub-optimal ROBDD minimisation

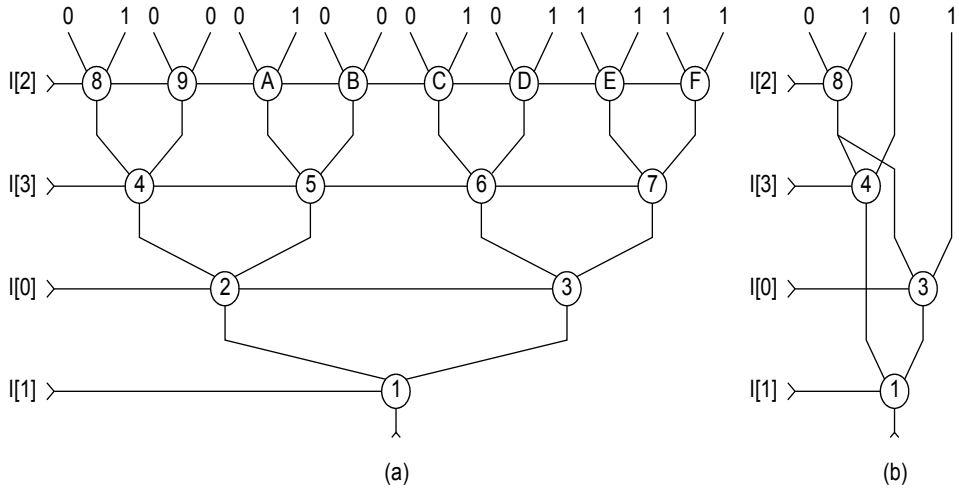

Figure 7.8: A reordered, optimal ROBDD implementation of q1t0b2

There are six paths through the reordered, optimal, four-node implementation of the q1t0b2 function. These are tabulated below in Table 7.3. The numbers are the node numbers from Figure 7.8, or the terminal nodes.

Path	Path Length
1, 4, 8, T_0	3
1, 4, 8, T_1	3
1, 4, T_0	2
1, 3, 8, T_0	3
1, 3, 8, T_1	3
1, 3, T_1	2

Table 7.3: Paths and Path Lengths for an optimal ROBDD minimisation

7.5 Summary

In this chapter the physical design of complex layout blocks has been presented. Part of a complex data-path from Twofish, a real world cryptographic algorithm, has been correctly implemented such that it passes LVS checking. This shows that a SPICE implementation of a design can be successfully checked against a layout implementation. It has been shown from manufacturing data that it is physically possible to implement the tank capacitors on-chip. The importance and usefulness of the ROBDD design methods has been shown, and a method has been presented that would be capable of automatically producing SPICE when supplied with a functional description in Verilog. It has also been practically demonstrated that for efficient designs, it is essential to optimise the variable order of an OBDD to obtain the optimal ROBDD.

Chapter 8 A more complex *Asynchrobatic* system

8.1 Introduction

To show that *Asynchrobatic* Logic can viably implement data processing applications, a system for calculating the Greatest Common Denominator (GCD) of two 16-bit numbers using Euclid's method [Eucl70] is presented.

The structure is presented in both Verilog and as a circuit. As noted previously in Chapter 6, the importance of being able to represent *Asynchrobatic* Logic using Verilog should not be underestimated. The lack of testability of a large SPICE netlist would preclude the implementation of complex systems, because of the excessive duration of testing and debugging. A Verilog design and testing methodology allows much larger structures to be created. Without a Verilog implementation, even a small system like this would have been extremely difficult to implement and debug.

Euclid's method calculates the GCD using repeated subtraction. It uses the two essential components of any computation system [Böhm66], iteration and decision. The pseudo-code for Euclid's method is very simple, and is shown below in Code Segment 6. Iteration can be seen in the outer "while" loop, and decision can be seen in the inner "if-then-else" construct. The actual implementation is different from this in that both variables are reassigned with one always containing the result of the subtraction, and the other containing the subtrahend.

```
while (a != b) do
  if (a > b) then
    a = a - b;
  else
    b = b - a;
  end if;
end while;
```

Code Segment 6: Pseudo-code for GCD algorithm

The complex data-path components required to implement the GCD calculator include a subtractor / reverse subtractor, and a comparator. Both of these use radix-four look-ahead structures. The higher-radix structures are used because they reduce the number of stages in the *Asynchrobatic* pipeline, but do not exceed radix-four because this would generally result in designs that have more than four FETs in series. In general this type of structure would not comply with the electrical rules of most CMOS processes. Simple data-path components like buffers and multiplexers were also required, but should need no detailed explanation.

8.2 Construction of the basic data-path cells

The adiabatic data-path cells required to construct the design are as follows:

- Buffer (also usable as an Inverter).
- Two-, Three- and Four-input AND (usable as INV-AND, NAND, NOR, etc).
- Three-, Five- and Seven-input $(AND\text{-}OR)^n$.
- Two-input XOR (usable as XNOR).
- Two-input MUX.

The NMOS trees for the four-input AND, the seven-input $(AND\text{-}OR)^n$, the two-input XOR and the two-input MUX are shown in Figures 8.1, 8.2, 8.3 and 8.4 respectively. The NMOS trees are not shown for the smaller versions of AND and $(AND\text{-}OR)^n$ with fewer inputs, but can be easily derived by

intelligently removing the inputs with the highest index or indices as necessary.

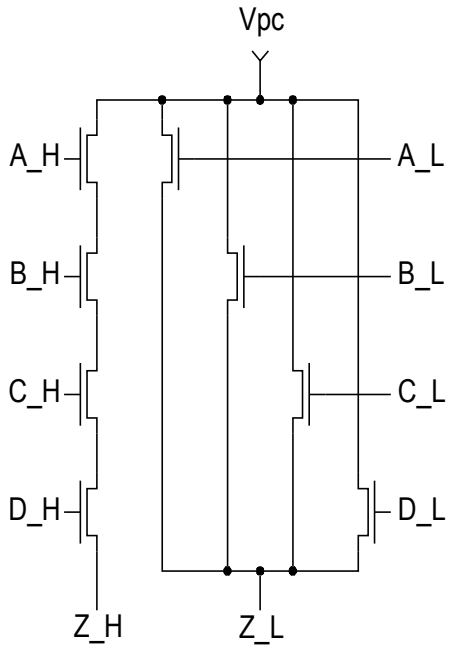

Figure 8.1: NMOS tree of 4-input AND [Will08a]

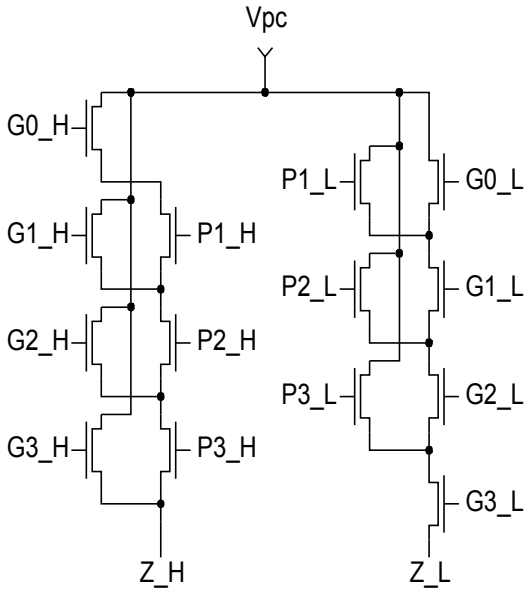

Figure 8.2: NMOS tree of 7-input (AND-OR)^4 [Will08a]

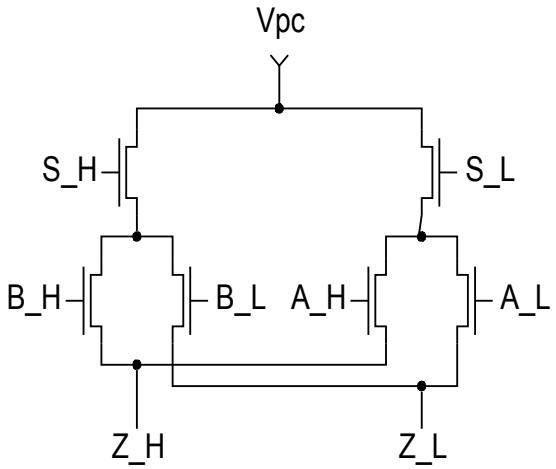

Figure 8.3: NMOS tree of 2-input MUX

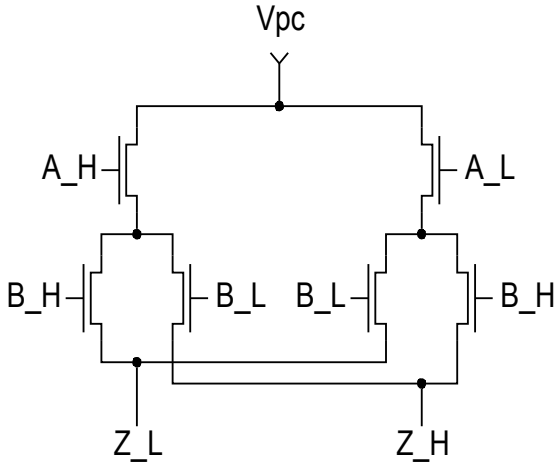

Figure 8.4: NMOS tree of 2-input XOR [Will08a]

8.3 The Comparator

The comparator is required to process an input of two complementary data-paths and reduce them to two complementary single-bit outputs, “Not equal” and “Greater than”. For a sixteen-bit wide data-path, using radix-four look-ahead, this can be achieved in three *Asynchrobatic* stages. For each quadrupling of the data-path width a single extra stage would be required. The first stage computes these functions for each pair of input bits. The “Not equal” output can be thought of as the inverse of “Equal”. “Equal” for each bit is the XNOR function. This can be merged using a multi-input AND for the whole data-path. Although the AND function is both commutative and associative, a regular structure is required so the equality results can be used in the “Greater than” part of the comparator. The “Greater than” function must start from the Most Significant Bit (MSB) and work towards the Least Significant Bit (LSB). The bitwise “Greater than” function is evaluated using an AND function with one input negated (a “free” operation in dual-rail logics). If the MSBs are not equal, then it is instantly determinable which of the two numbers is the greater. If the MSBs are equal, then the next pair of MSBs must be checked, and so-on. The function which implements this is the same $(\text{AND-OR})^n$ (where n is the radix of the comparator) which will also be used in the adder block. This computation can be performed on a block-by-block basis for use with a look-ahead structure, and the carry-generation logic from the subtractor can be reused for this purpose. The complete structure of the Comparator is shown in Figure 8.5. The boxes on the left contain gates to perform the bitwise comparison, and boxes marked “R-4 LA” contain the radix-4 Look-Ahead logic. The *Asynchrobatic* implementation of this circuit has the obvious disadvantage that the width of the data-path decreases logarithmically, but since buffered versions of the values being compared are also required, the comparator is operated in parallel with the buffers, keeping the data-path width suitably large.

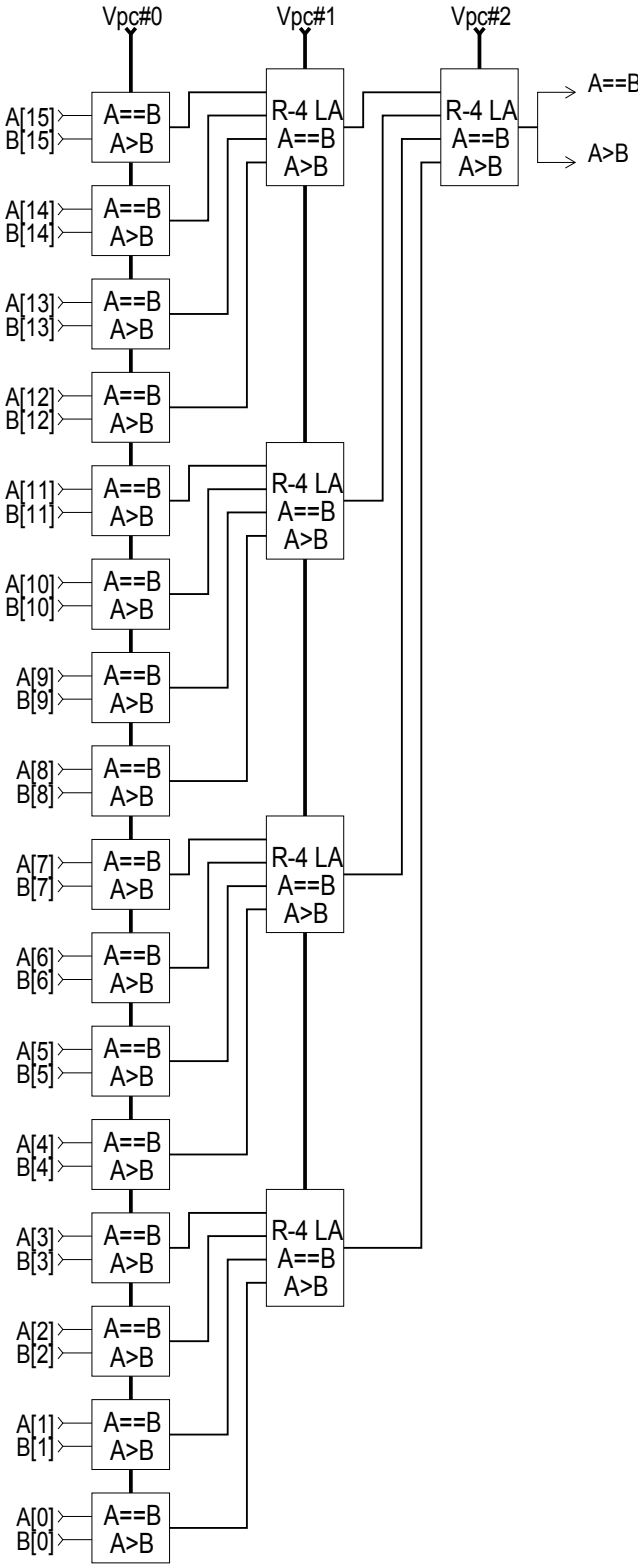

Figure 8.5: Comparator for GCD

8.4 The Subtractor / Reverse Subtractor

The subtractor / reverse subtractor is based upon the radix-four Carry Look-ahead Adder (CLA) that was presented in [Will08a]. Depending upon a single-bit input “R”, it will perform an unsigned integer subtraction between the two w-bit wide data-bus inputs “A” and “B”, with “R” selecting either “A” as the subtrahend and “B” as the minuend, or “A” as the minuend and “B” as the subtrahend, and with the output data-bus being “Z”, this is described by equation (8.1).

$$Z = \left\{ \begin{array}{l} (+A) + (-B) \;\; if \; R = 0 \\ (-A) + (+B) \;\; if \; R = 1 \end{array} \right\} \;\; \forall A, B, Z \in 0 \leqslant \mathbb{Z} < 2^{w} \qquad (8.1)$$

The implementation uses a plurality of XOR/XNOR gates (deployed as a complementary pair of n-bit wide programmable inverters) to selectively perform the ones’ complement of a single input bus, whilst leaving the other unmodified. The conversion to two’s complement is achieved by using a fixed carry-in of one (a “*hot-one*”) into the LSB of the subtractor. The initial “propagate” and “generate” signals are generated at the next stage, although because there are only three inputs, these stages could have been merged.

This is followed by the main carry look-ahead logic, which in this sixteen-bit example occupies two further stages, and would increase by a further stage every time the data-path width was quadrupled. This is then followed by a final assimilation stage where the initial “propagate” signals, which are buffered through the Carry Look-ahead (CL) logic, are XORed with the “generate” signal, calculated by the CL logic. The complete subtractor / reverse subtractor is shown in Figure 8.6.

This particular version is based upon the Sklansky's [Skla60] adder structure, although there is nothing to prevent the Kogge-Stone [Kogg73] adder structure, or the structure of any of the Knowles family of adders

[Know99] being used. The paper which disclosed the *Asynchrobatic* adder upon which this subtractor is based [Will08a], appears to be the first that explicitly suggests using Higher Radix Knowles Adders (HRKA) in an application. However, both higher radix Sklansky adders [Will99] and higher radix Kogge-Stone adders [Gurk00] have previously been proposed, and there is another previously published work that has alluded to a multi-dimensional design space for parallel prefix structures [Zieg04].

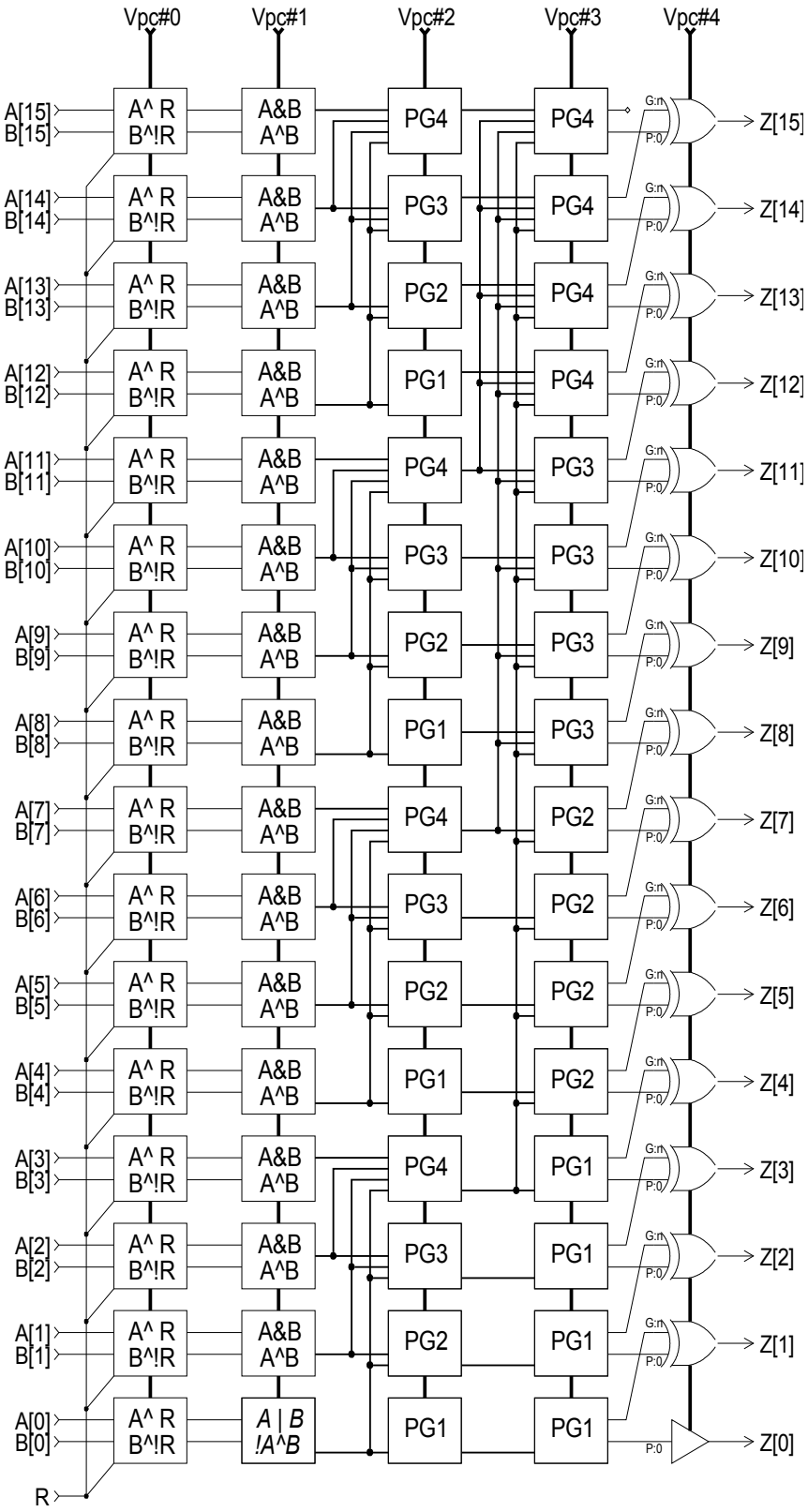

Figure 8.6: Subtractor / Reverse subtractor for GCD [Will08a]

8.5 Control logic

As well at the adiabatic data-path, there is also the asynchronous control logic. The required asynchronous control structures were as follows:

- Simple pipeline element
- Pipeline element with token (active after reset)
- Multiplexer
- De-Multiplexer

The comparator and subtractor are controlled as simple pipelines. The decision as to which of the two buses represents the minuend and which represents the subtrahend is simply implemented as a multiplexer in the data-path. However, the most complex control logic is the interface between the equality output of the comparator and the inputs to multiplexer where the while-loop is implemented. This structure needs to be seeded with an initial token so that the GCD circuit will operate correctly after a reset, and needs to move a single signal from the data-path domain, and integrate it into the control domain. The seeded token is marked “T0”, and it can be seen that the reset signal also directly drives into the data-path element.

The initialisation token is required to place the GCD calculator into a state where its input MUX is set so it is waiting for the two external data-path inputs and an external request signal. This is also the state that the GCD calculator would be in just after it had produced a result. The initial token is generated using the global reset signal. For most parts of the *Asynchrobatic* controller circuitry, this signal is just required to ensure that the asynchronous SWC logic is in the “Idle” phase. However, in locations where an initialisation token is required, the asynchronous SWC logic is required to be in the “Hold” phase, where the function is evaluated. Since the single-bit control logic is bi-stable and could initialise into either state, the reset signal must also ensure that the control logic is initialised to the correct value.

After the input MUX, there is the comparator stage, which also buffers the buses. The output from the comparator, which is in the data-path domain, drives a DeMUX, which is in the control domain. This selects between outputting the result of the GCD calculation if the comparator result shows both buses hold the same value, or performing a subtraction if the two buses contain different values. The output from the subtractor, along with the subtrahend, are fed-back into the internal inputs of the input MUX.

The simplified structural block diagram for the complete *Asynchrobatic* GCD circuit is shown in Figure 8.7. In that figure, the rectangles labelled "SWC" represent the standard Asynchronous Stepwise Charging control logic of a pipeline element. The rectangles marked "T0", "MUX and "DMX" represent a pipeline element with an initial token, a multiplexer and a demultiplexer respectively. In the data-path, the buffers and multiplexers are represented by their usual symbols (triangle and symmetrical trapezoid), and the complex data-path functions of the comparator and subtractor / reverse subtractor are annotated with their functions. The reset signal is can be seen to be entering the data-path buffer in the element with the initial token. The signals "**A**", "**B**" and "**Z**" are each 16-bit wide dual-rail buses. The request and acknowledge signals are the standard single-bit, positive logic handshaking signals used for asynchronous communication.

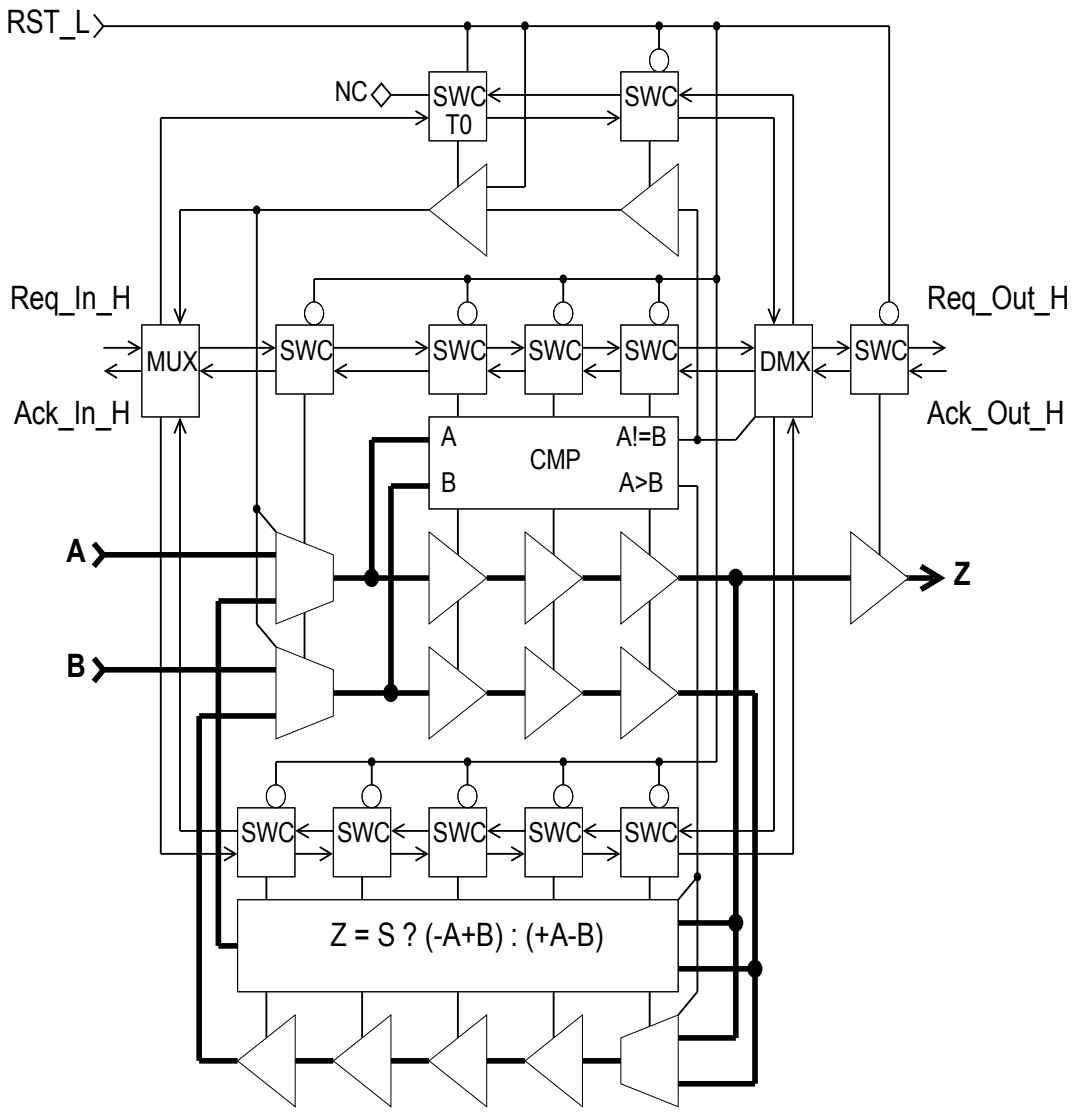

Figure 8.7: Asynchrobatic GCD [Will08c]

8.6 Performance

The functioning single-rail Verilog description was translated, mainly using regular expressions, into a SPICE netlist. This was simulated using Eldo MACH, a faster table-lookup-based circuit simulator from Mentor Graphics. The SPICE results were presented for the maximum length Fibonacci-based test. The results presented in [Will08c] and shown below in Table 8.1 kept the voltage and temperature fixed at their nominal values of

3.3V and 25ºC, and showed that as would be expected for an asynchronous design, the delay increased for a slower process. The presented work also noted that the power consumption of the asynchronous controller part of the circuit dominated the much larger adiabatic data-path by a factor of about three. It is obvious that the single-bit control path is inefficient in *Asynchrobatic* Logic, as was noted in that paper. However, it was later realised that the circuit's speed could have been improved by merging the logic of the first two pipeline stages of the subtractor into a single, more complex, logic function.

Process; (3.3V; 25ºC)	**Delay (µs)**	**Controller Power (nW)**	**Data-path Power (nW)**	**Total Power (nW)**
Fast-Fast	1.022	2.627	0.8034	3.430
Typical	2.067	2.577	0.6801	3.257
Slow-Slow	5.205	2.353	0.6252	2.978

Table 8.1: Performance results for GCD circuit [Will08c]

Three 10pF capacitors were used as the tank capacitors. Based upon the dimensions of the q-boxes in the previous chapter, and manufacturing data from the foundry [AMIS02] & [AMIS03], it is reasonable to believe that devices of this capacitance could be easily constructed on-chip. This assertion is based upon the typical capacitance of an on-chip capacitor constructed between the two layers of polysilicon, which is quoted as 1.1fF/µm² [AMIS02], these capacitors would need to occupy 9,090µm². This gives dimensions for these capacitors of 90.9µm × 100µm. If these are compared to the dimensions of the q-boxes from the previous chapter, it is clear from their similarity that this size of capacitor is perfectly viable as an on-chip device. Furthermore, since these capacitors do not need to be matched, and thus can have metal layers above them utilised, the parasitic capacitances that would naturally occur between the five metal layers above the capacitor can be exploited to increase the available capacitance. This can be done by filling these locations with a parasitic capacitor constructed from

three-dimensionally interdigitated pieces of metal laid out with the unusual aim of maximising the coupling, area and fringe capacitances of this parasitic capacitor.

8.7 Testing

The implementation of the GCD circuit was tested using two simple tests that exercise a reasonable proportion of the subtractor. The choice of test vectors for this circuit is important, because poorly chosen or random tests are likely to lead to the GCD becoming a down-counter, or, if one input is zero, getting stuck in an infinite loop! The worst case pair of inputs that will eventually produce a result would be the maximum representable value and either one or one less than the maximum representable value. This is an artefact of the algorithm, not a fault with the circuit.

There are a series of short tests that can be used to validate and verify the design. The tests are shown in Table 8.2 below. The first test shows that the GCD calculator will correctly exit, and does not exercise the subtractor. The second test performs one subtraction. The third test is the first where the output is different from both inputs, showing conclusively that the circuit is performing iterations. These tests can be extended to the required number of cycles by defining the relationship between the inputs as P:(n/n+1)P, with both values being integers.

Relationship between inputs	**Maximum input values (16-bit)**	**Required subtraction cycles**	**Expected result**
P : P	0xFFFF : 0xFFFF	0	0xFFFF
P : ½P	0xFFFE : 0x7FFF	1	0x7FFF
P : ⅔P	0xFFFF : 0xAAAA	2	0x5555

Table 8.2: Simple test vectors for the GCD circuit

The only requirement for (P) is that it is a positive integer within the range of the bit-width (w) of the GCD circuit. This is defined in equation (8.2).

$$0 < P < 2^W \tag{8.2}$$

The longer test generates the well known Fibonacci series [Pisa02] on the internal data-paths. This test would remain viable on ultra-wide versions of the GCD calculator, as lists of large Fibonacci numbers are available on-line. This test requires two Fibonacci numbers, $F_{(n)}$ and $F_{(n-1)}$ where $F_{(n)}$ is the n^{th} Fibonacci, and this must adhere to the inequality shown in equation (8.3).

$$F_{(n)} \leq 2^W - 1 \tag{8.3}$$

With a 16-bit wide data-path, the value of n of $F_{(n)}$ is 24, for a 32-bit wide data-path it is 47, and for a 64-bit data-path it would be 96. This shows that this test will scale to a usable length even for high data-path widths.

8.8 Simulation Results

8.8.1 Verilog

The behavioural Verilog for the GCD circuit was simulated, and data captured from the input and output ports of the Device Under Test (DUT) as would be possible for a physical implementation. Internal probes were placed on the outputs of the subtractor and subtractor bypass buffers so that visibility of the intermediate internal results was available. These output listings are shown after this paragraph and continued on the subsequent page. Input and Output transactions are labelled as such with the "*request*" and "*acknowledge*" signal being shown, and bundled-data being presented both in hexadecimal and decimal. Transactions on the internal probe point are marked "M8" and just show the bundled data, again, both in hexadecimal and decimal. During the start-up phases, signals shown as "X" are in an undefined state and after

operations have occurred on the data-bus, signals shown as “Z” are high-impedance, indicating that there is no valid data on the data-bus at that time. Code Sequence 7, which is continued over two pages, shows four truncated output sequences. The first is the initialisation that does not form part of the tests, but does show the inputs being initialised to the “*Idle*” state, and the reset signal forcing the outputs to go from an undefined state, also into the “*Idle*” state. After this initialisation, the tests follow, and are clearly delineated from each other. The former locations of the removed tests can be seen from the large gaps in the progression of simulation time. There is no ulterior motive for the removal of these tests as they produced the expected results; they have only been omitted for brevity.

```
Trigger   Time Req/Ack Hexadecimal   (Decimal)
=============================================
M8          60         zzzz zzzz (    z     z)
Inputs      70 R:0 A:0 zzzz zzzz (    z     z)
Outputs     90 R:0 A:x      zzzz       (    z)
Outputs    170 R:0 A:0      zzzz       (    z)
---------------------------------------------
Inputs    1050 R:0 A:0 FFFF FFFF (65535 65535)
Inputs    1260 R:1 A:0 FFFF FFFF (65535 65535)
Inputs    1410 R:0 A:1 FFFF FFFF (65535 65535)
Inputs    1660 R:0 A:0 zzzz zzzz (    z     z)
Outputs   1710 R:1 A:0      FFFF       (65535)
Outputs   1790 R:1 A:1      FFFF       (65535)
Outputs   2010 R:0 A:1      zzzz       (    z)
Outputs   2090 R:0 A:0      zzzz       (    z)
---------------------------------------------
Inputs  203490 R:0 A:0 AAAA FFFF (43690 65535)
Inputs  203700 R:1 A:0 AAAA FFFF (43690 65535)
Inputs  203850 R:0 A:1 AAAA FFFF (43690 65535)
Inputs  204100 R:0 A:0 zzzz zzzz (    z     z)
M8      204420         5555 AAAA (21845 43690)
M8      204720         zzzz zzzz (    z     z)
M8      205190         5555 5555 (21845 21845)
M8      205490         zzzz zzzz (    z     z)
Outputs 205700 R:1 A:0      5555       (21845)
Outputs 205780 R:1 A:1      5555       (21845)
Outputs 206000 R:0 A:1      zzzz       (    z)
Outputs 206080 R:0 A:0      zzzz       (    z)
---------------------------------------------
                Continued...
```

Code Segment 7: Verilog results of simple GCD tests

```
Trigger   Time Req/Ack Hexadecimal   (Decimal)
-------------------------------------------
Inputs  354710 R:0 A:0 6FF1 B520 (28657 46368)
Inputs  354920 R:1 A:0 6FF1 B520 (28657 46368)
Inputs  355070 R:0 A:1 6FF1 B520 (28657 46368)
Inputs  355320 R:0 A:0 zzzz zzzz (    z     z)
M8      355640         452F 6FF1 (17711 28657)
M8      355940         zzzz zzzz (    z     z)
M8      356410         2AC2 452F (10946 17711)
M8      356710         zzzz zzzz (    z     z)
M8      357190         1A6D 2AC2 ( 6765 10946)
M8      357490         zzzz zzzz (    z     z)
M8      357960         1055 1A6D ( 4181  6765)
M8      358260         zzzz zzzz (    z     z)
M8      358740         0A18 1055 ( 2584  4181)
M8      359040         zzzz zzzz (    z     z)
M8      359510         063D 0A18 ( 1597  2584)
M8      359810         zzzz zzzz (    z     z)
M8      360290         03DB 063D (  987  1597)
M8      360590         zzzz zzzz (    z     z)
M8      361060         0262 03DB (  610   987)
M8      361360         zzzz zzzz (    z     z)
M8      361840         0179 0262 (  377   610)
M8      362140         zzzz zzzz (    z     z)
M8      362610         00E9 0179 (  233   377)
M8      362910         zzzz zzzz (    z     z)
M8      363390         0090 00E9 (  144   233)
M8      363690         zzzz zzzz (    z     z)
M8      364160         0059 0090 (   89   144)
M8      364460         zzzz zzzz (    z     z)
M8      364940         0037 0059 (   55    89)
M8      365240         zzzz zzzz (    z     z)
M8      365710         0022 0037 (   34    55)
M8      366010         zzzz zzzz (    z     z)
M8      366490         0015 0022 (   21    34)
M8      366790         zzzz zzzz (    z     z)
M8      367260         000D 0015 (   13    21)
M8      367560         zzzz zzzz (    z     z)
M8      368040         0008 000D (    8    13)
M8      368340         zzzz zzzz (    z     z)
M8      368810         0005 0008 (    5     8)
M8      369110         zzzz zzzz (    z     z)
M8      369590         0003 0005 (    3     5)
M8      369890         zzzz zzzz (    z     z)
M8      370360         0002 0003 (    2     3)
M8      370660         zzzz zzzz (    z     z)
M8      371140         0001 0002 (    1     2)
M8      371440         zzzz zzzz (    z     z)
M8      371910         0001 0001 (    1     1)
M8      372210         zzzz zzzz (    z     z)
Outputs 372420 R:1 A:0      0001       (    1)
Outputs 372500 R:1 A:1      0001       (    1)
Outputs 372720 R:0 A:1      zzzz       (    z)
Outputs 372800 R:0 A:0      zzzz       (    z)
```

Code Segment 7: Verilog results of GCD Fibonacci test

8.8.2 SPICE

Having used the Verilog simulations to validate and verify the design and its results, the simulations can be now performed using SPICE.

If care is not taken, SPICE can generate vast quantities of simulation result data, so only minimal probing of internal nodes was performed. This limits which internal signals may be displayed.

Figure 8.8 shows an example of a simulation cycle that would be used to take power measurements. This figure and the following two are all from the same simulation run, which was performed using slow-n, slow-p models with nominal voltage (3.3V) and nominal temperature (25ºC). The uppermost graph shows the voltages on each of the tank capacitors (labelled V(VC1), V(VC2) and V(VC3)). The next graph shows one of the internally generated stepwise power-clock signals (labelled XDUT.ASWC0). This is the SWC waveform that drives the input MUX in the centre-left of Figure 8.7. The digital graphs show the input, output and internally probed signals, these clearly show that the data-path returns to zero when not driven, although due to the scale, only the transitions can be seen, rather than the values. The input and output "*request*" and "*acknowledge*" signals are shown in the penultimate graph, and the final graph was produced by the waveform processing tool, and shows the result of integrating the current drawn by the power supplies driving the ASWC control logic (labelled "idd") and the SWC circuit (labelled "ipc"). It can be seen that the majority of of the current is drawn by the ASWC control logic.

Figure 8.9 shows a close-up of the first two transactions of the Figure 8.8. The uppermost graph again shows the voltages on each of the tank capacitors (labelled V(VC1), V(VC2) and V(VC3)). The next graph shows one of the internally generated stepwise power-clock signals (labelled XDUT.ASWC0). The exponential charging within each charging step can be clearly seen. The final, digital graph shows the two transactions in sufficient detail that the hexadecimal arithmetic and subsequent return-to-zero can be seen to be occurring correctly.

Figure 8.10 shows the start-up cycle, which wasn't used for power or performance measurement. It can be clearly seen from this that the tank capacitors converge to operational voltages quickly after computation activity commences, although the speed of convergence will be determined by the size of the tank capacitors and the capacitive load being driven.

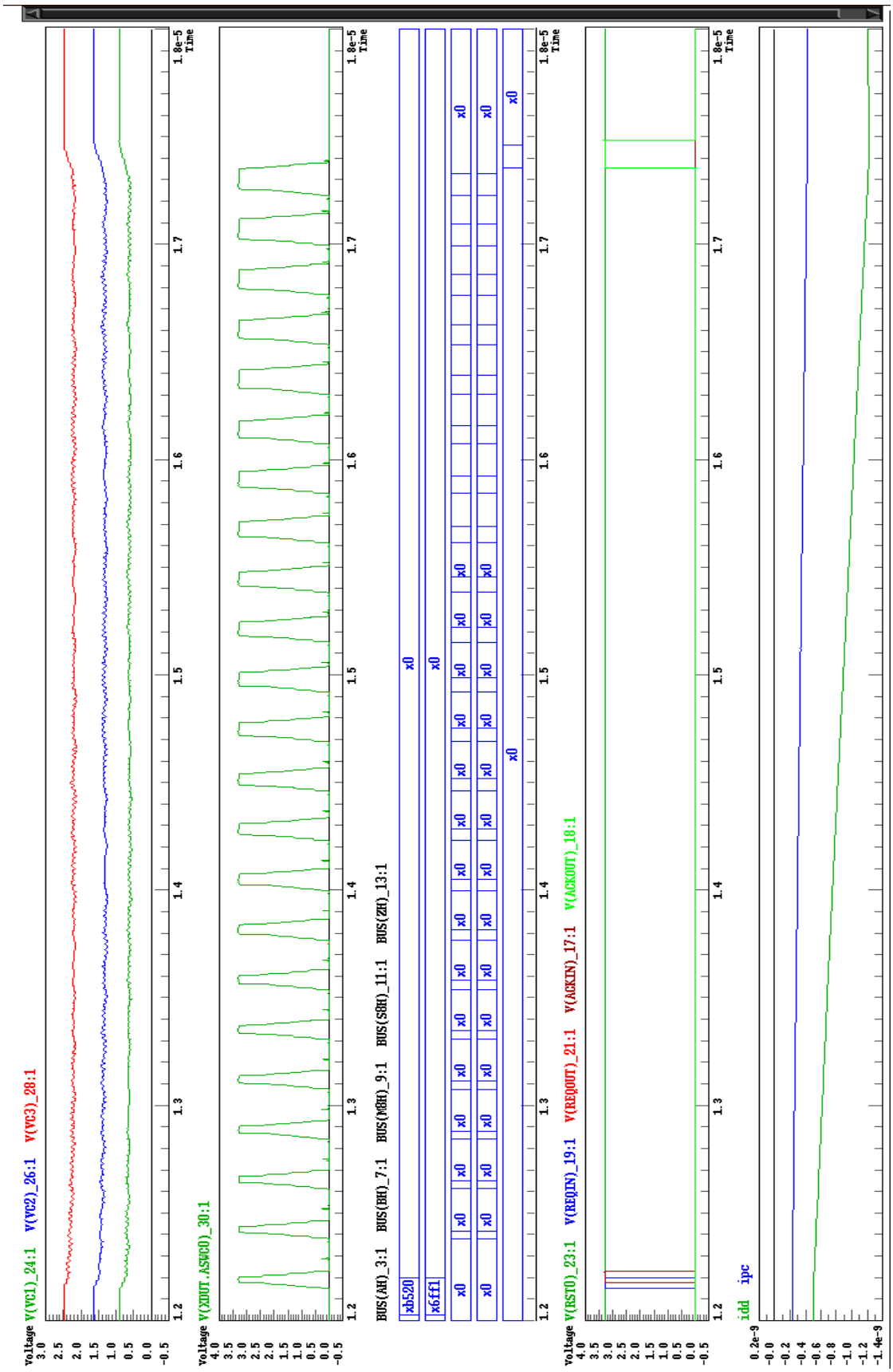

Figure 8.8: GCD complete trace

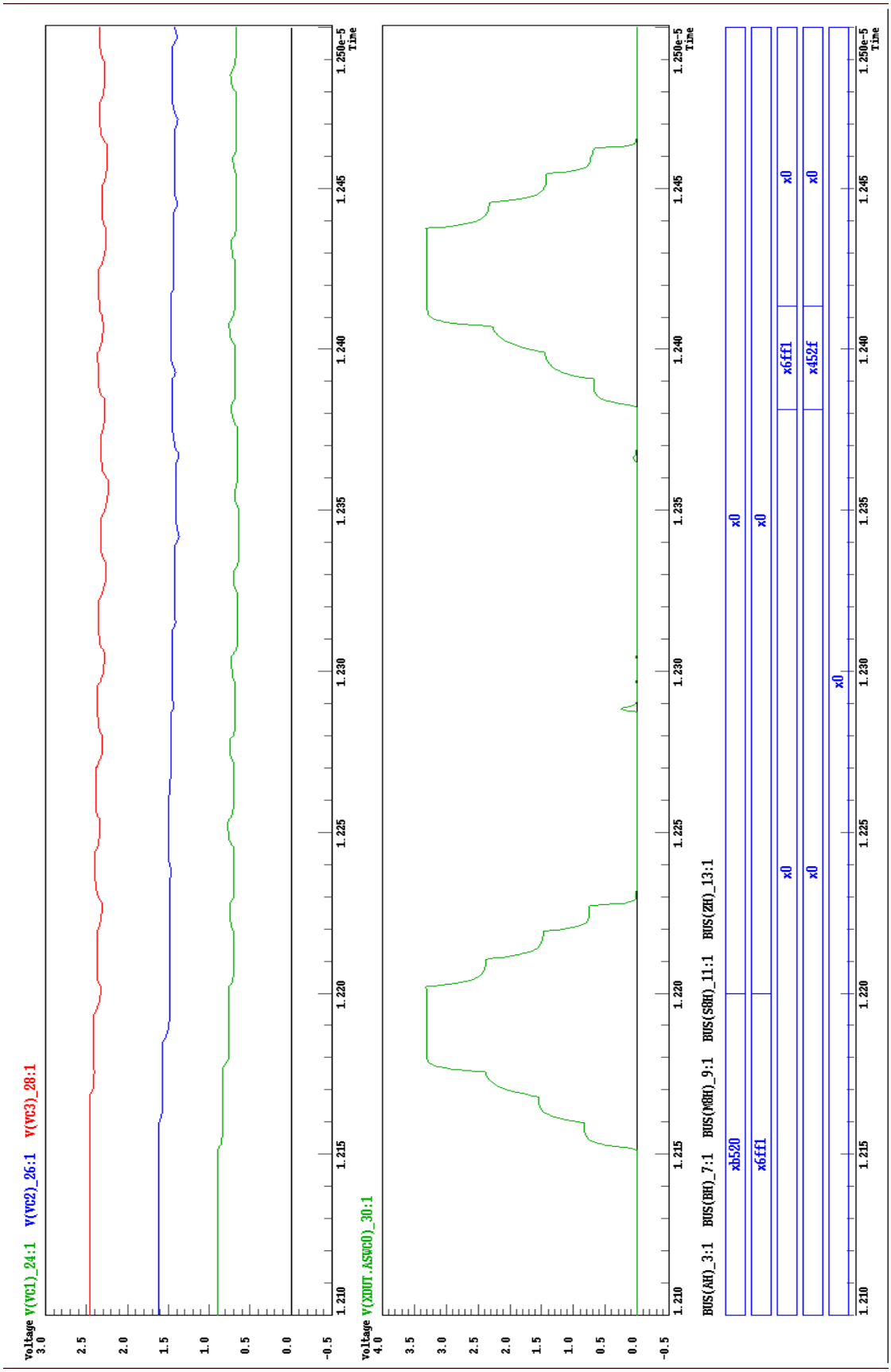

Figure 8.9: GCD close-up

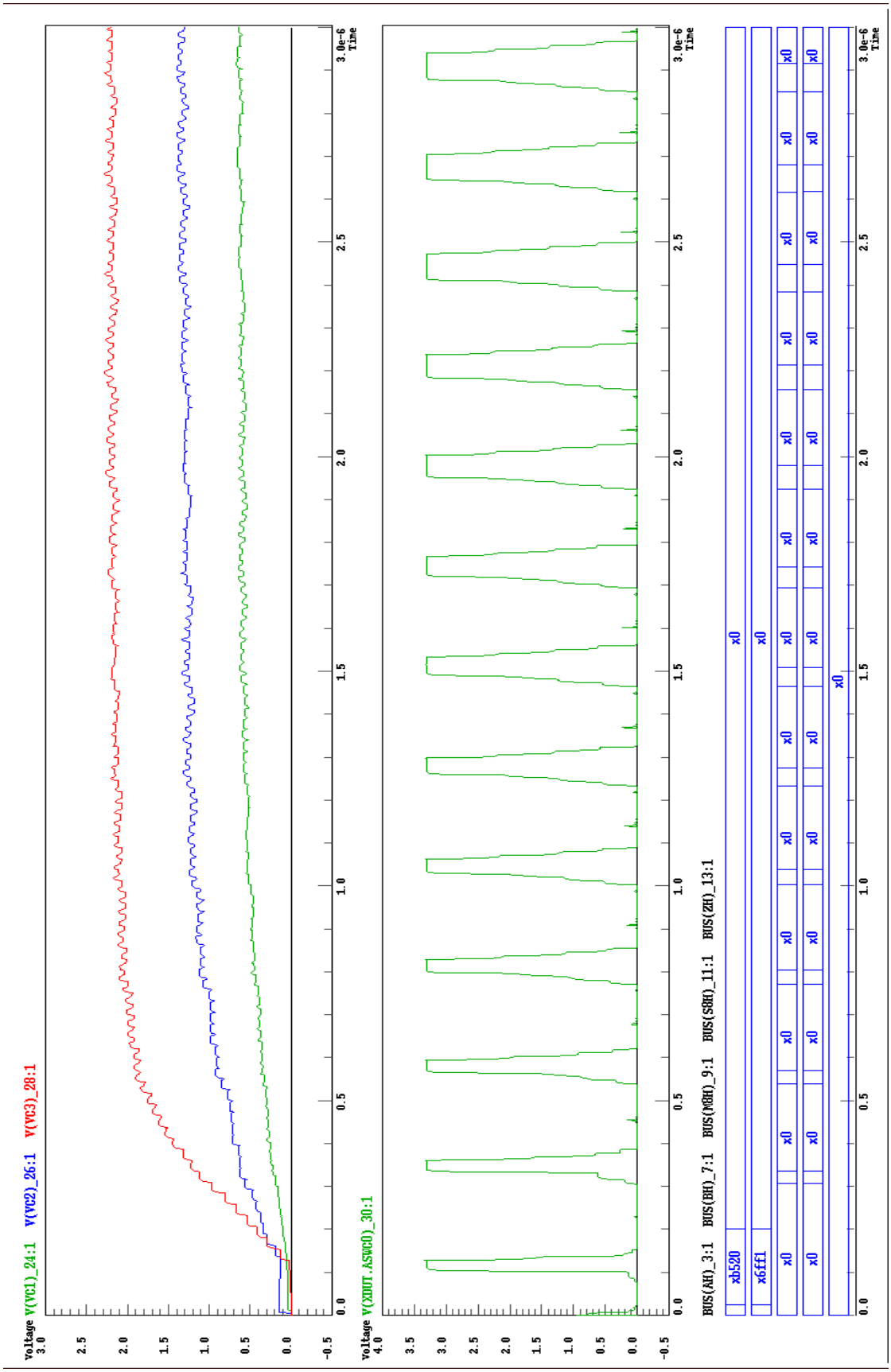

Figure 8.10: Start-up performance of GCD circuit

8.9 Summary

The ability to implement the GCD circuit is a major achievement, as it proves that *Asynchrobatic* Logic can be use to implement complex systems that include decision and iteration. Since these are two essential structures in computational systems, it is only a small step to extend this and confidently argue that this achievement shows that complex processing systems can be implemented using *Asynchrobatic* Logic.

As well as producing an *Asynchrobatic* logic circuit that could perform an iterative computation, it has been shown that complex *Asynchrobatic* logic circuits can be modelled using Verilog HDL, which shows that even larger systems could be constructed if so desired.

Chapter 9 Conclusions and Future work

9.1 Conclusions

In the first four chapters of this thesis, the foundations upon which *Asynchrobatic* Logic is built have been introduced. These are Adiabatic Logic, Asynchronous Logic, and the design methods for dual-rail logic. Although the majority of this material was previously known, this introduction extended knowledge by quantifying how the number of inputs causes the size of the search space for Free n-ary Decision Diagrams to increase, along with suggestion possible applications in Multi-Valued Logic (MVL).

Asynchrobatic Logic has been introduced. It has been shown to be a viable method for the implementation of novel, low-power, complex systems. The fact that an operational implementation of the Greatest Common Denominator (GCD) algorithm could be produced shows that systems capable of performing iteration and decision can be implemented. The significance of this is that it has been shown that these two constructs are required to allow structured programming [Böhm66]. This implies that *Asynchrobatic* Logic could be used to implement arbitrarily complex computational systems. Although the GCD was a netlist only simulation, the layout implementation of the q-boxes from the Twofish algorithm shows that viable physical implementations can also be produced.

As well as producing circuitry capable of implementing *Asynchrobatic* Logic, this work has shown that it is possible to model *Asynchrobatic* Logic using the Hardware Description Languages (HDLs) Verilog and VHDL. The ability to model complex systems using more abstract representations is essential because performing SPICE simulations would take too long, and any mistakes would be far more difficult to locate or diagnose.

As with any VLSI system or engineering project, *Asynchrobatic* Logic has made some compromises to achieve design goals. The main compromise made by *Asynchrobatic* Logic is the width of the low-power, charge recovering, adiabatic data-path needed to amortise the power used in the asynchronous controller. However, there can be no doubt that *Asynchrobatic* Logic achieved its basic aim of unifying the low-power benefits of asynchronous logic and adiabatic logic.

The relevance of the novel idea of *Asynchrobatic* Logic has been validated by others following the lead, with the initial paper that disclosed *Asynchrobatic* Logic [Will04], being cited in at least two other papers, one from Carleton University, Ottawa, Canada, where the main expertise appears to be in adiabatic logic [Arsa07] and one from Newcastle University, Great Britain, where the expertise is in asynchronous logic [Asim08]. Indeed, in his paper Arsalan quotes directly "*If the power reducing properties of these techniques could be combined, then it should be possible to produce a logic design methodology that is only active when it is performing useful computations , and recycles a large proportion of the energy used to perform those computations.*" The accolade of being directly quoted is obviously very welcome.

It is worth comparing *Asynchrobatic* Logic with the alternative suggestions for asynchronous and adiabatic logics from others. The suggestion by Arsalan *et al.* [Asra07] uses a "Control and Regeneration" block that appears to consist of "*a conventional CMOS OR*" gate. The idea appears to be to use a dual-rail asynchronous signalling method (similar to that suggested for narrow *Asynchrobatic* data-paths), with charge being forwarded from the asserted output of the previous stage to enable adiabatic operation. However, this method would seem to provide an approximation to adiabatic charging, but does not appear to be capable of charge recovery.

The suggestion of Asimakpoulos *et al.* [Asim08] can be summarised as asynchronously connecting an adiabatic data-path to a resonant clock. This is a perfectly reasonable alternative suggestion, but may be somewhat harder to implement on silicon than it is to simulate and to do this requires peak and trough detectors, which are probably just as complex as Asynchronous Stepwise Charging logic, and the design still requires an off-chip inductor.

Another area where this work has extended the state-of-the-art is with respect to Reversible Logic. If this work is compared with that of Khazamipour [Khaz05] & [Khaz06], who has investigated using Reversible Energy Recovery Logic (RERL), the improvements can be quantified. Moving from using RERL, which has eight-phase clocking, as a method for implementing reversible logic circuits, to using PFAL, which only requires four-phase clocking makes the clocking scheme half as complex. The Positive Feedback Adiabatic Logic (PFAL) solution requires 76 transistors. In comparison, the RERL solution appears to require at least 94 devices, which means that the proposed PFAL solution operates with at least 20% fewer transistors. This quantification remains in some doubt, as it is somewhat unclear from the previous work how many devices are required to make a two-input AND gate reversible. Clearly with either four-phase clocking, or *Asynchrobatic* operation, the use of PFAL to implement reversible logic circuits represents a reduction in complexity, and a clear advancement of knowledge.

9.2 Novelty claims and contributions

The main claim of novelty and advancement of the state-of-the-art is the concept of *Asynchrobatic* Logic. Until this concept had been introduced asynchronous design and adiabatic design had been separate and isolated fields or research. Therefore, the introduction of a asynchronous, adiabatic system is a major advancement in the field of low-power microelectronics.

The implementation of this using capacitor-based, asynchronous stepwise charging as a method for driving adiabatic data-path logic, discloses a viable implementation of *Asynchrobatic* logic. This was extended by showing that complex data-paths and complex control structures can also be implemented.

Methods for modelling *Asynchrobatic* Logic in industry standard Hardware Description Languages (HDLs), like Verilog and VHDL were proposed. This implemented code is novel, because to model *Asynchrobatic* logic correctly, both rising and falling edges of clock signals need to be considered.

A systematic identification of other potential adiabatic logic families based around cross-coupled pairs of PMOS transistors was performed. It identified a previously undocumented adiabatic logic family. However, it did not show any extra low-power benefits over previously disclosed technologies.

The use of the Positive Feedback Adiabatic Logic (PFAL) family to implement complex reversible processing logic represents a substantial step forward. It allows reversible logic, to be implemented using *Asynchrobatic* logic, thus allowing fully adiabatic systems, to be created.

A generalisation of results for the rate of growth of the search space for "Free *n*-ary Decision Diagrams". These sequences only appear to have been documented for binary and ternary decision diagrams, but could be usefully extended to Free Quaternary, Quinary or higher-order Decision Diagrams, with possible applications being the design of functions for Multi-Valued Logics (MVL).

9.3 Applications and future work.

There is no reason why *Asynchrobatic* Logic could not be commercialised. Obvious wide-data-path applications for *Asynchrobatic* logic include Very Long Instruction Word (VLIW), Floating-Point, Single Instruction Multiple Data (SIMD), Vector and cryptographic processors. In the field of cryptography, as well as obvious targets like block ciphers, there is potential to implement data-paths to allow processing for Elliptic Curve Cryptography (ECC). These use Galois Field (GF) arithmetic over prime bit-widths, for example GF(2^{173}) [Leun03]. Another potential benefit of using these systems for cryptography is the potential to reduce side channel information leakage. It is conjectured that the asynchronous nature of operations will make it harder to derive information from circuit timing, and that reversible (adiabatic) operation can reduce susceptibility to Differential Power Analysis (DPA) attacks [Thap06]. Furthermore, the dual-rail nature of the logic, the lower power consumption and what is effectively power-supply damping caused by the tank capacitors should also combine to further reduce this technology's susceptibility to power analysis. This dual-rail implementation may also make the circuit more tamper resistant, because attempting to inject data could cause the dual-rail wiring to enter an invalid state, allowing this unauthorised access to be detected.

The register-file structure presented by Moon et al. [Moon98] could be converted to *Asynchrobatic* operation. This would allow the efficient implementation of systems that require register-style storage. Whilst the availability of register-files is not absolutely essential, as they can be implemented by the feedback of reused values around a loop with a multiplexer (MUX) to allow new data to be written, this functional block would be one of the most desirable to implement as the suggested alternative is nowhere near as efficient as a randomly addressable register-file. There are some more design complications with register-file design, as the Static RAM (SRAM) cell would need to be margined to ensure that its contents can be

reliably written, stored and read. Register-files are essential components in most processors.

The power consumption of the asynchronous controller probably has potential for further optimisation. It is likely that it could be further reduced. This could be achieved for example by using a lower-power logic style than standard static CMOS. As an example, further work could consider using sub-threshold, current mode circuits.

With the demonstrated potential to implement fully reversible circuits using this technology, further research looking at implementing more complex reversible gates would be useful. Demonstrating more complex (and more useful) reversible gates is likely to cause interest in PFAL-based *Asynchrobatic* Logic from those researching reversible computation.

As noted previously, this work approached the idea from a position of intellectual strength that was more superior in terms of adiabatic logic. The steeper learning curve for asynchronous logic has alluded to further areas of crossover that may be exploitable in the future. There is a substantial tranche of work in asynchronous logic that is predicated upon the use of Differential Cascode Voltage Switch Logic (DCVSL) circuitry. Given that the adiabatic logic circuits are also predicated, albeit in a different way, on DCVSL circuits, there may be further exploitable potential in this area. The best example of this is at the control/data-path interface where the result of a data-path operation is a single bit that must influence the control structure. If this single bit cannot travel with other data in the wide data-path, then to operate it with an *Asynchrobatic* power-clock would not be the most efficient method for its propagation, and moving it into a purely asynchronous domain is likely to be more efficient.

The majority of the work performed in the evaluation of *Asynchrobatic* Logic was conducted using sub-micron, rather than deep sub-micron or

nanometre processes. However, there is evidence from simulations that adiabatic circuits will operate when implemented in deep sub-micron processes, and no reason to expect this situation to change for nanometre processes. In fact, the availability of devices with different threshold voltages (V_T) devices provide further optimisation potential for all parts to the design.

Given the extensive list of different adiabatic logic families that have been proposed, and which are catalogued in the appendix, it would be a worthwhile exercise to systematically compare the power, area and performance of a large number of these for several benchmark tests under defined process conditions on a variety of different CMOS processes. These benchmarking tests would ideally use components that are of use in production systems like arithmetic units or parts of cryptographic systems, rather than having tests based upon buffers or inverters.

The decision to limit decision tree depth to four NMOS devices is based upon the Electrical Rule Checker (ERC) limits that are imposed on standard static CMOS to prevent problems caused by CMOS switches being resistive, non-ideal switches. However, due to the different nature of operation of *Asynchrobatic* Logic and the underlying adiabatic logic families in its data-path, it may be possible to waive this arbitrary limit in some situations. However, further analysis of how this affects performance would be necessary.

Finally, other design methods for obtaining more optimal implementations of functions should be investigated as and when they are published.

Chapter 10 References and Bibliography

10.1 References

[Adam79] Douglas Adams, “The Hitch-Hiker's Guide to the Galaxy”, Pan Books, page 135, October 1979, ISBN: 0-330-25864-8.

[AMD09] Advanced Micro Devices Inc., “AMD Family 10h Desktop Processor Power and Thermal Data Sheet”, revision 3.18, February 2009. Available via the following URL: http://www.amd.com/.

[Amir00] E. Amirante, J. Fischer and D. Schmitt-Landsiedel, “Verlustleistungsschwankungen in adiabatischen Schaltungen”, ITG-Fachbericht 162, “Mikroelektronik für die Informations-technik”, Darmstadt, Germany, 20th-21st November 2000, pages 133-138. Translation of title is “Variations of the Power Dissipation in Adiabatic Circuits”.

[Amir04] Ettore Amirante, “Adiabatic Logic in Sub-Quartermicron CMOS Technologies”, Shaker Verlag GmbH, Aachen, Germany, 2004. ISBN: 3-8322-3487-X. ISBN13: 9783832-234874.

[AMIS02] AMI Semiconductor Belgium BVBA, “C035M Design rule manual supplement for analogue option (C035M-A)”, DS13337, Revision: 3.0, 7th August 2002.

[AMIS03] AMI Semiconductor Belgium BVBA, “C035M-D Design rule manual”, DS13330, Revision: 5.0, 16th January 2003.

[Arma98] A. Armah, and A. Jaekel, “An ordering-insensitive methodology for efficient DCVS circuit synthesis”, Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering (CCECE’98), volume 1, pages I:49-52, Waterloo, Ontario, Canada, 24th-28th May 1998.

[Arsa07] M. Arsalan and M. Shams, “Asynchronous Adiabatic Logic”, Proceedings of the International Symposium on Circuits and

Systems (ISCAS'07), pages 3720-3723, New Orleans, Louisiana, USA, 27th-30th May 2007.

[Asim08] P. Asimakpoulos and A. Yakovlev, “An Adiabatic Power-Supply Controller for Asynchronous Logic Circuits”, Web pages of the 20th UK Asynchronous Forum, Manchester, Great Britain, 1st-2nd September 2008. Available at the following URL: http://intranet.cs.man.ac.uk/apt/async/events/ukforum20/.

[Atha97] W. C. Athas, N. Tzartzanis, L. “J.” Svensson, L. Peterson, H. Li, P. Wang and W.-C. Liu, “AC-1: a clock-powered microprocessor”, Proceedings of the 1997 International Symposium on Low Power Electronics and Design (ISLPED’97), pages 328-333, Monterey, California, USA, 18th-20th August 1997.

[Benn73] C. H. Bennett, “Logical Reversibility of Computation”, IBM Journal of Research and Development, 17(6):525-532, 1973.

[Benn82] C. H. Bennett, “The thermodynamics of computation – a review”, International Journal of Theoretical Physics, 21, pages 905-940, 1982. Reprinted in [Leff03].

[Bern93] J. Bern, J. Gergov, C. Meinel and A. Slobodová, “Boolean Manipulation with Free BDD's. First Experimental Results.”, European Design and Test Conference, pages 200-207, Paris, France, 28th February - 3rd March 1994.

[Blac69] J. R. Black, “Electromigration – A brief survey and some recent results”, IEEE Transactions on Electron Devices, ED16(4):338-347, April 1969.

[Blot02] A. Blotti, S. Di Pascoli and R. Saletti, “A comparison of some circuit schemes for semi-reversible adiabatic logic”, International Journal of Electronics, 89(2):147-158, February 2002.

[Blot04] A. Blotti and R. Saletti, “Ultra low-power adiabatic circuit semi-custom design”, IEEE Transactions on VLSI, 12(11):1248-1253, November 2004.

[Böhm66] C. Böhm and G. Jacopini, “Flow diagrams, Turing machines and languages with only two formation rules”, Communications of the ACM, 9(5):366-371, May 1966.

[Boll96] B. Bollig and I. Wegener, “Improving the variable ordering of OBDDs is NP-complete”, IEEE Transactions on Computers, 45(9):993-1002, September 1996.

[Brya86] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation”, IEEE Transactions on Computers, 35(8):677-691, August 1986.

[Chan02] R. C. Chang, P.-C. Hung and I.-H. Wang, “Complementary pass-transistor energy recovery logic for low-power applications”, IEE Proceedings on Computers and Digital Techniques, 149(4):146-151, July 2002.

[Choe97] Swee Yew Choe, G. A. Rigby and G. R. Hellestrand, “Half-rail differential logic”, 44th IEEE International Solid-State Circuits Conference – Digest of Technical Papers (ISSCC'97), pages 420-421 and 498, San Francisco, California, USA, 6th-8th February 1997.

[Chu86] K. M. Chu and D. L. Pulfrey, “Design procedures for differential cascode voltage switch logic circuits”, IEEE Journal of Solid-State Circuits, 21(6):1082-1087, June 1986.

[De96a] V. K. De and J. D. Meindl, “Complementary Adiabatic and Fully Adiabatic MOS Logic Families for Gigascale Integration”, 43rd IEEE International Solid State Circuits Conference – Digest of Technical Papers (ISSCC’96), pages 300-301 and 462, San Francisco, California, USA, 8th-10th February 1996.

[De96b] V. K. De and J. D. Meindl, “Opportunities for non-dissipative computation [adiabatic logic]”, Proceedings of 9th annual IEEE ASIC Conference and Exhibit (ASIC’96), pages 297-300, Rochester, New York, USA, 23rd-27th September 1996.

[De96c] V. K. De and J. D. Meindl, “A dynamic energy recycling logic family for ultra-low-power gigascale integration (GSI)”, Proceedings of the 1996 International Symposium on Low Power Electronics and

Design (ISLPED'96), pages 371-375, Monterey, California, USA, 12th-14th August 1996.

[Denk94] J. S. Denker, "A review of adiabatic computing", 1994 IEEE Symposium on Low Power Electronics – Digest of Technical Papers (ISLPE'94), pages 94-97, San Diego, California, USA, 10th-12th October 1994.

[Dhar96] S. Dharmasena and L. Svensson, "Startup Energies in Energy-Recovery CMOS", Proceedings of the fourth workshop on Physics and Computation (PhysComp'96), Boston, Massachusetts, USA, 22nd-24th November 1996.

[Dick94] A. G. Dickinson and J. S. Denker, "Adiabatic Dynamic Logic", Proceedings of the IEEE 1994 Custom Integrated Circuits Conference (CICC'94), pages 282-285, San Diego, California, USA, 1st-4th May 1994.

[Dick95] A. G. Dickinson and J. S. Denker, "Adiabatic Dynamic Logic", IEEE Journal of Solid-State Circuits, 30(3):311-315, March 1995.

[Dijk65] E. W. Dijkstra, "Solution of a problem in concurrent programming control", Communications of the ACM, 8(9):569, September 1965.

[Eucl70] Euclid; H. Billingsley (translator), "The Elements of Geometrie", Book VII, John Daye, London, 1570.

[Feis73] Horst Feistel, "Cryptography and Computer Privacy", Scientific American, 228(5):15-23, May 1973.

[Feyn00] Richard P. Feynman; Tony Hey and Robin W. Allen (editors), "Feynman lectures on computation", Westview Press, 2000, ISBN:0-7382-0296-7.

[Fisc04] J. Fischer, E. Amirante, A. Bargagli-Stoffi and D. Schmitt-Landsiedel, "Improving the positive feedback adiabatic logic family", Kleinheubacher Berichte 2003, in Advances in Radio Science, volume 2, pages 221–225, May 2004.

[Fisc05] J. Fischer, P. Teichmann and D. Schmitt-Landsiedel, "Scaling trends in adiabatic logic", Proceedings of the 2nd Conference on Computing Frontiers, pages 427-434, Ischia, Italy, 4th-6th May 2005.

[Fisc06] Jürgen Fischer, “Adiabatische Schaltungen und Systeme in Deep-Submicron-CMOS-Technologien”, Shaker Verlag GmbH, Aachen, Germany, 2006. ISBN: 3-8322-5576-1. ISBN13: 9783832-255763. Title translates to “Adiabatic Circuits and Systems in Deep Submicron CMOS Technologies”.

[Fran95] D. J. Frank and P. M. Solomon, “Electroid-Orientated Adiabatic Switching Circuits”, Proceedings of the 1995 International Symposium on Low Power Design (ISLPD’95), pages 197-202, Dana Point, California, USA, 23rd-26th April 1995.

[Fred82] E. Fredkin and T. Toffoli, “Conservative Logic”, International Journal of Theoretical Physics, 21(3-4):219-253, April 1982.

[Gaba94a] T. Gabara, “Pulsed Low Power CMOS”, International Journal of High Speed Electronics and Systems, 5(2):159-177, June 1994

[Gaba94b] T. Gabara, “Pulsed Power Supply CMOS – PPS CMOS”, 1994 IEEE Symposium on Low Power Electronics - Digest of Technical Papers (ISLPE'94), pages 98-99, San Diego, California, USA, 10th-12th October 1994.

[Gaba95] T. Gabara and W. Fischer, “An Integrated System Consisting of an 8×8 Adiabatic-PPS Multiplier Powered by a Tank Circuit”, 42nd IEEE International Solid State Circuits Conference – Digest of Technical Papers (ISSCC’95), pages 316-317 & 387, San Francisco, California, USA, 1995.

[Gere99] Sabih H. Gerez, “Algorithms for VLSI design automation”, John Wiley & Sons Ltd, Great Britain, 1999, ISBN: 9-780471-984894.

[Gurk00] F. K. Gurkaynak, Y. Leblebici, L. Chaouati and P. J. McGuinness, “High radix Kogge-Stone parallel prefix adder architectures”, Proceedings of the International Symposium on Circuits and Systems (ISCAS’00), volume 5, pages V:609-612, Geneva, Switzerland, 28th-31st May 2000.

[Hall92] J. S. Hall, “An Electroid Switching Model for Reversible Computer Architectures”, Proceedings of the 2nd Workshop on Physics and

Computation (PhysComp'92), pages 237-247, Dallas, Texas, USA, 2nd-4th October 1992.

[Hahm94] M. Hahm, "Modest power savings for applications dominated by switching of large capacitive loads", 1994 IEEE Symposium on Low Power Electronics – Digest of Technical Papers (ISLPE'94), pages 60-61, San Diego, California, USA, 10th-12th October 1994.

[Hand04] Handshake Solutions, URL:http://www.handshakesolutions.com/.

[Harr63] M. A. Harrison, "The number of Equivalence Classes of Boolean Functions Under Groups Containing Negation", IEEE Transactions on Electronic Computers, 12(5):559-561, May 1963.

[He06] Y. He, J. Tian, X. Tan and H. Min, "Quasi-static adiabatic logic 2N-2N2P2D family", Electronics Letters, 42(16):905-906, 3rd August 2006.

[Hell84] L. Heller, W. Griffin, J. Davis and N. Thoma, "Cascode voltage switch logic: a differential CMOS logic family", IEEE International Solid State Circuits Conference – Digest of Technical Papers (ISSCC'84), pages 16–17, San Francisco, California, USA, February 1984.

[Hinm93] R. T. Hinman and M. F. Schlecht, "Recovered Energy Logic - A Highly Efficient Alternative to Today's Logic Circuits", Record of the 24th Annual IEEE Power Electronics Specialists Conference (PESC'93), pages 17-26, Seattle, Washington, USA, 20th-24th June 1993.

[Hong01] Dai Hongyu, Zhou Runde and Ge Yuanqing, "High Efficient energy recovery logic circuit for adiabatic computing", Proceedings of the 4th International Conference on ASIC (ASIC'01), pages 858-861, Shanghai, China, 23rd-25th October 2001.

[Hu03] Hu Jianping, Cen Lizhang and Liu Xiao, "A new type of low-power adiabatic circuit with complementary pass-transistor logic", Proceedings of 5th International Conference on ASIC (ASIC'03), volume 2, pages II:1235-1238, Beijing, China, 21st-24th October 2003 .

[Hu04] Hu Jianping, Wu Yangbo and Zhang Weiquang, “Complementary Pass-Transistor Adiabatic Logic Circuit Using Three-Phase Power Supply”, Chinese Journal of Semiconductors, 25(8):918-924, August 2004.

[Icar02] Icarus Verilog website, URL: http://www.icarus.com/eda/verilog/

[Inde94] T. Indermaur and M. Horowitz, “Evaluation of charge recovery circuits and adiabatic switching for low power CMOS design”, 1994 IEEE Symposium on Low Power Electronics - Digest of Technical Papers (ISLPE'94), pages 102-103, San Diego, California, USA, 10th-12th October 1994.

[Inte07] Intel Corp. Inc., “Intel's Transistor Technology Breakthrough Represents Biggest Change to Computer Chips in 40 Years”, press release, 27th January 2007. Available via the following URL: http://www.intel.com/ .

[Jaek97] A. Jaekel, "A New Model for Constructing DCVS Trees", IEEE Canadian Conference on Electrical and Computer Engineering (CCECE’97), volume 2, pages 641-645, St. John's, Newfoundland and Labrador, Canada, 25th-28th May 1997.

[Karo95] T. Karoubalis, G. Ph. Alexiou and N. Kanopoulos, “Optimal synthesis of differential cascode voltage switch (DCVS) logic circuits using ordered binary decision diagrams (OBDDs)”, Proceedings of the European Design Automation Conference (Euro DAC’95), pages 282-287, Brighton, Great Britain, 18th-22nd September 1995.

[Kent98] “Chipwise” web site, Department of Electronics, University of Kent, URL:http://www.ee.kent.ac.uk/chipwise/.

[Kern02] P. Kerntopf, “Synthesis of multipurpose reversible logic gates”, Proceedings of the 2002 Euromicro Symposium on Digital System Design, pages 259-266, Dortmund, Germany, 4th-6th September 2002.

[Khaz05] A. Khazamipour and K. Radecka, “Adiabatic Implementation of Reversible Logic”, Proceedings of the 48th Mid-West Symposium on

Circuits and Systems (MWSCAS'05), pages 291-294, Cincinnati, Ohio, USA, 7th -10th August 2005.

[Khaz06] A. Khazamipour and K. Radecka, “A New Architectures of Adiabatic Reversible Logic Gates”, Proceedings of the 2006 IEEE North-East Workshop on Circuits and Systems, pages 233-236, Gatineau, Québec, Canada, 18th-21st June 2006.

[Kim98] S. Kim and M. C. Papaefthymiou, “True single-phase energy-recovering logic for low-power, high-speed VLSI”, Proceedings of the 1998 International Symposium on Low Powr Electronics and Design (ISLPED'98), pages 167-172, August 1998.

[Kim98] S. Kim and M. C. Papaefthymiou, “Single-phase Source-Coupled Adiabatic Logic”, Proceedings of the 1999 International Symposium on Low Powr Electronics and Design (ISLPED'99), pages 97-99, San Diego, California, USA, 1999.

[Kim00] C. Kim, S. M. Yoo and S. M. Kang, “Low-power adiabatic computing with NMOS energy recovery logic”, Electronics Letters, 36(16):1349-1350, 3rd August 2000.

[Kioi97] K. Kioi ,H. Kotaki, “Forward body-bias MOS (FBMOS) dual rail logic using an adiabatic charging technique with sub -0.6V operation”, Electronics Letters, 33(14):1200-1201, 3rd July 1997.

[Know99] S. Knowles, “A Family of Adders”, Proceedings of the 14th IEEE Symposium on Computer Arithmetic, pages 30-34, Adelaide, Australia, 14th-16th April 1999.

[Kogg73] P. M. Kogge and H. S. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence equations”, IEEE Transactions on Computers, 22(8):786-793, August 1973.

[Koll92] J. G. Koller and W. C. Athas, “Adiabatic Switching, Low Energy Computing, and the Physics of Storing and Erasing Information”, Proceedings of the 2nd Workshop on Physics and Computation (PhysComp’92), pages 267-270, Dallas, Texas, USA, 2nd-4th October 1992.

[Kong96a] Bai-Sun Kong, Joo-Sun Choi, Seog-Jun Lee and Kwyro Lee, “Charge Recycling Differential Logic for Low-Power Application”, 43rd IEEE International Solid State Circuits Conference – Digest of Technical Papers (ISSCC’96), pages 302-303 and 462, San Francisco, California, USA, 8th-10th February 1996.

[Kong96b] Bai-Sun Kong, Joo-Sun Choi, Seog-Jun Lee and Kwyro Lee, “Charge Recycling Differential Logic (CRDL) for Low-Power Application”, IEEE Journal of Solid-State Circuits, 31(9):1267-1276, September 1996.

[Kong97] Bai-Sun Kong, Young-Hyun Jun and Kwyro Lee, “A true single-phase clocking scheme for low-power and high-speed VLSI”, Proceedings of 1997 IEEE International Symposium on Circuits and Systems (ISCAS'97), volume 3, pages III:1904-1907, Hong Kong, 9th-12th June 1997.

[Kram94] A. Kramer, J. S. Denker, S. C. Avery, A. G. Dickinson and T. R. Wik, “Adiabatic Computing with the 2n-2n2d Logic Family”, 1994 Symposium on VLSI Circuits – Digest of Technical Papers, pages 25-26, Honolulu, Hawaii, USA, 9th-11th June 1994.

[Kram95] A. Kramer, J. S. Denker, B. Flower and J. Moroney, “2nd Order adiabatic computation with 2n-2p and 2n-2n2p logic circuits”, Proceedings of IEEE Symposium Low Power Design (ISLPD’95), pages 191-196, Dana Point, California, USA, 23rd-26th April 1995.

[Kwon98] K. Kwon and S.-I. Chae, “Simple reversible energy recovery logic using NMOS switch networks with cross-coupled PMOS pair”, Electronics Letters, 34(23):2215-2216, 12th November 1998.

[Land61] R. Landauer, “Irreversibility and Heat Generation in the computing process“, IBM Journal of Research and Development, volume 5, pages 183-191, July 1961.

[Lau96] K. T. Lau and W. Y. Wang, “Transmission gate-interfaced APDL design”, Electronics Letters, 32(4):317-318, 15th February 1996.

[Lau97] K. T. Lau and F. Liu, “Improved adiabatic pseudo-domino logic family”, Electronics Letters, 33(25):2113-2114, 4th December 1997.

[Lee59] C. Y. Lee, “Representation of Switching Circuits by Binary-Decision Programs”, Bell System Technical Journal, volume 38, pages 985-999, July 1959.

[Leff03] Harvey S. Leff and Andrew F. Rex (editors), “Maxwell’s Demon 2”, IoP Publishing, London, 2003. ISBN13: 9-780750-307598.

[Lega01] J.-D. Legat, “Optimisation de la logique différentielle adiabatique ADCVSL par utilisation du DTMOS”, Journées francophones d’études Faible Tension Faible Consommation (FTFC’2001), Paris, 30th-31st May 2001, pages 57-62. Title approximately translates to “Optimisation of ADCVSL differential adiabatic logic using DTMOS”.

[Lega03] J.-D. Legat, “Comparaison des logiques différentielles à faible consommation et à amplitude réduite”, Journées francophones d’études Faible Tension Faible Consommation (FTFC’2003), Paris, 15th-16th May 2003, pages 69-74. Title approximately translates to “Comparison of differential logics at low power and reduced amplitude”.

[Leun03] P.-K. Leung, C.-S. Choy, C.-F. Chan and K.-P. Pun, “A low power asynchronous GF(2^{173}) ALU for elliptic curve crypto-processor”, Proceedings of the International Symposium on Circuits and Systems (ISCAS'03), volume 5, pages V:337-340, Bangkok, Thailand, 25th-28th May 2003.

[Li00] Li Xiao-min, Qiu Yu-lin and Chen Chao-shu, “A Type of Bootstrapped Charge-Recovery Logic Circuit”, Chinese Journal of Semiconductors, 21(9):887-891, September 2000. Article is in Chinese as "一□ 利用自□ 效□ 的Charge-Recovery □ □ □ 路".

[Li01] Li Xiao-min, Qiu Yu-lin and Chen Chao-shu, “Design of Low Voltage Charge-Recovery Logic Circuit”, Chinese Journal of Semiconductors, 22(10):1352-1356, October 2001. Article is in Chinese as "低□ □ Charge-Recovery □ □ □ 路的□ □ ".

[Li07] Shun Li, Feng Zhou, Chunhong Chen, Hua Chen and Yipin Wu; “Quasi-Static Energy Recovery Logic with Single Power-Clock Supply”, Proceedings of International Symposium on Circuits and

Systems (ISCAS 2007), pages 2124-2127, New Orleans, Louisiana, USA, 27th-30th May 2007.

[Lim98] Joonho Lim, Kipaek Kwon and Soo-Ik Chae, “Reversible energy recovery logic circuit without non-adiabatic energy loss”, Electronics Letters, 34(4):344-346, 19th February 1998.

[Liu98a] F. Liu and K. T. Lau, “Pass-transistor adiabatic logic with NMOS pull-down configuration”, Electronics Letters, 34(8):739-741, 16th April 1998.

[Liu98b] F. Liu and K. T. Lau, “Improved structure for efficient charge recovery logic”, Electronics Letters, 34(18):1731-1732, 3rd September 1998.

[Lo98] Chun-Keung Lo and Philip C. H. Chan, “Design of Low Power Differential Logic Using Adiabatic Switching Technique”, Proceedings of the International Symposium on Circuits and Systems (ISCAS'98), volume 2, pages II:33-36, Monterey, California, USA, 31st May - 3rd June 1998.

[Maks95] D. Maksimović and V. G. Oklobdžija, “Integrated Power Clock Generators for Low Energy Logic”, Record of the 26th Annual IEEE Power Electronics Specialists Conference (PESC’95), volume 1, pages I:61-67, Atlanta, Georgia, USA, 18th -22nd June 1995.

[Maks97a] D. Maksimović, V. G. Oklobdžija, B. Nikolic and K. W. Current, “Design and Experimental Verification of a CMOS Adiabatic Logic with Single-Phase Power-Clock supply”, Proceedings of the 40th Midwest Symposium on Circuits and Systems (MWCAS'97), volume 1, pages I:417-420, Sacramento, California, USA, 3rd-6th August 1997.

[Maks97b] D. Maksimović, V. G. Oklobdžija, B. Nikolic and K. W. Current, “Clocked CMOS Adiabatic Logic with Integrated Single-Phase Power-Clock Supply: Experimental Results”, Proceedings of the 1997 International Symposium on Low Power Electronics and Design (ISLPED’97), pages 323-327, Monterey, California, USA, 18th-20th August 1997.

[Mate96] D. Mateo and A. Rubio, “Quasi-adiabatic ternary CMOS logic”, Electronics Letters, 32(2):317-318, 18th January 1996.

[Mate97] D. Mateo and A. Rubio, “Implementation of a 5×5 trits multiplier in quasi-adiabatic ternary CMOS logic”,Proceedings of the 23rd European Solid-State Circuits Conference (ESSCIRC'97), pages 120-123, Southampton, Great Britain, 16th -18th September 1997.

[Maxw71] James Clerk Maxwell, ”Theory of Heat”, pages 308-309, London, 1871.

[MIT04] Massachusetts Institute of Technology, “6.111 Introductory Digital Systems Laboratory; Tutorial problems; L02: Logic Gates; Problem 6”, Fall (autumn) 2004, URL: http://web.mit.edu/6.111/www/f2004/

[Moon95] Yong Moon and Deog-Kyoon Jeong, “Efficient Charge Recovery Logic”, 1995 Symposium on VLSI Circuits – Digest of Technical Papers, pages 129-130, Kyoto, Japan, 8th-10th June 1995.

[Moon96] Yong Moon and Deog-Kyoon Jeong, “An efficient charge recovery logic circuit”, IEEE Journal of Solid-State Circuits, 31(4):514-522, April 1996.

[Moon98] Yong Moon and Deog-Kyoon Jeong, “A 32×32-b adiabatic register file with supply clock generator”, IEEE Journal of Solid-State Circuits, 33(5):696-701, May 1998.

[Mull59] D. E. Muller and W. S. Bartky, “A theory of asynchronous circuits”, Proceedings of International Symposium Theory of Switching, pages 204-243, 1959.

[Naka00] Shunji Nakata, Takakuni Douseki, Yuichi Kado and Junzo Yamada, “A Low Power Multiplier Using Adiabatic Charging Binary Decision Diagram Circuit”, Japanese Journal of Applied Physics, 39(4B):2305-2311, part 1, April 2000.

[Naka04] Shunji Nakata, “Adiabatic Charging Reversible Logic Using a Switched Capacitor Regenerator”, IEICE Transactions on Electronics, E87-C(11):1837-1846, November 2004.

[OEIS02] Online Encyclopaedia of Integer Sequences, ID: A052129, URL:http://www.research.att.com/~njas/sequences/A052129/, accessed 2009-07-03.

[OEIS06] Online Encyclopaedia of Integer Sequences, ID: A123851, URL:http://www.research.att.com/~njas/sequences/A123851/, accessed 2009-08-13.

[OEIS09a] Online Encyclopaedia of Integer Sequences, ID: A164334, URL:http://www.research.att.com/~njas/sequences/A164334/, accessed 2009-08-20.

[OEIS09b] Online Encyclopaedia of Integer Sequences, ID: A164335, URL:http://www.research.att.com/~njas/sequences/A164335/, accessed 2009-08-20.

[Oklo97] V. G. Oklobdžija, D. Maksimović and Fengcheng Lin, "Pass-Transistor Adiabatic Logic Using Single Power-Clock Supply", IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, 44(10):842-846, October 1997.

[Pave94] Nigel C. Paver, "The Design and Implementation of an Asynchronous Microprocessor", Ph.D. Thesis, University of Manchester, UK, 1994.

[Pisa02] Leonardo Pisano (also known as Leonardo Fibonacci), "*Il Liber Abbaci*", Italy, 1202.

[Qian04] Yang Qian and Zhou Runde, "ERCCL: Energy Recovery Capacitance Coupling Logic", Proceedings of the 7th International Conference on Solid-State and Integrated Circuits Technology (ICSICT'04), volume 3, pages III:2090-2093, Beijing, China, 18th -21st October 2004.

[Rabi03] S. Rabii and S. Limotyrakis, "Measuring Average Power Dissipation in SPICE", PDF document "power_meas_v2.pdf" downloaded 13th January 2003.

[Schn98] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall and N. Ferguson, "Twofish: A 128-Bit Block Cipher", Published 15th June 1998, Presented at the First Advanced Encryption Standard

Candidate Conference, Ventura, California, USA, 20th-22nd August 1998.

[Seit85] C. L. Seitz, A. H. Frey, S. Mattisson, S. D. Rabin, and J. L. A. Van-de-Snepscheut, “Hot-clock nMOS”, in H. Fuchs (editor), 1985 Chappel Hill Conference on VLSI, Computer Science Press, pages 1-17, 1985.

[Sham96] M. Shams, J. C. Ebergen and M. I. Elmasry, “A comparison of CMOS implementations of an asynchronous circuits primitive: the C-element”, Proceedings of the 1996 International Symposium on Low Power Electronics and Design (ISLPED’96), pages 93-96, Monterey, California, USA, 12th-14th August 1996.

[Shin03] Youngjoon Shin, Hanseung Lee, Yong Moon and Chanho Lee, “A design of 16-bit adiabatic Microprocessor core”, Journal of Semiconductor Technology and Science, 3(4):194-198, December 2003.

[Shin04] Youngjoon Shin, Chanho Lee and Yong Moon, “A Low Power 16-bit RISC Microprocessor Using ECRL Circuits”, ETRI Journal, 26(6):513-519, December 2004.

[Skla60] J. Sklansky, “Conditional Sum Addition Logic”, IRE Transactions on Electronic Computers, EC9(6):226-231, June 1960.

[Song04] Hee-sup Song and Jin-ku Kang, “A CMOS Adiabatic Logic for Low Power Circuit Design”, Proceedings of 2004 IEEE Asia-Pacific Conference on Advanced System Integrated Circuits (AP-ASIC 2004), pages 348-351, Fukuoka, Japan, 4th-5th August 2004.

[Spar01] Jens Sparsø and Steve Furber (editors.), “Principles of Asynchronous Circuit Design: A Systems Perspective”, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001. ISBN: 0-7923-7613-7. ISBN13: 9780792-376132.

[Star02] V. I. Starosel’skii, “Adiabatic Logic Circuits: A Review”, Russian Microelectronics, 31(1):37-58, 2002. Also with Russian citation of: В. И. Старосельский, “Адиабатическая логика (Обзор)”, Микроэлектроника, 31(1) 2002.

[Suva00] D. Suvakovic and C. Salama, “Two phase non-overlapping clock adiabatic differential cascode voltage switch logic”, 2000 IEEE International Solid-State Circuits Conference – Digest of Technical Papers (ISSCC'2000), pages 364-365, San Francisco, California, USA, 7th-9th February 2000.

[Sven94a] L. “J.” Svensson and J. G. Koller, “Adiabatic charging without inductors”, Report ACMOS-TR-3a, Information Sciences Institute, University of Southern California, USA, 8th February 1994.

[Sven94b] L. “J.” Svensson and J. G. Koller, “Driving a capacitive load without dissipating fCV^2”, 1994 IEEE Symposium on Low Power Electronics – Digest of Technical Papers (ISLPE’94), pages 100-101, San Diego, California, USA, 10th-12th October1994.

[Sven95] L. “J.” Svensson, W. C. Athas and J. G. Koller, “System and method for power-efficient charging and discharging of a capacitive load form a single source”, US Patent 5,473,526, 5th December 1995.

[Sven96] L. “J.” Svensson, W. C. Athas, R. S.-C. Wen, “A sub-CV^2 pad driver with 10ns transition time”, Proceedings of IEEE Symposium on Low Power Electronics and Design (ISLPED'96), pages 105-108, Monterey, California, USA, 12th-14th August 1996.

[Suth89] I. E. Sutherland, “Micropipelines”, Communications of ACM 32(6):720-738, January 1989.

[Taka00] Kazukiyo Takahashi and Mitsuru Mizunuma, “Adiabatic Dynamic CMOS Logic Circuit”, Electronics and Communications in Japan, part 2, 83(5):50-58, May 2000.

[Teic07] P. Teichmann, J. Fischer, F. R. Chouard and D. Schmitt-Landsiedel, “Design of Ultra-Low-Power Arithmetic Structures in Adiabatic Logic”, Proceedings of 2007 IEEE International Symposium on Integrated Circuits (ISIC'07), pages 365-368, Singapore, 26th-28th September 2007.

[Thap06] Himanshu Thapliyal and M. Zwolinski, “Reversible Logic to Cryptographic Hardware: A New Paradigm”, Proceedings of the 49th

IEEE International Midwest Symposium on Circuits and Systems (MWCAS'06), volume 1, pages I:342-346, San Juan, Puerto Rico, 6th-9th August 2006.

[Tzar97] N. Tzartzanis and W. C. Athas, “Clock-Powered Logic for a 50MHz Low-Power RISC Datapath”, ISSCC'97), pages 338-339 & 482, 1997.

[Tzar99] N. Tzartzanis and W. C. Athas, “Retractile Clock-Powered Logic”, ISLPED'99), pages 18-23, 1999

[Varg01a] L. Varga, F. Kovács and G. Hosszú, “An Efficient Adiabatic Charge-Recovery Logic”, Proceedings of the IEEE SouthEastCon 2001, pages 17-20, Clemson, South Carolina, USA, 30th March-1st April 2001.

[Varg01b] L. Varga, F. Kovács and G. Hosszú, “An Improved Pass-Gate Adiabatic Logic”, Proceedings 14th Annual International ASIC/SOC Conference, pages 208-211, Arlington, Virginia, USA, 12th-15th September 2001.

[VBer94] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij and A. Peeters, “Asynchronous circuits for low power: a DCC error corrector”, IEEE Design and Test of Computers, 11(2):22-32, Summer 1994.

[Vetu96] A. Vetuli, S. D. Pascoli and L. M. Reyneri, “Positive feedback in adiabatic logic”, Electronics Letters, 32(20):1867-1869, 26th September 1996.

[Vija07a] Ponnusamy Vijayakumar, M. Shanthanalakshmi and Kandasamy Gunavathi, “Optimizing CMOS Circuits for Performance Improvements Using Adiabatic Logic“, Information Technology Journal, 6(3):325-331, 2007.

[Vija07b] Ponnusamy Vijayakumar and Kandasamy Gunavathi, “Energy Efficient Charge Recovery for Positive Feedback Adiabatic Logic”, IETE Technical Review, 24(2):127-133, March-April 2007.

[Wang95] W. Y. Wang and K. T. Lau, “Adiabatic pseudo-domino logic”, Electronics Letters, 31(23):1982-1983, 9th November 1995.

[Weis93] M. Weiser, “Hot topics-ubiquitous computing”, IEEE Computer, 26(10):71-72, October 1993.

[West94] Neil H. E. Weste and Kamran Eshraghian, “Principles of CMOS VLSI Design; A Systems Perspective”, 2nd Edition, Addison-Wesley, USA, 1994, ISBN 0-201-53376-6, ISBN13 9-780201-533767.

[Widj03] B. W. Widjaja and K. T. Lau, “Improved Adiabatic Pseudo Domino Logic 2 (IAPDL-2), Electronics Letters, 39(16):1167-1169, 7th August 2003.

[Will99] David J. Willingham, “Adiabatic CMOS 8×8 multiplier”, MSc project report, University of Westminster, London, 1999.

[Will04] David J. Willingham and İzzet Kale, “Asynchronous, quasi-Adiabatic (*Asynchrobatic*) logic for low power very wide data width applications”, Proceedings of International Symposium on Circuits and Systems (ISCAS 2004), volume 2, pages II:257-260, Vancouver, Canada, 23rd-26th May 2004.

[Will05] David J. Willingham, “Variations in Power Consumption of Adiabatic Logic Families Depending Upon the Charging Scheme Used”, 2005, unpublished.

[Will08a] David J. Willingham and İzzet Kale, “An *Asynchrobatic*, Radix-four, Carry Look-ahead Adder”, Proceedings of PhD Research in Microelectronics and Electronics (PRIME 2008), pages 105-108, İstanbul, Turkey, 22nd-25th June 2008.

[Will08b] David J. Willingham and İzzet Kale, “Using Positive Feedback Adiabatic Logic to implement Reversible Toffoli Gates”, Proceedings of Norchip 2008, pages 5-8, Tallinn, Estonia, 17th-18th November 2008.

[Will08c] David J. Willingham and İzzet Kale, “A system for calculating the Greatest Common Denominator implemented using *Asynchrobatic* Logic”, Proceedings of Norchip 2008, pages 194-197, Tallinn, Estonia, 17th-18th November 2008.

[Won98] Jae-Hee Won and Kiyoung Choi, “Modified Half Rail Differential Logic for Reduced Internal Logic Swing”, Proceedings of the

International Symposium on Circuits and Systems (ISCAS'98), volume 2, pages II:157-160, 1998.

[Ye97] Y. Ye, K. Roy and G. I. Stamoulis, “Quasi-Static Energy Recovery Logic and Supply-Clock Generation Circuits”, Proceedings of the 1997 International Symposium on Low Power Electronics and Design (ISLPED’97), pages 96-99, Monterey, California, USA, 18th-20th August 1997.

[Yeh97] C. C. Yeh, J. H. Lou and J. B. Kuo, “1.5V CMOS full-swing energy efficient logic (EEL) circuit suitable for low-voltage and low-power VLSI applications”, Electronics Letters, 33(16):1375-1376, 31st July 1997.

[Yoo98] Seung-Moon Yoo and Sung-Mo (Steve) Kang, “A Bootstrapped NMOS Charge Recovery Logic”, Proceedings of the 8th Great Lakes Symposium on VLSI (GLS-VLSI’98), pages 30-33, Lafayette, Louisiana, USA, 19th-21st February 1998.

[Yoo99a] Seung-Moon Yoo and Sung-Mo (Steve) Kang, “CMOS Pass-gate No-race Charge-recycling Logic (CPNCL)”, Proceedings of International Symposium on Circuits and Systems (ISCAS'99), volume 1, pages I:226-229, Orlando, Florida, USA, 30th May-2nd June 1999.

[Yoo99b] Seung-Moon Yoo and Sung-Mo (Steve) Kang, “No-race charge-recycling differential logic (NCDL)”, Proceedings of the 9th Great Lakes Symposium on VLSI (GLS-VLSI'99), pages 202-205, Ypsilanti, Michigan, USA, 4th-6th March 1999.

[Youn93] S. G. Younis and T. F. Knight Jnr., “Practical Implementation of Charge Recovering Asymptotically Zero Power CMOS”, Proceedings of the 1993 Symposium on Research on Integrated Systems, Seattle, USA, pages 234-250, 1993.

[Youn94a] Saed G. Younis and T. F. Knight Jnr., “Asymptotically Zero Energy Computing Split-Level Charge Recovery Logic”, International Workshop on Low Power Design, pages 177-182, Napa, California, USA, 24th-27th April 1994.

[Youn94b] Saed G. Younis, “Asymptotically Zero Energy Computing Using Split-Level Charge Recovery Logic”, Ph.D. Thesis, Massachusetts Institute of Technology, June 1994.

[Zieg04] M. M. Ziegler and M. R. Stan, “A Unified Design Space for Regular Parallel Prefix Adders”, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 2004 (DATE'04), volume 2, pages II:1386-1387, Paris, France, 16th-20th February 2004.

10.2 Bibliography

Behind each of these books, there's a man. That's what interests me.
Ray Bradbury, Fahrenheit 451.

This sections lists unreferenced books that may provide useful background reading.

- Nazeih M. Botros, “HDL Programming Fundamentals; VHDL and Verilog”, Charles River Media, Massachusetts, USA, 2006, ISBN: 9-781584-508557.
- Anantha P. Chandrakasan and Robert W. Brodersen, “Low Power Digital CMOS Design”, Kluwer Academic Publishers, Massachusetts, USA, 1995, ISBN: 9-780792-395768, Chapter 6 (Lars Svensson, “Adiabatic Switching”), pages 181-218.
- Anthony J.G. Hey (editor), “Feynman and Computation”, Westview Press, USA, ISBN: 978-08133-4039-5.
- Hubert Kaeslin, “Digital Integrated Circuit Design; From VLSI Architectures to CMOS Fabrication”, Cambridge University Press, 2008, ISBN: 978-0-521-88267-5.
- Carver Mead and Lynn Conway, “Introduction to VLSI Systems”, Addison-Wesley, USA, 1980, ISBN: 9-780201-043587.
- Jan M. Rabaey and Massoud Pedram, “Low Power Design Methodologies”, Kluwer Academic Publishers, Massachusetts, USA, 1996, ISBN: 9-780792-396307, Chapter 4 (William C. Athas, “Energy-Recovery CMOS”), pages 65-100.
- Kaushik Roy and Sharat C. Prasad, “Low-Power CMOS VLSI Circuit Design”, John Wiley & Sons, Inc., USA, 2000, ISBN: 9-780471-114888, Chapter 7, pages 272-320.
- Michael J. S. Smith, “Application-Specific Integrated Circuits”, Addison Wesley Longman, USA, 1997, ISBN: 9-780201-500226.

Appendices

Appendix A Verilog source-code

A.1 Single-rail GCD

```
`timescale 1ns / 100ps
`define RESET_DELAY 1.5
// The assignment delay following pseudo-power-clock
`define DP_RESET_VAL 1'bz
// The normal reset value applied to data-path cells
`define STAGE_DELAY 2
// The assignment delay following pseudo-power-clock
`define NEG_EDGE 1
// Are flop outputs reset on falling edge?
`define NEG_EDGE_VAL 1'bz
// The normal value assigned upon the falling pseudo-power-clock edge
`define LOGIC_DELAY 2.5
// The assignment delay following pseudo-power-clock

// Static CMOS cell names use ALL CAPS
// Adiabatic cell names use lower case

module C_ELE2R0 (A, B, Z, rst0);
// 2-input C-Element with reset to 0
input  A, B, rst0;
output Z;
reg    Z;

always @(A or B or rst0)
  if (~rst0)
    #`RESET_DELAY Z <= 0;
  else if (A == B)
    #`LOGIC_DELAY Z <= A;
endmodule

module C_ELE2R1 (A, B, Z, rst0);
// 2-input C-Element with reset to 1
input  A, B, rst0;
output Z;
reg    Z;

always @(A or B or rst0)
  if (~rst0)
    #`RESET_DELAY Z <= 1;
  else if (A == B)
    #`LOGIC_DELAY Z <= A;
endmodule

module C_ELE3R0 (A, B, C, Z, rst0);
// 3-input C-Element with reset to 0
input  A, B, C, rst0;
output Z;
reg    Z;
```

```
always @(A or B or C or rst0)
  if (~rst0)
    #`RESET_DELAY Z <= 0;
  else if ((A == B) & (B == C))
    #`LOGIC_DELAY Z <= A;
endmodule

module BUF1 (A, Z);
// Buffer
input  A;
output Z;
reg    Z;

always @(A)
  #`LOGIC_DELAY Z <= A;
endmodule

module INV1 (A, Z);
// Inverter
input  A;
output Z;
reg    Z;

always @(A)
  #`LOGIC_DELAY Z <= ~A;
endmodule

module OR2 (A, B, Z);
// 2-input OR gate
input  A, B;
output Z;
reg    Z;

always @(A or B)
  #`LOGIC_DELAY Z <= A | B;
endmodule

module AN2 (A, B, Z);
// 2-input AND gate
input  A, B;
output Z;
reg    Z;

always @(A or B)
  #`LOGIC_DELAY Z <= A & B;
endmodule

module MUX2 (S0req, S0ack, S1req, S1ack,
             CT0req, CT1req, Ctack, Zreq, Zack, rst0);
// 2-input MUX for Asynchronous Controller
input  S0req, S1req, CT0req, CT1req, Zack;
input  rst0;
output S0ack, S1ack, CTack, Zreq;
wire   S0CT0req, S1CT1req, ZreqI;

C_ELE2R0 cel1 (S0req, CT0req, S0CT0req, rst0);
C_ELE2R0 cel2 (S1req, CT1req, S1CT1req, rst0);
C_ELE2R0 cel3 (Zack, S0CT0req, S0ack, rst0);
```

```
C_ELE2R0 cel4 (Zack, S1CT1req, S1ack, rst0);
OR2      or2  (S0CT0req, S1CT1req, Zreq);
BUF1     buf1 (Zack, CTack);
endmodule

module DMX2 (S0req, S0ack, S1req, S1ack,
             CT0req, CT1req, Ctack, Ireq, Iack, rst0);
// 2-output DeMUX for Asynchronous Controller
input  S0ack, S1ack, CT0req, CT1req, Ireq;
input  rst0;
output S0req, S1req, CTack, Iack;
wire   S0CT0req, S1CT1req;

C_ELE2R0 cel1 (Ireq, CT0req, S0req, rst0);
C_ELE2R0 cel2 (Ireq, CT1req, S1req, rst0);
OR2      or2  (S0ack, S1ack, Iack);
BUF1     buf1 (Iack, CTack);
endmodule

module PIPELINE_ELER0 (Ireq, Iack, Zreq, Zack, aswc, rst0);
// Asynchronous pipeline element
input  Ireq, Zack;
input  rst0;
output Zreq, Iack;
output aswc;
wire   temp;

INV1     inv1 (Zack, Zack_n);
C_ELE2R0 cel1 (Ireq, Zack_n, aswc,rst0);
BUF1     buf1 (aswc, temp);
BUF1     buf2 (temp, Zreq);
BUF1     buf3 (temp, Iack);
endmodule

module PIPELINE_ELER1 (Ireq, Iack, Zreq, Zack, aswc, rst0);
// Asynchronous pipeline element with reset to active
//(for initial tokens)
input  Ireq, Zack;
input  rst0;
output Zreq, Iack;
output aswc;
wire   temp;

INV1     inv1 (Zack, Zack_n);
C_ELE2R1 cel1 (Ireq, Zack_n, aswc,rst0);
BUF1     buf1 (aswc, temp);
BUF1     buf2 (temp, Zreq);
BUF1     buf3 (temp, Iack);
endmodule

// Single Rail models of PFAL Cells
// Can be expanded to Dual Rail
// using regular expression substitution
// CMOS models can be created using
// rpmos and nmos primitives
// aswc is the pseudo-power-clock
// rst0 is reset asserted low (for simulation purposes)
```

```
// Basic cells

module buf1 (A, Z, aswc, rst0);
// Buffer
input  A;
input  aswc, rst0;
output Z;
reg    Z;

always @(posedge aswc or negedge rst0)
  if (~rst0)
    #`RESET_DELAY Z <= `DP_RESET_VAL;
  else
    #`STAGE_DELAY Z <= A;

always @(negedge aswc)
  if (`NEG_EDGE)
    #`STAGE_DELAY Z <= `NEG_EDGE_VAL;
endmodule

module buf1r1 (A, Z, aswc, rst0);
// Buffer with reset
input  A;
input  aswc, rst0;
output Z;
reg    Z;

always @(posedge aswc or negedge rst0)
  if (~rst0)
    #`RESET_DELAY Z <= 1;
  else
    #`STAGE_DELAY Z <= A;

always @(negedge aswc)
  if (`NEG_EDGE)
    #`STAGE_DELAY Z <= `NEG_EDGE_VAL;
endmodule

module xor2 (A, B, Z, aswc, rst0);
// 2-input XOR
input  A, B;
input  aswc, rst0;
output Z;
reg    Z;

always @(posedge aswc or negedge rst0)
  if (~rst0)
    #`RESET_DELAY Z <= `DP_RESET_VAL;
  else
    #`STAGE_DELAY Z <= A ^ B;

always @(negedge aswc)
  if (`NEG_EDGE)
    #`STAGE_DELAY Z <= `NEG_EDGE_VAL;
endmodule
```

```
module xnor2 (A, B, Z, aswc, rst0);
// 2-input XNOR
input  A, B;
input  aswc, rst0;
output Z;
reg    Z;

always @(posedge aswc or negedge rst0)
  if (~rst0)
    #`RESET_DELAY Z <= `DP_RESET_VAL;
  else
    #`STAGE_DELAY Z <= ~(A ^ B);

always @(negedge aswc)
  if (`NEG_EDGE)
    #`STAGE_DELAY Z <= `NEG_EDGE_VAL;
endmodule

module and2 (A, B, Z, aswc, rst0);
// 2-input AND
input  A, B;
input  aswc, rst0;
output Z;
reg    Z;

always @(posedge aswc or negedge rst0)
  if (~rst0)
    #`RESET_DELAY Z <= `DP_RESET_VAL;
  else
    #`STAGE_DELAY Z <= A & B;

always @(negedge aswc)
  if (`NEG_EDGE)
    #`STAGE_DELAY Z <= `NEG_EDGE_VAL;
endmodule

module or2 (A, B, Z, aswc, rst0);
// 2-input OR
input  A, B;
input  aswc, rst0;
output Z;
reg    Z;

always @(posedge aswc or negedge rst0)
  if (~rst0)
    #`RESET_DELAY Z <= `DP_RESET_VAL;
  else
    #`STAGE_DELAY Z <= A | B;

always @(negedge aswc)
  if (`NEG_EDGE)
    #`STAGE_DELAY Z <= `NEG_EDGE_VAL;
endmodule
```

```
module mux2 (A, B, S, Z, aswc, rst0);
// 2-way MUX
input  A, B, S;
input  aswc, rst0;
output Z;
reg    Z;

always @(posedge aswc or negedge rst0)
  if (~rst0)
    #`RESET_DELAY Z <= `DP_RESET_VAL;
  else
    #`STAGE_DELAY Z <= S ? B : A;

always @(negedge aswc)
  if (`NEG_EDGE)
    #`STAGE_DELAY Z <= `NEG_EDGE_VAL;
endmodule

module and3 (A, B, C, Z, aswc, rst0);
// 3-input AND
input  A, B, C;
input  aswc, rst0;
output Z;
reg    Z;

always @(posedge aswc or negedge rst0)
  if (~rst0)
    #`RESET_DELAY Z <= `DP_RESET_VAL;
  else
    #`STAGE_DELAY Z <= A & B & C;

always @(negedge aswc)
  if (`NEG_EDGE)
    #`STAGE_DELAY Z <= `NEG_EDGE_VAL;
endmodule

module and4 (A, B, C, D, Z, aswc, rst0);
// 4-input AND
input  A, B, C, D;
input  aswc, rst0;
output Z;
reg    Z;

always @(posedge aswc or negedge rst0)
if (~rst0)
  #`RESET_DELAY Z <= `DP_RESET_VAL;
else
  #`STAGE_DELAY Z <= A & B & C & D;

always @(negedge aswc)
if (`NEG_EDGE)
  #`STAGE_DELAY Z <= `NEG_EDGE_VAL;
endmodule
```

```
module gpp2 (G1, G0, P1, Gp, aswc, rst0);
// 3-input AND2-OR
input  G1, G0, P1;
input  aswc, rst0;
output Gp;
reg    Gp;

always @(posedge aswc or negedge rst0)
if (~rst0)
  #`RESET_DELAY Gp <= `DP_RESET_VAL;
else
  #`STAGE_DELAY Gp <= G1 |(P1 & G0);

always @(negedge aswc)
if (`NEG_EDGE)
  #`STAGE_DELAY Gp <= `NEG_EDGE_VAL;
endmodule

module gpp3 (G2, G1, G0, P2, P1, Gp, aswc, rst0);
// 5-input AND2-OR-AND-OR
input  G2, G1, G0, P2, P1;
input  aswc, rst0;
output Gp;
reg    Gp;

always @(posedge aswc or negedge rst0)
if (~rst0)
  #`RESET_DELAY Gp <= `DP_RESET_VAL;
else
  #`STAGE_DELAY Gp <= G2 |(P2 &(G1 |(P1 & G0)));

always @(negedge aswc)
if (`NEG_EDGE)
  #`STAGE_DELAY Gp <= `NEG_EDGE_VAL;
endmodule

module gpp4 (G3, G2, G1, G0, P3, P2, P1, Gp, aswc, rst0);
// 7-input AND2-OR-AND-OR-AND-OR
input  G3, G2, G1, G0, P3, P2, P1;
input  aswc, rst0;
output Gp;
reg    Gp;

always @(posedge aswc or negedge rst0)
if (~rst0)
  #`RESET_DELAY Gp <= `DP_RESET_VAL;
else
  #`STAGE_DELAY Gp <= G3 |(P3 &(G2 |(P2 &(G1 |(P1 & G0)))));

always @(negedge aswc)
  if (`NEG_EDGE)
    #`STAGE_DELAY Gp <= `NEG_EDGE_VAL;
endmodule

/*
        The structural components of the subtractor
*/
```

```
module HALF_ADDER(A, B, P0, G0, aswc, rst0);
// Half Adder to provide "Generate" and "Propagate" signals
input  A, B;
input  aswc, rst0;
output G0, P0;

xor2 Pro (A, B, P0, aswc, rst0);
and2 Gen (A, B, G0, aswc, rst0);
endmodule

module CALF_ADDER(A, B, P0, G0, aswc, rst0);
// Carry-set Half Adder - Used for LSB of Subtractor
input  A, B;
input  aswc, rst0;
output G0, P0;

xnor2 Pro (A, B, P0, aswc, rst0);
or2   Gen (A, B, G0, aswc, rst0);
endmodule

module PP1 (Q_in, G_in, Q_ot, G_ot, aswc, rst0);
// Level 1 Propagate/Generate Look-Ahead logic
input  G_in, Q_in;
input  aswc, rst0;
output G_ot, Q_ot;

buf1 Pr0 (Q_in, Q_ot, aswc, rst0);
buf1 Gen (G_in, G_ot, aswc, rst0);
endmodule

module PP2 (Q_in, P1_in, P0_in, G1_in, G0_in,
            Q_ot, P_ot, G_ot, aswc, rst0);
// Level 2 Propagate/Generate Look-Ahead logic
// Includes Buffer for initial propagate
input  Q_in;
input  G1_in, G0_in;
input  P1_in, P0_in;
input  aswc, rst0;
output Q_ot, P_ot, G_ot;

buf1 Pr0 (Q_in,                Q_ot, aswc, rst0);
and2 Pro (P1_in, P0_in,        P_ot, aswc, rst0);
gpp2 Gen (G1_in, G0_in, P1_in,G_ot, aswc, rst0);
endmodule

module PP3 (Q_in, P2_in, P1_in, P0_in, G2_in, G1_in, G0_in,
            Q_ot, P_ot, G_ot, aswc, rst0);
// Level 3 Propagate/Generate Look-Ahead logic
// Includes Buffer for initial propagate
input  Q_in;
input  G2_in, G1_in, G0_in;
input  P2_in, P1_in, P0_in;
input  aswc, rst0;
output Q_ot, P_ot, G_ot;

buf1 Pr0 (Q_in,                                Q_ot, aswc, rst0);
and3 Pro (P2_in, P1_in, P0_in,                 P_ot, aswc, rst0);
gpp3 Gen (G2_in, G1_in, G0_in, P2_in, P1_in, G_ot, aswc, rst0);
```

```
endmodule
module PP4 (Q_in, P3_in, P2_in, P1_in, P0_in,
            G3_in, G2_in, G1_in, G0_in,
            Q_ot, P_ot, G_ot, aswc, rst0);
// Level 4 Propagate/Generate Look-Ahead logic
// Includes Buffer for initial propagate
input  Q_in;
input  G3_in, G2_in, G1_in, G0_in;
input  P3_in, P2_in, P1_in, P0_in;
input  aswc, rst0;
output Q_ot, P_ot, G_ot;
buf1 Pr0 (Q_in,                                        Q_ot,aswc,rst0);
and4 Pro (P3_in,P2_in,P1_in,P0_in,                     P_ot,aswc,rst0);
gpp4 Gen (G3_in,G2_in,G1_in,G0_in,P3_in,P2_in,P1_in,G_ot,aswc,rst0);
endmodule

module SUBRSB16(A, B, R, Z, aswc, rst0);
// 16-bit, radix-4, carry look-ahead, two's complement, selectable
//subtractor or reverse subtractor
// 16-bit inputs A and B are selectable as to which is
// the subtrahend and minuend
// Input R selects between the following operations: +A-B or -A+B
// Subtraction performed using two's complement,
// Input selected as subtrahend is complemented using XOR
// (to give ones complement)
// Two's complement obtained with fixed Carry-in incorporated
// into initial Propagate/Generate logic
input  [15:0]  A, B;
input          R;
input  [0:4]   aswc;
input          rst0;
output [15:0]  Z;

wire   [15:0]  A0, B0;
wire   [15:0]  Q0, G0;
wire   [15:0]  Q1, G1;
wire   [15:0]  Q2, G2;
wire   [15:1]  P1;
wire   [15:4]  P2;

// A0 is ones complement of A if R==0

xor2 ia0        (~R, A[00], A0[00], aswc[0], rst0);
xor2 ia1        (~R, A[01], A0[01], aswc[0], rst0);
xor2 ia2        (~R, A[02], A0[02], aswc[0], rst0);
xor2 ia3        (~R, A[03], A0[03], aswc[0], rst0);
xor2 ia4        (~R, A[04], A0[04], aswc[0], rst0);
xor2 ia5        (~R, A[05], A0[05], aswc[0], rst0);
xor2 ia6        (~R, A[06], A0[06], aswc[0], rst0);
xor2 ia7        (~R, A[07], A0[07], aswc[0], rst0);
xor2 ia8        (~R, A[08], A0[08], aswc[0], rst0);
xor2 ia9        (~R, A[09], A0[09], aswc[0], rst0);
xor2 iaa        (~R, A[10], A0[10], aswc[0], rst0);
xor2 iab        (~R, A[11], A0[11], aswc[0], rst0);
xor2 iac        (~R, A[12], A0[12], aswc[0], rst0);
xor2 iad        (~R, A[13], A0[13], aswc[0], rst0);
xor2 iae        (~R, A[14], A0[14], aswc[0], rst0);
xor2 iaf        (~R, A[15], A0[15], aswc[0], rst0);
```

```
// B0 is ones complement of B if R==1

xor2 ib0        ( R, B[00], B0[00], aswc[0], rst0);
xor2 ib1        ( R, B[01], B0[01], aswc[0], rst0);
xor2 ib2        ( R, B[02], B0[02], aswc[0], rst0);
xor2 ib3        ( R, B[03], B0[03], aswc[0], rst0);
xor2 ib4        ( R, B[04], B0[04], aswc[0], rst0);
xor2 ib5        ( R, B[05], B0[05], aswc[0], rst0);
xor2 ib6        ( R, B[06], B0[06], aswc[0], rst0);
xor2 ib7        ( R, B[07], B0[07], aswc[0], rst0);
xor2 ib8        ( R, B[08], B0[08], aswc[0], rst0);
xor2 ib9        ( R, B[09], B0[09], aswc[0], rst0);
xor2 iba        ( R, B[10], B0[10], aswc[0], rst0);
xor2 ibb        ( R, B[11], B0[11], aswc[0], rst0);
xor2 ibc        ( R, B[12], B0[12], aswc[0], rst0);
xor2 ibd        ( R, B[13], B0[13], aswc[0], rst0);
xor2 ibe        ( R, B[14], B0[14], aswc[0], rst0);
xor2 ibf        ( R, B[15], B0[15], aswc[0], rst0);

// Produce initial Propagate and Generate signals
// LSB behaves like Full Adder with Carry in tied to 1 e.g.
//FULL_ADDER fa0 (A0[00], B0[00],1'b1, Q0[00],G0[00], aswc[1], rst0);

CALF_ADDER ha0  (A0[00], B0[00], Q0[00], G0[00], aswc[1], rst0);
HALF_ADDER ha1  (A0[01], B0[01], Q0[01], G0[01], aswc[1], rst0);
HALF_ADDER ha2  (A0[02], B0[02], Q0[02], G0[02], aswc[1], rst0);
HALF_ADDER ha3  (A0[03], B0[03], Q0[03], G0[03], aswc[1], rst0);
HALF_ADDER ha4  (A0[04], B0[04], Q0[04], G0[04], aswc[1], rst0);
HALF_ADDER ha5  (A0[05], B0[05], Q0[05], G0[05], aswc[1], rst0);
HALF_ADDER ha6  (A0[06], B0[06], Q0[06], G0[06], aswc[1], rst0);
HALF_ADDER ha7  (A0[07], B0[07], Q0[07], G0[07], aswc[1], rst0);
HALF_ADDER ha8  (A0[08], B0[08], Q0[08], G0[08], aswc[1], rst0);
HALF_ADDER ha9  (A0[09], B0[09], Q0[09], G0[09], aswc[1], rst0);
HALF_ADDER ha10 (A0[10], B0[10], Q0[10], G0[10], aswc[1], rst0);
HALF_ADDER ha11 (A0[11], B0[11], Q0[11], G0[11], aswc[1], rst0);
HALF_ADDER ha12 (A0[12], B0[12], Q0[12], G0[12], aswc[1], rst0);
HALF_ADDER ha13 (A0[13], B0[13], Q0[13], G0[13], aswc[1], rst0);
HALF_ADDER ha14 (A0[14], B0[14], Q0[14], G0[14], aswc[1], rst0);
HALF_ADDER ha15 (A0[15], B0[15], Q0[15], G0[15], aswc[1], rst0);

PP1 s00 (Q0[00], G0[00], Q1[00], G1[00], aswc[2], rst0);
PP2 s01 (Q0[01], Q0[01], Q0[00], G0[01], G0[00],
         Q1[01], P1[01], G1[01], aswc[2], rst0);
PP3 s02 (Q0[02], Q0[02], Q0[01], Q0[00], G0[02], G0[01], G0[00],
         Q1[02], P1[02], G1[02], aswc[2], rst0);
PP4 s03 (Q0[03], Q0[03], Q0[02], Q0[01], Q0[00],
         G0[03], G0[02], G0[01], G0[00],
         Q1[03], P1[03], G1[03], aswc[2], rst0);
PP1 s04 (Q0[04], G0[04], Q1[04], G1[04], aswc[2], rst0);
PP2 s05 (Q0[05], Q0[05], Q0[04], G0[05], G0[04],
         Q1[05], P1[05], G1[05], aswc[2], rst0);
PP3 s06 (Q0[06], Q0[06], Q0[05], Q0[04], G0[06], G0[05], G0[04],
         Q1[06], P1[06], G1[06], aswc[2], rst0);
PP4 s07 (Q0[07], Q0[07], Q0[06], Q0[05], Q0[04],
         G0[07], G0[06], G0[05], G0[04],
         Q1[07], P1[07], G1[07], aswc[2], rst0);
```

```
PP1 s08 (Q0[08], G0[08], Q1[08], G1[08], aswc[2], rst0);
PP2 s09 (Q0[09], Q0[09], Q0[08], G0[09], G0[08],
         Q1[09], P1[09], G1[09], aswc[2],rst0);
PP3 s0a (Q0[10], Q0[10], Q0[09], Q0[08], G0[10], G0[09], G0[08],
         Q1[10], P1[10], G1[10], aswc[2], rst0);
PP4 s0b (Q0[11], Q0[11], Q0[10], Q0[09], Q0[08],
         G0[11], G0[10], G0[09], G0[08],
         Q1[11], P1[11], G1[11], aswc[2], rst0);
PP1 s0c (Q0[12], G0[12], Q1[12], G1[12], aswc[2], rst0);
PP2 s0d (Q0[13], Q0[13], Q0[12], G0[13], G0[12],
         Q1[13], P1[13], G1[13], aswc[2], rst0);
PP3 s0e (Q0[14], Q0[14], Q0[13], Q0[12], G0[14], G0[13], G0[12],
         Q1[14], P1[14], G1[14], aswc[2], rst0);
PP4 s0f (Q0[15], Q0[15], Q0[14], Q0[13], Q0[12],
         G0[15], G0[14], G0[13], G0[12],
         Q1[15], P1[15], G1[15], aswc[2], rst0);

PP1 s10 (Q1[00], G1[00], Q2[00], G2[00], aswc[3], rst0);
PP1 s11 (Q1[01], G1[01], Q2[01], G2[01], aswc[3], rst0);
PP1 s12 (Q1[02], G1[02], Q2[02], G2[02], aswc[3], rst0);
PP1 s13 (Q1[03], G1[03], Q2[03], G2[03], aswc[3], rst0);
PP2 s14 (Q1[04], Q1[04], P1[03], G1[04], G1[03],
         Q2[04], P2[04], G2[04], aswc[3], rst0);
PP2 s15 (Q1[05], P1[05], P1[03], G1[05], G1[03],
         Q2[05], P2[05], G2[05], aswc[3], rst0);
PP2 s16 (Q1[06], P1[06], P1[03], G1[06], G1[03],
         Q2[06], P2[06], G2[06], aswc[3], rst0);
PP2 s17 (Q1[07], P1[07], P1[03], G1[07], G1[03],
         Q2[07], P2[07], G2[07], aswc[3], rst0);
PP3 s18 (Q1[08], Q1[08], P1[07], P1[3], G1[08], G1[07], G1[3],
         Q2[08], P2[08], G2[08], aswc[3], rst0);
PP3 s19 (Q1[09], P1[09], P1[07], P1[3], G1[09], G1[07], G1[3],
         Q2[09], P2[09], G2[09], aswc[3], rst0);
PP3 s1a (Q1[10], P1[10], P1[07], P1[3], G1[10], G1[07], G1[3],
         Q2[10], P2[10], G2[10], aswc[3], rst0);
PP3 s1b (Q1[11], P1[11], P1[07], P1[3], G1[11], G1[07], G1[3],
         Q2[11], P2[11], G2[11], aswc[3], rst0);
PP4 s1c (Q1[12], Q1[12], P1[11], P1[7], P1[3],
         G1[12], G1[11], G1[7], G1[3],
         Q2[12], P2[12], G2[12], aswc[3], rst0);
PP4 s1d (Q1[13], P1[13], P1[11], P1[7], P1[3],
         G1[13], G1[11], G1[7], G1[3],
         Q2[13], P2[13], G2[13], aswc[3], rst0);
PP4 s1e (Q1[14], P1[14], P1[11], P1[7], P1[3],
         G1[14], G1[11], G1[7], G1[3],
         Q2[14], P2[14], G2[14], aswc[3], rst0);
PP4 s1f (Q1[15], P1[15], P1[11], P1[7], P1[3],
         G1[15], G1[11], G1[7], G1[3],
         Q2[15], P2[15], G2[15], aswc[3], rst0);

buf1 o0 (        Q2[00], Z[00], aswc[4], rst0);
xor2 o1 (G2[00], Q2[01], Z[01], aswc[4], rst0);
xor2 o2 (G2[01], Q2[02], Z[02], aswc[4], rst0);
xor2 o3 (G2[02], Q2[03], Z[03], aswc[4], rst0);
xor2 o4 (G2[03], Q2[04], Z[04], aswc[4], rst0);
xor2 o5 (G2[04], Q2[05], Z[05], aswc[4], rst0);
xor2 o6 (G2[05], Q2[06], Z[06], aswc[4], rst0);
xor2 o7 (G2[06], Q2[07], Z[07], aswc[4], rst0);
```

```
xor2 o8 (G2[07], Q2[08], Z[08], aswc[4], rst0);
xor2 o9 (G2[08], Q2[09], Z[09], aswc[4], rst0);
xor2 oa (G2[09], Q2[10], Z[10], aswc[4], rst0);
xor2 ob (G2[10], Q2[11], Z[11], aswc[4], rst0);
xor2 oc (G2[11], Q2[12], Z[12], aswc[4], rst0);
xor2 od (G2[12], Q2[13], Z[13], aswc[4], rst0);
xor2 oe (G2[13], Q2[14], Z[14], aswc[4], rst0);
xor2 of (G2[14], Q2[15], Z[15], aswc[4], rst0);
endmodule

module BUF16(I, Z, aswc, rst0);
// 16-bit wide buffer
input  [15:0]  I;
input          aswc;
input          rst0;
output [15:0]  Z;

buf1 b00 (I[00], Z[00], aswc, rst0);
buf1 b01 (I[01], Z[01], aswc, rst0);
buf1 b02 (I[02], Z[02], aswc, rst0);
buf1 b03 (I[03], Z[03], aswc, rst0);
buf1 b04 (I[04], Z[04], aswc, rst0);
buf1 b05 (I[05], Z[05], aswc, rst0);
buf1 b06 (I[06], Z[06], aswc, rst0);
buf1 b07 (I[07], Z[07], aswc, rst0);
buf1 b08 (I[08], Z[08], aswc, rst0);
buf1 b09 (I[09], Z[09], aswc, rst0);
buf1 b0a (I[10], Z[10], aswc, rst0);
buf1 b0b (I[11], Z[11], aswc, rst0);
buf1 b0c (I[12], Z[12], aswc, rst0);
buf1 b0d (I[13], Z[13], aswc, rst0);
buf1 b0e (I[14], Z[14], aswc, rst0);
buf1 b0f (I[15], Z[15], aswc, rst0);
endmodule

module MUX16(I1, I2, S, Z, aswc, rst0);
// 16-bit wide 2-way MUX
input  [15:0]  I1, I2;
input          S;
input          aswc;
input          rst0;
output [15:0]  Z;

mux2 m00 (I1[00], I2[00], S, Z[00], aswc, rst0);
mux2 m01 (I1[01], I2[01], S, Z[01], aswc, rst0);
mux2 m02 (I1[02], I2[02], S, Z[02], aswc, rst0);
mux2 m03 (I1[03], I2[03], S, Z[03], aswc, rst0);
mux2 m04 (I1[04], I2[04], S, Z[04], aswc, rst0);
mux2 m05 (I1[05], I2[05], S, Z[05], aswc, rst0);
mux2 m06 (I1[06], I2[06], S, Z[06], aswc, rst0);
mux2 m07 (I1[07], I2[07], S, Z[07], aswc, rst0);
mux2 m08 (I1[08], I2[08], S, Z[08], aswc, rst0);
mux2 m09 (I1[09], I2[09], S, Z[09], aswc, rst0);
mux2 m0a (I1[10], I2[10], S, Z[10], aswc, rst0);
mux2 m0b (I1[11], I2[11], S, Z[11], aswc, rst0);
mux2 m0c (I1[12], I2[12], S, Z[12], aswc, rst0);
mux2 m0d (I1[13], I2[13], S, Z[13], aswc, rst0);
mux2 m0e (I1[14], I2[14], S, Z[14], aswc, rst0);
```

```
mux2 m0f (I1[15], I2[15], S, Z[15], aswc, rst0);
endmodule

module CMP16(A, B, A_EQ_B, A_GT_B, aswc, rst0);
// 16-bit, radix-4, look-ahead comparator
input  [15:0] A, B;
input  [0:2]  aswc;
input         rst0;
output        A_EQ_B, A_GT_B;

wire [15:0] EQ0, GT0;
wire  [3:0] EQ1, GT1;

// Bitwise equality
xnor2 e00 (A[00], B[00], EQ0[00], aswc[0], rst0);
xnor2 e01 (A[01], B[01], EQ0[01], aswc[0], rst0);
xnor2 e02 (A[02], B[02], EQ0[02], aswc[0], rst0);
xnor2 e03 (A[03], B[03], EQ0[03], aswc[0], rst0);
xnor2 e04 (A[04], B[04], EQ0[04], aswc[0], rst0);
xnor2 e05 (A[05], B[05], EQ0[05], aswc[0], rst0);
xnor2 e06 (A[06], B[06], EQ0[06], aswc[0], rst0);
xnor2 e07 (A[07], B[07], EQ0[07], aswc[0], rst0);
xnor2 e08 (A[08], B[08], EQ0[08], aswc[0], rst0);
xnor2 e09 (A[09], B[09], EQ0[09], aswc[0], rst0);
xnor2 e0a (A[10], B[10], EQ0[10], aswc[0], rst0);
xnor2 e0b (A[11], B[11], EQ0[11], aswc[0], rst0);
xnor2 e0c (A[12], B[12], EQ0[12], aswc[0], rst0);
xnor2 e0d (A[13], B[13], EQ0[13], aswc[0], rst0);
xnor2 e0e (A[14], B[14], EQ0[14], aswc[0], rst0);
xnor2 e0f (A[15], B[15], EQ0[15], aswc[0], rst0);

// Bitwise A>B
and2  g00 (A[00], ~B[00], GT0[00], aswc[0], rst0);
and2  g01 (A[01], ~B[01], GT0[01], aswc[0], rst0);
and2  g02 (A[02], ~B[02], GT0[02], aswc[0], rst0);
and2  g03 (A[03], ~B[03], GT0[03], aswc[0], rst0);
and2  g04 (A[04], ~B[04], GT0[04], aswc[0], rst0);
and2  g05 (A[05], ~B[05], GT0[05], aswc[0], rst0);
and2  g06 (A[06], ~B[06], GT0[06], aswc[0], rst0);
and2  g07 (A[07], ~B[07], GT0[07], aswc[0], rst0);
and2  g08 (A[08], ~B[08], GT0[08], aswc[0], rst0);
and2  g09 (A[09], ~B[09], GT0[09], aswc[0], rst0);
and2  g0a (A[10], ~B[10], GT0[10], aswc[0], rst0);
and2  g0b (A[11], ~B[11], GT0[11], aswc[0], rst0);
and2  g0c (A[12], ~B[12], GT0[12], aswc[0], rst0);
and2  g0d (A[13], ~B[13], GT0[13], aswc[0], rst0);
and2  g0e (A[14], ~B[14], GT0[14], aswc[0], rst0);
and2  g0f (A[15], ~B[15], GT0[15], aswc[0], rst0);

and4  e10 (EQ0[00], EQ0[01], EQ0[02], EQ0[03],
           EQ1[00], aswc[1], rst0);
and4  e11 (EQ0[04], EQ0[05], EQ0[06], EQ0[07],
           EQ1[01], aswc[1], rst0);
and4  e12 (EQ0[08], EQ0[09], EQ0[10], EQ0[11],
           EQ1[02], aswc[1], rst0);
and4  e13 (EQ0[12], EQ0[13], EQ0[14], EQ0[15],
           EQ1[03], aswc[1], rst0);
```

```
gpp4 g10 (GT0[03], GT0[02], GT0[01], GT0[00],
          EQ0[03], EQ0[02], EQ0[01], GT1[0], aswc[1], rst0);
gpp4 g11 (GT0[07], GT0[06], GT0[05], GT0[04],
          EQ0[07], EQ0[06], EQ0[05], GT1[1], aswc[1], rst0);
gpp4 g12 (GT0[11], GT0[10], GT0[09], GT0[08],
          EQ0[11], EQ0[10], EQ0[09], GT1[2], aswc[1], rst0);
gpp4 g13 (GT0[15], GT0[14], GT0[13], GT0[12],
          EQ0[15], EQ0[14], EQ0[13], GT1[3], aswc[1], rst0);

and4 e20 (EQ1[3], EQ1[2], EQ1[1], EQ1[0], A_EQ_B, aswc[2], rst0);
gpp4 g20 (GT1[3], GT1[2], GT1[1], GT1[0],
          EQ1[3], EQ1[2], EQ1[1], A_GT_B, aswc[2], rst0);
endmodule

module gcd (A, B, Z, reqI, ackI, reqO, ackO, rst0);
input  [15:0] A, B;
input         reqI, ackO, rst0;
output [15:0] Z;
output        ackI, reqO;

wire [0:11] aswc;
wire [13:0] req, ack;

wire  [1:2] AEQB;

wire [15:0] A0,B0,A1,B1,A2,B2,A3,B3,S8,M8,M7,M6,M5,M4;

PIPELINE_ELER0 ctb (req[3],ack3a,req[12],ack[12],aswc[10],rst0);
buf1 fbb (A_EQ_B, AEQB[1], aswc[10], rst0);

PIPELINE_ELER1 ctc (req[12],ack[12],req[13],ack[13],aswc[11],rst0);
buf1r1 fbc (AEQB[1], AEQB[2], aswc[11], rst0);

AN2  mx00 ( AEQB[2], req[13], A_EQ_Bmx0);
AN2  mx01 (~AEQB[2], req[13], A_EQ_Bmx1);
MUX2 mx0  (reqI, ackI, req[9], ack[9], A_EQ_Bmx0, A_EQ_Bmx1, ack[13],
           req[10], ack[10], rst0);

PIPELINE_ELER0 ct0 (req[10], ack[10], req[0], ack[0], aswc[0], rst0);
PIPELINE_ELER0 ct1 (req[0], ack[0], req[1], ack[1], aswc[1], rst0);
PIPELINE_ELER0 ct2 (req[1], ack[1], req[2], ack[2], aswc[2], rst0);
PIPELINE_ELER0 ct3 (req[2], ack[2], req[3], ack[3], aswc[3], rst0);
C_ELE3R0 ct3a (ack3a, ack3b, ack3c, ack[3], rst0);

MUX16 amx (A, S8, A_EQ_Bmx1, A0, aswc[0], rst0);
MUX16 bmx (B, M8, A_EQ_Bmx1, B0, aswc[0], rst0);
CMP16 cmp (A0,B0, A_EQ_B, A_GT_B, aswc[1:3], rst0);
BUF16 ab1 (A0, A1, aswc[1], rst0);
BUF16 bb1 (B0, B1, aswc[1], rst0);
BUF16 ab2 (A1, A2, aswc[2], rst0);
BUF16 bb2 (B1, B2, aswc[2], rst0);
BUF16 ab3 (A2, A3, aswc[3], rst0);
BUF16 bb3 (B2, B3, aswc[3], rst0);

AN2  dx00 ( A_EQ_B,req[3], A_EQ_Bdx0);
AN2  dx01 (~A_EQ_B,req[3], A_EQ_Bdx1);
DMX2 dx0  (req[11], ack[11], req[4], ack[4], A_EQ_Bdx0, A_EQ_Bdx1,
           ack3b, req[3], ack3c, rst0);
```

```
//Subtractor and subtrahend feedback path
PIPELINE_ELER0 ct4 (req[4], ack[4], req[5], ack[5], aswc[4], rst0);
PIPELINE_ELER0 ct5 (req[5], ack[5], req[6], ack[6], aswc[5], rst0);
PIPELINE_ELER0 ct6 (req[6], ack[6], req[7], ack[7], aswc[6], rst0);
PIPELINE_ELER0 ct7 (req[7], ack[7], req[8], ack[8], aswc[7], rst0);
PIPELINE_ELER0 ct8 (req[8], ack[8], req[9], ack[9], aswc[8], rst0);

SUBRSB16 sub (A3, B3, A_GT_B, S8, aswc[4:8], rst0);
MUX16    smx (A3, B3, A_GT_B, M4, aswc[4], rst0);
BUF16 sb1 (M4, M5, aswc[5], rst0);
BUF16 sb2 (M5, M6, aswc[6], rst0);
BUF16 sb3 (M6, M7, aswc[7], rst0);
BUF16 sb4 (M7, M8, aswc[8], rst0);

// Result output buffer
PIPELINE_ELER0 ct9 (req[11], ack[11], reqO, ackO, aswc[9], rst0);
BUF16 sb5 (A3, Z, aswc[9], rst0);

always @(M8 or S8)
begin
  #5 $display("S8/M8   %t %H %H (%D %D)",$time,S8,M8,S8,M8);
end

endmodule

module test;
reg  [15:0] A, B;
wire [15:0] Z;
reg         reqI;
wire        ackI;
wire        reqO;
wire        ackO;
reg         rst0;

initial
begin
$display("Running %m");
$dumpfile("gcd.vcd");
$dumpvars(0,test);

// Initialise
rst0 = 0;  // Activate reset
A = 16'hZZZZ;
B = 16'hZZZZ;
reqI = 0;  // Invalid data on input buses

#100 rst0 = 1; // Remove reset state
$display("------------------------------------------------------------");
A <= 16'hFFFF; B <= 16'hFFFF;
#20 reqI=1;

#20 reqI=0;
#20 A <= 16'hZZZZ; B <= 16'hZZZZ;
$display("------------------------------------------------------------");
#5000 A <= 16'h7FFF; B <= 16'hFFFE;
#20 reqI=1;

#20 reqI=0;
```

```
#20 A <= 16'hZZZZ; B <= 16'hZZZZ;
$display("------------------------------------------------------------");
#5000 A <= 16'hFFFE; B <= 16'h7FFF;
#20 reqI=1;

#20 reqI=0;
#20 A <= 16'hZZZZ; B <= 16'hZZZZ;
$display("------------------------------------------------------------");
#5000 A <= 16'hFFFF; B <= 16'hAAAA;
#20 reqI=1;

#20 reqI=0;
#20 A <= 16'hZZZZ; B <= 16'hZZZZ;
$display("------------------------------------------------------------");
#5000 A <= 16'hAAAA; B <= 16'hFFFF;
#20 reqI=1;

#20 reqI=0;
#20 A <= 16'hZZZZ; B <= 16'hZZZZ;
$display("------------------------------------------------------------");
#5000 A <= 46368;B <= 28657;
#20 reqI=1;

#20 reqI=0;
#20 A <= 16'hZZZZ; B <= 16'hZZZZ;
$display("------------------------------------------------------------");
#10000 A <= 28657;B <= 46368;
#20 reqI=1;

#20 reqI=0;
#20 A <= 16'hZZZZ; B <= 16'hZZZZ;
$display("------------------------------------------------------------");

#10000
$finish(2);
end

always @(reqI or ackI or A or B)
begin
  #5 $display("Inputs  %t Rq:%B Ak:%B A:%H B:%H (%D %D)",
                   $time, reqI, ackI,   A,   B,  A, B);
end

always @(reqO or ackO or Z)
begin
  #5 $display("Outputs %t Rq:%B Ak:%B Z:%H (%D)",
                   $time, reqO, ackO,   Z,  Z);
end

gcd  dut (A, B, Z, reqI, ackI, reqO, ackO, rst0);
BUF1 o0  (reqO, temp);
BUF1 o1  (temp, ackO);

endmodule
```

A.2 Dual-rail GCD

```
`timescale 1ns / 100ps
`define RESET_DELAY 1.5
// The assignment delay following pseudo-power-clock
`define DP_RESET_VAL 1'bx
// The normal reset value applied to data-path cells
`define STAGE_DELAY 2
// The assignment delay following pseudo-power-clock
`define NEG_EDGE 1
// Are flop outputs reset on falling edge?
`define NEG_EDGE_VAL 1'bz
// The normal value assigned upon the falling pseudo-power-clock edge
`define LOGIC_DELAY 2.5
// The assignment delay following pseudo-power-clock
`define DEBUG 0
`define DEBUG_DELAY 2.2
`define INTNODE 1

// Static CMOS cell names use ALL CAPS
// Adiabatic cell names use lower case
module C_ELE2R0 (A, B, Z, rst0);
// 2-input C-Element with reset to 0
input   A, B, rst0;
output  Z;
reg     Z;

always @(A or B or rst0)
  if (~rst0)
    #`RESET_DELAY Z <= 0;
  else if (A == B)
    #`LOGIC_DELAY Z <= A;
endmodule

module C_ELE2R1 (A, B, Z, rst0);
// 2-input C-Element with reset to 1
input   A, B, rst0;
output  Z;
reg     Z;

always @(A or B or rst0)
  if (~rst0)
    #`RESET_DELAY Z <= 1;
  else if (A == B)
    #`LOGIC_DELAY Z <= A;
endmodule

module C_ELE3R0 (A, B, C, Z, rst0);
// 3-input C-Element with reset to 0
input   A, B, C, rst0;
output  Z;
reg     Z;

always @(A or B or C or rst0)
  if (~rst0)
    #`RESET_DELAY Z <= 0;
  else if ((A == B) & (B == C))
    #`LOGIC_DELAY Z <= A;
```

```
endmodule

module BUF1 (A, Z);
// Buffer
input   A;
output  Z;
reg     Z;

always @(A)
  #`LOGIC_DELAY Z <= A;
endmodule

module INV1 (A, Z);
// Inverter
input   A;
output  Z;
reg     Z;

always @(A)
  #`LOGIC_DELAY Z <= ~A;
endmodule

module OR2 (A, B, Z);
// 2-input OR gate
input   A, B;
output  Z;
reg     Z;

always @(A or B)
  #`LOGIC_DELAY Z <= A | B;
endmodule

module AN2 (A, B, Z);
// 2-input AND gate
input   A, B;
output  Z;
reg     Z;

always @(A or B)
  #`LOGIC_DELAY Z <= A & B;
endmodule

module MUX2 (S0req, S0ack, S1req, S1ack,
             CT0req, CT1req, Ctack, Zreq, Zack, rst0);
// 2-input MUX for Asynchronous Controller
input   S0req, S1req, CT0req, CT1req, Zack;
input   rst0;
output  S0ack, S1ack, Ctack, Zreq;
wire    S0CT0req, S1CT1req, ZreqI;

C_ELE2R0 cel1 (S0req, CT0req, S0CT0req, rst0);
C_ELE2R0 cel2 (S1req, CT1req, S1CT1req, rst0);
C_ELE2R0 cel3 (Zack, S0CT0req, S0ack, rst0);
C_ELE2R0 cel4 (Zack, S1CT1req, S1ack, rst0);
OR2      or2  (S0CT0req, S1CT1req, Zreq);
BUF1     buf1 (Zack, CTack);
endmodule
```

```
module DMX2 (S0req, S0ack, S1req, S1ack,
             CT0req, CT1req, Ctack, Ireq, Iack, rst0);
// 2-output DeMUX for Asynchronous Controller
input   S0ack, S1ack, CT0req, CT1req, Ireq;
input   rst0;
output  S0req, S1req, Ctack, Iack;
wire    S0CT0req, S1CT1req;

C_ELE2R0 cel1 (Ireq, CT0req, S0req, rst0);
C_ELE2R0 cel2 (Ireq, CT1req, S1req, rst0);
OR2      or2  (S0ack, S1ack, Iack);
BUF1     buf1 (Iack, CTack);
endmodule

module PIPELINE_ELER0 (Ireq, Iack, Zreq, Zack, aswc, rst0);
// Asynchronous pipeline element
input   Ireq, Zack;
input   rst0;
output  Zreq, Iack;
output  aswc;
wire    temp;

INV1     inv1 (Zack, Zack_n);
C_ELE2R0 cel1 (Ireq, Zack_n, aswc,rst0);
BUF1     buf1 (aswc, temp);
BUF1     buf2 (temp, Zreq);
BUF1     buf3 (temp, Iack);
endmodule

module PIPELINE_ELER1 (Ireq, Iack, Zreq, Zack, aswc, rst0);
// Asynchronous pipeline element with reset to active
// (for initial tokens)
input   Ireq, Zack;
input   rst0;
output  Zreq, Iack;
output  aswc;
wire    temp;

INV1     inv1 (Zack, Zack_n);
C_ELE2R1 cel1 (Ireq, Zack_n, aswc,rst0);
BUF1     buf1 (aswc, temp);
BUF1     buf2 (temp, Zreq);
BUF1     buf3 (temp, Iack);
endmodule

// Dual Rail models of PFAL Cells
// Could be replaced with switch level models.
// aswc is the pseudo-power-clock
// rst0 is reset asserted low (for simulation purposes)

// Basic cells
```

```
module buf1 (A_L, A_H, Z_L, Z_H, aswc, rst0);
// Buffer
input  A_L, A_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;

always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (A_H == A_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b",A_L,A_H);
    end
    #`STAGE_DELAY Z_H <= A_H; Z_L <= A_L;
end

always @(negedge aswc)
if (`NEG_EDGE) begin
  #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
end
endmodule

module buf1r1 (A_L, A_H, Z_L, Z_H, aswc, rst0);
// Buffer with reset
input  A_L, A_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;

always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= 0; Z_H <= 1;
  end else begin
    if (A_H == A_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b", A_L, A_H);
    end
    #`STAGE_DELAY Z_H <= A_H; Z_L <= A_L;
end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

module xor2 (A_L, A_H, B_L, B_H, Z_L, Z_H, aswc, rst0);
// 2-input XOR
input  A_L, A_H, B_L, B_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;
```

```
always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (A_H == A_L || B_H == B_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b", A_L, A_H, B_L, B_H);
    end
    #`STAGE_DELAY Z_H <= A_H ^ B_H; Z_L <= ~(A_L ^ B_L);
  end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

module xnor2 (A_L, A_H, B_L, B_H, Z_L, Z_H, aswc, rst0);
// 2-input XNOR
input  A_L, A_H, B_L, B_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;

always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (A_H == A_L || B_H == B_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b", A_L, A_H, B_L, B_H);
    end
    #`STAGE_DELAY Z_H <= ~(A_H ^ B_H); Z_L <= A_L ^ B_L;
  end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

module and2 (A_L, A_H, B_L, B_H, Z_L, Z_H, aswc, rst0);
// 2-input AND
input  A_L, A_H, B_L, B_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;

always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (A_H == A_L || B_H == B_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b", A_L, A_H, B_L, B_H);
    end
    #`STAGE_DELAY Z_H <= A_H & B_H; Z_L <= A_L | B_L;
  end
```

```
always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

module or2 (A_L, A_H, B_L, B_H, Z_L, Z_H, aswc, rst0);
// 2-input OR
input  A_L, A_H, B_L, B_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;

always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (A_H == A_L || B_H == B_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b", A_L, A_H, B_L, B_H);
    end
    #`STAGE_DELAY Z_H <= A_H | B_H; Z_L <= A_L & B_L;
  end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

module mux2 (A_L, A_H, B_L, B_H, S_L, S_H, Z_L, Z_H, aswc, rst0);
// 2-way MUX
input  A_L, A_H, B_L, B_H, S_L, S_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;

always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (S_H == S_L || (S_L == 1'b1 && A_H == A_L )
                   || (S_H == 1'b1 && B_H == B_L )) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b %b %b", A_L, A_H, B_L, B_H, S_L, S_H);
    end
    #`STAGE_DELAY Z_H <= S_H ? B_H : A_H; Z_L <= S_L ? A_L : B_L;
  end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule
```

```
module and3 (A_L, A_H, B_L, B_H, C_L, C_H, Z_L, Z_H, aswc, rst0);
// 3-input AND
input  A_L, A_H, B_L, B_H, C_L, C_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;

always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (A_H == A_L || B_H == B_L || C_H == C_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b %b %b", A_L, A_H, B_L, B_H, C_L, C_H);
    end
    #`STAGE_DELAY Z_H <= A_H & B_H & C_H;Z_L <= A_L | B_L | C_L;
  end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

module and4 (A_L, A_H, B_L, B_H, C_L, C_H, D_L, D_H,
             Z_L, Z_H, aswc, rst0);
// 4-input AND
input  A_L, A_H, B_L, B_H, C_L, C_H, D_L, D_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;

always @(posedge aswc or negedge rst0)
  if (~rst0) begin
     #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (A_H == A_L || B_H == B_L || C_H == C_L || D_H == D_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b %b %b %b %b", A_L, A_H, B_L, B_H,
                                         C_L, C_H, D_L, D_H);
    end
    #`STAGE_DELAY Z_H <= A_H & B_H & C_H & D_H;
                  Z_L <= A_L | B_L | C_L | D_L;
  end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

module gpp2 (G1_L, G1_H, G0_L, G0_H, P1_L, P1_H,
             Z_L, Z_H, aswc, rst0);
// 3-input AND2-OR
input  G1_L, G1_H, G0_L, G0_H, P1_L, P1_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;
```

```
always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (G1_H == G1_L || G0_H == G0_L || P1_H == P1_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b %b %b",G1_H,G1_L,G0_H,G0_L,P1_H,P1_L);
    end
    #`STAGE_DELAY Z_H <= G1_H |(P1_H & G0_H);
                  Z_L <= G1_L &(P1_L | G0_L);
  end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

module gpp3 (G2_L, G2_H, G1_L, G1_H, G0_L, G0_H,
             P2_L, P2_H, P1_L, P1_H, Z_L, Z_H, aswc, rst0);
// 5-input AND2-OR-AND-OR
input  G2_L, G2_H, G1_L, G1_H, G0_L, G0_H, P2_L, P2_H, P1_L, P1_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;

always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (G2_H == G2_L || G1_H == G1_L || G0_H == G0_L ||
        P2_H == P2_L || P1_H == P1_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b %b %b %b %b %b %b",
         G2_H, G2_L, G1_H, G1_L, G0_H, G0_L, P2_H, P2_L, P1_H, P1_L);
    end
    #`STAGE_DELAY
    Z_H <= G2_H |(P2_H &(G1_H |(P1_H & G0_H)));
    Z_L <= G2_L &(P2_L |(G1_L &(P1_L | G0_L)));
  end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

module gpp4 (G3_L, G3_H, G2_L, G2_H, G1_L, G1_H, G0_L, G0_H,
             P3_L, P3_H, P2_L, P2_H, P1_L, P1_H,
             Z_L, Z_H, aswc, rst0);
// 7-input AND2-OR-AND-OR-AND-OR
input  G3_L, G3_H, G2_L, G2_H, G1_L, G1_H, G0_L, G0_H,
       P3_L, P3_H, P2_L, P2_H, P1_L, P1_H;
input  aswc, rst0;
output Z_L, Z_H;
reg    Z_L, Z_H;
```

```
always @(posedge aswc or negedge rst0)
  if (~rst0) begin
    #`RESET_DELAY Z_L <= `DP_RESET_VAL ; Z_H <= `DP_RESET_VAL ;
  end else begin
    if (G3_H == G3_L || G2_H == G2_L || G1_H == G1_L || G0_H == G0_L
     || P3_H == P3_L || P2_H == P2_L || P1_H == P1_L) begin
      $display("Complementary input violation in %m at %t",$time);
      $display("%b %b %b %b %b %b %b %b %b %b %b %b %b %b",
                      G3_H, G3_L, G2_H, G2_L, G1_H, G1_L, G0_H, G0_L,
                      P3_H, P3_L, P2_H, P2_L, P1_H, P1_L);
    end
    #`STAGE_DELAY
    Z_H <= G3_H |(P3_H &(G2_H |(P2_H &(G1_H |(P1_H & G0_H)))));
    Z_L <= G3_L &(P3_L |(G2_L &(P2_L |(G1_L &(P1_L | G0_L)))));
  end

always @(negedge aswc)
  if (`NEG_EDGE) begin
    #`STAGE_DELAY Z_L <= `NEG_EDGE_VAL; Z_H <= `NEG_EDGE_VAL;
  end
endmodule

/*
  The structural components of the subtractor
*/

module HALF_ADDER(A_L, A_H, B_L, B_H,
                  P0_L, P0_H, G0_L, G0_H, aswc, rst0);
// Half Adder to provide "Generate" and "Propagate" signals
input  A_L, A_H, B_L, B_H;
input  aswc, rst0;
output P0_L, P0_H, G0_L, G0_H;

xor2 Pro (A_L, A_H, B_L, B_H, P0_L, P0_H, aswc, rst0);
and2 Gen (A_L, A_H, B_L, B_H, G0_L, G0_H, aswc, rst0);
endmodule

module CALF_ADDER(A_L, A_H, B_L, B_H,
                  P0_L, P0_H, G0_L, G0_H, aswc, rst0);
// Carry-set Half Adder
// Used for LSB of Subtractor
input  A_L, A_H, B_L, B_H;
input  aswc, rst0;
output P0_L, P0_H, G0_L, G0_H;

xnor2 Pro (A_L, A_H, B_L, B_H, P0_L, P0_H, aswc, rst0);
or2   Gen (A_L, A_H, B_L, B_H, G0_L, G0_H, aswc, rst0);
endmodule


module PP1(Qi_L, Qi_H, Gi_L, Gi_H,
          Qo_L, Qo_H, Go_L, Go_H, aswc, rst0);
// Level 1 Propagate/Generate Look-Ahead logic
input  Qi_L, Qi_H;
input  Gi_L, Gi_H;
input  aswc, rst0;
output Qo_L, Qo_H, Go_L, Go_H;
```

```
buf1 Pr0 (Qi_L, Qi_H, Qo_L, Qo_H, aswc, rst0);
buf1 Gen (Gi_L, Gi_H, Go_L, Go_H, aswc, rst0);
endmodule

module PP2(Qi_L, Qi_H, P1i_L, P1i_H, P0i_L, P0i_H,
           G1i_L, G1i_H, G0i_L, G0i_H,
           Qo_L, Qo_H, Po_L, Po_H, Go_L, Go_H, aswc, rst0);
// Level 2 Propagate/Generate Look-Ahead logic
// Includes Buffer for initial propagate
input  Qi_L, Qi_H;
input  G1i_L, G1i_H, G0i_L, G0i_H;
input  P1i_L, P1i_H, P0i_L, P0i_H;
input  aswc, rst0;
output Qo_L, Qo_H, Po_L, Po_H, Go_L, Go_H;

buf1 Pr0 (Qi_L, Qi_H, Qo_L, Qo_H, aswc, rst0);
and2 Pro (P1i_L, P1i_H, P0i_L,P0i_H, Po_L, Po_H, aswc, rst0);
gpp2 Gen (G1i_L, G1i_H, G0i_L, G0i_H, P1i_L, P1i_H,
          Go_L, Go_H, aswc, rst0);
endmodule

module PP3(Qi_L, Qi_H, P2i_L, P2i_H, P1i_L, P1i_H, P0i_L, P0i_H,
           G2i_L, G2i_H, G1i_L, G1i_H, G0i_L, G0i_H,
           Qo_L, Qo_H, Po_L, Po_H, Go_L, Go_H, aswc, rst0);
// Level 3 Propagate/Generate Look-Ahead logic
// Includes Buffer for initial propagate
input  Qi_L, Qi_H;
input  G2i_L, G2i_H, G1i_L, G1i_H, G0i_L, G0i_H;
input  P2i_L, P2i_H, P1i_L, P1i_H, P0i_L, P0i_H;
input  aswc, rst0;
output Qo_L, Qo_H, Po_L, Po_H, Go_L, Go_H;

buf1 Pr0 (Qi_L, Qi_H, Qo_L, Qo_H, aswc, rst0);
and3 Pro (P2i_L, P2i_H, P1i_L, P1i_H, P0i_L, P0i_H,
          Po_L, Po_H, aswc, rst0);
gpp3 Gen (G2i_L, G2i_H, G1i_L, G1i_H, G0i_L, G0i_H,
          P2i_L, P2i_H, P1i_L, P1i_H,
          Go_L, Go_H, aswc, rst0);
endmodule

module PP4(Qi_L, Qi_H,
           P3i_L, P3i_H, P2i_L, P2i_H, P1i_L, P1i_H, P0i_L, P0i_H,
           G3i_L, G3i_H, G2i_L, G2i_H, G1i_L, G1i_H, G0i_L, G0i_H,
           Qo_L, Qo_H, Po_L, Po_H, Go_L, Go_H, aswc, rst0);
// Level 4 Propagate/Generate Look-Ahead logic
// Includes Buffer for initial propagate
input  Qi_L, Qi_H;
input  G3i_L, G3i_H, G2i_L, G2i_H, G1i_L, G1i_H, G0i_L, G0i_H;
input  P3i_L, P3i_H, P2i_L, P2i_H, P1i_L, P1i_H, P0i_L, P0i_H;
input  aswc, rst0;
output Qo_L, Qo_H, Po_L, Po_H, Go_L, Go_H;

buf1 Pr0 (Qi_L, Qi_H, Qo_L, Qo_H, aswc, rst0);
and4 Pro (P3i_L, P3i_H, P2i_L, P2i_H, P1i_L, P1i_H, P0i_L, P0i_H,
          Po_L, Po_H, aswc, rst0);
gpp4 Gen (G3i_L, G3i_H, G2i_L, G2i_H, G1i_L, G1i_H, G0i_L, G0i_H,
          P3i_L, P3i_H, P2i_L, P2i_H, P1i_L, P1i_H,
```

```
          Go_L, Go_H, aswc, rst0);
endmodule
module SUBRSB16(A_L,A_H,B_L,B_H,R_L,R_H,Z_L,Z_H,aswc,rst0);
// 16-bit, radix-4, carry look-ahead, two's complement,
// subtractor/reverse subtractor 16-bit inputs A and B are selectable
// as to which is the subtrahend and minuend
// Input R selects between the following operations: +A-B or -A+B
// Subtraction performed using two's complement,
// Input selected as subtrahend is complemented using XOR
// (to give ones complement)
// Two's complement obtained with fixed Carry-in incorporated into
// initial Propagate/Generate logic
input  [15:0] A_L, A_H, B_L, B_H;
input         R_L, R_H;
input  [0:4]  aswc;
input         rst0;
output [15:0] Z_L, Z_H;

wire   [15:0] A0_L, A0_H, B0_L, B0_H;
wire   [15:0] Q0_L, Q0_H, G0_L, G0_H;
wire   [15:0] Q1_L, Q1_H, G1_L, G1_H;
wire   [15:0] Q2_L, Q2_H, G2_L, G2_H;
wire   [15:1] P1_L, P1_H;
wire   [15:4] P2_L, P2_H;

// A0 is ones complement of A if R==0
xor2 ia0 (R_H,R_L, A_L[00],A_H[00], A0_L[00],A0_H[00], aswc[0],rst0);
xor2 ia1 (R_H,R_L, A_L[01],A_H[01], A0_L[01],A0_H[01], aswc[0],rst0);
xor2 ia2 (R_H,R_L, A_L[02],A_H[02], A0_L[02],A0_H[02], aswc[0],rst0);
xor2 ia3 (R_H,R_L, A_L[03],A_H[03], A0_L[03],A0_H[03], aswc[0],rst0);
xor2 ia4 (R_H,R_L, A_L[04],A_H[04], A0_L[04],A0_H[04], aswc[0],rst0);
xor2 ia5 (R_H,R_L, A_L[05],A_H[05], A0_L[05],A0_H[05], aswc[0],rst0);
xor2 ia6 (R_H,R_L, A_L[06],A_H[06], A0_L[06],A0_H[06], aswc[0],rst0);
xor2 ia7 (R_H,R_L, A_L[07],A_H[07], A0_L[07],A0_H[07], aswc[0],rst0);
xor2 ia8 (R_H,R_L, A_L[08],A_H[08], A0_L[08],A0_H[08], aswc[0],rst0);
xor2 ia9 (R_H,R_L, A_L[09],A_H[09], A0_L[09],A0_H[09], aswc[0],rst0);
xor2 iaa (R_H,R_L, A_L[10],A_H[10], A0_L[10],A0_H[10], aswc[0],rst0);
xor2 iab (R_H,R_L, A_L[11],A_H[11], A0_L[11],A0_H[11], aswc[0],rst0);
xor2 iac (R_H,R_L, A_L[12],A_H[12], A0_L[12],A0_H[12], aswc[0],rst0);
xor2 iad (R_H,R_L, A_L[13],A_H[13], A0_L[13],A0_H[13], aswc[0],rst0);
xor2 iae (R_H,R_L, A_L[14],A_H[14], A0_L[14],A0_H[14], aswc[0],rst0);
xor2 iaf (R_H,R_L, A_L[15],A_H[15], A0_L[15],A0_H[15], aswc[0],rst0);

// B0 is ones complement of B if R==1
xor2 ib0 (R_L,R_H, B_L[00],B_H[00], B0_L[00],B0_H[00], aswc[0],rst0);
xor2 ib1 (R_L,R_H, B_L[01],B_H[01], B0_L[01],B0_H[01], aswc[0],rst0);
xor2 ib2 (R_L,R_H, B_L[02],B_H[02], B0_L[02],B0_H[02], aswc[0],rst0);
xor2 ib3 (R_L,R_H, B_L[03],B_H[03], B0_L[03],B0_H[03], aswc[0],rst0);
xor2 ib4 (R_L,R_H, B_L[04],B_H[04], B0_L[04],B0_H[04], aswc[0],rst0);
xor2 ib5 (R_L,R_H, B_L[05],B_H[05], B0_L[05],B0_H[05], aswc[0],rst0);
xor2 ib6 (R_L,R_H, B_L[06],B_H[06], B0_L[06],B0_H[06], aswc[0],rst0);
xor2 ib7 (R_L,R_H, B_L[07],B_H[07], B0_L[07],B0_H[07], aswc[0],rst0);
xor2 ib8 (R_L,R_H, B_L[08],B_H[08], B0_L[08],B0_H[08], aswc[0],rst0);
xor2 ib9 (R_L,R_H, B_L[09],B_H[09], B0_L[09],B0_H[09], aswc[0],rst0);
xor2 iba (R_L,R_H, B_L[10],B_H[10], B0_L[10],B0_H[10], aswc[0],rst0);
xor2 ibb (R_L,R_H, B_L[11],B_H[11], B0_L[11],B0_H[11], aswc[0],rst0);
xor2 ibc (R_L,R_H, B_L[12],B_H[12], B0_L[12],B0_H[12], aswc[0],rst0);
xor2 ibd (R_L,R_H, B_L[13],B_H[13], B0_L[13],B0_H[13], aswc[0],rst0);
```

```
xor2 ibe (R_L,R_H, B_L[14],B_H[14], B0_L[14],B0_H[14], aswc[0],rst0);
xor2 ibf (R_L,R_H, B_L[15],B_H[15], B0_L[15],B0_H[15], aswc[0],rst0);

// Produce initial Propagate and Generate signals
// LSB behaves like Full Adder with Carry in tied to 1
CALF_ADDER ha0  (A0_L[00], A0_H[00], B0_L[00], B0_H[00],
              Q0_L[00], Q0_H[00], G0_L[00], G0_H[00], aswc[1], rst0);
HALF_ADDER ha1  (A0_L[01], A0_H[01], B0_L[01], B0_H[01],
              Q0_L[01], Q0_H[01], G0_L[01], G0_H[01], aswc[1], rst0);
HALF_ADDER ha2  (A0_L[02], A0_H[02], B0_L[02], B0_H[02],
              Q0_L[02], Q0_H[02], G0_L[02], G0_H[02], aswc[1], rst0);
HALF_ADDER ha3  (A0_L[03], A0_H[03], B0_L[03], B0_H[03],
              Q0_L[03], Q0_H[03], G0_L[03], G0_H[03], aswc[1], rst0);
HALF_ADDER ha4  (A0_L[04], A0_H[04], B0_L[04], B0_H[04],
              Q0_L[04], Q0_H[04], G0_L[04], G0_H[04], aswc[1], rst0);
HALF_ADDER ha5  (A0_L[05], A0_H[05], B0_L[05], B0_H[05],
              Q0_L[05], Q0_H[05], G0_L[05], G0_H[05], aswc[1], rst0);
HALF_ADDER ha6  (A0_L[06], A0_H[06], B0_L[06], B0_H[06],
              Q0_L[06], Q0_H[06], G0_L[06], G0_H[06], aswc[1], rst0);
HALF_ADDER ha7  (A0_L[07], A0_H[07], B0_L[07], B0_H[07],
              Q0_L[07], Q0_H[07], G0_L[07], G0_H[07], aswc[1], rst0);
HALF_ADDER ha8  (A0_L[08], A0_H[08], B0_L[08], B0_H[08],
              Q0_L[08], Q0_H[08], G0_L[08], G0_H[08], aswc[1], rst0);
HALF_ADDER ha9  (A0_L[09], A0_H[09], B0_L[09], B0_H[09],
              Q0_L[09], Q0_H[09], G0_L[09], G0_H[09], aswc[1], rst0);
HALF_ADDER ha10 (A0_L[10], A0_H[10], B0_L[10], B0_H[10],
              Q0_L[10], Q0_H[10], G0_L[10], G0_H[10], aswc[1], rst0);
HALF_ADDER ha11 (A0_L[11], A0_H[11], B0_L[11], B0_H[11],
              Q0_L[11], Q0_H[11], G0_L[11], G0_H[11], aswc[1], rst0);
HALF_ADDER ha12 (A0_L[12], A0_H[12], B0_L[12], B0_H[12],
              Q0_L[12], Q0_H[12], G0_L[12], G0_H[12], aswc[1], rst0);
HALF_ADDER ha13 (A0_L[13], A0_H[13], B0_L[13], B0_H[13],
              Q0_L[13], Q0_H[13], G0_L[13], G0_H[13], aswc[1], rst0);
HALF_ADDER ha14 (A0_L[14], A0_H[14], B0_L[14], B0_H[14],
              Q0_L[14], Q0_H[14], G0_L[14], G0_H[14], aswc[1], rst0);
HALF_ADDER ha15 (A0_L[15], A0_H[15], B0_L[15], B0_H[15],
              Q0_L[15], Q0_H[15], G0_L[15], G0_H[15], aswc[1], rst0);

PP1 s00 (Q0_L[00], Q0_H[00], G0_L[00], G0_H[00],
         Q1_L[00], Q1_H[00], G1_L[00], G1_H[00], aswc[2], rst0);
PP2 s01 (Q0_L[01], Q0_H[01], Q0_L[01], Q0_H[01], Q0_L[00], Q0_H[00],
         G0_L[01], G0_H[01], G0_L[00], G0_H[00],
         Q1_L[01], Q1_H[01], P1_L[01], P1_H[01], G1_L[01], G1_H[01],
         aswc[2], rst0);
PP3 s02 (Q0_L[02], Q0_H[02],
         Q0_L[02], Q0_H[02], Q0_L[01], Q0_H[01], Q0_L[00], Q0_H[00],
         G0_L[02], G0_H[02], G0_L[01], G0_H[01], G0_L[00], G0_H[00],
         Q1_L[02], Q1_H[02], P1_L[02], P1_H[02], G1_L[02], G1_H[02],
         aswc[2], rst0);
PP4 s03 (Q0_L[03], Q0_H[03], Q0_L[03], Q0_H[03], Q0_L[02], Q0_H[02],
         Q0_L[01], Q0_H[01], Q0_L[00], Q0_H[00], G0_L[03], G0_H[03],
         G0_L[02], G0_H[02], G0_L[01], G0_H[01], G0_L[00], G0_H[00],
         Q1_L[03], Q1_H[03], P1_L[03], P1_H[03], G1_L[03], G1_H[03],
         aswc[2], rst0);
PP1 s04 (Q0_L[04], Q0_H[04], G0_L[04], G0_H[04],
         Q1_L[04], Q1_H[04], G1_L[04], G1_H[04], aswc[2], rst0);
PP2 s05 (Q0_L[05], Q0_H[05], Q0_L[05], Q0_H[05], Q0_L[04], Q0_H[04],
         G0_L[05], G0_H[05], G0_L[04], G0_H[04],
```

```
        Q1_L[05], Q1_H[05], P1_L[05], P1_H[05], G1_L[05], G1_H[05],
        aswc[2], rst0);


PP3 s06 (Q0_L[06], Q0_H[06],
        Q0_L[06], Q0_H[06], Q0_L[05], Q0_H[05], Q0_L[04], Q0_H[04],
        G0_L[06], G0_H[06], G0_L[05], G0_H[05], G0_L[04], G0_H[04],
        Q1_L[06], Q1_H[06], P1_L[06], P1_H[06], G1_L[06], G1_H[06],
        aswc[2], rst0);
PP4 s07 (Q0_L[07], Q0_H[07], Q0_L[07], Q0_H[07], Q0_L[06], Q0_H[06],
        Q0_L[05], Q0_H[05], Q0_L[04], Q0_H[04], G0_L[07], G0_H[07],
        G0_L[06], G0_H[06], G0_L[05], G0_H[05], G0_L[04], G0_H[04],
        Q1_L[07], Q1_H[07], P1_L[07], P1_H[07], G1_L[07], G1_H[07],
        aswc[2], rst0);
PP1 s08 (Q0_L[08], Q0_H[08], G0_L[08], G0_H[08],
        Q1_L[08], Q1_H[08], G1_L[08], G1_H[08], aswc[2],rst0);
PP2 s09 (Q0_L[09], Q0_H[09], Q0_L[09], Q0_H[09], Q0_L[08], Q0_H[08],
        G0_L[09], G0_H[09], G0_L[08], G0_H[08],
        Q1_L[09], Q1_H[09], P1_L[09], P1_H[09], G1_L[09], G1_H[09],
        aswc[2], rst0);
PP3 s0a (Q0_L[10], Q0_H[10],
        Q0_L[10], Q0_H[10], Q0_L[09], Q0_H[09], Q0_L[08], Q0_H[08],
        G0_L[10], G0_H[10], G0_L[09], G0_H[09], G0_L[08], G0_H[08],
        Q1_L[10], Q1_H[10], P1_L[10], P1_H[10], G1_L[10], G1_H[10],
        aswc[2], rst0);
PP4 s0b (Q0_L[11], Q0_H[11], Q0_L[11], Q0_H[11], Q0_L[10], Q0_H[10],
        Q0_L[09], Q0_H[09], Q0_L[08], Q0_H[08], G0_L[11], G0_H[11],
        G0_L[10], G0_H[10], G0_L[09], G0_H[09], G0_L[08], G0_H[08],
        Q1_L[11], Q1_H[11], P1_L[11], P1_H[11], G1_L[11], G1_H[11],
        aswc[2], rst0);
PP1 s0c (Q0_L[12], Q0_H[12], G0_L[12], G0_H[12],
        Q1_L[12], Q1_H[12], G1_L[12], G1_H[12], aswc[2], rst0);
PP2 s0d (Q0_L[13], Q0_H[13], Q0_L[13], Q0_H[13], Q0_L[12], Q0_H[12],
        G0_L[13], G0_H[13], G0_L[12], G0_H[12],
        Q1_L[13], Q1_H[13], P1_L[13], P1_H[13], G1_L[13], G1_H[13],
        aswc[2], rst0);
PP3 s0e (Q0_L[14], Q0_H[14],
        Q0_L[14], Q0_H[14], Q0_L[13], Q0_H[13], Q0_L[12], Q0_H[12],
        G0_L[14], G0_H[14], G0_L[13], G0_H[13], G0_L[12], G0_H[12],
        Q1_L[14], Q1_H[14], P1_L[14], P1_H[14], G1_L[14], G1_H[14],
        aswc[2], rst0);
PP4 s0f (Q0_L[15], Q0_H[15], Q0_L[15], Q0_H[15], Q0_L[14], Q0_H[14],
        Q0_L[13], Q0_H[13], Q0_L[12], Q0_H[12], G0_L[15], G0_H[15],
        G0_L[14], G0_H[14], G0_L[13], G0_H[13], G0_L[12], G0_H[12],
        Q1_L[15], Q1_H[15], P1_L[15], P1_H[15], G1_L[15], G1_H[15],
        aswc[2], rst0);

PP1 s10 (Q1_L[00], Q1_H[00], G1_L[00], G1_H[00],
        Q2_L[00], Q2_H[00], G2_L[00], G2_H[00], aswc[3], rst0);
PP1 s11 (Q1_L[01], Q1_H[01], G1_L[01], G1_H[01],
        Q2_L[01], Q2_H[01], G2_L[01], G2_H[01], aswc[3], rst0);
PP1 s12 (Q1_L[02], Q1_H[02], G1_L[02], G1_H[02],
        Q2_L[02], Q2_H[02], G2_L[02], G2_H[02], aswc[3], rst0);
PP1 s13 (Q1_L[03], Q1_H[03], G1_L[03], G1_H[03],
        Q2_L[03], Q2_H[03], G2_L[03], G2_H[03], aswc[3], rst0);
PP2 s14 (Q1_L[04], Q1_H[04], Q1_L[04], Q1_H[04], P1_L[03], P1_H[03],
        G1_L[04], G1_H[04], G1_L[03], G1_H[03],
        Q2_L[04], Q2_H[04], P2_L[04], P2_H[04], G2_L[04], G2_H[04],
```

```
        aswc[3], rst0);
PP2 s15 (Q1_L[05], Q1_H[05], P1_L[05], P1_H[05], P1_L[03], P1_H[03],
        G1_L[05], G1_H[05], G1_L[03], G1_H[03],
        Q2_L[05], Q2_H[05], P2_L[05], P2_H[05], G2_L[05], G2_H[05],
        aswc[3], rst0);
PP2 s16 (Q1_L[06], Q1_H[06], P1_L[06], P1_H[06], P1_L[03], P1_H[03],
        G1_L[06], G1_H[06], G1_L[03], G1_H[03],
        Q2_L[06], Q2_H[06], P2_L[06], P2_H[06], G2_L[06], G2_H[06],
        aswc[3], rst0);
PP2 s17 (Q1_L[07], Q1_H[07], P1_L[07], P1_H[07], P1_L[03], P1_H[03],
        G1_L[07], G1_H[07], G1_L[03], G1_H[03],
        Q2_L[07], Q2_H[07], P2_L[07], P2_H[07], G2_L[07], G2_H[07],
        aswc[3], rst0);
PP3 s18 (Q1_L[08], Q1_H[08],
        Q1_L[08], Q1_H[08], P1_L[07], P1_H[07], P1_L[03], P1_H[03],
        G1_L[08], G1_H[08], G1_L[07], G1_H[07], G1_L[03], G1_H[03]
        Q2_L[08], Q2_H[08], P2_L[08], P2_H[08], G2_L[08], G2_H[08],
        aswc[3], rst0);
PP3 s19 (Q1_L[09], Q1_H[09],
        P1_L[09], P1_H[09], P1_L[07], P1_H[07], P1_L[03], P1_H[03],
        G1_L[09], G1_H[09], G1_L[07], G1_H[07], G1_L[03], G1_H[03],
        Q2_L[09], Q2_H[09], P2_L[09], P2_H[09], G2_L[09], G2_H[09],
        aswc[3], rst0);
PP3 s1a (Q1_L[10], Q1_H[10],
        P1_L[10], P1_H[10], P1_L[07], P1_H[07], P1_L[03], P1_H[03],
        G1_L[10], G1_H[10], G1_L[07], G1_H[07], G1_L[03], G1_H[03],
        Q2_L[10], Q2_H[10], P2_L[10], P2_H[10], G2_L[10], G2_H[10],
        aswc[3], rst0);
PP3 s1b (Q1_L[11], Q1_H[11],
        P1_L[11], P1_H[11], P1_L[07], P1_H[07], P1_L[03], P1_H[03],
        G1_L[11], G1_H[11], G1_L[07], G1_H[07], G1_L[03], G1_H[03],
        Q2_L[11], Q2_H[11], P2_L[11], P2_H[11], G2_L[11], G2_H[11],
        aswc[3], rst0);
PP4 s1c (Q1_L[12], Q1_H[12], Q1_L[12], Q1_H[12], P1_L[11], P1_H[11],
        P1_L[07], P1_H[07], P1_L[03], P1_H[03], G1_L[12], G1_H[12],
        G1_L[11], G1_H[11], G1_L[07], G1_H[07], G1_L[03], G1_H[03],
        Q2_L[12], Q2_H[12], P2_L[12], P2_H[12], G2_L[12], G2_H[12],
        aswc[3], rst0);
PP4 s1d (Q1_L[13], Q1_H[13], P1_L[13], P1_H[13], P1_L[11], P1_H[11],
        P1_L[07], P1_H[07], P1_L[03], P1_H[03], G1_L[13], G1_H[13],
        G1_L[11], G1_H[11], G1_L[07], G1_H[07], G1_L[03], G1_H[03],
        Q2_L[13], Q2_H[13], P2_L[13], P2_H[13], G2_L[13], G2_H[13],
        aswc[3], rst0);
PP4 s1e (Q1_L[14], Q1_H[14], P1_L[14], P1_H[14], P1_L[11], P1_H[11],
        P1_L[07], P1_H[07], P1_L[03], P1_H[03], G1_L[14], G1_H[14],
        G1_L[11], G1_H[11], G1_L[07], G1_H[07], G1_L[03], G1_H[03],
        Q2_L[14], Q2_H[14], P2_L[14], P2_H[14], G2_L[14], G2_H[14],
        aswc[3], rst0);
PP4 s1f (Q1_L[15], Q1_H[15], P1_L[15], P1_H[15], P1_L[11], P1_H[11],
        P1_L[07], P1_H[07], P1_L[03], P1_H[03], G1_L[15], G1_H[15],
        G1_L[11], G1_H[11], G1_L[07], G1_H[07], G1_L[03], G1_H[03],
        Q2_L[15], Q2_H[15], P2_L[15], P2_H[15], G2_L[15], G2_H[15],
        aswc[3], rst0);
```

```
buf1 o0 (                   Q2_L[00], Q2_H[00], Z_L[00], Z_H[00],
         aswc[4], rst0);
xor2 o1 (G2_L[00], G2_H[00], Q2_L[01], Q2_H[01], Z_L[01], Z_H[01],
         aswc[4], rst0);
xor2 o2 (G2_L[01], G2_H[01], Q2_L[02], Q2_H[02], Z_L[02], Z_H[02],
         aswc[4], rst0);
xor2 o3 (G2_L[02], G2_H[02], Q2_L[03], Q2_H[03], Z_L[03], Z_H[03],
         aswc[4], rst0);
xor2 o4 (G2_L[03], G2_H[03], Q2_L[04], Q2_H[04], Z_L[04], Z_H[04],
         aswc[4], rst0);
xor2 o5 (G2_L[04], G2_H[04], Q2_L[05], Q2_H[05], Z_L[05], Z_H[05],
         aswc[4], rst0);
xor2 o6 (G2_L[05], G2_H[05], Q2_L[06], Q2_H[06], Z_L[06], Z_H[06],
         aswc[4], rst0);
xor2 o7 (G2_L[06], G2_H[06], Q2_L[07], Q2_H[07], Z_L[07], Z_H[07],
         aswc[4], rst0);
xor2 o8 (G2_L[07], G2_H[07], Q2_L[08], Q2_H[08], Z_L[08], Z_H[08],
         aswc[4], rst0);
xor2 o9 (G2_L[08], G2_H[08], Q2_L[09], Q2_H[09], Z_L[09], Z_H[09],
         aswc[4], rst0);
xor2 oa (G2_L[09], G2_H[09], Q2_L[10], Q2_H[10], Z_L[10], Z_H[10],
         aswc[4], rst0);
xor2 ob (G2_L[10], G2_H[10], Q2_L[11], Q2_H[11], Z_L[11], Z_H[11],
         aswc[4], rst0);
xor2 oc (G2_L[11], G2_H[11], Q2_L[12], Q2_H[12], Z_L[12], Z_H[12],
         aswc[4], rst0);
xor2 od (G2_L[12], G2_H[12], Q2_L[13], Q2_H[13], Z_L[13], Z_H[13],
         aswc[4], rst0);
xor2 oe (G2_L[13], G2_H[13], Q2_L[14], Q2_H[14], Z_L[14], Z_H[14],
         aswc[4], rst0);
xor2 of (G2_L[14], G2_H[14], Q2_L[15], Q2_H[15], Z_L[15], Z_H[15],
         aswc[4], rst0);
endmodule

module BUF16(I_L, I_H, Z_L, Z_H, aswc, rst0);
// 16-bit wide buffer
input  [15:0] I_L, I_H;
input         aswc;
input         rst0;
output [15:0] Z_L, Z_H;

buf1 b00 (I_L[00], I_H[00], Z_L[00], Z_H[00], aswc, rst0);
buf1 b01 (I_L[01], I_H[01], Z_L[01], Z_H[01], aswc, rst0);
buf1 b02 (I_L[02], I_H[02], Z_L[02], Z_H[02], aswc, rst0);
buf1 b03 (I_L[03], I_H[03], Z_L[03], Z_H[03], aswc, rst0);
buf1 b04 (I_L[04], I_H[04], Z_L[04], Z_H[04], aswc, rst0);
buf1 b05 (I_L[05], I_H[05], Z_L[05], Z_H[05], aswc, rst0);
buf1 b06 (I_L[06], I_H[06], Z_L[06], Z_H[06], aswc, rst0);
buf1 b07 (I_L[07], I_H[07], Z_L[07], Z_H[07], aswc, rst0);
buf1 b08 (I_L[08], I_H[08], Z_L[08], Z_H[08], aswc, rst0);
buf1 b09 (I_L[09], I_H[09], Z_L[09], Z_H[09], aswc, rst0);
buf1 b0a (I_L[10], I_H[10], Z_L[10], Z_H[10], aswc, rst0);
buf1 b0b (I_L[11], I_H[11], Z_L[11], Z_H[11], aswc, rst0);
buf1 b0c (I_L[12], I_H[12], Z_L[12], Z_H[12], aswc, rst0);
buf1 b0d (I_L[13], I_H[13], Z_L[13], Z_H[13], aswc, rst0);
buf1 b0e (I_L[14], I_H[14], Z_L[14], Z_H[14], aswc, rst0);
buf1 b0f (I_L[15], I_H[15], Z_L[15], Z_H[15], aswc, rst0);
endmodule
```

```
module MUX16(I1_L, I1_H, I2_L, I2_H, S_L, S_H, Z_L, Z_H, aswc, rst0);
// 16-bit wide 2-way MUX
input  [15:0] I1_L, I1_H, I2_L, I2_H;
input         S_L, S_H;
input         aswc, rst0;
output [15:0] Z_L, Z_H;

mux2 m00 (I1_L[00], I1_H[00], I2_L[00], I2_H[00], S_L, S_H,
          Z_L[00], Z_H[00], aswc, rst0);
mux2 m01 (I1_L[01], I1_H[01], I2_L[01], I2_H[01], S_L, S_H,
          Z_L[01], Z_H[01], aswc, rst0);
mux2 m02 (I1_L[02], I1_H[02], I2_L[02], I2_H[02], S_L, S_H,
          Z_L[02], Z_H[02], aswc, rst0);
mux2 m03 (I1_L[03], I1_H[03], I2_L[03], I2_H[03], S_L, S_H,
          Z_L[03], Z_H[03], aswc, rst0);
mux2 m04 (I1_L[04], I1_H[04], I2_L[04], I2_H[04], S_L, S_H,
          Z_L[04], Z_H[04], aswc, rst0);
mux2 m05 (I1_L[05], I1_H[05], I2_L[05], I2_H[05], S_L, S_H,
          Z_L[05], Z_H[05], aswc, rst0);
mux2 m06 (I1_L[06], I1_H[06], I2_L[06], I2_H[06], S_L, S_H,
          Z_L[06], Z_H[06], aswc, rst0);
mux2 m07 (I1_L[07], I1_H[07], I2_L[07], I2_H[07], S_L, S_H,
          Z_L[07], Z_H[07], aswc, rst0);
mux2 m08 (I1_L[08], I1_H[08], I2_L[08], I2_H[08], S_L, S_H,
          Z_L[08], Z_H[08], aswc, rst0);
mux2 m09 (I1_L[09], I1_H[09], I2_L[09], I2_H[09], S_L, S_H,
          Z_L[09], Z_H[09], aswc, rst0);
mux2 m0a (I1_L[10], I1_H[10], I2_L[10], I2_H[10], S_L, S_H,
          Z_L[10], Z_H[10], aswc, rst0);
mux2 m0b (I1_L[11], I1_H[11], I2_L[11], I2_H[11], S_L, S_H,
          Z_L[11], Z_H[11], aswc, rst0);
mux2 m0c (I1_L[12], I1_H[12], I2_L[12], I2_H[12], S_L, S_H,
          Z_L[12], Z_H[12], aswc, rst0);
mux2 m0d (I1_L[13], I1_H[13], I2_L[13], I2_H[13], S_L, S_H,
          Z_L[13], Z_H[13], aswc, rst0);
mux2 m0e (I1_L[14], I1_H[14], I2_L[14], I2_H[14], S_L, S_H,
          Z_L[14], Z_H[14], aswc, rst0);
mux2 m0f (I1_L[15], I1_H[15], I2_L[15], I2_H[15], S_L, S_H,
          Z_L[15], Z_H[15], aswc, rst0);

always @(posedge aswc)
  if (`DEBUG)
    #`DEBUG_DELAY $display("MUX     %t %H %H %H %H %H %H %H %H",
                  $time, S_L, S_H, I1_L, I1_H, I2_L, I2_H, Z_L, Z_H);

endmodule

module CMP16(A_L, A_H, B_L, B_H,
             A_EQ_B_L, A_EQ_B_H, A_GT_B_L, A_GT_B_H, aswc, rst0);
// 16-bit, radix-4, look-ahead comparator
input [15:0] A_L, A_H, B_L, B_H;
input  [0:2] aswc;
input        rst0;
output       A_EQ_B_L, A_EQ_B_H, A_GT_B_L, A_GT_B_H;

wire [15:0] EQ0_L, EQ0_H, GT0_L, GT0_H;
wire  [3:0] EQ1_L, EQ1_H, GT1_L, GT1_H;
```

```
// Bitwise equality
xnor2 e00 (A_L[00], A_H[00], B_L[00], B_H[00],
           EQ0_L[00], EQ0_H[00], aswc[0], rst0);
xnor2 e01 (A_L[01], A_H[01], B_L[01], B_H[01],
           EQ0_L[01], EQ0_H[01], aswc[0], rst0);
xnor2 e02 (A_L[02], A_H[02], B_L[02], B_H[02],
           EQ0_L[02], EQ0_H[02], aswc[0], rst0);
xnor2 e03 (A_L[03], A_H[03], B_L[03], B_H[03],
           EQ0_L[03], EQ0_H[03], aswc[0], rst0);
xnor2 e04 (A_L[04], A_H[04], B_L[04], B_H[04],
           EQ0_L[04], EQ0_H[04], aswc[0], rst0);
xnor2 e05 (A_L[05], A_H[05], B_L[05], B_H[05],
           EQ0_L[05], EQ0_H[05], aswc[0], rst0);
xnor2 e06 (A_L[06],A_H[06], B_L[06], B_H[06],
           EQ0_L[06], EQ0_H[06], aswc[0], rst0);
xnor2 e07 (A_L[07], A_H[07], B_L[07],B_H[07],
           EQ0_L[07], EQ0_H[07], aswc[0], rst0);
xnor2 e08 (A_L[08], A_H[08], B_L[08], B_H[08],
           EQ0_L[08], EQ0_H[08], aswc[0], rst0);
xnor2 e09 (A_L[09], A_H[09], B_L[09], B_H[09],
           EQ0_L[09], EQ0_H[09], aswc[0], rst0);
xnor2 e0a (A_L[10], A_H[10], B_L[10], B_H[10],
           EQ0_L[10], EQ0_H[10], aswc[0], rst0);
xnor2 e0b (A_L[11], A_H[11], B_L[11], B_H[11],
           EQ0_L[11], EQ0_H[11], aswc[0], rst0);
xnor2 e0c (A_L[12], A_H[12], B_L[12], B_H[12],
           EQ0_L[12], EQ0_H[12], aswc[0], rst0);
xnor2 e0d (A_L[13], A_H[13], B_L[13], B_H[13],
           EQ0_L[13], EQ0_H[13], aswc[0], rst0);
xnor2 e0e (A_L[14], A_H[14], B_L[14], B_H[14],
           EQ0_L[14], EQ0_H[14], aswc[0], rst0);
xnor2 e0f (A_L[15], A_H[15], B_L[15], B_H[15],
           EQ0_L[15], EQ0_H[15], aswc[0], rst0);

// Bitwise A>B
and2 g00 (A_L[00], A_H[00], B_H[00], B_L[00],
          GT0_L[00], GT0_H[00], aswc[0], rst0);
and2 g01 (A_L[01], A_H[01], B_H[01], B_L[01],
          GT0_L[01], GT0_H[01], aswc[0], rst0);
and2 g02 (A_L[02], A_H[02], B_H[02], B_L[02],
          GT0_L[02], GT0_H[02], aswc[0], rst0);
and2 g03 (A_L[03], A_H[03], B_H[03], B_L[03],
          GT0_L[03], GT0_H[03], aswc[0], rst0);
and2 g04 (A_L[04], A_H[04], B_H[04], B_L[04],
          GT0_L[04], GT0_H[04], aswc[0], rst0);
and2 g05 (A_L[05], A_H[05], B_H[05], B_L[05],
          GT0_L[05], GT0_H[05], aswc[0], rst0);
and2 g06 (A_L[06], A_H[06], B_H[06], B_L[06],
          GT0_L[06], GT0_H[06], aswc[0], rst0);
and2 g07 (A_L[07], A_H[07], B_H[07], B_L[07],
          GT0_L[07], GT0_H[07], aswc[0], rst0);
and2 g08 (A_L[08], A_H[08], B_H[08], B_L[08],
          GT0_L[08], GT0_H[08], aswc[0], rst0);
and2 g09 (A_L[09], A_H[09], B_H[09], B_L[09],
          GT0_L[09], GT0_H[09], aswc[0], rst0);
and2 g0a (A_L[10], A_H[10], B_H[10], B_L[10],
          GT0_L[10], GT0_H[10], aswc[0], rst0);
```

```
and2 g0b (A_L[11], A_H[11], B_H[11], B_L[11],
          GT0_L[11], GT0_H[11], aswc[0], rst0);
and2 g0c (A_L[12], A_H[12], B_H[12], B_L[12],
          GT0_L[12], GT0_H[12], aswc[0], rst0);
and2 g0d (A_L[13], A_H[13], B_H[13], B_L[13],
          GT0_L[13], GT0_H[13], aswc[0], rst0);
and2 g0e (A_L[14], A_H[14], B_H[14], B_L[14],
          GT0_L[14], GT0_H[14], aswc[0], rst0);
and2 g0f (A_L[15], A_H[15], B_H[15], B_L[15],
          GT0_L[15], GT0_H[15], aswc[0], rst0);

and4 e10 (EQ0_L[00], EQ0_H[00], EQ0_L[01], EQ0_H[01],
          EQ0_L[02], EQ0_H[02], EQ0_L[03], EQ0_H[03],
          EQ1_L[00], EQ1_H[00], aswc[1], rst0);
and4 e11 (EQ0_L[04], EQ0_H[04], EQ0_L[05], EQ0_H[05],
          EQ0_L[06], EQ0_H[06], EQ0_L[07], EQ0_H[07],
          EQ1_L[01], EQ1_H[01], aswc[1], rst0);
and4 e12 (EQ0_L[08], EQ0_H[08], EQ0_L[09], EQ0_H[09],
          EQ0_L[10], EQ0_H[10], EQ0_L[11], EQ0_H[11],
          EQ1_L[02], EQ1_H[02], aswc[1], rst0);
and4 e13 (EQ0_L[12], EQ0_H[12], EQ0_L[13], EQ0_H[13],
          EQ0_L[14], EQ0_H[14], EQ0_L[15], EQ0_H[15],
          EQ1_L[03], EQ1_H[03], aswc[1], rst0);

gpp4 g10 (GT0_L[03], GT0_H[03], GT0_L[02], GT0_H[02],
          GT0_L[01], GT0_H[01], GT0_L[00], GT0_H[00],
          EQ0_L[03], EQ0_H[03], EQ0_L[02], EQ0_H[02],
          EQ0_L[01], EQ0_H[01], GT1_L[0], GT1_H[0], aswc[1], rst0);
gpp4 g11 (GT0_L[07], GT0_H[07], GT0_L[06], GT0_H[06],
          GT0_L[05], GT0_H[05], GT0_L[04], GT0_H[04],
          EQ0_L[07], EQ0_H[07], EQ0_L[06], EQ0_H[06],
          EQ0_L[05], EQ0_H[05], GT1_L[1], GT1_H[1], aswc[1], rst0);
gpp4 g12 (GT0_L[11], GT0_H[11], GT0_L[10], GT0_H[10],
          GT0_L[09], GT0_H[09], GT0_L[08], GT0_H[08],
          EQ0_L[11], EQ0_H[11], EQ0_L[10], EQ0_H[10],
          EQ0_L[09], EQ0_H[09], GT1_L[2], GT1_H[2], aswc[1], rst0);
gpp4 g13 (GT0_L[15], GT0_H[15], GT0_L[14], GT0_H[14],
          GT0_L[13], GT0_H[13], GT0_L[12], GT0_H[12],
          EQ0_L[15], EQ0_H[15], EQ0_L[14], EQ0_H[14],
          EQ0_L[13], EQ0_H[13], GT1_L[3], GT1_H[3], aswc[1], rst0);

and4 e20 (EQ1_L[3], EQ1_H[3], EQ1_L[2], EQ1_H[2],
          EQ1_L[1], EQ1_H[1], EQ1_L[0], EQ1_H[0],
          A_EQ_B_L, A_EQ_B_H, aswc[2], rst0);
gpp4 g20 (GT1_L[3], GT1_H[3], GT1_L[2], GT1_H[2],
          GT1_L[1], GT1_H[1], GT1_L[0], GT1_H[0],
          EQ1_L[3], EQ1_H[3], EQ1_L[2], EQ1_H[2],
          EQ1_L[1], EQ1_H[1], A_GT_B_L, A_GT_B_H, aswc[2], rst0);

always @(GT0_L or GT0_H or EQ0_L or EQ0_H)
  if (`DEBUG)
    #`DEBUG_DELAY $display("cmp0    %t %B %B %B %B",
                           $time, GT0_H, GT0_L, EQ0_H, EQ0_L);
always @(GT1_L or GT1_H or EQ1_L or EQ1_H)
  if (`DEBUG)
    #`DEBUG_DELAY $display("cmp1    %t %B %B %B %B",
                            $time, GT1_H, GT1_L, EQ1_H, EQ1_L);
endmodule
```

```
module gcd (A_L, A_H, B_L, B_H, Z_L, Z_H, reqI, ackI, reqO, ackO,
rst0);
input  [15:0] A_L,A_H,B_L,B_H;
input         reqI,ackO,rst0;
output [15:0] Z_L,Z_H;
output        ackI,reqO;

wire [0:11] aswc;
wire [13:0] req,ack;
wire [1:2]  AEQB_L,AEQB_H;
wire [15:0] A0_L, A0_H, B0_L, B0_H, A1_L, A1_H, B1_L, B1_H,
            A2_L, A2_H, B2_L, B2_H, A3_L, A3_H, B3_L, B3_H,
            S8_L, S8_H, M8_L, M8_H,
            M7_L, M7_H, M6_L, M6_H, M5_L, M5_H, M4_L, M4_H;

PIPELINE_ELER0 ctb (req[3],ack3a,req[12],ack[12],aswc[10],rst0);
buf1 fbb (A_EQ_B_L,A_EQ_B_H,AEQB_L[1],AEQB_H[1],aswc[10],rst0);

PIPELINE_ELER1 ctc (req[12],ack[12],req[13], ack[13], aswc[11],
rst0);
buf1r1 fbc (AEQB_L[1],AEQB_H[1],AEQB_L[2],AEQB_H[2], aswc[11], rst0);

AN2  mx00 (AEQB_H[2],req[13],A_EQ_Bmx0);
AN2  mx01 (AEQB_L[2],req[13],A_EQ_Bmx1);
MUX2 mx0  (reqI, ackI, req[9], ack[9], A_EQ_Bmx0, A_EQ_Bmx1, ack[13],
req[10], ack[10], rst0);

PIPELINE_ELER0 ct0 (req[10], ack[10], req[0], ack[0], aswc[0], rst0);
PIPELINE_ELER0 ct1 (req[0], ack[0], req[1], ack[1], aswc[1], rst0);
PIPELINE_ELER0 ct2 (req[1], ack[1], req[2], ack[2], aswc[2], rst0);
PIPELINE_ELER0 ct3 (req[2], ack[2], req[3], ack[3], aswc[3], rst0);
C_ELE3R0 ct3a (ack3a, ack3b, ack3c, ack[3], rst0);

MUX16 amx (A_L, A_H, S8_L, S8_H, AEQB_H[2], AEQB_L[2], A0_L, A0_H,
aswc[0], rst0);
MUX16 bmx (B_L, B_H, M8_L, M8_H, AEQB_H[2], AEQB_L[2], B0_L, B0_H,
aswc[0], rst0);
CMP16 cmp (A0_L, A0_H, B0_L, B0_H, A_EQ_B_L, A_EQ_B_H, A_GT_B_L,
A_GT_B_H, aswc[1:3], rst0);
BUF16 ab1 (A0_L, A0_H, A1_L, A1_H, aswc[1], rst0);
BUF16 bb1 (B0_L, B0_H, B1_L, B1_H, aswc[1], rst0);
BUF16 ab2 (A1_L, A1_H, A2_L, A2_H, aswc[2], rst0);
BUF16 bb2 (B1_L, B1_H, B2_L, B2_H, aswc[2], rst0);
BUF16 ab3 (A2_L, A2_H, A3_L, A3_H, aswc[3], rst0);
BUF16 bb3 (B2_L, B2_H, B3_L, B3_H, aswc[3], rst0);

AN2  dx00 (A_EQ_B_H, req[3], A_EQ_Bdx0);
AN2  dx01 (A_EQ_B_L, req[3], A_EQ_Bdx1);
DMX2 dx0  (req[11], ack[11], req[4], ack[4], A_EQ_Bdx0, A_EQ_Bdx1,
ack3b, req[3], ack3c, rst0);


//Subtractor and subtrahend feedback path
PIPELINE_ELER0 ct4 (req[4], ack[4], req[5], ack[5], aswc[4], rst0);
PIPELINE_ELER0 ct5 (req[5], ack[5], req[6], ack[6], aswc[5], rst0);
PIPELINE_ELER0 ct6 (req[6], ack[6], req[7], ack[7], aswc[6], rst0);
PIPELINE_ELER0 ct7 (req[7], ack[7], req[8], ack[8], aswc[7], rst0);
```

```
PIPELINE_ELER0 ct8 (req[8], ack[8], req[9], ack[9], aswc[8], rst0);

SUBRSB16 sub (A3_L, A3_H, B3_L, B3_H, A_GT_B_L, A_GT_B_H, S8_L, S8_H,
aswc[4:8], rst0);
MUX16 smx (A3_L, A3_H, B3_L, B3_H, A_GT_B_L, A_GT_B_H, M4_L, M4_H,
aswc[4],rst0);
BUF16 sb1 (M4_L, M4_H, M5_L, M5_H, aswc[5], rst0);
BUF16 sb2 (M5_L, M5_H, M6_L, M6_H, aswc[6], rst0);
BUF16 sb3 (M6_L, M6_H, M7_L, M7_H, aswc[7], rst0);
BUF16 sb4 (M7_L, M7_H, M8_L, M8_H, aswc[8], rst0);

// Result output buffer
PIPELINE_ELER0 ct9 (req[11], ack[11], reqO, ackO, aswc[9], rst0);
BUF16 sb5 (A3_L, A3_H, Z_L, Z_H, aswc[9], rst0);

always @(aswc)
  if (`DEBUG)
    #`DEBUG_DELAY $display("ASWC    %t %B", $time, aswc);

always @(req or ack)
  if (`DEBUG)
    #`DEBUG_DELAY $display("RqAk %t %B %B", $time, req, ack);

always @(ack3a or ack3b or ack3c)
  if (`DEBUG)
    #`DEBUG_DELAY $display("Ack3 %t %B %B %B",
                           $time, ack3a, ack3b, ack3c);

always @(A0_L or A0_H or B0_L or B0_H)
  if (`DEBUG)
    #`DEBUG_DELAY $display("A0B0    %t %D %D %D %D",
                           $time, A0_H, A0_L, B0_H, B0_L);

always @(A1_L or A1_H or B1_L or B1_H)
  if (`DEBUG)
    #`DEBUG_DELAY $display("A1B1    %t %D %D %D %D",
                           $time, A1_H, A1_L, B1_H, B1_L);

always @(A2_L or A2_H or B2_L or B2_H)
  if (`DEBUG)
    #`DEBUG_DELAY $display("A2B2    %t %D %D %D %D",
                           $time, A2_H, A2_L, B2_H, B2_L);

always @(A3_L or A3_H or B3_L or B3_H)
  if (`DEBUG)
    #`DEBUG_DELAY $display("A3B3    %t %D %D %D %D",
                           $time, A3_H, A3_L, B3_H, B3_L);

always @(A_EQ_B_L or A_EQ_B_H or A_GT_B_L or A_GT_B_H)
  if (`DEBUG)
    #`DEBUG_DELAY $display("GTEQ    %t EQ:%B %B GT:%B %B",
                      $time, A_EQ_B_H, A_EQ_B_L, A_GT_B_H, A_GT_B_L);

always @(M8_L or M8_H or S8_L or S8_H)
  if (`DEBUG || `INTNODE)
    #`DEBUG_DELAY $display("M8      %t %H %H %H %H (%D %D %D %D)",
              $time, S8_H, S8_L, M8_H, M8_L, S8_H, S8_L, M8_H, M8_L);
endmodule
```

```
module test;
reg  [15:0] A_L, A_H, B_L, B_H;
wire [15:0] Z_L, Z_H;
reg         reqI;
wire        ackI;
wire        reqO;
wire        ackO;
reg         rst0;

wire        temp;
initial
begin
$display("Running %m");
$dumpfile("gcd-dr.vcd");
$dumpvars(0,test);

// Initialise
rst0 = 0;  // Activate reset
A_H = 16'hZZZZ; A_L = 16'hZZZZ; B_H = 16'hZZZZ; B_L = 16'hZZZZ;
reqI = 0;  // Invalid data on input buses

#100 rst0 = 1; // Remove reset state
$display("-----------------------------------------------------------");
A_H <=16'hFFFF; B_H <=16'hFFFF; #1 A_L <= ~A_H; B_L <= ~B_H;
#20 reqI=1;

#20 reqI=0;
#20 A_H <=16'hZZZZ; B_H <=16'hZZZZ; A_L <=16'hZZZZ; B_L <=16'hZZZZ;
$display("-----------------------------------------------------------");
#5000 A_H <=16'h7FFF; B_H <=16'hFFFE; #1 A_L <= ~A_H; B_L <= ~B_H;
#20 reqI=1;

#20 reqI=0;
#20 A_H <=16'hZZZZ; B_H <=16'hZZZZ; A_L <=16'hZZZZ; B_L <= 6'hZZZZ;
$display("-----------------------------------------------------------");
#5000 A_H <= 16'hFFFE; B_H <= 16'h7FFF; #1 A_L <= ~A_H; B_L <= ~B_H;
#20 reqI=1;

#20 reqI=0;
#20 A_H <=16'hZZZZ; B_H <=16'hZZZZ; A_L <= 16'hZZZZ; B_L <= 16'hZZZZ;
$display("-----------------------------------------------------------");
#5000 A_H <=16'hFFFF; B_H <=16'hAAAA; #1 A_L <= ~A_H; B_L <= ~B_H;
#20 reqI=1;

#20 reqI=0;
#20 A_H <= 16'hZZZZ; B_H <= 16'hZZZZ; A_L <= 16'hZZZZ; B_L <=
16'hZZZZ;
$display("-----------------------------------------------------------");
#5000 A_H <= 16'hAAAA; B_H <= 16'hFFFF; #1 A_L <= ~A_H; B_L <= ~B_H;
#20 reqI=1;

#20 reqI=0;
#20 A_H <=16'hZZZZ; B_H <=16'hZZZZ; A_L <=16'hZZZZ; B_L <=16'hZZZZ;
$display("-----------------------------------------------------------");
#5000 A_H <= 46368;B_H <= 28657; #1 A_L <= ~A_H; B_L <= ~B_H;
#20 reqI=1;
```

```
#20 reqI=0;
#20 A_H <=16'hZZZZ; B_H <=16'hZZZZ; A_L <=16'hZZZZ; B_L <=16'hZZZZ;
$display("-----------------------------------------------------");
#10000 A_H <= 28657;B_H <= 46368; #1 A_L <= ~A_H; B_L <= ~B_H;
#20 reqI=1;

#20 reqI=0;
#20 A_H <=16'hZZZZ; B_H <=16'hZZZZ; A_L <=16'hZZZZ; B_L <=16'hZZZZ;
$display("-----------------------------------------------------");

#10000
$finish(2);
end

always @(reqI or ackI or A_L or A_H or B_L or B_H)
begin
  #5 $display("Inputs  %t Rq:%B Ak:%B A:%H %H B:%H %H (%D %D %D %D)",
          $time, reqI, ackI, A_H, A_L, B_H, B_L, A_H, A_L, B_H, B_L);
end

always @(reqO or ackO or Z_H)
begin
  #5 $display("Outputs %t Rq:%B Ak:%B Z:%H %H (%D %D)",
                              $time, reqO, ackO, Z_H, Z_L, Z_H, Z_L);
end

gcd  dut (A_L,A_H, B_L,B_H, Z_L,Z_H, reqI,ackI, reqO,ackO, rst0);
BUF1 o0  (reqO,temp);
BUF1 o1  (temp,ackO);

endmodule
```

Appendix B C source-code

```
/* OBDD reduction of functions */
#include <stdio.h>
#include <stdlib.h>

#define NULL 0

#define VERTEX_ZERO 0
#define VERTEX_ONE  1

#define DEBUG 0

/*#define ECRL  0*/
/*#define IECRL 0*/
#define PFAL  1

#define LEVELS 4
#define SIZE 16
#define OTFUN 65536

/*
#define LEVELS 3
#define SIZE 8
#define OTFUN 256
*/
/*
#define LEVELS 2
#define SIZE 4
#define OTFUN 16
*/
/*
#define LEVELS 1
#define SIZE 2
#define OTFUN 4
*/
/*
SIZE = 1<<LEVELS
OTFUN= 1<<SIZE
*/

struct node {
  int    chooser;
  char   name;
  struct node *zero;
  struct node *one;
};

struct table {
  struct node  *node;
  struct table *next;
};

struct expr {
  int value;
  int size;
};
```

```
struct choice {
  int logical;
  int physical;
};


static struct node *v0;
static struct node *v1;

static struct table *t0;

static struct choice var_ord[LEVELS];

static char node_name;

static const char *pmosl = "0.35u";
static const char *pmosw = "0.50u";
static const char *pmosm = "P";
static const char *nmosl = "0.35u";
static const char *nmosw = "0.50u";
static const char *nmosm = "N";

/* allocate storage for vertex */
struct node *obdd_alloc(void)
{
  return (struct node *) malloc(sizeof(struct node));
}

struct table *talloc(void)
{
  return (struct table *) malloc(sizeof(struct table));
}

/*
  A tree factorial is used to calculate the complete search
  space of FBDD trees to find the minimal implementation
*/
int factorial(int in)
{
  int i;
  int fact=1;

  for(i=1;i<=in;i++)
    fact*=i;
  return fact;
}

int tree_fact(int in)
{
  int i,j;
  int fact=1;

  for(i=1;i<=in;i++)
    for(j=1;j<=(1<<(in-i));j++)
      fact*=i;
  return fact;
}
```

```
void fact_ord_nr(int in, int size)
{
  int h;
  int i;
  int j;
  int k;
  int l;
  int used=0;

  i=in;
  for (l=size;l>0;l--) {
    h=i%l;
    k=0;
    for (j=0;j<=h;j++)
      do {
        k++;
      } while (used & (1<<k));
    var_ord[l-1].logical=h;
    var_ord[l-1].physical=k;
    used |= (1<<k);
    i/=l;
  }
}

void fact_ord_nrp(int in, int size)
{
  int h;
  int i;
  int j;
  int k;
  int l;
  int used=0;

  i=in;
  for (l=size;l>0;l--) {
    h=i%l;
    k=0;
    for (j=0;j<=h;j++)
      do {
        k++;
      } while (used & (1<<k));
    var_ord[l-1].logical=h;
    var_ord[l-1].physical=k;
    used |= (1<<k);
    i/=l;
  }
  printf("* Logical  ");
  for (l=size;l>0;l--)
    printf("%d",var_ord[l-1].logical);
  printf(" Physical ");
  for (l=size;l>0;l--)
    printf("%d",var_ord[l-1].physical);
  printf("\n");
}
```

```
/* Can this be converted to be non-recursive? */
struct node *find_or_add(struct table *tab,
                         int chooser,
                         struct node *zero,
                         struct node *one)
{
  struct node  *v;
  struct table *t;

  if (tab->node==NULL) { /* We're at the end of the linked list */
    v = obdd_alloc();
    v->chooser=chooser;
    v->name=node_name++;
    v->zero=zero;
    v->one=one;
    t = talloc();
    t->next=NULL;
    t->node=NULL;
    tab->node=v;
    tab->next=t;
    #if DEBUG
      printf("FA: Adding   %x %p %p\n", chooser, zero, one);
    #endif
  } else if (tab->node->chooser==chooser &&
             tab->node->zero   ==zero    &&
             tab->node->one    ==one) {
    v=tab->node;
    #if DEBUG
      printf("FA: Found %p\n",tab->node);
    #endif
  } else {
    v=find_or_add(tab->next, chooser, zero, one);
    #if DEBUG
      printf("FA: Next\n");
    #endif
  }
  return v;
}

void choose(struct expr f, struct expr *g, int chooser)
{
  int i;
  int new_exprl=0;
  int new_exprh=0;

  #if DEBUG
    printf("CH: Size %d\n",f.size-1);
    printf("CH: ");
  #endif
  for(i=f.size-1;i>=0;i--){
    #if DEBUG
      printf("%d ",(f.value)&(1<<i)?1:0);
    #endif
    if (i & (1<<chooser)) {
      new_exprl<<=1;
      new_exprl|=(f.value)&(1<<i)?1:0;
    } else {
      new_exprh<<=1;
```

```
      new_exprh|=(f.value)&(1<<i)?1:0;
    }
  }
  #if DEBUG
    printf("\n");
  #endif
  #if DEBUG
    printf("CH: Split %x on %d, %x %x\n",
                                (f.value),chooser,new_exprl,new_exprh);
  #endif
  g[0].size = g[1].size = (f.size)>>1;
  g[0].value=new_exprl;
  g[1].value=new_exprh;
}

/* Can this be converted to be non-recursive? */
struct node *robdd_build(struct table *tab, struct expr f, int ix)
{
  struct node *zero, *one;
  struct expr g[2];

  #if DEBUG
    printf("RB: Level %d\n",ix);
  #endif
  if (f.size==1) {
    if (f.value==0) {
      #if DEBUG
        printf ("RB: Returning V0\n");
      #endif
      return v0;
    } else if (f.value==1) {
      #if DEBUG
        printf ("RB: Returning V1\n");
      #endif
      return v1;
    } else {
      printf("RB: f=%d\n",f.value);
      printf("RB: ERROR Can't return V0 or V1 @ Top Level\n");
      }
    } else {
/* The decision can be made on i, 1 or some arbitrary number between
 these limits */
    #if DEBUG
      printf("RB: Level %d lvo %d pvo %d\n",
                ix,(var_ord[ix-1].logical),(var_ord[ix-1].physical));
    #endif
    choose(f,&g[0],(var_ord[ix-1].logical));
    #if DEBUG
      printf("RB: f %x o %x l %x\n",f.value,g[0].value,g[1].value);
    #endif
    zero = robdd_build(tab, g[0], ix-1);
    one  = robdd_build(tab, g[1], ix-1);
    if (zero == one) {
      #if DEBUG
        printf ("RB: No Node\n");
      #endif
      return zero;
    } else {
```

```
      #if DEBUG
        printf ("RB: Add node %x %p %p\n",ix,zero,one);
      #endif
      return find_or_add(tab, var_ord[ix-1].physical, zero, one);
    }
  }
}

/* Can this be converted to be non-recursive? */
int obdd_print(struct node *node)
{
  int totals =0;

  if (node->chooser==0) {
    if (node==v0) {
      printf("0 "); /* Zero function */
      return 0;
    }
    if (node==v1) {
      printf("1 "); /* Ones function */
      return 0;
    }
    printf("ERROR (obdd print)\n");
    }
  printf("%d=L: ",node->chooser);
  if (node->zero==v0)
    printf("0 ");
  else if (node->zero==v1)
    printf("1 ");
  else
    totals+=obdd_print(node->zero);
  printf("%d=H: ",node->chooser);
  if (node->one==v0)
    printf("0 ");
  else if (node->one==v1)
    printf("1 ");
  else
    totals+=obdd_print(node->one);
  return totals+1;
}

int obdd_count(struct node *node)
{
  int totals =0;

  if (node->chooser==0) {
    if (node==v0) {
      return 1;
    }
    if (node==v1) {
      return 1;
    }
    printf("ERROR (obdd count)\n");
  }
  totals+=obdd_count(node->zero);
  totals+=obdd_count(node->one);
  return totals;
}
```

```
int table_count(struct table *entry)
{
  int count =0;

  while (entry->next!=NULL) {
    count++;
    entry=entry->next;
  }
  return count;
}

/* Can this be converted to be non-recursive? */
int obdd_print_tab(struct table *entry)
{
  int totals =0;
  struct node *left;
  struct node *right;

  if (entry->node==NULL)
    return 0;
  else {
    printf("Node %c ",entry->node->name);
    left=entry->node->zero;
    right=entry->node->one;
    printf("I%d=L: ",entry->node->chooser);
    if (left==v0)
      printf("0 ");
    else if (left==v1)
      printf("1 ");
    else
      printf("%c ",left->name);
    printf("I%d=H: ",entry->node->chooser);
    if (right==v0)
      printf("0 ");
    else if (right==v1)
      printf("1 ");
    else
      printf("%c ",right->name);
    totals +=obdd_print_tab(entry->next);
  }
  return totals+1;
}

void print_spice_header(int func_id)
{
  int i;

  printf(".Subckt X%dX%0*X\n+",LEVELS,SIZE>>2,func_id);
  for(i=1;i<=LEVELS;i++)
    printf("I%d_H I%d_L ",i,i);
  printf("Z_H Z_L vpc gnd\n");
  printf("MP0  Z_L Z_H  vpc vpc %s L=%s W=%s\n",pmosm,pmosl,pmosw);
  printf("MP1  Z_H Z_L  vpc vpc %s L=%s W=%s\n",pmosm,pmosl,pmosw);
  #ifdef IECRL
    printf("MN0  Z_L Z_H  gnd gnd %s L=%s W=%s\n",nmosm,nmosl,nmosw);
    printf("MN1  Z_H Z_L  gnd gnd %s L=%s W=%s\n",nmosm,nmosl,nmosw);
  #endif
```

```
  #ifdef PFAL
    printf("MN0  Z_L Z_H  gnd gnd %s L=%s W=%s\n",nmosm,nmosl,nmosw);
    printf("MN1  Z_H Z_L  gnd gnd %s L=%s W=%s\n",nmosm,nmosl,nmosw);
  #endif
}

/* Can this be converted to be non-recursive? */
int obdd_print_spice(int func_id,
                     struct table *entry,
                     struct node *root)
{
  int totals =0;
  struct node *left;
  struct node *right;

  if (entry->node==NULL) {
    print_spice_header(func_id);
    return 0;
  } else {
    totals +=obdd_print_spice(func_id,entry->next,root);

    left=entry->node->zero;
    if(entry->node==root) {
      #ifdef ECRL
        printf("MN%cL gnd I%d_L ",
                          entry->node->name, entry->node->chooser);
      #endif
      #ifdef IECRL
        printf("MN%cL gnd I%d_L ",
                         entry->node->name,  entry->node->chooser);
      #endif
      #ifdef PFAL
        printf("MN%cL vpc I%d_L ",
                          entry->node->name, entry->node->chooser);
      #endif
  } else
      printf("MN%cL %c   I%d_L ",
         entry->node->name, entry->node->name, entry->node->chooser);
    if (left==v0) {
      #ifdef ECRL
        printf("Z_H ");
      #endif
      #ifdef IECRL
        printf("Z_H ");
      #endif
      #ifdef PFAL
        printf("Z_L ");
      #endif
  } else if (left==v1) {
      #ifdef ECRL
        printf("Z_L ");
      #endif
      #ifdef IECRL
        printf("Z_L ");
      #endif
      #ifdef PFAL
        printf("Z_H ");
      #endif
```

```
    } else
      printf("%c   ",left->name);
    printf("gnd %s L=%s W=%s\n",nmosm,nmosl,nmosw);

    right=entry->node->one;
    if(entry->node==root) {
      #ifdef ECRL
        printf("MN%cH gnd I%d_H ",
                          entry->node->name, entry->node->chooser);
      #endif
      #ifdef IECRL
        printf("MN%cH gnd I%d_H ",
                          entry->node->name, entry->node->chooser);
      #endif
      #ifdef PFAL
        printf("MN%cH vpc I%d_H ",
                          entry->node->name, entry->node->chooser);
      #endif
    } else
      printf("MN%cH %c   I%d_H ",
         entry->node->name, entry->node->name, entry->node->chooser);
    if (right==v0) {
      #ifdef ECRL
        printf("Z_H ");
      #endif
      #ifdef IECRL
        printf("Z_H ");
      #endif
      #ifdef PFAL
        printf("Z_L ");
      #endif
    } else if (right==v1) {
      #ifdef ECRL
        printf("Z_L ");
      #endif
      #ifdef IECRL
        printf("Z_L ");
      #endif
      #ifdef PFAL
        printf("Z_H ");
      #endif
    } else
    printf("%c   ",right->name);
  printf("gnd %s L=%s W=%s\n",nmosm,nmosl,nmosw);
  return totals+1;
  }
}

/*
  To free the nodes (avoiding excessive memory usage) both the
  linked list and the ROBDD tree nodes need to be removed.
*/
void free_table(struct table *entry)
{
  struct table *tab;

  while (entry->next!=NULL) {
    free(entry->node);
```

```
    tab=entry;
    entry=entry->next;
    free(tab);
  }
  free(entry);
}

void print_spice_footer(struct table *entry, struct node *root)
{
  while (entry->node!=NULL)
    entry=entry->next;
  if (root->chooser==0) {
    if (root==v0) {
      #ifdef ECRL
        printf("R0 gnd Z_H 1m\n"); /*Zero function*/
      #endif
      #ifdef IECRL
        printf("R0 gnd Z_H 1m\n"); /*Zero function*/
      #endif
      #ifdef PFAL
        printf("R0 vpc Z_H 1m\n"); /*Zero function:PFAL*/
      #endif
      printf(".Ends\n");
    } else if (root==v1) {
      #ifdef ECRL
        printf("R1 gnd Z_L 1m\n"); /*Ones function*/
      #endif
      #ifdef IECRL
        printf("R1 gnd Z_L 1m\n"); /*Ones function*/
      #endif
      #ifdef PFAL
        printf("R1 vpc Z_L 1m\n"); /*Ones function:PFAL*/
      #endif
      printf(".Ends\n");
    } else {
      printf("ERROR (print spice footer)\n");
    }
  } else {
    printf(".Ends\n");
  }
}

int main ()
{
  struct node *root;
  struct table *tab;
  struct expr expr;

  int i;
  int j;
  #if DEBUG
    int t;
  #endif

  int decs;
  int mindecs;
  int bestj;
```

```
int paths;
int minpaths;

int qval[32]={
  0x7670, 0x2F4A, 0x3C2D, 0x945E,
  0x2B8B, 0xA6B8, 0xC1D9, 0xF0A3,
  0xA633, 0xD83A, 0x3CA6, 0xD6E0,
  0xE8E4, 0x6761, 0xF306, 0xA1CB,
  0x3CE1, 0xAF84, 0x1F13, 0x7926,
  0x935C, 0x73A8, 0x4DD8, 0x546D,
  0x3A27, 0x254F, 0xF461, 0x4375,
  0xF630, 0x85B9, 0x2BF0, 0xCB13};

int q_equivs[32][2]={
  {0x035F, 0x1F13}, {0x036F, 0x2B8B},
  {0x036F, 0xE8E4}, {0x036F, 0xF630},
  {0x037E, 0x254F}, {0x037E, 0x2F4A},
  {0x03D7, 0x7670}, {0x03D7, 0xAF84},
  {0x03DD, 0xF0A3}, {0x03DE, 0xA633},
  {0x03DE, 0xF306}, {0x067B, 0x546D},
  {0x067B, 0xD6E0}, {0x067B, 0xF461},
  {0x067E, 0x4DD8}, {0x06B7, 0x3A27},
  {0x06B7, 0x4375}, {0x06BD, 0xA6B8},
  {0x077A, 0x6761}, {0x07B5, 0x73A8},
  {0x07B5, 0xC1D9}, {0x07E3, 0xCB13},
  {0x07E6, 0xD83A}, {0x07F1, 0x2BF0},
  {0x07F8, 0x3C2D}, {0x169B, 0x945E},
  {0x16AD, 0x7926}, {0x16BC, 0x3CA6},
  {0x16BC, 0x935C}, {0x179A, 0xA1CB},
  {0x17AC, 0x85B9}, {0x19E6, 0x3CE1}};

int equ;

v0=obdd_alloc();
v0->chooser=0;
v0->zero=NULL;
v0->one=NULL;
v0->name='0';

v1=obdd_alloc();
v1->chooser=0;
v1->zero=NULL;
v1->one=NULL;
v1->name='1';

t0=talloc();
t0->node=NULL;
t0->next=t0;

node_name='A';

expr.size=SIZE;
for (i=0;i<32;i++){
  expr.value=qval[i];
  /* Repeat for every order to find minimum F/OBDD tree */
  mindecs=OTFUN;
  minpaths=OTFUN;
  bestj=0;
```

```
    for (j=0;j<factorial(LEVELS);j++) {
      tab=talloc();
      tab->node=NULL;
      tab->next=NULL;
      fact_ord_nr(j, LEVELS);
      node_name='A';
      #if DEBUG
        printf("MN: i %x  j %x  tab %p  root %p\n", i, j, tab, root);
        for (t=0;t<LEVELS;t++)
          printf("MN: pv%d %d lv%d %d\n",
                       t,var_ord[t].physical,t,var_ord[t].logical);
        printf("MN: Levels %d\n",LEVELS);
      #endif
      root=robdd_build(tab,expr,LEVELS);
      decs=table_count(tab);
      paths=obdd_count(root);
      #if DEBUG
        printf("MN: j %d decs %d mind %d best %d\n",
                      j,      decs,   mindecs, bestj);
      #endif
      if (decs<mindecs) {
        mindecs=decs;bestj=j;
      }
      if (decs==mindecs && paths<minpaths) {
        minpaths=paths;bestj=j;
      }

      free_table(tab);
      node_name='A';
    }
    tab=talloc();
    tab->node=NULL;
    tab->next=NULL;
    fact_ord_nrp(bestj, LEVELS);
    node_name='A';
    root=robdd_build(tab,expr,LEVELS);
    obdd_print_spice(qval[i],tab,root);
    decs=table_count(tab);
    paths=obdd_count(root);
    print_spice_footer(tab,root);
    printf("* Value: %X Order: %2d Nodes: %d Paths: %2d\n",
                    qval[i],  bestj,     decs,     paths);
    equ=0;
    while (q_equivs[equ][1]!=qval[i])
      equ++;
    printf("* Base:  %04X Equivs: ",q_equivs[equ][0]);
    for (j=0;j<32;j++)
      if (q_equivs[equ][0]==q_equivs[j][0])
        printf("%X ", q_equivs[j][1]);
    printf("\n\n");
    free_table(tab);
    node_name='A';
  }
  return 0;
}
```

Appendix C SPICE source-code

C.1 SPICE for q-boxes

```
.SUBCKT xor2 I1_H I1_L I2_H I2_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNCL vpc I1_L A   gnd N L=0.35u W=0.50u
MNCH vpc I1_H B   gnd N L=0.35u W=0.50u
MNBL B   I2_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I2_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I2_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I2_H Z_H gnd N L=0.35u W=0.50u
.ENDS

.SUBCKT xor3 I1_H I1_L I2_H I2_L I3_H I3_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNEL vpc I1_L C   gnd N L=0.35u W=0.50u
MNEH vpc I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L B   gnd N L=0.35u W=0.50u
MNDH D   I2_H A   gnd N L=0.35u W=0.50u
MNCL C   I2_L A   gnd N L=0.35u W=0.50u
MNCH C   I2_H B   gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I3_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I3_H Z_H gnd N L=0.35u W=0.50u
.ENDS

* Logical  0010 Physical 1243
.SUBCKT q0t0b0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNEL vpc I1_L C   gnd N L=0.35u W=0.50u
MNEH vpc I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L B   gnd N L=0.35u W=0.50u
MNDH D   I2_H A   gnd N L=0.35u W=0.50u
MNCL C   I2_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I2_H B   gnd N L=0.35u W=0.50u
MNBL B   I4_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I4_H A   gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I3_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: 7670 Order: 12 Nodes: 5 Paths:  9
* Base:  03D7 Equivs: 7670 AF84
```

```
* Logical  2000 Physical 3124
.SUBCKT q0t0b1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I3_L E   gnd N L=0.35u W=0.50u
MNGH vpc I3_H F   gnd N L=0.35u W=0.50u
MNFL F   I1_L Z_H gnd N L=0.35u W=0.50u
MNFH F   I1_H A   gnd N L=0.35u W=0.50u
MNEL E   I1_L B   gnd N L=0.35u W=0.50u
MNEH E   I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L C   gnd N L=0.35u W=0.50u
MNDH D   I2_H Z_L gnd N L=0.35u W=0.50u
MNCL C   I4_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I4_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I2_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I2_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: 2F4A Order:  2 Nodes: 7 Paths:  9
* Base:  037E Equivs: 254F 2F4A

* Logical  3000 Physical 4123
.SUBCKT q0t0b2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNEL vpc I4_L C   gnd N L=0.35u W=0.50u
MNEH vpc I4_H D   gnd N L=0.35u W=0.50u
MNDL D   I1_L C   gnd N L=0.35u W=0.50u
MNDH D   I1_H A   gnd N L=0.35u W=0.50u
MNCL C   I2_L A   gnd N L=0.35u W=0.50u
MNCH C   I2_H B   gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I3_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I3_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 3C2D Order:  3 Nodes: 5 Paths: 10
* Base:  07F8 Equivs: 3C2D

* Logical  2000 Physical 3124
.SUBCKT q0t0b3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNHL vpc I3_L E   gnd N L=0.35u W=0.50u
MNHH vpc I3_H G   gnd N L=0.35u W=0.50u
MNGL G   I1_L C   gnd N L=0.35u W=0.50u
MNGH G   I1_H F   gnd N L=0.35u W=0.50u
MNFL F   I2_L Z_H gnd N L=0.35u W=0.50u
```

```
MNFH F   I2_H Z_L gnd N L=0.35u W=0.50u
MNEL E   I1_L B   gnd N L=0.35u W=0.50u
MNEH E   I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L C   gnd N L=0.35u W=0.50u
MNDH D   I2_H Z_H gnd N L=0.35u W=0.50u
MNCL C   I4_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I4_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I2_L A   gnd N L=0.35u W=0.50u
MNBH B   I2_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: 945E Order:  2 Nodes: 8 Paths: 10
* Base:  169B Equivs: 945E

* Logical  0200 Physical 1423
.SUBCKT q0t1b0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNFL vpc I1_L D   gnd N L=0.35u W=0.50u
MNFH vpc I1_H E   gnd N L=0.35u W=0.50u
MNEL E   I2_L Z_L gnd N L=0.35u W=0.50u
MNEH E   I2_H A   gnd N L=0.35u W=0.50u
MNDL D   I4_L B   gnd N L=0.35u W=0.50u
MNDH D   I4_H C   gnd N L=0.35u W=0.50u
MNCL C   I2_L Z_H gnd N L=0.35u W=0.50u
MNCH C   I2_H A   gnd N L=0.35u W=0.50u
MNBL B   I2_L A   gnd N L=0.35u W=0.50u
MNBH B   I2_H Z_H gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I3_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 2B8B Order:  8 Nodes: 6 Paths:  9
* Base:  036F Equivs: 2B8B E8E4 F630

* Logical  3100 Physical 4213
.SUBCKT q0t1b1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I4_L E   gnd N L=0.35u W=0.50u
MNGH vpc I4_H F   gnd N L=0.35u W=0.50u
MNFL F   I2_L D   gnd N L=0.35u W=0.50u
MNFH F   I2_H A   gnd N L=0.35u W=0.50u
MNEL E   I2_L C   gnd N L=0.35u W=0.50u
MNEH E   I2_H D   gnd N L=0.35u W=0.50u
MNDL D   I1_L Z_H gnd N L=0.35u W=0.50u
MNDH D   I1_H Z_L gnd N L=0.35u W=0.50u
MNCL C   I1_L A   gnd N L=0.35u W=0.50u
MNCH C   I1_H B   gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I3_H Z_H gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_H gnd N L=0.35u W=0.50u
```

```
MNAH A   I3_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: A6B8 Order:  7 Nodes: 7 Paths: 10
* Base:  06BD Equivs: A6B8

* Logical  0000 Physical 1234
.SUBCKT q0t1b2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I1_L C   gnd N L=0.35u W=0.50u
MNGH vpc I1_H F   gnd N L=0.35u W=0.50u
MNFL F   I2_L D   gnd N L=0.35u W=0.50u
MNFH F   I2_H E   gnd N L=0.35u W=0.50u
MNEL E   I3_L A   gnd N L=0.35u W=0.50u
MNEH E   I3_H Z_H gnd N L=0.35u W=0.50u
MNDL D   I3_L Z_H gnd N L=0.35u W=0.50u
MNDH D   I3_H Z_L gnd N L=0.35u W=0.50u
MNCL C   I2_L B   gnd N L=0.35u W=0.50u
MNCH C   I2_H Z_L gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I3_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: C1D9 Order:  0 Nodes: 7 Paths:  9
* Base:  07B5 Equivs: 73A8 C1D9

* Logical  2000 Physical 3124
.SUBCKT q0t1b3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNEL vpc I3_L B   gnd N L=0.35u W=0.50u
MNEH vpc I3_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L Z_L gnd N L=0.35u W=0.50u
MNDH D   I2_H C   gnd N L=0.35u W=0.50u
MNCL C   I4_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I4_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I1_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I1_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: F0A3 Order:  2 Nodes: 5 Paths:  6
* Base:  03DD Equivs: F0A3

* Logical  3200 Physical 4312
.SUBCKT q0t2b0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
```

```
MNFL vpc I4_L E   gnd N L=0.35u W=0.50u
MNFH vpc I4_H B   gnd N L=0.35u W=0.50u
MNEL E   I3_L A   gnd N L=0.35u W=0.50u
MNEH E   I3_H D   gnd N L=0.35u W=0.50u
MNDL D   I1_L B   gnd N L=0.35u W=0.50u
MNDH D   I1_H C   gnd N L=0.35u W=0.50u
MNCL C   I2_L Z_H gnd N L=0.35u W=0.50u
MNCH C   I2_H Z_L gnd N L=0.35u W=0.50u
MNBL B   I2_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I2_H Z_H gnd N L=0.35u W=0.50u
MNAL A   I1_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I1_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: A633 Order: 11 Nodes: 6 Paths:  8
* Base:  03DE Equivs: A633 F306

* Logical  0000 Physical 1234
.SUBCKT q0t2b1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNHL vpc I1_L D   gnd N L=0.35u W=0.50u
MNHH vpc I1_H G   gnd N L=0.35u W=0.50u
MNGL G   I2_L E   gnd N L=0.35u W=0.50u
MNGH G   I2_H F   gnd N L=0.35u W=0.50u
MNFL F   I3_L Z_H gnd N L=0.35u W=0.50u
MNFH F   I3_H Z_L gnd N L=0.35u W=0.50u
MNEL E   I3_L A   gnd N L=0.35u W=0.50u
MNEH E   I3_H Z_L gnd N L=0.35u W=0.50u
MNDL D   I2_L B   gnd N L=0.35u W=0.50u
MNDH D   I2_H C   gnd N L=0.35u W=0.50u
MNCL C   I4_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I4_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I3_L A   gnd N L=0.35u W=0.50u
MNBH B   I3_H Z_H gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: D83A Order:  0 Nodes: 8 Paths: 10
* Base:  07E6 Equivs: D83A

* Logical  3000 Physical 4123
.SUBCKT q0t2b2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I4_L C   gnd N L=0.35u W=0.50u
MNGH vpc I4_H F   gnd N L=0.35u W=0.50u
MNFL F   I1_L D   gnd N L=0.35u W=0.50u
MNFH F   I1_H E   gnd N L=0.35u W=0.50u
MNEL E   I2_L A   gnd N L=0.35u W=0.50u
MNEH E   I2_H Z_L gnd N L=0.35u W=0.50u
MNDL D   I2_L B   gnd N L=0.35u W=0.50u
MNDH D   I2_H Z_H gnd N L=0.35u W=0.50u
```

```
MNCL C   I2_L A   gnd N L=0.35u W=0.50u
MNCH C   I2_H B   gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I3_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I3_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 3CA6 Order:  3 Nodes: 7 Paths: 10
* Base:  16BC Equivs: 3CA6 935C

* Logical  1100 Physical 2314
.SUBCKT q0t2b3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNHL vpc I2_L C   gnd N L=0.35u W=0.50u
MNHH vpc I2_H G   gnd N L=0.35u W=0.50u
MNGL G   I3_L E   gnd N L=0.35u W=0.50u
MNGH G   I3_H F   gnd N L=0.35u W=0.50u
MNFL F   I1_L A   gnd N L=0.35u W=0.50u
MNFH F   I1_H Z_L gnd N L=0.35u W=0.50u
MNEL E   I1_L D   gnd N L=0.35u W=0.50u
MNEH E   I1_H A   gnd N L=0.35u W=0.50u
MNDL D   I4_L Z_L gnd N L=0.35u W=0.50u
MNDH D   I4_H Z_H gnd N L=0.35u W=0.50u
MNCL C   I3_L Z_H gnd N L=0.35u W=0.50u
MNCH C   I3_H B   gnd N L=0.35u W=0.50u
MNBL B   I1_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I1_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: D6E0 Order:  5 Nodes: 8 Paths: 11
* Base:  067B Equivs: 546D D6E0 F461

* Logical  1200 Physical 2413
.SUBCKT q0t3b0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNFL vpc I2_L D   gnd N L=0.35u W=0.50u
MNFH vpc I2_H E   gnd N L=0.35u W=0.50u
MNEL E   I1_L A   gnd N L=0.35u W=0.50u
MNEH E   I1_H Z_L gnd N L=0.35u W=0.50u
MNDL D   I4_L B   gnd N L=0.35u W=0.50u
MNDH D   I4_H C   gnd N L=0.35u W=0.50u
MNCL C   I1_L A   gnd N L=0.35u W=0.50u
MNCH C   I1_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I1_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I1_H A   gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I3_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: E8E4 Order:  9 Nodes: 6 Paths:  9
```

```
* Base:  036F Equivs: 2B8B E8E4 F630

* Logical  0000 Physical 1234
.SUBCKT q0t3b1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNFL vpc I1_L C   gnd N L=0.35u W=0.50u
MNFH vpc I1_H E   gnd N L=0.35u W=0.50u
MNEL E   I2_L B   gnd N L=0.35u W=0.50u
MNEH E   I2_H D   gnd N L=0.35u W=0.50u
MNDL D   I3_L Z_L gnd N L=0.35u W=0.50u
MNDH D   I3_H Z_H gnd N L=0.35u W=0.50u
MNCL C   I2_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I2_H B   gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I3_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: 6761 Order:  0 Nodes: 6 Paths:  9
* Base:  077A Equivs: 6761

* Logical  2000 Physical 3124
.SUBCKT q0t3b2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNFL vpc I3_L A   gnd N L=0.35u W=0.50u
MNFH vpc I3_H E   gnd N L=0.35u W=0.50u
MNEL E   I1_L B   gnd N L=0.35u W=0.50u
MNEH E   I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L C   gnd N L=0.35u W=0.50u
MNDH D   I2_H A   gnd N L=0.35u W=0.50u
MNCL C   I4_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I4_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I2_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I2_H Z_H gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: F306 Order:  2 Nodes: 6 Paths:  8
* Base:  03DE Equivs: A633 F306

* Logical  2000 Physical 3124
.SUBCKT q0t3b3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNHL vpc I3_L E   gnd N L=0.35u W=0.50u
MNHH vpc I3_H G   gnd N L=0.35u W=0.50u
MNGL G   I1_L C   gnd N L=0.35u W=0.50u
```

```
MNGH G   I1_H F   gnd N L=0.35u W=0.50u
MNFL F   I2_L Z_L gnd N L=0.35u W=0.50u
MNFH F   I2_H Z_H gnd N L=0.35u W=0.50u
MNEL E   I1_L B   gnd N L=0.35u W=0.50u
MNEH E   I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L C   gnd N L=0.35u W=0.50u
MNDH D   I2_H Z_L gnd N L=0.35u W=0.50u
MNCL C   I4_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I4_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I2_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I2_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: A1CB Order:  2 Nodes: 8 Paths: 10
* Base:  179A Equivs: A1CB

* Logical  3000 Physical 4123
.SUBCKT q1t0b0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNFL vpc I4_L C   gnd N L=0.35u W=0.50u
MNFH vpc I4_H E   gnd N L=0.35u W=0.50u
MNEL E   I1_L B   gnd N L=0.35u W=0.50u
MNEH E   I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L B   gnd N L=0.35u W=0.50u
MNDH D   I2_H A   gnd N L=0.35u W=0.50u
MNCL C   I2_L A   gnd N L=0.35u W=0.50u
MNCH C   I2_H B   gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I3_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I3_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 3CE1 Order:  3 Nodes: 6 Paths: 10
* Base:  19E6 Equivs: 3CE1

* Logical  2000 Physical 3124
.SUBCKT q1t0b1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNEL vpc I3_L C   gnd N L=0.35u W=0.50u
MNEH vpc I3_H D   gnd N L=0.35u W=0.50u
MNDL D   I1_L A   gnd N L=0.35u W=0.50u
MNDH D   I1_H B   gnd N L=0.35u W=0.50u
MNCL C   I1_L B   gnd N L=0.35u W=0.50u
MNCH C   I1_H Z_L gnd N L=0.35u W=0.50u
MNBL B   I2_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I2_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
```

```
* Value: AF84 Order:  2 Nodes: 5 Paths:  9
* Base:  03D7 Equivs: 7670 AF84

* Logical  1200 Physical 2413
.SUBCKT q1t0b2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNDL vpc I2_L B   gnd N L=0.35u W=0.50u
MNDH vpc I2_H C   gnd N L=0.35u W=0.50u
MNCL C   I1_L A   gnd N L=0.35u W=0.50u
MNCH C   I1_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I4_L A   gnd N L=0.35u W=0.50u
MNBH B   I4_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I3_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 1F13 Order:  9 Nodes: 4 Paths:  6
* Base:  035F Equivs: 1F13

* Logical  0000 Physical 1234
.SUBCKT q1t0b3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNHL vpc I1_L E   gnd N L=0.35u W=0.50u
MNHH vpc I1_H G   gnd N L=0.35u W=0.50u
MNGL G   I2_L F   gnd N L=0.35u W=0.50u
MNGH G   I2_H A   gnd N L=0.35u W=0.50u
MNFL F   I3_L A   gnd N L=0.35u W=0.50u
MNFH F   I3_H C   gnd N L=0.35u W=0.50u
MNEL E   I2_L B   gnd N L=0.35u W=0.50u
MNEH E   I2_H D   gnd N L=0.35u W=0.50u
MNDL D   I3_L Z_H gnd N L=0.35u W=0.50u
MNDH D   I3_H C   gnd N L=0.35u W=0.50u
MNCL C   I4_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I4_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I3_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: 7926 Order:  0 Nodes: 8 Paths: 12
* Base:  16AD Equivs: 7926

* Logical  2000 Physical 3124
.SUBCKT q1t1b0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I3_L E   gnd N L=0.35u W=0.50u
MNGH vpc I3_H F   gnd N L=0.35u W=0.50u
```

```
MNFL F   I2_L C   gnd N L=0.35u W=0.50u
MNFH F   I2_H A   gnd N L=0.35u W=0.50u
MNEL E   I1_L B   gnd N L=0.35u W=0.50u
MNEH E   I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L C   gnd N L=0.35u W=0.50u
MNDH D   I2_H Z_H gnd N L=0.35u W=0.50u
MNCL C   I4_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I4_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I2_L A   gnd N L=0.35u W=0.50u
MNBH B   I2_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: 935C Order:  2 Nodes: 7 Paths: 10
* Base:  16BC Equivs: 3CA6 935C

* Logical  3000 Physical 4123
.SUBCKT q1t1b1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I4_L D   gnd N L=0.35u W=0.50u
MNGH vpc I4_H F   gnd N L=0.35u W=0.50u
MNFL F   I1_L E   gnd N L=0.35u W=0.50u
MNFH F   I1_H Z_L gnd N L=0.35u W=0.50u
MNEL E   I2_L Z_H gnd N L=0.35u W=0.50u
MNEH E   I2_H B   gnd N L=0.35u W=0.50u
MNDL D   I1_L A   gnd N L=0.35u W=0.50u
MNDH D   I1_H C   gnd N L=0.35u W=0.50u
MNCL C   I2_L B   gnd N L=0.35u W=0.50u
MNCH C   I2_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I3_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I2_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I2_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 73A8 Order:  3 Nodes: 7 Paths:  9
* Base:  07B5 Equivs: 73A8 C1D9

* Logical  2200 Physical 3412
.SUBCKT q1t1b2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I3_L D   gnd N L=0.35u W=0.50u
MNGH vpc I3_H F   gnd N L=0.35u W=0.50u
MNFL F   I4_L C   gnd N L=0.35u W=0.50u
MNFH F   I4_H E   gnd N L=0.35u W=0.50u
MNEL E   I1_L A   gnd N L=0.35u W=0.50u
MNEH E   I1_H Z_L gnd N L=0.35u W=0.50u
MNDL D   I4_L B   gnd N L=0.35u W=0.50u
MNDH D   I4_H C   gnd N L=0.35u W=0.50u
MNCL C   I1_L A   gnd N L=0.35u W=0.50u
MNCH C   I1_H Z_H gnd N L=0.35u W=0.50u
```

```
MNBL B   I1_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I1_H A   gnd N L=0.35u W=0.50u
MNAL A   I2_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I2_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: 4DD8 Order: 10 Nodes: 7 Paths: 12
* Base:  067E Equivs: 4DD8

* Logical  0000 Physical 1234
.SUBCKT q1t1b3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNHL vpc I1_L D   gnd N L=0.35u W=0.50u
MNHH vpc I1_H G   gnd N L=0.35u W=0.50u
MNGL G   I2_L Z_H gnd N L=0.35u W=0.50u
MNGH G   I2_H F   gnd N L=0.35u W=0.50u
MNFL F   I3_L E   gnd N L=0.35u W=0.50u
MNFH F   I3_H A   gnd N L=0.35u W=0.50u
MNEL E   I4_L Z_H gnd N L=0.35u W=0.50u
MNEH E   I4_H Z_L gnd N L=0.35u W=0.50u
MNDL D   I2_L B   gnd N L=0.35u W=0.50u
MNDH D   I2_H C   gnd N L=0.35u W=0.50u
MNCL C   I3_L A   gnd N L=0.35u W=0.50u
MNCH C   I3_H Z_L gnd N L=0.35u W=0.50u
MNBL B   I3_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I3_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 546D Order:  0 Nodes: 8 Paths: 11
* Base:  067B Equivs: 546D D6E0 F461

* Logical  2200 Physical 3412
.SUBCKT q1t2b0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I3_L C   gnd N L=0.35u W=0.50u
MNGH vpc I3_H F   gnd N L=0.35u W=0.50u
MNFL F   I4_L D   gnd N L=0.35u W=0.50u
MNFH F   I4_H E   gnd N L=0.35u W=0.50u
MNEL E   I1_L A   gnd N L=0.35u W=0.50u
MNEH E   I1_H Z_H gnd N L=0.35u W=0.50u
MNDL D   I1_L Z_H gnd N L=0.35u W=0.50u
MNDH D   I1_H Z_L gnd N L=0.35u W=0.50u
MNCL C   I4_L A   gnd N L=0.35u W=0.50u
MNCH C   I4_H B   gnd N L=0.35u W=0.50u
MNBL B   I1_L A   gnd N L=0.35u W=0.50u
MNBH B   I1_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I2_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I2_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 3A27 Order: 10 Nodes: 7 Paths: 10
```

```
* Base:  06B7 Equivs: 3A27 4375

* Logical  2000 Physical 3124
.SUBCKT q1t2b1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I3_L E   gnd N L=0.35u W=0.50u
MNGH vpc I3_H F   gnd N L=0.35u W=0.50u
MNFL F   I1_L C   gnd N L=0.35u W=0.50u
MNFH F   I1_H Z_H gnd N L=0.35u W=0.50u
MNEL E   I1_L B   gnd N L=0.35u W=0.50u
MNEH E   I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I2_L C   gnd N L=0.35u W=0.50u
MNDH D   I2_H Z_L gnd N L=0.35u W=0.50u
MNCL C   I4_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I4_H Z_H gnd N L=0.35u W=0.50u
MNBL B   I2_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I2_H A   gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: 254F Order:  2 Nodes: 7 Paths:  9
* Base:  037E Equivs: 254F 2F4A

* Logical  2000 Physical 3124
.SUBCKT q1t2b2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNHL vpc I3_L D   gnd N L=0.35u W=0.50u
MNHH vpc I3_H G   gnd N L=0.35u W=0.50u
MNGL G   I1_L Z_L gnd N L=0.35u W=0.50u
MNGH G   I1_H F   gnd N L=0.35u W=0.50u
MNFL F   I2_L A   gnd N L=0.35u W=0.50u
MNFH F   I2_H E   gnd N L=0.35u W=0.50u
MNEL E   I4_L Z_L gnd N L=0.35u W=0.50u
MNEH E   I4_H Z_H gnd N L=0.35u W=0.50u
MNDL D   I1_L B   gnd N L=0.35u W=0.50u
MNDH D   I1_H C   gnd N L=0.35u W=0.50u
MNCL C   I2_L Z_H gnd N L=0.35u W=0.50u
MNCH C   I2_H A   gnd N L=0.35u W=0.50u
MNBL B   I2_L A   gnd N L=0.35u W=0.50u
MNBH B   I2_H Z_H gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: F461 Order:  2 Nodes: 8 Paths: 11
* Base:  067B Equivs: 546D D6E0 F461

* Logical  1110 Physical 2341
.SUBCKT q1t2b3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
```

```
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I2_L C   gnd N L=0.35u W=0.50u
MNGH vpc I2_H F   gnd N L=0.35u W=0.50u
MNFL F   I3_L D   gnd N L=0.35u W=0.50u
MNFH F   I3_H E   gnd N L=0.35u W=0.50u
MNEL E   I4_L Z_H gnd N L=0.35u W=0.50u
MNEH E   I4_H A   gnd N L=0.35u W=0.50u
MNDL D   I4_L Z_L gnd N L=0.35u W=0.50u
MNDH D   I4_H Z_H gnd N L=0.35u W=0.50u
MNCL C   I3_L A   gnd N L=0.35u W=0.50u
MNCH C   I3_H B   gnd N L=0.35u W=0.50u
MNBL B   I4_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I4_H A   gnd N L=0.35u W=0.50u
MNAL A   I1_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I1_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 4375 Order: 17 Nodes: 7 Paths: 10
* Base:  06B7 Equivs: 3A27 4375

* Logical  2200 Physical 3412
.SUBCKT q1t3b0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNFL vpc I3_L B   gnd N L=0.35u W=0.50u
MNFH vpc I3_H E   gnd N L=0.35u W=0.50u
MNEL E   I4_L D   gnd N L=0.35u W=0.50u
MNEH E   I4_H Z_L gnd N L=0.35u W=0.50u
MNDL D   I1_L A   gnd N L=0.35u W=0.50u
MNDH D   I1_H C   gnd N L=0.35u W=0.50u
MNCL C   I2_L Z_H gnd N L=0.35u W=0.50u
MNCH C   I2_H Z_L gnd N L=0.35u W=0.50u
MNBL B   I4_L Z_H gnd N L=0.35u W=0.50u
MNBH B   I4_H A   gnd N L=0.35u W=0.50u
MNAL A   I2_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I2_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: F630 Order: 10 Nodes: 6 Paths:  8
* Base:  036F Equivs: 2B8B E8E4 F630

* Logical  1110 Physical 2341
.SUBCKT q1t3b1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I2_L D   gnd N L=0.35u W=0.50u
MNGH vpc I2_H F   gnd N L=0.35u W=0.50u
MNFL F   I3_L E   gnd N L=0.35u W=0.50u
MNFH F   I3_H B   gnd N L=0.35u W=0.50u
MNEL E   I4_L Z_L gnd N L=0.35u W=0.50u
MNEH E   I4_H Z_H gnd N L=0.35u W=0.50u
MNDL D   I3_L A   gnd N L=0.35u W=0.50u
```

```
MNDH D   I3_H C   gnd N L=0.35u W=0.50u
MNCL C   I4_L B   gnd N L=0.35u W=0.50u
MNCH C   I4_H A   gnd N L=0.35u W=0.50u
MNBL B   I1_L Z_L gnd N L=0.35u W=0.50u
MNBH B   I1_H Z_H gnd N L=0.35u W=0.50u
MNAL A   I1_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I1_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: 85B9 Order: 17 Nodes: 7 Paths: 10
* Base:  17AC Equivs: 85B9

* Logical  3000 Physical 4123
.SUBCKT q1t3b2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNFL vpc I4_L D   gnd N L=0.35u W=0.50u
MNFH vpc I4_H E   gnd N L=0.35u W=0.50u
MNEL E   I3_L Z_H gnd N L=0.35u W=0.50u
MNEH E   I3_H Z_L gnd N L=0.35u W=0.50u
MNDL D   I1_L B   gnd N L=0.35u W=0.50u
MNDH D   I1_H C   gnd N L=0.35u W=0.50u
MNCL C   I2_L Z_L gnd N L=0.35u W=0.50u
MNCH C   I2_H A   gnd N L=0.35u W=0.50u
MNBL B   I2_L A   gnd N L=0.35u W=0.50u
MNBH B   I2_H Z_H gnd N L=0.35u W=0.50u
MNAL A   I3_L Z_L gnd N L=0.35u W=0.50u
MNAH A   I3_H Z_H gnd N L=0.35u W=0.50u
.ENDS
* Value: 2BF0 Order:  3 Nodes: 6 Paths:  8
* Base:  07F1 Equivs: 2BF0

* Logical  1100 Physical 2314
.SUBCKT q1t3b3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z_H Z_L vpc gnd
MP0  Z_L Z_H  vpc vpc P L=0.35u W=0.50u
MP1  Z_H Z_L  vpc vpc P L=0.35u W=0.50u
MN0  Z_L Z_H  gnd gnd N L=0.35u W=0.50u
MN1  Z_H Z_L  gnd gnd N L=0.35u W=0.50u
MNGL vpc I2_L C   gnd N L=0.35u W=0.50u
MNGH vpc I2_H F   gnd N L=0.35u W=0.50u
MNFL F   I3_L E   gnd N L=0.35u W=0.50u
MNFH F   I3_H Z_H gnd N L=0.35u W=0.50u
MNEL E   I1_L Z_L gnd N L=0.35u W=0.50u
MNEH E   I1_H D   gnd N L=0.35u W=0.50u
MNDL D   I4_L Z_L gnd N L=0.35u W=0.50u
MNDH D   I4_H Z_H gnd N L=0.35u W=0.50u
MNCL C   I3_L A   gnd N L=0.35u W=0.50u
MNCH C   I3_H B   gnd N L=0.35u W=0.50u
MNBL B   I1_L A   gnd N L=0.35u W=0.50u
MNBH B   I1_H Z_L gnd N L=0.35u W=0.50u
MNAL A   I4_L Z_H gnd N L=0.35u W=0.50u
MNAH A   I4_H Z_L gnd N L=0.35u W=0.50u
.ENDS
* Value: CB13 Order:  5 Nodes: 7 Paths:  9
* Base:  07E3 Equivs: CB13
```

```
*.Include ./xor_cells.cdl
*.Include ./q_cells.cdl

.SUBCKT q0t0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L
+ Z0_H Z0_L Z1_H Z1_L Z2_H Z2_L Z3_H Z3_L
+ vpc gnd
XX4X7670 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z0_H Z0_L vpc gnd
+ q0t0b0
XX4X2F4A I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z1_H Z1_L vpc gnd
+ q0t0b1
XX4X3C2D I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z2_H Z2_L vpc gnd
+ q0t0b2
XX4X945E I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z3_H Z3_L vpc gnd
+ q0t0b3
.ENDS

.SUBCKT q0t1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L
+ Z0_H Z0_L Z1_H Z1_L Z2_H Z2_L Z3_H Z3_L
+ vpc gnd
XX4X2B8B I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z0_H Z0_L vpc gnd
+ q0t1b0
XX4XA6B8 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z1_H Z1_L vpc gnd
+ q0t1b1
XX4XC1D9 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z2_H Z2_L vpc gnd
+ q0t1b2
XX4XF0A3 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z3_H Z3_L vpc gnd
+ q0t1b3
.ENDS

.SUBCKT q0t2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L
+ Z0_H Z0_L Z1_H Z1_L Z2_H Z2_L Z3_H Z3_L
+ vpc gnd
XX4XA633 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z0_H Z0_L vpc gnd
+ q0t2b0
XX4XD83A I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z1_H Z1_L vpc gnd
+ q0t2b1
XX4X3CA6 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z2_H Z2_L vpc gnd
+ q0t2b2
XX4XD6E0 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z3_H Z3_L vpc gnd
+ q0t2b3
.ENDS

.SUBCKT q0t3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L
+ Z0_H Z0_L Z1_H Z1_L Z2_H Z2_L Z3_H Z3_L
+ vpc gnd
XX4XE8E4 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z0_H Z0_L vpc gnd
+ q0t3b0
XX4X6761 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z1_H Z1_L vpc gnd
+ q0t3b1
XX4XF306 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z2_H Z2_L vpc gnd
+ q0t3b2
XX4XA1CB I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z3_H Z3_L vpc gnd
+ q0t3b3
```

```
.ENDS

.SUBCKT q1t0
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L
+ Z0_H Z0_L Z1_H Z1_L Z2_H Z2_L Z3_H Z3_L
+ vpc gnd
XX4X3CE1 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z0_H Z0_L vpc gnd
+ q1t0b0
XX4XAF84 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z1_H Z1_L vpc gnd
+ q1t0b1
XX4X1F13 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z2_H Z2_L vpc gnd
+ q1t0b2
XX4X7926 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z3_H Z3_L vpc gnd
+ q1t0b3
.ENDS

.SUBCKT q1t1
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L
+ Z0_H Z0_L Z1_H Z1_L Z2_H Z2_L Z3_H Z3_L
+ vpc gnd
XX4X935C I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z0_H Z0_L vpc gnd
+ q1t1b0
XX4X73A8 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z1_H Z1_L vpc gnd
+ q1t1b1
XX4X4DD8 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z2_H Z2_L vpc gnd
+ q1t1b2
XX4X546D I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z3_H Z3_L vpc gnd
+ q1t1b3
.ENDS

.SUBCKT q1t2
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L
+ Z0_H Z0_L Z1_H Z1_L Z2_H Z2_L Z3_H Z3_L
+ vpc gnd
XX4X3A27 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z0_H Z0_L vpc gnd
+ q1t2b0
XX4X254F I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z1_H Z1_L vpc gnd
+ q1t2b1
XX4XF461 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z2_H Z2_L vpc gnd
+ q1t2b2
XX4X4375 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z3_H Z3_L vpc gnd
+ q1t2b3
.ENDS

.SUBCKT q1t3
+ I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L
+ Z0_H Z0_L Z1_H Z1_L Z2_H Z2_L Z3_H Z3_L
+ vpc gnd
XX4XF630 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z0_H Z0_L vpc gnd
+ q1t3b0
XX4X85B9 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z1_H Z1_L vpc gnd
+ q1t3b1
XX4X2BF0 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z2_H Z2_L vpc gnd
+ q1t3b2
XX4XCB13 I1_H I1_L I2_H I2_L I3_H I3_L I4_H I4_L Z3_H Z3_L vpc gnd
+ q1t3b3
.ENDS
```

```
.SUBCKT q0
+ X0_H X0_L X1_H X1_L X2_H X2_L X3_H X3_L
+ X4_H X4_L X5_H X5_L X6_H X6_L X7_H X7_L
+ Y0_H Y0_L Y1_H Y1_L Y2_H Y2_L Y3_H Y3_L
+ Y4_H Y4_L Y5_H Y5_L Y6_H Y6_L Y7_H Y7_L
+ vpc0 vpc1 vpc2 vpc3 gnd

Xa03 X7_H X7_L X3_H X3_L           a13_H a13_L vpc0 gnd xor2
Xa02 X2_H X2_L X6_H X6_L           a12_H a12_L vpc0 gnd xor2
Xa01 X5_H X5_L X1_H X1_L           a11_H a11_L vpc0 gnd xor2
Xa00 X0_H X0_L X4_H X4_L           a10_H a10_L vpc0 gnd xor2

Xb03 X7_H X7_L X0_H X0_L X4_H X4_L b13_H b13_L vpc0 gnd xor3
Xb02 X6_H X6_L X3_H X3_L           b12_H b12_L vpc0 gnd xor2
Xb01 X5_H X5_L X2_H X2_L           b11_H b11_L vpc0 gnd xor2
Xb00 X1_H X1_L X4_H X4_L           b10_H b10_L vpc0 gnd xor2

Xqt0
+ a10_H a10_L a11_H a11_L a12_H a12_L a13_H a13_L
+ a20_H a20_L a21_H a21_L a22_H a22_L a23_H a23_L
+ vpc1 gnd q0t0
Xqt1
+ b10_H b10_L b11_H b11_L b12_H b12_L b13_H b13_L
+ b20_H b20_L b21_H b21_L b22_H b22_L b23_H b23_L
+ vpc1 gnd q0t1

Xa13 a23_H a23_L b23_H b23_L             a33_H a33_L vpc2 gnd xor2
Xa12 b22_H b22_L a22_H a22_L             a32_H a32_L vpc2 gnd xor2
Xa11 a21_H a21_L b21_H b21_L             a31_H a31_L vpc2 gnd xor2
Xa10 b20_H b20_L a20_H a20_L             a30_H a30_L vpc2 gnd xor2

Xb13 a23_H a23_L b20_H b20_L a20_H a20_L b33_H b33_L vpc2 gnd xor3
Xb12 a22_H a22_L b23_H b23_L             b32_H b32_L vpc2 gnd xor2
Xb11 a21_H a21_L b22_H b22_L             b31_H b31_L vpc2 gnd xor2
Xb10 b21_H b21_L a20_H a20_L             b30_H b30_L vpc2 gnd xor2

Xqt2
+ a30_H a30_L a31_H a31_L a32_H a32_L a33_H a33_L
+  Y0_H  Y0_L  Y1_H  Y1_L  Y2_H  Y2_L  Y3_H  Y3_L
+ vpc3 gnd q0t2
Xqt3
+ b30_H b30_L b31_H b31_L b32_H b32_L b33_H b33_L
+  Y4_H  Y4_L  Y5_H  Y5_L  Y6_H  Y6_L  Y7_H  Y7_L
+ vpc3 gnd q0t3
.ENDS

.SUBCKT q1
+ X0_H X0_L X1_H X1_L X2_H X2_L X3_H X3_L
+ X4_H X4_L X5_H X5_L X6_H X6_L X7_H X7_L
+ Y0_H Y0_L Y1_H Y1_L Y2_H Y2_L Y3_H Y3_L
+ Y4_H Y4_L Y5_H Y5_L Y6_H Y6_L Y7_H Y7_L
+ vpc0 vpc1 vpc2 vpc3 gnd

Xa03 X7_H X7_L X3_H X3_L           a13_H a13_L vpc0 gnd xor2
Xa02 X2_H X2_L X6_H X6_L           a12_H a12_L vpc0 gnd xor2
Xa01 X5_H X5_L X1_H X1_L           a11_H a11_L vpc0 gnd xor2
Xa00 X0_H X0_L X4_H X4_L           a10_H a10_L vpc0 gnd xor2
```

```
Xb03 X7_H X7_L X0_H X0_L X4_H X4_L b13_H b13_L vpc0 gnd xor3
Xb02 X6_H X6_L X3_H X3_L           b12_H b12_L vpc0 gnd xor2
Xb01 X5_H X5_L X2_H X2_L           b11_H b11_L vpc0 gnd xor2
Xb00 X1_H X1_L X4_H X4_L           b10_H b10_L vpc0 gnd xor2

Xqt0
+ a10_H a10_L a11_H a11_L a12_H a12_L a13_H a13_L
+ a20_H a20_L a21_H a21_L a22_H a22_L a23_H a23_L
+ vpc1 gnd q1t0
Xqt1
+ b10_H b10_L b11_H b11_L b12_H b12_L b13_H b13_L
+ b20_H b20_L b21_H b21_L b22_H b22_L b23_H b23_L
+ vpc1 gnd q1t1

Xa13 a23_H a23_L b23_H b23_L             a33_H a33_L vpc2 gnd xor2
Xa12 b22_H b22_L a22_H a22_L             a32_H a32_L vpc2 gnd xor2
Xa11 a21_H a21_L b21_H b21_L             a31_H a31_L vpc2 gnd xor2
Xa10 b20_H b20_L a20_H a20_L             a30_H a30_L vpc2 gnd xor2

Xb13 a23_H a23_L b20_H b20_L a20_H a20_L b33_H b33_L vpc2 gnd xor3
Xb12 a22_H a22_L b23_H b23_L             b32_H b32_L vpc2 gnd xor2
Xb11 a21_H a21_L b22_H b22_L             b31_H b31_L vpc2 gnd xor2
Xb10 b21_H b21_L a20_H a20_L             b30_H b30_L vpc2 gnd xor2

Xqt2
+ a30_H a30_L a31_H a31_L a32_H a32_L a33_H a33_L
+  Y0_H  Y0_L  Y1_H  Y1_L  Y2_H  Y2_L  Y3_H  Y3_L
+ vpc3 gnd q1t2
Xqt3
+ b30_H b30_L b31_H b31_L b32_H b32_L b33_H b33_L
+  Y4_H  Y4_L  Y5_H  Y5_L  Y6_H  Y6_L  Y7_H  Y7_L
+ vpc3 gnd q1t3
.ENDS
```

C.2 LVS summaries

```
==> q0_lvs_summary <==

           Matched    Matched    Unmatched    Unmatched   Component
           Layout     Source     Layout       Source       Type
           -------    -------    ---------    ---------    ---------
Nets:          210        210           0            0
Instances:     378        378           0            0    MN(N)
                64         64           0            0    MP(P)
           -------    -------    ---------    ---------
Total Inst:    442        442           0            0


==> q1_lvs_summary <==

          Matched    Matched    Unmatched    Unmatched    Component
          Layout     Source     Layout       Source       Type
          -------    -------    ---------    ---------    ---------
Nets:         212        212           0            0
Instances:    382        382           0            0    MN(N)
               64         64           0            0    MP(P)
          -------    -------    ---------    ---------
Total Inst:   446        446           0            0
```

C.3 SPICE for GCD

```
* tt 3.3V 25C
*****************************************************************
* Process: tt
** including /home/spice/FAB_AME/ntyp.dat
.MODEL EN3 NMOS             LEVEL   = 53
* Full SPICE Models redacted pursuant to Non Disclosure Agreement
*****************************************************************
** including /home/spice/FAB_AME/ptyp.dat
.MODEL EP3 PMOS        LEVEL   = 53
* Full SPICE Models redacted pursuant to Non Disclosure Agreement
*****************************************************************

* Voltage: 3.3V
.PARAM VVDD=3.3
* Temperature: 25C
.TEMP 25
* Include Testbench
** including ./Testbench.spi
* GCD Testbench
* eldo -power -i Testbench.spi

* Include Models
* Process
*.INCLUDE '/home/spice/FAB_AME/nslow.dat'
*.INCLUDE '/home/spice/FAB_AME/pslow.dat'
*.INCLUDE '/home/spice/FAB_AME/ntyp.dat'
*.INCLUDE '/home/spice/FAB_AME/ptyp.dat'
*.INCLUDE '/home/spice/FAB_AME/nfast.dat'
*.INCLUDE '/home/spice/FAB_AME/pfast.dat'
* Voltage
*.PARAM vvdd=3.0
*.PARAM vvdd=3.3
*.PARAM vvdd=3.6
* Temperature
*.TEMP 125
*.TEMP 25
*.TEMP -40
* Include Model Aliases
** including ./defmod.spi
* DEFMOD statements to correct models
.DEFMOD N EN3
.DEFMOD P EP3

.OPTION Mach
.OPTION LIMPROBE = 3000
.OPTION Mach_MaxDcIterations = 750
.OPTION mach_DcAlgorithm = AUTO

* Include Subcircuits
** including ././AN2.spi
* AN2
.SUBCKT ND2 A B Z VDD GND
MPA Z A VDD VDD P L=0.35U W=2.40U
MPB Z B VDD VDD P L=0.35U W=2.40U
MNA NAB A GND GND N L=0.35U W=1.60U
MNB Z B NAB GND N L=0.35U W=1.60U
```

```
.ENDS

*.INCLUDE './INV1.spi'

.SUBCKT AN2 A B Z VDD GND
XND2 A  B ZN VDD GND ND2
XIV1 ZN    Z VDD GND IV1
.ENDS

** including ././CE2.spi
* CE2
.SUBCKT CE2 A B Z VDD GND
MPAI PAB A VDD VDD EP3 L=0.35U W=2.40U
MPBI ZN  B PAB VDD EP3 L=0.35U W=2.40U
MPAR PZN A VDD VDD EP3 L=0.35U W=0.50U
MPBR PZN B VDD VDD EP3 L=0.35U W=0.50U
MPFZ ZN ZB PZN VDD EP3 L=0.35U W=0.50U
MPZB ZB ZN VDD VDD EP3 L=0.35U W=0.50U
MPZO Z  ZN VDD VDD EP3 L=0.35U W=2.40U
MNZO Z  ZN GND GND EN3 L=0.35U W=1.60U
MNZB ZB ZN GND GND EN3 L=0.35U W=0.50U
MNFZ ZN ZB NZN GND EN3 L=0.35U W=0.50U
MNBR NZN B GND GND EN3 L=0.35U W=0.50U
MNAR NZN A GND GND EN3 L=0.35U W=0.50U
MNBI ZN  B NAB GND EN3 L=0.35U W=1.20U
MNAI NAB A GND GND EN3 L=0.35U W=1.20U
.ENDS

** including ././CE2R0.spi
* CE2R0
.SUBCKT CE2R0 A B RN Z VDD GND
MPAI PAB A VDD VDD EP3 L=0.35U W=2.40U
MPBI ZN  B PAB VDD EP3 L=0.35U W=2.40U
MPAR PZN A VDD VDD EP3 L=0.35U W=0.50U
MPBR PZN B VDD VDD EP3 L=0.35U W=0.50U
MPFZ ZN ZB PZN VDD EP3 L=0.35U W=0.50U
MPZB ZB ZN VDD VDD EP3 L=0.35U W=0.50U
MPZO Z  ZN VDD VDD EP3 L=0.35U W=2.40U
MNZO Z  ZN GND GND EN3 L=0.35U W=1.60U
MNZB ZB ZN GND GND EN3 L=0.35U W=0.50U
MNFZ ZN ZB NZN GND EN3 L=0.35U W=0.50U
MNBR NZN B NN  GND EN3 L=0.35U W=0.50U
MNAR NZN A NN  GND EN3 L=0.35U W=0.50U
MNBI ZN  B NAB GND EN3 L=0.35U W=1.20U
MNAI NAB A NN  GND EN3 L=0.35U W=1.20U

MPRS ZN RN VDD VDD EP3 L=0.35U W=0.50U
MNRS NN RN GND GND EN3 L=0.35U W=2.40U
.ENDS

** including ././CE2R1.spi
* CE2R0
.SUBCKT CE2R1 A B RN Z VDD GND
MPAI PAB A PN  VDD EP3 L=0.35U W=2.40U
MPBI ZN  B PAB VDD EP3 L=0.35U W=2.40U
MPAR PZN A PN  VDD EP3 L=0.35U W=0.50U
MPBR PZN B PN  VDD EP3 L=0.35U W=0.50U
MPFZ ZN ZB PZN VDD EP3 L=0.35U W=0.50U
```

```
MPZB ZB ZN VDD VDD EP3 L=0.35U W=0.50U
MPZO Z  ZN VDD VDD EP3 L=0.35U W=2.40U
MNZO Z  ZN GND GND EN3 L=0.35U W=1.60U
MNZB ZB ZN GND GND EN3 L=0.35U W=0.50U
MNFZ ZN ZB NZN GND EN3 L=0.35U W=0.50U
MNBR NZN B GND GND EN3 L=0.35U W=0.50U
MNAR NZN A GND GND EN3 L=0.35U W=0.50U
MNBI ZN  B NAB GND EN3 L=0.35U W=1.20U
MNAI NAB A GND GND EN3 L=0.35U W=1.20U

MPRS PN RB VDD VDD EP3 L=0.35U W=3.20U
MNRS ZN RB GND GND EN3 L=0.35U W=0.50U

MPRI RB RN VDD VDD EP3 L=0.35U W=0.50U
MNRI RB RN GND GND EN3 L=0.35U W=0.50U
.ENDS

** including ././CE3.spi
* CE3
.SUBCKT CE3 A B C Z VDD GND
MPAI PAB A VDD VDD EP3 L=0.35U W=2.40U
MPBI PBC B PAB VDD EP3 L=0.35U W=2.40U
MPCI ZN  C PBC VDD EP3 L=0.35U W=2.40U
MPAR PZN A VDD VDD EP3 L=0.35U W=0.50U
MPBR PZN B VDD VDD EP3 L=0.35U W=0.50U
MPCR PZN C VDD VDD EP3 L=0.35U W=0.50U
MPFZ ZN ZB PZN VDD EP3 L=0.35U W=0.50U
MPZB ZB ZN VDD VDD EP3 L=0.35U W=0.50U
MPZO Z  ZN VDD VDD EP3 L=0.35U W=2.40U
MNZO Z  ZN GND GND EN3 L=0.35U W=1.60U
MNZB ZB ZN GND GND EN3 L=0.35U W=0.50U
MNFZ ZN ZB NZN GND EN3 L=0.35U W=0.50U
MNCR NZN C GND GND EN3 L=0.35U W=0.50U
MNBR NZN B GND GND EN3 L=0.35U W=0.50U
MNAR NZN A GND GND EN3 L=0.35U W=0.50U
MNCI ZN  C NBC GND EN3 L=0.35U W=1.20U
MNBI NBC B NAB GND EN3 L=0.35U W=1.20U
MNAI NAB A GND GND EN3 L=0.35U W=1.20U
.ENDS

** including ././DLY.spi
* DLY
.SUBCKT DLY A Z VPB VNB VDD GND
MPAR PAV VPB VDD VDD P L=0.35U W=1.20U
MPAO B   A   PAV VDD P L=0.35U W=1.20U
MNAO B   A   NAV GND N L=0.35U W=0.80U
MNAR NAV VNB GND GND N L=0.35U W=0.80U

MPBR PBV VPB VDD VDD P L=0.35U W=1.20U
MPBO C   B   PBV VDD P L=0.35U W=1.20U
MNBO C   B   NBV GND N L=0.35U W=0.80U
MNBR NBV VNB GND GND N L=0.35U W=0.80U

MPCR PCV VPB VDD VDD P L=0.35U W=1.20U
MPCO D   C   PCV VDD P L=0.35U W=1.20U
MNCO D   C   NCV GND N L=0.35U W=0.80U
MNCR NCV VNB GND GND N L=0.35U W=0.80U
```

```
*MPDo Z D vdd vdd P L=0.35u W=1.20u
*MNDo Z D gnd gnd N L=0.35u W=0.80u

MPDR PBV VPB VDD VDD P L=0.35U W=1.20U
MPDO E   D   PBV VDD P L=0.35U W=1.20U
MNDO E   D   NBV GND N L=0.35U W=0.80U
MNDR NBV VNB GND GND N L=0.35U W=0.80U

MPER PCV VPB VDD VDD P L=0.35U W=1.20U
MPEO F   E   PCV VDD P L=0.35U W=1.20U
MNEO F   E   NCV GND N L=0.35U W=0.80U
MNER NCV VNB GND GND N L=0.35U W=0.80U

MPFO Z F VDD VDD P L=0.35U W=1.20U
MNFO Z F GND GND N L=0.35U W=0.80U

.ENDS

** including ././DMX.spi
* DMX DeMultiplexor

.SUBCKT DMX S0REQ S0ACK S1REQ S1ACK CT0REQ CT1REQ CTACK
+ IREQ IACK VDD GND
XS0 IREQ CT0REQ S0REQ VDD GND CE2
XS1 IREQ CT1REQ S1REQ VDD GND CE2
XO1 S0ACK S1ACK IACK VDD GND OR2
XB1 IACK CTACK VDD GND BF1
.ENDS

** including ././INV1.spi
* INV1
.SUBCKT IV1 A Z VDD GND
MP Z A VDD VDD P L=0.35U W=2.40U
MN Z A GND GND N L=0.35U W=1.60U
.ENDS

.SUBCKT BF1 A Z VDD GND
XI0 A NET VDD GND IV1
XI1 NET Z VDD GND IV1
.ENDS

** including ././MUX.spi
* MUX
.SUBCKT MUX S0REQ S0ACK S1REQ S1ACK CT0REQ CT1REQ CTACK
+ ZREQ ZACK VDD GND
XS0  S0REQ CT0REQ S0CT0REQ VDD GND CE2
XS1  S1REQ CT1REQ S1CT1REQ VDD GND CE2
XZ0  ZACK S0CT0REQ S0ACK VDD GND CE2
XZ1  ZACK S1CT1REQ S1ACK VDD GND CE2
XO1  S0CT0REQ S1CT1REQ ZREQ VDD GND OR2
XB1  ZACK CTACK VDD GND BF1
.ENDS

** including ././OR2.spi
* OR2
.SUBCKT NR2 A B Z VDD GND
MPA PAB A VDD VDD P L=0.35U W=2.40U
MPB Z B PAB VDD P L=0.35U W=2.40U
```

```
MNA Z A GND GND N L=0.35U W=1.60U
MNB Z B GND GND N L=0.35U W=1.60U
.ENDS

*.INCLUDE './INV1.spi'

.SUBCKT OR2 A B Z VDD GND
XNR2 A  B ZN VDD GND NR2
XIV1 ZN    Z VDD GND IV1
.ENDS

** including ././XR2.spi
* XR2
.SUBCKT XR2 A B Z VDD GND

MPN1 Q A VDD VDD P L=0.35U W=2.30U
MPN2 I B Q   VDD P L=0.35U W=2.30U

MNN1 I A GND GND N L=0.35U W=0.80U
MNN2 I B GND GND N L=0.35U W=0.80U

MPAI P I VDD VDD P L=0.35U W=2.30U
MPA1 Z A P   VDD P L=0.35U W=2.30U
MPA2 Z B P   VDD P L=0.35U W=2.30U

MNAI Z I GND GND N L=0.35U W=0.80U
MNA1 Z A N   GND N L=0.35U W=1.20U
MNA2 N B GND GND N L=0.35U W=1.20U

.ENDS


** including ././SWCR0.spi
* SWCR0
* Stepwise charging circuit - reset to 0

.SUBCKT SWCR0 REQI ACKI REQO ACKO RST0
+ VPB VNB VDDL VDDS VC3 VC2 VC1 GND VPC

XINVA ACKO NACKO VDDL GND IV1
XCE2R0 REQI NACKO RST0 RISE VDDL GND CE2R0
XINVR RISE FALL VDDL GND IV1
XINVF FALL DI0 VDDL GND IV1

XDLY1A DI0 DI1 VPB VNB VDDL GND DLY
XDLY1B DI1 DI2 VPB VNB VDDL GND DLY
XDLY1C DI2 DI3 VPB VNB VDDL GND DLY

XINVO DI3 DLO  VDDL GND IV1
XINVQ DLO REQO VDDL GND IV1
XINVB DLO ACKI VDDL GND IV1

XXR1A DI0 DI1 P1 VDDL GND XR2
XXR1B DI1 DI2 P2 VDDL GND XR2
XXR1C DI2 DI3 P3 VDDL GND XR2

XND2S4 RISE DI3 S4L VDDL GND ND2
```

```
MPR1 P1 RISE S3H VDDL P L=0.35U W=1.20U
MNR1 P1 FALL S3H GND  N L=0.35U W=0.80U
MPF1 P3 FALL S3H VDDL P L=0.35U W=1.20U
MNF1 P3 RISE S3H GND  N L=0.35U W=0.80U
XINVS3 S3H S3L VDDL GND IV1

R0 P2 S2H 1M
XINVS2 S2H S2L VDDL GND IV1

MPR3 P1 FALL S1H VDDL P L=0.35U W=1.20U
MNR3 P1 RISE S1H GND  N L=0.35U W=0.80U
MPF3 P3 RISE S1H VDDL P L=0.35U W=1.20U
MNF3 P3 FALL S1H GND  N L=0.35U W=0.80U

XNR2S0 RISE DI3 S0H VDDL GND NR2

MPSW4 VDDS S4L VPC VDDS P L=0.35U W=4.80U
MPSW3 VC3  S3L VPC VDDS P L=0.35U W=3.60U
MPSW2 VC2  S2L VPC VDDS P L=0.35U W=2.40U
MNSW2 VC2  S2H VPC GND  N L=0.35U W=1.60U
MNSW1 VC1  S1H VPC GND  N L=0.35U W=1.60U
MNSW0 GND  S0H VPC GND  N L=0.35U W=1.60U
.ENDS


** including ././SWCR1.spi
* SWCR1
* Stepwise charging circuit - reset to 0

.SUBCKT SWCR1 REQI ACKI REQO ACKO RST0
+ VPB VNB VDDL VDDS VC3 VC2 VC1 GND VPC

XINVA ACKO NACKO VDDL GND IV1
XCE2R1 REQI NACKO RST0 RISE VDDL GND CE2R1
XINVR RISE FALL VDDL GND IV1
XINVF FALL DI0 VDDL GND IV1

XDLY1A DI0 DI1 VPB VNB VDDL GND DLY
XDLY1B DI1 DI2 VPB VNB VDDL GND DLY
XDLY1C DI2 DI3 VPB VNB VDDL GND DLY

XINVO DI3 DLO  VDDL GND IV1
XINVQ DLO REQO VDDL GND IV1
XINVB DLO ACKI VDDL GND IV1

XXR1A DI0 DI1 P1 VDDL GND XR2
XXR1B DI1 DI2 P2 VDDL GND XR2
XXR1C DI2 DI3 P3 VDDL GND XR2

XND2S4 RISE DI3 S4L VDDL GND ND2

MPR1 P1 RISE S3H VDDL P L=0.35U W=1.20U
MNR1 P1 FALL S3H GND  N L=0.35U W=0.80U
MPF1 P3 FALL S3H VDDL P L=0.35U W=1.20U
MNF1 P3 RISE S3H GND  N L=0.35U W=0.80U
XINVS3 S3H S3L VDDL GND IV1

R0 P2 S2H 1M
```

```
XINVS2 S2H S2L VDDL GND IV1

MPR3 P1 FALL S1H VDDL P L=0.35U W=1.20U
MNR3 P1 RISE S1H GND  N L=0.35U W=0.80U
MPF3 P3 RISE S1H VDDL P L=0.35U W=1.20U
MNF3 P3 FALL S1H GND  N L=0.35U W=0.80U

XNR2S0 RISE DI3 S0H VDDL GND NR2

MPSW4 VDDS S4L VPC VDDS P L=0.35U W=1.60U
MPSW3 VC3  S3L VPC VDDS P L=0.35U W=1.60U
MPSW2 VC2  S2L VPC VDDS P L=0.35U W=1.60U
MNSW2 VC2  S2H VPC GND  N L=0.35U W=1.60U
MNSW1 VC1  S1H VPC GND  N L=0.35U W=1.60U
MNSW0 GND  S0H VPC GND  N L=0.35U W=1.60U
.ENDS

** including ././a2o.spi
* A2O
.SUBCKT PG2
+ G1_H G1_L P1_H P1_L
+ G0_H G0_L Z_H Z_L VPC GND
MP0 Z_L Z_H  VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L  VPC VPC P L=3.5E-07 W=5E-07
MN0 Z_L Z_H  GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L  GND GND N L=3.5E-07 W=5E-07
MU0 GL0 G0_L VPC GND N L=3.5E-07 W=5E-07
MU1 GH0 G0_H VPC GND N L=3.5E-07 W=5E-07
MU2 GL0 P1_L VPC GND N L=3.5E-07 W=5E-07
MU3 Z_H P1_H GH0 GND N L=3.5E-07 W=5E-07
MU4 Z_L G1_L GL0 GND N L=3.5E-07 W=5E-07
MU5 Z_H G1_H VPC GND N L=3.5E-07 W=5E-07
.ENDS

** including ././a2oao.spi
* A2OAO
.SUBCKT PG3
+ G2_H G2_L P2_H P2_L
+ G1_H G1_L P1_H P1_L
+ G0_H G0_L Z_H Z_L VPC GND
MP0 Z_L Z_H  VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L  VPC VPC P L=3.5E-07 W=5E-07
MN0 Z_L Z_H  GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L  GND GND N L=3.5E-07 W=5E-07
MU0 GL0 G0_L VPC GND N L=3.5E-07 W=5E-07
MU1 GH0 G0_H VPC GND N L=3.5E-07 W=5E-07
MU2 GL0 P1_L VPC GND N L=3.5E-07 W=5E-07
MU3 GH1 P1_H GH0 GND N L=3.5E-07 W=5E-07
MU4 GL1 G1_L GL0 GND N L=3.5E-07 W=5E-07
MU5 GH1 G1_H VPC GND N L=3.5E-07 W=5E-07
MU6 GL1 P2_L VPC GND N L=3.5E-07 W=5E-07
MU7 Z_H P2_H GH1 GND N L=3.5E-07 W=5E-07
MU8 Z_L G2_L GL1 GND N L=3.5E-07 W=5E-07
MU9 Z_H G2_H VPC GND N L=3.5E-07 W=5E-07
.ENDS

** including ././a2oaoao.spi
* A2OAOAO
```

```
.SUBCKT PG4
+ G3_H G3_L P3_H P3_L
+ G2_H G2_L P2_H P2_L
+ G1_H G1_L P1_H P1_L
+ G0_H G0_L Z_H Z_L VPC GND
MP0 Z_L Z_H  VPC VPC P L=3.5E-07 W=10E-07
MP1 Z_H Z_L  VPC VPC P L=3.5E-07 W=10E-07
MN0 Z_L Z_H  GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L  GND GND N L=3.5E-07 W=5E-07
MU0 GL0 G0_L VPC GND N L=3.5E-07 W=5E-07
MU1 GH0 G0_H VPC GND N L=3.5E-07 W=5E-07
MU2 GL0 P1_L VPC GND N L=3.5E-07 W=5E-07
MU3 GH1 P1_H GH0 GND N L=3.5E-07 W=5E-07
MU4 GL1 G1_L GL0 GND N L=3.5E-07 W=5E-07
MU5 GH1 G1_H VPC GND N L=3.5E-07 W=5E-07
MU6 GL1 P2_L VPC GND N L=3.5E-07 W=5E-07
MU7 GH2 P2_H GH1 GND N L=3.5E-07 W=5E-07
MU8 GL2 G2_L GL1 GND N L=3.5E-07 W=5E-07
MU9 GH2 G2_H VPC GND N L=3.5E-07 W=5E-07
MUA GL2 P3_L VPC GND N L=3.5E-07 W=5E-07
MUB Z_H P3_H GH2 GND N L=3.5E-07 W=5E-07
MUC Z_L G3_L GL2 GND N L=3.5E-07 W=5E-07
MUD Z_H G3_H VPC GND N L=3.5E-07 W=5E-07
.ENDS

** including ././and2.spi
* AND2
.SUBCKT AND2 A_H A_L B_H B_L Z_H Z_L VPC GND
MP0 Z_L Z_H VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L VPC VPC P L=3.5E-07 W=5E-07
MN0 Z_L Z_H GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L GND GND N L=3.5E-07 W=5E-07
MU0 Z_L A_L VPC GND N L=3.5E-07 W=5E-07
MU1 Z_H A_H ABH GND N L=3.5E-07 W=5E-07
MU2 Z_L B_L VPC GND N L=3.5E-07 W=5E-07
MU3 ABH B_H VPC GND N L=3.5E-07 W=5E-07
.ENDS

** including ././and3.spi
* AND3
.SUBCKT AND3 A_H A_L B_H B_L C_H C_L Z_H Z_L VPC GND
MP0 Z_L Z_H VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L VPC VPC P L=3.5E-07 W=5E-07
MN0 Z_L Z_H GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L GND GND N L=3.5E-07 W=5E-07
MU0 Z_L A_L VPC GND N L=3.5E-07 W=5E-07
MU1 Z_H A_H ABH GND N L=3.5E-07 W=5E-07
MU2 Z_L B_L VPC GND N L=3.5E-07 W=5E-07
MU3 ABH B_H BCH GND N L=3.5E-07 W=5E-07
MU4 Z_L C_L VPC GND N L=3.5E-07 W=5E-07
MU5 BCH C_H VPC GND N L=3.5E-07 W=5E-07
.ENDS

** including ././and4.spi
* AND4
.SUBCKT AND4 A_H A_L B_H B_L C_H C_L D_H D_L Z_H Z_L VPC GND
MP0 Z_L Z_H VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L VPC VPC P L=3.5E-07 W=5E-07
```

```
MN0 Z_L Z_H GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L GND GND N L=3.5E-07 W=5E-07
MU0 Z_L A_L VPC GND N L=3.5E-07 W=5E-07
MU1 Z_H A_H ABH GND N L=3.5E-07 W=5E-07
MU2 Z_L B_L VPC GND N L=3.5E-07 W=5E-07
MU3 ABH B_H BCH GND N L=3.5E-07 W=5E-07
MU4 Z_L C_L VPC GND N L=3.5E-07 W=5E-07
MU5 BCH C_H CDH GND N L=3.5E-07 W=5E-07
MU6 Z_L D_L VPC GND N L=3.5E-07 W=5E-07
MU7 CDH D_H VPC GND N L=3.5E-07 W=5E-07
.ENDS

** including ././buf1.spi
* BUF1
.SUBCKT BUF1 A_H A_L Z_H Z_L VPC GND
MP0 Z_L Z_H VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L VPC VPC P L=3.5E-07 W=5E-07
MN0 Z_L Z_H GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L GND GND N L=3.5E-07 W=5E-07
MU0 Z_L A_L VPC GND N L=3.5E-07 W=5E-07
MU1 Z_H A_H VPC GND N L=3.5E-07 W=5E-07
.ENDS

** including ././buf1r0.spi
* BUF1
.SUBCKT BUF1R0 A_H A_L Z_H Z_L R_L VPC GND
MP0 Z_L Z_H VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L VPC VPC P L=3.5E-07 W=5E-07
MN0 Z_L Z_H GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L GND GND N L=3.5E-07 W=5E-07
MU0 Z_L A_L VPC GND N L=3.5E-07 W=5E-07
MU1 Z_H A_H VPC GND N L=3.5E-07 W=5E-07
MPR Z_L R_L VPC VPC P L=3.5E-07 W=2E-06
.ENDS

** including ././buf1r1.spi
* BUF1
.SUBCKT BUF1R1 A_H A_L Z_H Z_L R_L VPC GND
MP0 Z_L Z_H VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L VPC VPC P L=3.5E-07 W=5E-07
MN0 Z_L Z_H GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L GND GND N L=3.5E-07 W=5E-07
MU0 Z_L A_L VPC GND N L=3.5E-07 W=5E-07
MU1 Z_H A_H VPC GND N L=3.5E-07 W=5E-07
MPR Z_H R_L VPC VPC P L=3.5E-07 W=5E-07
.ENDS

** including ././buf1x16.spi
* BUF1 x16

*.INCLUDE './buf1.spi'

.SUBCKT BUF1X16
+ A15_H A14_H A13_H A12_H A11_H A10_H A09_H A08_H
+ A07_H A06_H A05_H A04_H A03_H A02_H A01_H A00_H
+ A15_L A14_L A13_L A12_L A11_L A10_L A09_L A08_L
+ A07_L A06_L A05_L A04_L A03_L A02_L A01_L A00_L
+ Z15_H Z14_H Z13_H Z12_H Z11_H Z10_H Z09_H Z08_H
```

```
+ Z07_H Z06_H Z05_H Z04_H Z03_H Z02_H Z01_H Z00_H
+ Z15_L Z14_L Z13_L Z12_L Z11_L Z10_L Z09_L Z08_L
+ Z07_L Z06_L Z05_L Z04_L Z03_L Z02_L Z01_L Z00_L
+ VPC GND
X00 A00_H A00_L Z00_H Z00_L VPC GND BUF1
X01 A01_H A01_L Z01_H Z01_L VPC GND BUF1
X02 A02_H A02_L Z02_H Z02_L VPC GND BUF1
X03 A03_H A03_L Z03_H Z03_L VPC GND BUF1
X04 A04_H A04_L Z04_H Z04_L VPC GND BUF1
X05 A05_H A05_L Z05_H Z05_L VPC GND BUF1
X06 A06_H A06_L Z06_H Z06_L VPC GND BUF1
X07 A07_H A07_L Z07_H Z07_L VPC GND BUF1
X08 A08_H A08_L Z08_H Z08_L VPC GND BUF1
X09 A09_H A09_L Z09_H Z09_L VPC GND BUF1
X10 A10_H A10_L Z10_H Z10_L VPC GND BUF1
X11 A11_H A11_L Z11_H Z11_L VPC GND BUF1
X12 A12_H A12_L Z12_H Z12_L VPC GND BUF1
X13 A13_H A13_L Z13_H Z13_L VPC GND BUF1
X14 A14_H A14_L Z14_H Z14_L VPC GND BUF1
X15 A15_H A15_L Z15_H Z15_L VPC GND BUF1
.ENDS

** including ././mux2.spi
* MUX2
.SUBCKT MUX2 A_H A_L B_H B_L S_H S_L Z_H Z_L VPC GND
MP0 Z_L Z_H VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L VPC VPC P L=3.5E-07 W=5E-07
MN0 Z_L Z_H GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L GND GND N L=3.5E-07 W=5E-07
MU0 SLA S_L VPC GND N L=3.5E-07 W=5E-07
MU1 SHB S_H VPC GND N L=3.5E-07 W=5E-07
MU2 Z_L A_L SLA GND N L=3.5E-07 W=5E-07
MU3 Z_H A_H SLA GND N L=3.5E-07 W=5E-07
MU4 Z_L B_L SHB GND N L=3.5E-07 W=5E-07
MU5 Z_H B_H SHB GND N L=3.5E-07 W=5E-07
.ENDS

** including ././mux2x16.spi
* BUF1 x16

*.INCLUDE './mux2.spi'

.SUBCKT MUX2X16
+ A15_H A14_H A13_H A12_H A11_H A10_H A09_H A08_H
+ A07_H A06_H A05_H A04_H A03_H A02_H A01_H A00_H
+ A15_L A14_L A13_L A12_L A11_L A10_L A09_L A08_L
+ A07_L A06_L A05_L A04_L A03_L A02_L A01_L A00_L
+ B15_H B14_H B13_H B12_H B11_H B10_H B09_H B08_H
+ B07_H B06_H B05_H B04_H B03_H B02_H B01_H B00_H
+ B15_L B14_L B13_L B12_L B11_L B10_L B09_L B08_L
+ B07_L B06_L B05_L B04_L B03_L B02_L B01_L B00_L
+ S_H S_L
+ Z15_H Z14_H Z13_H Z12_H Z11_H Z10_H Z09_H Z08_H
+ Z07_H Z06_H Z05_H Z04_H Z03_H Z02_H Z01_H Z00_H
+ Z15_L Z14_L Z13_L Z12_L Z11_L Z10_L Z09_L Z08_L
+ Z07_L Z06_L Z05_L Z04_L Z03_L Z02_L Z01_L Z00_L
+ VPC GND
X00 A00_H A00_L B00_H B00_L S_H S_L Z00_H Z00_L VPC GND MUX2
```

```
X01 A01_H A01_L B01_H B01_L S_H S_L Z01_H Z01_L VPC GND MUX2
X02 A02_H A02_L B02_H B02_L S_H S_L Z02_H Z02_L VPC GND MUX2
X03 A03_H A03_L B03_H B03_L S_H S_L Z03_H Z03_L VPC GND MUX2
X04 A04_H A04_L B04_H B04_L S_H S_L Z04_H Z04_L VPC GND MUX2
X05 A05_H A05_L B05_H B05_L S_H S_L Z05_H Z05_L VPC GND MUX2
X06 A06_H A06_L B06_H B06_L S_H S_L Z06_H Z06_L VPC GND MUX2
X07 A07_H A07_L B07_H B07_L S_H S_L Z07_H Z07_L VPC GND MUX2
X08 A08_H A08_L B08_H B08_L S_H S_L Z08_H Z08_L VPC GND MUX2
X09 A09_H A09_L B09_H B09_L S_H S_L Z09_H Z09_L VPC GND MUX2
X10 A10_H A10_L B10_H B10_L S_H S_L Z10_H Z10_L VPC GND MUX2
X11 A11_H A11_L B11_H B11_L S_H S_L Z11_H Z11_L VPC GND MUX2
X12 A12_H A12_L B12_H B12_L S_H S_L Z12_H Z12_L VPC GND MUX2
X13 A13_H A13_L B13_H B13_L S_H S_L Z13_H Z13_L VPC GND MUX2
X14 A14_H A14_L B14_H B14_L S_H S_L Z14_H Z14_L VPC GND MUX2
X15 A15_H A15_L B15_H B15_L S_H S_L Z15_H Z15_L VPC GND MUX2
.ENDS

** including ././or2.spi
* OR2
.SUBCKT ORR2 A_H A_L B_H B_L Z_H Z_L VPC GND
XAND2 A_L A_H B_L B_H Z_L Z_H VPC GND AND2
.ENDS

** including ././xnor2.spi
* XNOR2
.SUBCKT XNOR2 A_H A_L B_H B_L Z_H Z_L VPC GND
XXOR2 A_H A_L B_H B_L Z_L Z_H VPC GND XOR2
.ENDS

** including ././xor2.spi
* XOR2
.SUBCKT XOR2 A_H A_L B_H B_L Z_H Z_L VPC GND
MP0 Z_L Z_H VPC VPC P L=3.5E-07 W=5E-07
MP1 Z_H Z_L VPC VPC P L=3.5E-07 W=5E-07
MN0 Z_L Z_H GND GND N L=3.5E-07 W=5E-07
MN1 Z_H Z_L GND GND N L=3.5E-07 W=5E-07
MU0 ABL A_L VPC GND N L=3.5E-07 W=5E-07
MU1 ABH A_H VPC GND N L=3.5E-07 W=5E-07
MU2 Z_L B_L ABL GND N L=3.5E-07 W=5E-07
MU3 Z_H B_H ABL GND N L=3.5E-07 W=5E-07
MU4 Z_H B_L ABH GND N L=3.5E-07 W=5E-07
MU5 Z_L B_H ABH GND N L=3.5E-07 W=5E-07
.ENDS

** including ././cmp16.spi
* CMP16

*.INCLUDE './and2.spi'
*.INCLUDE './xor2.spi'
*.INCLUDE './a2oaoao.spi'
*.INCLUDE './and4.spi'

.SUBCKT CMP16
+ A15_H A14_H A13_H A12_H A11_H A10_H A09_H A08_H
+ A07_H A06_H A05_H A04_H A03_H A02_H A01_H A00_H
+ A15_L A14_L A13_L A12_L A11_L A10_L A09_L A08_L
+ A07_L A06_L A05_L A04_L A03_L A02_L A01_L A00_L
+ B15_H B14_H B13_H B12_H B11_H B10_H B09_H B08_H
```

```
+ B07_H B06_H B05_H B04_H B03_H B02_H B01_H B00_H
+ B15_L B14_L B13_L B12_L B11_L B10_L B09_L B08_L
+ B07_L B06_L B05_L B04_L B03_L B02_L B01_L B00_L
+ NE_H NE_L GT_H GT_L VPC0 VPC1 VPC2 GND

* Stage 0

* Note inversion of output.
XE000 A00_H A00_L B00_H B00_L EQ000_L EQ000_H VPC0 GND XOR2
XE001 A01_H A01_L B01_H B01_L EQ001_L EQ001_H VPC0 GND XOR2
XE002 A02_H A02_L B02_H B02_L EQ002_L EQ002_H VPC0 GND XOR2
XE003 A03_H A03_L B03_H B03_L EQ003_L EQ003_H VPC0 GND XOR2
XE004 A04_H A04_L B04_H B04_L EQ004_L EQ004_H VPC0 GND XOR2
XE005 A05_H A05_L B05_H B05_L EQ005_L EQ005_H VPC0 GND XOR2
XE006 A06_H A06_L B06_H B06_L EQ006_L EQ006_H VPC0 GND XOR2
XE007 A07_H A07_L B07_H B07_L EQ007_L EQ007_H VPC0 GND XOR2
XE008 A08_H A08_L B08_H B08_L EQ008_L EQ008_H VPC0 GND XOR2
XE009 A09_H A09_L B09_H B09_L EQ009_L EQ009_H VPC0 GND XOR2
XE010 A10_H A10_L B10_H B10_L EQ010_L EQ010_H VPC0 GND XOR2
XE011 A11_H A11_L B11_H B11_L EQ011_L EQ011_H VPC0 GND XOR2
XE012 A12_H A12_L B12_H B12_L EQ012_L EQ012_H VPC0 GND XOR2
XE013 A13_H A13_L B13_H B13_L EQ013_L EQ013_H VPC0 GND XOR2
XE014 A14_H A14_L B14_H B14_L EQ014_L EQ014_H VPC0 GND XOR2
XE015 A15_H A15_L B15_H B15_L EQ015_L EQ015_H VPC0 GND XOR2

* Note inversion of B-input.
XG000 A00_H A00_L B00_L B00_H GT000_H GT000_L VPC0 GND AND2
XG001 A01_H A01_L B01_L B01_H GT001_H GT001_L VPC0 GND AND2
XG002 A02_H A02_L B02_L B02_H GT002_H GT002_L VPC0 GND AND2
XG003 A03_H A03_L B03_L B03_H GT003_H GT003_L VPC0 GND AND2
XG004 A04_H A04_L B04_L B04_H GT004_H GT004_L VPC0 GND AND2
XG005 A05_H A05_L B05_L B05_H GT005_H GT005_L VPC0 GND AND2
XG006 A06_H A06_L B06_L B06_H GT006_H GT006_L VPC0 GND AND2
XG007 A07_H A07_L B07_L B07_H GT007_H GT007_L VPC0 GND AND2
XG008 A08_H A08_L B08_L B08_H GT008_H GT008_L VPC0 GND AND2
XG009 A09_H A09_L B09_L B09_H GT009_H GT009_L VPC0 GND AND2
XG010 A10_H A10_L B10_L B10_H GT010_H GT010_L VPC0 GND AND2
XG011 A11_H A11_L B11_L B11_H GT011_H GT011_L VPC0 GND AND2
XG012 A12_H A12_L B12_L B12_H GT012_H GT012_L VPC0 GND AND2
XG013 A13_H A13_L B13_L B13_H GT013_H GT013_L VPC0 GND AND2
XG014 A14_H A14_L B14_L B14_H GT014_H GT014_L VPC0 GND AND2
XG015 A15_H A15_L B15_L B15_H GT015_H GT015_L VPC0 GND AND2

* Stage 1

XE100
+ EQ003_H EQ003_L EQ002_H EQ002_L
+ EQ001_H EQ001_L EQ000_H EQ000_L
+ EQ100_H EQ100_L VPC1 GND AND4
XE101
+ EQ007_H EQ007_L EQ006_H EQ006_L
+ EQ005_H EQ005_L EQ004_H EQ004_L
+ EQ101_H EQ101_L VPC1 GND AND4
XE102
+ EQ011_H EQ011_L EQ010_H EQ010_L
+ EQ009_H EQ009_L EQ008_H EQ008_L
+ EQ102_H EQ102_L VPC1 GND AND4
XE103
```

```
+ EQ015_H EQ015_L EQ014_H EQ014_L
+ EQ013_H EQ013_L EQ012_H EQ012_L
+ EQ103_H EQ103_L VPC1 GND AND4

XG100
+ GT003_H GT003_L EQ003_H EQ003_L
+ GT002_H GT002_L EQ002_H EQ002_L
+ GT001_H GT001_L EQ001_H EQ001_L
+ GT000_H GT000_L GT100_H GT100_L VPC1 GND PG4
XG101
+ GT007_H GT007_L EQ007_H EQ007_L
+ GT006_H GT006_L EQ006_H EQ006_L
+ GT005_H GT005_L EQ005_H EQ005_L
+ GT004_H GT004_L GT101_H GT101_L VPC1 GND PG4
XG102
+ GT011_H GT011_L EQ011_H EQ011_L
+ GT010_H GT010_L EQ010_H EQ010_L
+ GT009_H GT009_L EQ009_H EQ009_L
+ GT008_H GT008_L GT102_H GT102_L VPC1 GND PG4
XG103
+ GT015_H GT015_L EQ015_H EQ015_L
+ GT014_H GT014_L EQ014_H EQ014_L
+ GT013_H GT013_L EQ013_H EQ013_L
+ GT012_H GT012_L GT103_H GT103_L VPC1 GND PG4

* Stage 2

* Note inversion of output.
XE200
+ EQ103_H EQ103_L EQ102_H EQ102_L
+ EQ101_H EQ101_L EQ100_H EQ100_L
+ NE_L NE_H VPC2 GND AND4

XG200
+ GT103_H GT103_L EQ103_H EQ103_L
+ GT102_H GT102_L EQ102_H EQ102_L
+ GT101_H GT101_L EQ101_H EQ101_L
+ GT100_H GT100_L GT_H GT_L VPC2 GND PG4

.ENDS

** including ./sub16.spi
* SUB16

*.INCLUDE './buf1.spi'
*.INCLUDE './xor2.spi'
*.INCLUDE './xnor2.spi'
*.INCLUDE './or2.spi'
*.INCLUDE './and2.spi'
*.INCLUDE './and3.spi'
*.INCLUDE './and4.spi'
*.INCLUDE './a2o.spi'
*.INCLUDE './a2oao.spi'
*.INCLUDE './a2oaoao.spi'


* Half Adder
.SUBCKT HA A_H A_L B_H B_L G_H G_L P_H P_L VPC GND
```

```
XG0 A_H A_L B_H B_L G_H G_L VPC GND AND2
XP0 A_H A_L B_H B_L P_H P_L VPC GND XOR2
.ENDS

* Not Half Adder
.SUBCKT NHA A_H A_L B_H B_L G_H G_L P_H P_L VPC GND
XG0 A_H A_L B_H B_L G_H G_L VPC GND ORR2
XP0 A_H A_L B_H B_L P_H P_L VPC GND XNOR2
.ENDS

* Radix-4 Propagate/Generate cells
.SUBCKT GPG1
+ G0_H G0_L P0_H P0_L
+ GO_H GO_L PO_H PO_L VPC GND
XG1 G0_H G0_L GO_H GO_L VPC GND BUF1
XP1 P0_H P0_L PO_H PO_L VPC GND BUF1
.ENDS

.SUBCKT GPG2
+ G1_H G1_L P1_H P1_L
+ G0_H G0_L P0_H P0_L
+ GO_H GO_L PO_H PO_L VPC GND
XG2 G1_H G1_L P1_H P1_L G0_H G0_L GO_H GO_L VPC GND PG2
XP2 P1_H P1_L P0_H P0_L PO_H PO_L VPC GND AND2
.ENDS

.SUBCKT GPG3
+ G2_H G2_L P2_H P2_L
+ G1_H G1_L P1_H P1_L
+ G0_H G0_L P0_H P0_L
+ GO_H GO_L PO_H PO_L VPC GND
XG3 G2_H G2_L P2_H P2_L G1_H G1_L P1_H P1_L G0_H G0_L
+ GO_H GO_L VPC GND PG3
XP3 P2_H P2_L P1_H P1_L P0_H P0_L PO_H PO_L VPC GND AND3
.ENDS

.SUBCKT GPG4
+ G3_H G3_L P3_H P3_L
+ G2_H G2_L P2_H P2_L
+ G1_H G1_L P1_H P1_L
+ G0_H G0_L P0_H P0_L
+ GO_H GO_L PO_H PO_L VPC GND
XG4 G3_H G3_L P3_H P3_L G2_H G2_L P2_H P2_L
+ G1_H G1_L P1_H P1_L G0_H G0_L GO_H GO_L VPC GND PG4
XP4 P3_H P3_L P2_H P2_L P1_H P1_L P0_H P0_L PO_H PO_L VPC GND AND4
.ENDS

* Subtract/Reverse subtract
.SUBCKT SUB16
+ A15_H A14_H A13_H A12_H A11_H A10_H A09_H A08_H
+ A07_H A06_H A05_H A04_H A03_H A02_H A01_H A00_H
+ A15_L A14_L A13_L A12_L A11_L A10_L A09_L A08_L
+ A07_L A06_L A05_L A04_L A03_L A02_L A01_L A00_L
+ B15_H B14_H B13_H B12_H B11_H B10_H B09_H B08_H
+ B07_H B06_H B05_H B04_H B03_H B02_H B01_H B00_H
+ B15_L B14_L B13_L B12_L B11_L B10_L B09_L B08_L
+ B07_L B06_L B05_L B04_L B03_L B02_L B01_L B00_L
+ R_H R_L
```

```
+ Z15_H Z14_H Z13_H Z12_H Z11_H Z10_H Z09_H Z08_H
+ Z07_H Z06_H Z05_H Z04_H Z03_H Z02_H Z01_H Z00_H
+ Z15_L Z14_L Z13_L Z12_L Z11_L Z10_L Z09_L Z08_L
+ Z07_L Z06_L Z05_L Z04_L Z03_L Z02_L Z01_L Z00_L
+ VPC0 VPC1 VPC2 VPC3 VPC4 GND

* Stage 0

* This could be merged with stage 1 using a complex gate.
XIA00 R_L R_H A00_H A00_L A000_H A000_L VPC0 GND XOR2
XIA01 R_L R_H A01_H A01_L A001_H A001_L VPC0 GND XOR2
XIA02 R_L R_H A02_H A02_L A002_H A002_L VPC0 GND XOR2
XIA03 R_L R_H A03_H A03_L A003_H A003_L VPC0 GND XOR2
XIA04 R_L R_H A04_H A04_L A004_H A004_L VPC0 GND XOR2
XIA05 R_L R_H A05_H A05_L A005_H A005_L VPC0 GND XOR2
XIA06 R_L R_H A06_H A06_L A006_H A006_L VPC0 GND XOR2
XIA07 R_L R_H A07_H A07_L A007_H A007_L VPC0 GND XOR2
XIA08 R_L R_H A08_H A08_L A008_H A008_L VPC0 GND XOR2
XIA09 R_L R_H A09_H A09_L A009_H A009_L VPC0 GND XOR2
XIA10 R_L R_H A10_H A10_L A010_H A010_L VPC0 GND XOR2
XIA11 R_L R_H A11_H A11_L A011_H A011_L VPC0 GND XOR2
XIA12 R_L R_H A12_H A12_L A012_H A012_L VPC0 GND XOR2
XIA13 R_L R_H A13_H A13_L A013_H A013_L VPC0 GND XOR2
XIA14 R_L R_H A14_H A14_L A014_H A014_L VPC0 GND XOR2
XIA15 R_L R_H A15_H A15_L A015_H A015_L VPC0 GND XOR2

XIB00 R_H R_L B00_H B00_L B000_H B000_L VPC0 GND XOR2
XIB01 R_H R_L B01_H B01_L B001_H B001_L VPC0 GND XOR2
XIB02 R_H R_L B02_H B02_L B002_H B002_L VPC0 GND XOR2
XIB03 R_H R_L B03_H B03_L B003_H B003_L VPC0 GND XOR2
XIB04 R_H R_L B04_H B04_L B004_H B004_L VPC0 GND XOR2
XIB05 R_H R_L B05_H B05_L B005_H B005_L VPC0 GND XOR2
XIB06 R_H R_L B06_H B06_L B006_H B006_L VPC0 GND XOR2
XIB07 R_H R_L B07_H B07_L B007_H B007_L VPC0 GND XOR2
XIB08 R_H R_L B08_H B08_L B008_H B008_L VPC0 GND XOR2
XIB09 R_H R_L B09_H B09_L B009_H B009_L VPC0 GND XOR2
XIB10 R_H R_L B10_H B10_L B010_H B010_L VPC0 GND XOR2
XIB11 R_H R_L B11_H B11_L B011_H B011_L VPC0 GND XOR2
XIB12 R_H R_L B12_H B12_L B012_H B012_L VPC0 GND XOR2
XIB13 R_H R_L B13_H B13_L B013_H B013_L VPC0 GND XOR2
XIB14 R_H R_L B14_H B14_L B014_H B014_L VPC0 GND XOR2
XIB15 R_H R_L B15_H B15_L B015_H B015_L VPC0 GND XOR2

* Stage 1

* Note XHA00 includes two's complement correction.
XHA00 A000_H A000_L B000_H B000_L
+ G000_H G000_L P000_H P000_L VPC1 GND NHA
XHA01 A001_H A001_L B001_H B001_L
+ G001_H G001_L P001_H P001_L VPC1 GND HA
XHA02 A002_H A002_L B002_H B002_L
+ G002_H G002_L P002_H P002_L VPC1 GND HA
XHA03 A003_H A003_L B003_H B003_L
+ G003_H G003_L P003_H P003_L VPC1 GND HA
XHA04 A004_H A004_L B004_H B004_L
+ G004_H G004_L P004_H P004_L VPC1 GND HA
XHA05 A005_H A005_L B005_H B005_L
+ G005_H G005_L P005_H P005_L VPC1 GND HA
```

```
XHA06 A006_H A006_L B006_H B006_L
+ G006_H G006_L P006_H P006_L VPC1 GND HA
XHA07 A007_H A007_L B007_H B007_L
+ G007_H G007_L P007_H P007_L VPC1 GND HA
XHA08 A008_H A008_L B008_H B008_L
+ G008_H G008_L P008_H P008_L VPC1 GND HA
XHA09 A009_H A009_L B009_H B009_L
+ G009_H G009_L P009_H P009_L VPC1 GND HA
XHA10 A010_H A010_L B010_H B010_L
+ G010_H G010_L P010_H P010_L VPC1 GND HA
XHA11 A011_H A011_L B011_H B011_L
+ G011_H G011_L P011_H P011_L VPC1 GND HA
XHA12 A012_H A012_L B012_H B012_L
+ G012_H G012_L P012_H P012_L VPC1 GND HA

XHA13 A013_H A013_L B013_H B013_L
+ G013_H G013_L P013_H P013_L VPC1 GND HA
XHA14 A014_H A014_L B014_H B014_L
+ G014_H G014_L P014_H P014_L VPC1 GND HA
XHA15 A015_H A015_L B015_H B015_L
+ G015_H G015_L P015_H P015_L VPC1 GND HA

* Stage 2

XS000 G000_H G000_L P000_H P000_L
+ G100_H G100_L P100_H P100_L VPC2 GND GPG1
XS001 G001_H G001_L P001_H P001_L
+ G000_H G000_L P000_H P000_L
+ G101_H G101_L P101_H P101_L VPC2 GND GPG2
XS002 G002_H G002_L P002_H P002_L
+ G001_H G001_L P001_H P001_L
+ G000_H G000_L P000_H P000_L
+ G102_H G102_L P102_H P102_L VPC2 GND GPG3
XS003 G003_H G003_L P003_H P003_L
+ G002_H G002_L P002_H P002_L
+ G001_H G001_L P001_H P001_L
+ G000_H G000_L P000_H P000_L
+ G103_H G103_L P103_H P103_L VPC2 GND GPG4
XS004 G004_H G004_L P004_H P004_L
+ G104_H G104_L P104_H P104_L VPC2 GND GPG1
XS005 G005_H G005_L P005_H P005_L
+ G004_H G004_L P004_H P004_L
+ G105_H G105_L P105_H P105_L VPC2 GND GPG2
XS006 G006_H G006_L P006_H P006_L
+ G005_H G005_L P005_H P005_L
+ G004_H G004_L P004_H P004_L
+ G106_H G106_L P106_H P106_L VPC2 GND GPG3
XS007 G007_H G007_L P007_H P007_L
+ G006_H G006_L P006_H P006_L
+ G005_H G005_L P005_H P005_L
+ G004_H G004_L P004_H P004_L
+ G107_H G107_L P107_H P107_L VPC2 GND GPG4
XS008 G008_H G008_L P008_H P008_L
+ G108_H G108_L P108_H P108_L VPC2 GND GPG1
XS009 G009_H G009_L P009_H P009_L
+ G008_H G008_L P008_H P008_L
+ G109_H G109_L P109_H P109_L VPC2 GND GPG2
XS010 G010_H G010_L P010_H P010_L
```

```
+ G009_H G009_L P009_H P009_L
+ G008_H G008_L P008_H P008_L
+ G110_H G110_L P110_H P110_L VPC2 GND GPG3
XS011 G011_H G011_L P011_H P011_L
+ G010_H G010_L P010_H P010_L
+ G009_H G009_L P009_H P009_L
+ G008_H G008_L P008_H P008_L
+ G111_H G111_L P111_H P111_L VPC2 GND GPG4
XS012 G012_H G012_L P012_H P012_L
+ G112_H G112_L P112_H P112_L VPC2 GND GPG1
XS013 G013_H G013_L P013_H P013_L
+ G012_H G012_L P012_H P012_L
+ G113_H G113_L P113_H P113_L VPC2 GND GPG2
XS014 G014_H G014_L P014_H P014_L
+ G013_H G013_L P013_H P013_L
+ G012_H G012_L P012_H P012_L
+ G114_H G114_L P114_H P114_L VPC2 GND GPG3
XS015 G015_H G015_L P015_H P015_L
+ G014_H G014_L P014_H P014_L
+ G013_H G013_L P013_H P013_L
+ G012_H G012_L P012_H P012_L
+ G115_H G115_L P115_H P115_L VPC2 GND GPG4

* Q100 == P100
XB001 P001_H P001_L Q101_H Q101_L VPC2 GND BUF1
XB002 P002_H P002_L Q102_H Q102_L VPC2 GND BUF1
XB003 P003_H P003_L Q103_H Q103_L VPC2 GND BUF1
* Q104 == P104
XB005 P005_H P005_L Q105_H Q105_L VPC2 GND BUF1
XB006 P006_H P006_L Q106_H Q106_L VPC2 GND BUF1
XB007 P007_H P007_L Q107_H Q107_L VPC2 GND BUF1
* Q108 == P108
XB009 P009_H P009_L Q109_H Q109_L VPC2 GND BUF1
XB010 P010_H P010_L Q110_H Q110_L VPC2 GND BUF1
XB011 P011_H P011_L Q111_H Q111_L VPC2 GND BUF1
* Q112 == P112
XB013 P013_H P013_L Q113_H Q113_L VPC2 GND BUF1
XB014 P014_H P014_L Q114_H Q114_L VPC2 GND BUF1
XB015 P015_H P015_L Q115_H Q115_L VPC2 GND BUF1

* Stage 3

XS100 G100_H G100_L P100_H P100_L
+ G200_H G200_L P200_H P200_L VPC3 GND GPG1
XS101 G101_H G101_L Q101_H Q101_L
+ G201_H G201_L P201_H P201_L VPC3 GND GPG1
XS102 G102_H G102_L Q102_H Q102_L
+ G202_H G202_L P202_H P202_L VPC3 GND GPG1
XS103 G103_H G103_L Q103_H Q103_L
+ G203_H G203_L P203_H P203_L VPC3 GND GPG1
XS104 G104_H G104_L P104_H P104_L
+ G103_H G103_L P103_H P103_L
+ G204_H G204_L P204_H P204_L VPC3 GND GPG2
XS105 G105_H G105_L P105_H P105_L
+ G103_H G103_L P103_H P103_L
+ G205_H G205_L P205_H P205_L VPC3 GND GPG2
XS106 G106_H G106_L P106_H P106_L
+ G103_H G103_L P103_H P103_L
```

```
+ G206_H G206_L P206_H P206_L VPC3 GND GPG2
XS107 G107_H G107_L P107_H P107_L
+ G103_H G103_L P103_H P103_L
+ G207_H G207_L P207_H P207_L VPC3 GND GPG2
XS108 G108_H G108_L P108_H P108_L
+ G107_H G107_L P107_H P107_L
+ G103_H G103_L P103_H P103_L
+ G208_H G208_L P208_H P208_L VPC3 GND GPG3
XS109 G109_H G109_L P109_H P109_L
+ G107_H G107_L P107_H P107_L
+ G103_H G103_L P103_H P103_L
+ G209_H G209_L P209_H P209_L VPC3 GND GPG3
XS110 G110_H G110_L P110_H P110_L
+ G107_H G107_L P107_H P107_L
+ G103_H G103_L P103_H P103_L
+ G210_H G210_L P210_H P210_L VPC3 GND GPG3
XS111 G111_H G111_L P111_H P111_L
+ G107_H G107_L P107_H P107_L
+ G103_H G103_L P103_H P103_L
+ G211_H G211_L P211_H P211_L VPC3 GND GPG3
XS112 G112_H G112_L P112_H P112_L
+ G111_H G111_L P111_H P111_L
+ G107_H G107_L P107_H P107_L
+ G103_H G103_L P103_H P103_L
+ G212_H G212_L P212_H P212_L VPC3 GND GPG4
XS113 G113_H G113_L P113_H P113_L
+ G111_H G111_L P111_H P111_L
+ G107_H G107_L P107_H P107_L
+ G103_H G103_L P103_H P103_L
+ G213_H G213_L P213_H P213_L VPC3 GND GPG4
XS114 G114_H G114_L P114_H P114_L
+ G111_H G111_L P111_H P111_L
+ G107_H G107_L P107_H P107_L
+ G103_H G103_L P103_H P103_L
+ G214_H G214_L P214_H P214_L VPC3 GND GPG4
XS115 G115_H G115_L P115_H P115_L
+ G111_H G111_L P111_H P111_L
+ G107_H G107_L P107_H P107_L
+ G103_H G103_L P103_H P103_L
+ G215_H G215_L P215_H P215_L VPC3 GND GPG4

*
*
*
*
XB104 P104_H P104_L Q204_H Q204_L VPC3 GND BUF1
XB105 Q105_H Q105_L Q205_H Q205_L VPC3 GND BUF1
XB106 Q106_H Q106_L Q206_H Q206_L VPC3 GND BUF1
XB107 Q107_H Q107_L Q207_H Q207_L VPC3 GND BUF1
XB108 P108_H P108_L Q208_H Q208_L VPC3 GND BUF1
XB109 Q109_H Q109_L Q209_H Q209_L VPC3 GND BUF1
XB110 Q110_H Q110_L Q210_H Q210_L VPC3 GND BUF1
XB111 Q111_H Q111_L Q211_H Q211_L VPC3 GND BUF1
XB112 P112_H P112_L Q212_H Q212_L VPC3 GND BUF1
XB113 Q113_H Q113_L Q213_H Q213_L VPC3 GND BUF1
XB114 Q114_H Q114_L Q214_H Q214_L VPC3 GND BUF1
XB115 Q115_H Q115_L Q215_H Q215_L VPC3 GND BUF1
```

```
* Stage 4

XO000 P200_H P200_L             Z00_H Z00_L VPC4 GND BUF1
XO001 P201_H P201_L G200_H G200_L Z01_H Z01_L VPC4 GND XOR2
XO002 P202_H P202_L G201_H G201_L Z02_H Z02_L VPC4 GND XOR2
XO003 P203_H P203_L G202_H G202_L Z03_H Z03_L VPC4 GND XOR2
XO004 Q204_H Q204_L G203_H G203_L Z04_H Z04_L VPC4 GND XOR2
XO005 Q205_H Q205_L G204_H G204_L Z05_H Z05_L VPC4 GND XOR2
XO006 Q206_H Q206_L G205_H G205_L Z06_H Z06_L VPC4 GND XOR2
XO007 Q207_H Q207_L G206_H G206_L Z07_H Z07_L VPC4 GND XOR2
XO008 Q208_H Q208_L G207_H G207_L Z08_H Z08_L VPC4 GND XOR2
XO009 Q209_H Q209_L G208_H G208_L Z09_H Z09_L VPC4 GND XOR2
XO010 Q210_H Q210_L G209_H G209_L Z10_H Z10_L VPC4 GND XOR2
XO011 Q211_H Q211_L G210_H G210_L Z11_H Z11_L VPC4 GND XOR2
XO012 Q212_H Q212_L G211_H G211_L Z12_H Z12_L VPC4 GND XOR2
XO013 Q213_H Q213_L G212_H G212_L Z13_H Z13_L VPC4 GND XOR2
XO014 Q214_H Q214_L G213_H G213_L Z14_H Z14_L VPC4 GND XOR2
XO015 Q215_H Q215_L G214_H G214_L Z15_H Z15_L VPC4 GND XOR2

.ENDS

** including ././GCD.spi
* GCD

.SUBCKT GCD REQIN ACKIN REQOUT ACKOUT RST0
+ A15_H A14_H A13_H A12_H A11_H A10_H A09_H A08_H
+ A07_H A06_H A05_H A04_H A03_H A02_H A01_H A00_H
+ A15_L A14_L A13_L A12_L A11_L A10_L A09_L A08_L
+ A07_L A06_L A05_L A04_L A03_L A02_L A01_L A00_L
+ B15_H B14_H B13_H B12_H B11_H B10_H B09_H B08_H
+ B07_H B06_H B05_H B04_H B03_H B02_H B01_H B00_H
+ B15_L B14_L B13_L B12_L B11_L B10_L B09_L B08_L
+ B07_L B06_L B05_L B04_L B03_L B02_L B01_L B00_L
+ Z15_H Z14_H Z13_H Z12_H Z11_H Z10_H Z09_H Z08_H
+ Z07_H Z06_H Z05_H Z04_H Z03_H Z02_H Z01_H Z00_H
+ Z15_L Z14_L Z13_L Z12_L Z11_L Z10_L Z09_L Z08_L
+ Z07_L Z06_L Z05_L Z04_L Z03_L Z02_L Z01_L Z00_L
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND

XCTC REQ3 ACK3A REQ12 ACK12 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC10 SWCR0
XFBC ANEB0_H ANEB0_L ANEB1_H ANEB1_L ASWC10 GND BUF1

XCTB REQ12 ACK12 REQ13 ACK13 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC11 SWCR1
XFBB ANEB1_H ANEB1_L ANEB2_H ANEB2_L RST0 ASWC11 GND BUF1R0

XMX0 REQIN ACKIN REQ9 ACK9 ANEB2_L ANEB2_H ACK13 REQ10 ACK10
+ VDD GND MUX

XCT0 REQ10 ACK10 REQ0 ACK0 RST0
+ VPB VNB VDD VDDSWC  VC3 VC2 VC1 GND ASWC0 SWCR0
XCT1 REQ0 ACK0 REQ1 ACK1 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC1 SWCR0
XCT2 REQ1 ACK1 REQ2 ACK2 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC2 SWCR0
XCT3 REQ2 ACK2 REQ3 ACK3 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC3 SWCR0
```

```
XCT3A ACK3A ACK3B ACK3C ACK3 VDD GND CE3

XAMX
+ A15_H A14_H A13_H A12_H A11_H A10_H A09_H A08_H
+ A07_H A06_H A05_H A04_H A03_H A02_H A01_H A00_H
+ A15_L A14_L A13_L A12_L A11_L A10_L A09_L A08_L
+ A07_L A06_L A05_L A04_L A03_L A02_L A01_L A00_L
+ S8_15_H S8_14_H S8_13_H S8_12_H S8_11_H S8_10_H S8_09_H S8_08_H
+ S8_07_H S8_06_H S8_05_H S8_04_H S8_03_H S8_02_H S8_01_H S8_00_H
+ S8_15_L S8_14_L S8_13_L S8_12_L S8_11_L S8_10_L S8_09_L S8_08_L
+ S8_07_L S8_06_L S8_05_L S8_04_L S8_03_L S8_02_L S8_01_L S8_00_L
+ ANEB2_H ANEB2_L
+ A0_15_H A0_14_H A0_13_H A0_12_H A0_11_H A0_10_H A0_09_H A0_08_H
+ A0_07_H A0_06_H A0_05_H A0_04_H A0_03_H A0_02_H A0_01_H A0_00_H
+ A0_15_L A0_14_L A0_13_L A0_12_L A0_11_L A0_10_L A0_09_L A0_08_L
+ A0_07_L A0_06_L A0_05_L A0_04_L A0_03_L A0_02_L A0_01_L A0_00_L
+ ASWC0 GND MUX2X16

XBMX
+ B15_H B14_H B13_H B12_H B11_H B10_H B09_H B08_H
+ B07_H B06_H B05_H B04_H B03_H B02_H B01_H B00_H
+ B15_L B14_L B13_L B12_L B11_L B10_L B09_L B08_L
+ B07_L B06_L B05_L B04_L B03_L B02_L B01_L B00_L
+ M8_15_H M8_14_H M8_13_H M8_12_H M8_11_H M8_10_H M8_09_H M8_08_H
+ M8_07_H M8_06_H M8_05_H M8_04_H M8_03_H M8_02_H M8_01_H M8_00_H
+ M8_15_L M8_14_L M8_13_L M8_12_L M8_11_L M8_10_L M8_09_L M8_08_L
+ M8_07_L M8_06_L M8_05_L M8_04_L M8_03_L M8_02_L M8_01_L M8_00_L
+ ANEB2_H ANEB2_L
+ B0_15_H B0_14_H B0_13_H B0_12_H B0_11_H B0_10_H B0_09_H B0_08_H
+ B0_07_H B0_06_H B0_05_H B0_04_H B0_03_H B0_02_H B0_01_H B0_00_H
+ B0_15_L B0_14_L B0_13_L B0_12_L B0_11_L B0_10_L B0_09_L B0_08_L
+ B0_07_L B0_06_L B0_05_L B0_04_L B0_03_L B0_02_L B0_01_L B0_00_L
+ ASWC0 GND MUX2X16

XCMP
+ A0_15_H A0_14_H A0_13_H A0_12_H A0_11_H A0_10_H A0_09_H A0_08_H
+ A0_07_H A0_06_H A0_05_H A0_04_H A0_03_H A0_02_H A0_01_H A0_00_H
+ A0_15_L A0_14_L A0_13_L A0_12_L A0_11_L A0_10_L A0_09_L A0_08_L
+ A0_07_L A0_06_L A0_05_L A0_04_L A0_03_L A0_02_L A0_01_L A0_00_L
+ B0_15_H B0_14_H B0_13_H B0_12_H B0_11_H B0_10_H B0_09_H B0_08_H
+ B0_07_H B0_06_H B0_05_H B0_04_H B0_03_H B0_02_H B0_01_H B0_00_H
+ B0_15_L B0_14_L B0_13_L B0_12_L B0_11_L B0_10_L B0_09_L B0_08_L
+ B0_07_L B0_06_L B0_05_L B0_04_L B0_03_L B0_02_L B0_01_L B0_00_L
+ ANEB0_H ANEB0_L AGTB_H AGTB_L ASWC1 ASWC2 ASWC3 GND CMP16

XAB1
+ A0_15_H A0_14_H A0_13_H A0_12_H A0_11_H A0_10_H A0_09_H A0_08_H
+ A0_07_H A0_06_H A0_05_H A0_04_H A0_03_H A0_02_H A0_01_H A0_00_H
+ A0_15_L A0_14_L A0_13_L A0_12_L A0_11_L A0_10_L A0_09_L A0_08_L
+ A0_07_L A0_06_L A0_05_L A0_04_L A0_03_L A0_02_L A0_01_L A0_00_L
+ A1_15_H A1_14_H A1_13_H A1_12_H A1_11_H A1_10_H A1_09_H A1_08_H
+ A1_07_H A1_06_H A1_05_H A1_04_H A1_03_H A1_02_H A1_01_H A1_00_H
+ A1_15_L A1_14_L A1_13_L A1_12_L A1_11_L A1_10_L A1_09_L A1_08_L
+ A1_07_L A1_06_L A1_05_L A1_04_L A1_03_L A1_02_L A1_01_L A1_00_L
+ ASWC1 GND BUF1X16
XBB1
+ B0_15_H B0_14_H B0_13_H B0_12_H B0_11_H B0_10_H B0_09_H B0_08_H
```

```
+ B0_07_H B0_06_H B0_05_H B0_04_H B0_03_H B0_02_H B0_01_H B0_00_H
+ B0_15_L B0_14_L B0_13_L B0_12_L B0_11_L B0_10_L B0_09_L B0_08_L
+ B0_07_L B0_06_L B0_05_L B0_04_L B0_03_L B0_02_L B0_01_L B0_00_L
+ B1_15_H B1_14_H B1_13_H B1_12_H B1_11_H B1_10_H B1_09_H B1_08_H
+ B1_07_H B1_06_H B1_05_H B1_04_H B1_03_H B1_02_H B1_01_H B1_00_H
+ B1_15_L B1_14_L B1_13_L B1_12_L B1_11_L B1_10_L B1_09_L B1_08_L
+ B1_07_L B1_06_L B1_05_L B1_04_L B1_03_L B1_02_L B1_01_L B1_00_L
+ ASWC1 GND BUF1X16

XAB2
+ A1_15_H A1_14_H A1_13_H A1_12_H A1_11_H A1_10_H A1_09_H A1_08_H
+ A1_07_H A1_06_H A1_05_H A1_04_H A1_03_H A1_02_H A1_01_H A1_00_H
+ A1_15_L A1_14_L A1_13_L A1_12_L A1_11_L A1_10_L A1_09_L A1_08_L
+ A1_07_L A1_06_L A1_05_L A1_04_L A1_03_L A1_02_L A1_01_L A1_00_L
+ A2_15_H A2_14_H A2_13_H A2_12_H A2_11_H A2_10_H A2_09_H A2_08_H
+ A2_07_H A2_06_H A2_05_H A2_04_H A2_03_H A2_02_H A2_01_H A2_00_H
+ A2_15_L A2_14_L A2_13_L A2_12_L A2_11_L A2_10_L A2_09_L A2_08_L
+ A2_07_L A2_06_L A2_05_L A2_04_L A2_03_L A2_02_L A2_01_L A2_00_L
+ ASWC2 GND BUF1X16
XBB2
+ B1_15_H B1_14_H B1_13_H B1_12_H B1_11_H B1_10_H B1_09_H B1_08_H
+ B1_07_H B1_06_H B1_05_H B1_04_H B1_03_H B1_02_H B1_01_H B1_00_H
+ B1_15_L B1_14_L B1_13_L B1_12_L B1_11_L B1_10_L B1_09_L B1_08_L
+ B1_07_L B1_06_L B1_05_L B1_04_L B1_03_L B1_02_L B1_01_L B1_00_L
+ B2_15_H B2_14_H B2_13_H B2_12_H B2_11_H B2_10_H B2_09_H B2_08_H
+ B2_07_H B2_06_H B2_05_H B2_04_H B2_03_H B2_02_H B2_01_H B2_00_H
+ B2_15_L B2_14_L B2_13_L B2_12_L B2_11_L B2_10_L B2_09_L B2_08_L
+ B2_07_L B2_06_L B2_05_L B2_04_L B2_03_L B2_02_L B2_01_L B2_00_L
+ ASWC2 GND BUF1X16

XAB3
+ A2_15_H A2_14_H A2_13_H A2_12_H A2_11_H A2_10_H A2_09_H A2_08_H
+ A2_07_H A2_06_H A2_05_H A2_04_H A2_03_H A2_02_H A2_01_H A2_00_H
+ A2_15_L A2_14_L A2_13_L A2_12_L A2_11_L A2_10_L A2_09_L A2_08_L
+ A2_07_L A2_06_L A2_05_L A2_04_L A2_03_L A2_02_L A2_01_L A2_00_L
+ A3_15_H A3_14_H A3_13_H A3_12_H A3_11_H A3_10_H A3_09_H A3_08_H
+ A3_07_H A3_06_H A3_05_H A3_04_H A3_03_H A3_02_H A3_01_H A3_00_H
+ A3_15_L A3_14_L A3_13_L A3_12_L A3_11_L A3_10_L A3_09_L A3_08_L
+ A3_07_L A3_06_L A3_05_L A3_04_L A3_03_L A3_02_L A3_01_L A3_00_L
+ ASWC3 GND BUF1X16
XBB3
+ B2_15_H B2_14_H B2_13_H B2_12_H B2_11_H B2_10_H B2_09_H B2_08_H
+ B2_07_H B2_06_H B2_05_H B2_04_H B2_03_H B2_02_H B2_01_H B2_00_H
+ B2_15_L B2_14_L B2_13_L B2_12_L B2_11_L B2_10_L B2_09_L B2_08_L
+ B2_07_L B2_06_L B2_05_L B2_04_L B2_03_L B2_02_L B2_01_L B2_00_L
+ B3_15_H B3_14_H B3_13_H B3_12_H B3_11_H B3_10_H B3_09_H B3_08_H
+ B3_07_H B3_06_H B3_05_H B3_04_H B3_03_H B3_02_H B3_01_H B3_00_H
+ B3_15_L B3_14_L B3_13_L B3_12_L B3_11_L B3_10_L B3_09_L B3_08_L
+ B3_07_L B3_06_L B3_05_L B3_04_L B3_03_L B3_02_L B3_01_L B3_00_L
+ ASWC3 GND BUF1X16

XDX0 REQ11 ACK11 REQ4 ACK4 ANEB0_L ANEB0_H ACK3B REQ3 ACK3C
+ VDD GND DMX

XCT4 REQ4 ACK4 REQ5 ACK5 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC4 SWCR0
XCT5 REQ5 ACK5 REQ6 ACK6 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC5 SWCR0
```

```
XCT6 REQ6 ACK6 REQ7 ACK7 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC6 SWCR0
XCT7 REQ7 ACK7 REQ8 ACK8 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC7 SWCR0
XCT8 REQ8 ACK8 REQ9 ACK9 RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC8 SWCR0

XSUB4
+ A3_15_H A3_14_H A3_13_H A3_12_H A3_11_H A3_10_H A3_09_H A3_08_H
+ A3_07_H A3_06_H A3_05_H A3_04_H A3_03_H A3_02_H A3_01_H A3_00_H
+ A3_15_L A3_14_L A3_13_L A3_12_L A3_11_L A3_10_L A3_09_L A3_08_L
+ A3_07_L A3_06_L A3_05_L A3_04_L A3_03_L A3_02_L A3_01_L A3_00_L
+ B3_15_H B3_14_H B3_13_H B3_12_H B3_11_H B3_10_H B3_09_H B3_08_H
+ B3_07_H B3_06_H B3_05_H B3_04_H B3_03_H B3_02_H B3_01_H B3_00_H
+ B3_15_L B3_14_L B3_13_L B3_12_L B3_11_L B3_10_L B3_09_L B3_08_L
+ B3_07_L B3_06_L B3_05_L B3_04_L B3_03_L B3_02_L B3_01_L B3_00_L
+ AGTB_H AGTB_L
+ S8_15_H S8_14_H S8_13_H S8_12_H S8_11_H S8_10_H S8_09_H S8_08_H
+ S8_07_H S8_06_H S8_05_H S8_04_H S8_03_H S8_02_H S8_01_H S8_00_H
+ S8_15_L S8_14_L S8_13_L S8_12_L S8_11_L S8_10_L S8_09_L S8_08_L
+ S8_07_L S8_06_L S8_05_L S8_04_L S8_03_L S8_02_L S8_01_L S8_00_L
+ ASWC4 ASWC5 ASWC6 ASWC7 ASWC8 GND SUB16

XSMX4
+ A3_15_H A3_14_H A3_13_H A3_12_H A3_11_H A3_10_H A3_09_H A3_08_H
+ A3_07_H A3_06_H A3_05_H A3_04_H A3_03_H A3_02_H A3_01_H A3_00_H
+ A3_15_L A3_14_L A3_13_L A3_12_L A3_11_L A3_10_L A3_09_L A3_08_L
+ A3_07_L A3_06_L A3_05_L A3_04_L A3_03_L A3_02_L A3_01_L A3_00_L
+ B3_15_H B3_14_H B3_13_H B3_12_H B3_11_H B3_10_H B3_09_H B3_08_H
+ B3_07_H B3_06_H B3_05_H B3_04_H B3_03_H B3_02_H B3_01_H B3_00_H
+ B3_15_L B3_14_L B3_13_L B3_12_L B3_11_L B3_10_L B3_09_L B3_08_L
+ B3_07_L B3_06_L B3_05_L B3_04_L B3_03_L B3_02_L B3_01_L B3_00_L
+ AGTB_H AGTB_L
+ M4_15_H M4_14_H M4_13_H M4_12_H M4_11_H M4_10_H M4_09_H M4_08_H
+ M4_07_H M4_06_H M4_05_H M4_04_H M4_03_H M4_02_H M4_01_H M4_00_H
+ M4_15_L M4_14_L M4_13_L M4_12_L M4_11_L M4_10_L M4_09_L M4_08_L
+ M4_07_L M4_06_L M4_05_L M4_04_L M4_03_L M4_02_L M4_01_L M4_00_L
+ ASWC4 GND MUX2X16

XSB5
+ M4_15_H M4_14_H M4_13_H M4_12_H M4_11_H M4_10_H M4_09_H M4_08_H
+ M4_07_H M4_06_H M4_05_H M4_04_H M4_03_H M4_02_H M4_01_H M4_00_H
+ M4_15_L M4_14_L M4_13_L M4_12_L M4_11_L M4_10_L M4_09_L M4_08_L
+ M4_07_L M4_06_L M4_05_L M4_04_L M4_03_L M4_02_L M4_01_L M4_00_L
+ M5_15_H M5_14_H M5_13_H M5_12_H M5_11_H M5_10_H M5_09_H M5_08_H
+ M5_07_H M5_06_H M5_05_H M5_04_H M5_03_H M5_02_H M5_01_H M5_00_H
+ M5_15_L M5_14_L M5_13_L M5_12_L M5_11_L M5_10_L M5_09_L M5_08_L
+ M5_07_L M5_06_L M5_05_L M5_04_L M5_03_L M5_02_L M5_01_L M5_00_L
+ ASWC5 GND BUF1X16

XSB6
+ M5_15_H M5_14_H M5_13_H M5_12_H M5_11_H M5_10_H M5_09_H M5_08_H
+ M5_07_H M5_06_H M5_05_H M5_04_H M5_03_H M5_02_H M5_01_H M5_00_H
+ M5_15_L M5_14_L M5_13_L M5_12_L M5_11_L M5_10_L M5_09_L M5_08_L
+ M5_07_L M5_06_L M5_05_L M5_04_L M5_03_L M5_02_L M5_01_L M5_00_L
+ M6_15_H M6_14_H M6_13_H M6_12_H M6_11_H M6_10_H M6_09_H M6_08_H
+ M6_07_H M6_06_H M6_05_H M6_04_H M6_03_H M6_02_H M6_01_H M6_00_H
+ M6_15_L M6_14_L M6_13_L M6_12_L M6_11_L M6_10_L M6_09_L M6_08_L
```

```
+ M6_07_L M6_06_L M6_05_L M6_04_L M6_03_L M6_02_L M6_01_L M6_00_L
+ ASWC6 GND BUF1X16

XSB7
+ M6_15_H M6_14_H M6_13_H M6_12_H M6_11_H M6_10_H M6_09_H M6_08_H
+ M6_07_H M6_06_H M6_05_H M6_04_H M6_03_H M6_02_H M6_01_H M6_00_H
+ M6_15_L M6_14_L M6_13_L M6_12_L M6_11_L M6_10_L M6_09_L M6_08_L
+ M6_07_L M6_06_L M6_05_L M6_04_L M6_03_L M6_02_L M6_01_L M6_00_L
+ M7_15_H M7_14_H M7_13_H M7_12_H M7_11_H M7_10_H M7_09_H M7_08_H
+ M7_07_H M7_06_H M7_05_H M7_04_H M7_03_H M7_02_H M7_01_H M7_00_H
+ M7_15_L M7_14_L M7_13_L M7_12_L M7_11_L M7_10_L M7_09_L M7_08_L
+ M7_07_L M7_06_L M7_05_L M7_04_L M7_03_L M7_02_L M7_01_L M7_00_L
+ ASWC7 GND BUF1X16

XSB8
+ M7_15_H M7_14_H M7_13_H M7_12_H M7_11_H M7_10_H M7_09_H M7_08_H
+ M7_07_H M7_06_H M7_05_H M7_04_H M7_03_H M7_02_H M7_01_H M7_00_H
+ M7_15_L M7_14_L M7_13_L M7_12_L M7_11_L M7_10_L M7_09_L M7_08_L
+ M7_07_L M7_06_L M7_05_L M7_04_L M7_03_L M7_02_L M7_01_L M7_00_L
+ M8_15_H M8_14_H M8_13_H M8_12_H M8_11_H M8_10_H M8_09_H M8_08_H
+ M8_07_H M8_06_H M8_05_H M8_04_H M8_03_H M8_02_H M8_01_H M8_00_H
+ M8_15_L M8_14_L M8_13_L M8_12_L M8_11_L M8_10_L M8_09_L M8_08_L
+ M8_07_L M8_06_L M8_05_L M8_04_L M8_03_L M8_02_L M8_01_L M8_00_L
+ ASWC8 GND BUF1X16

XCT9 REQ11 ACK11 REQOUT ACKOUT RST0
+ VPB VNB VDD VDDSWC VC3 VC2 VC1 GND ASWC9 SWCR0
XSB9
+ A3_15_H A3_14_H A3_13_H A3_12_H A3_11_H A3_10_H A3_09_H A3_08_H
+ A3_07_H A3_06_H A3_05_H A3_04_H A3_03_H A3_02_H A3_01_H A3_00_H
+ A3_15_L A3_14_L A3_13_L A3_12_L A3_11_L A3_10_L A3_09_L A3_08_L
+ A3_07_L A3_06_L A3_05_L A3_04_L A3_03_L A3_02_L A3_01_L A3_00_L
+ Z15_H Z14_H Z13_H Z12_H Z11_H Z10_H Z09_H Z08_H
+ Z07_H Z06_H Z05_H Z04_H Z03_H Z02_H Z01_H Z00_H
+ Z15_L Z14_L Z13_L Z12_L Z11_L Z10_L Z09_L Z08_L
+ Z07_L Z06_L Z05_L Z04_L Z03_L Z02_L Z01_L Z00_L
+ ASWC9 GND BUF1X16

.ENDS


* Capacitors
C1 VC1 0 10p
C2 VC2 0 10p
C3 VC3 0 10p
.IC V(VC3)=0 V(VC2)=0 V(VC1)=0

* Bias
VPB VPB 0 DC '0.725*VVDD'
VNB VNB 0 DC '0.275*VVDD'

* PSU
VDDEXT VDDEXT 0 DC 'VVDD'
VDD    VDD    0 DC 'VVDD'
VDDSWC VDDSWC 0 DC 'VVDD'
VSS    GND    0 DC 0

* Reset
```

```
VR0 RST0 0 DC 0 PULSE VVDD 0 0N 0.1N 0.1N 40N 1

XDUT REQIN ACKIN REQOUT ACKOUT RST0
+A15_H A14_H A13_H A12_H A11_H A10_H A09_H A08_H
+A07_H A06_H A05_H A04_H A03_H A02_H A01_H A00_H
+A15_L A14_L A13_L A12_L A11_L A10_L A09_L A08_L
+A07_L A06_L A05_L A04_L A03_L A02_L A01_L A00_L
+B15_H B14_H B13_H B12_H B11_H B10_H B09_H B08_H
+B07_H B06_H B05_H B04_H B03_H B02_H B01_H B00_H
+B15_L B14_L B13_L B12_L B11_L B10_L B09_L B08_L
+B07_L B06_L B05_L B04_L B03_L B02_L B01_L B00_L
+Z15_H Z14_H Z13_H Z12_H Z11_H Z10_H Z09_H Z08_H
+Z07_H Z06_H Z05_H Z04_H Z03_H Z02_H Z01_H Z00_H
+Z15_L Z14_L Z13_L Z12_L Z11_L Z10_L Z09_L Z08_L
+Z07_L Z06_L Z05_L Z04_L Z03_L Z02_L Z01_L Z00_L
+VPB VNB VDD VDDSWC VC3 VC2 VC1 GND GCD

XIV1  REQOUT RA0 VDDEXT GND IV1
XIV2  RA0 RA1    VDDEXT GND IV1
XIV3  RA1 RA2    VDDEXT GND IV1
XIV4  RA2 ACKOUT VDDEXT GND IV1

.SETBUS AH
+A15_H A14_H A13_H A12_H A11_H A10_H A09_H A08_H
+A07_H A06_H A05_H A04_H A03_H A02_H A01_H A00_H
.SETBUS AL
+A15_L A14_L A13_L A12_L A11_L A10_L A09_L A08_L
+A07_L A06_L A05_L A04_L A03_L A02_L A01_L A00_L
.SETBUS BH
+B15_H B14_H B13_H B12_H B11_H B10_H B09_H B08_H
+B07_H B06_H B05_H B04_H B03_H B02_H B01_H B00_H
.SETBUS BL
+B15_L B14_L B13_L B12_L B11_L B10_L B09_L B08_L
+B07_L B06_L B05_L B04_L B03_L B02_L B01_L B00_L

.SETBUS ZH
+Z15_H Z14_H Z13_H Z12_H Z11_H Z10_H Z09_H Z08_H
+Z07_H Z06_H Z05_H Z04_H Z03_H Z02_H Z01_H Z00_H
.SETBUS ZL
+Z15_L Z14_L Z13_L Z12_L Z11_L Z10_L Z09_L Z08_L
+Z07_L Z06_L Z05_L Z04_L Z03_L Z02_L Z01_L Z00_L

* Fibonacci:   F24 &   F23
* Fibonacci: 46368 & 28657
*          :  B520 &  6FF1
*          :  4ADF &  900E

.SIGBUS AH VHI=VVDD VLO=0 TFALL=10N TRISE=10N BASE=HEXA
+     0  0    25N B520   150N B520   200N 0
+ 12000N 0 12025N B520 12150N B520 12200N 0
.SIGBUS AL VHI=VVDD VLO=0 TFALL=10N TRISE=10N BASE=HEXA
+     0  0    25N 4ADF   150N 4ADF   200N 0
+ 12000N 0 12025N 4ADF 12150N 4ADF 12200N 0

.SIGBUS BH VHI=VVDD VLO=0 TFALL=10N TRISE=10N BASE=HEXA
+     0  0    25N 6FF1   150N 6FF1   200N 0
+ 12000N 0 12025N 6FF1 12150N 6FF1 12200N 0
.SIGBUS BL VHI=VVDD VLO=0 TFALL=10N TRISE=10N BASE=HEXA
```

```
+     0  0    25N 900E   150N 900E   200N 0
+ 12000N 0 12025N 900E 12150N 900E 12200N 0

.SETBUS A0H
+XDUT.A0_15_H XDUT.A0_14_H XDUT.A0_13_H XDUT.A0_12_H
+XDUT.A0_11_H XDUT.A0_10_H XDUT.A0_09_H XDUT.A0_08_H
+XDUT.A0_07_H XDUT.A0_06_H XDUT.A0_05_H XDUT.A0_04_H
+XDUT.A0_03_H XDUT.A0_02_H XDUT.A0_01_H XDUT.A0_00_H
.SETBUS A0L
+XDUT.A0_15_L XDUT.A0_14_L XDUT.A0_13_L XDUT.A0_12_L
+XDUT.A0_11_L XDUT.A0_10_L XDUT.A0_09_L XDUT.A0_08_L
+XDUT.A0_07_L XDUT.A0_06_L XDUT.A0_05_L XDUT.A0_04_L
+XDUT.A0_03_L XDUT.A0_02_L XDUT.A0_01_L XDUT.A0_00_L
.SETBUS B0H
+XDUT.B0_15_H XDUT.B0_14_H XDUT.B0_13_H XDUT.B0_12_H
+XDUT.B0_11_H XDUT.B0_10_H XDUT.B0_09_H XDUT.B0_08_H
+XDUT.B0_07_H XDUT.B0_06_H XDUT.B0_05_H XDUT.B0_04_H
+XDUT.B0_03_H XDUT.B0_02_H XDUT.B0_01_H XDUT.B0_00_H
.SETBUS B0L
+XDUT.B0_15_L XDUT.B0_14_L XDUT.B0_13_L XDUT.B0_12_L
+XDUT.B0_11_L XDUT.B0_10_L XDUT.B0_09_L XDUT.B0_08_L
+XDUT.B0_07_L XDUT.B0_06_L XDUT.B0_05_L XDUT.B0_04_L
+XDUT.B0_03_L XDUT.B0_02_L XDUT.B0_01_L XDUT.B0_00_L
.SETBUS M8H
+XDUT.M8_15_H XDUT.M8_14_H XDUT.M8_13_H XDUT.M8_12_H
+XDUT.M8_11_H XDUT.M8_10_H XDUT.M8_09_H XDUT.M8_08_H
+XDUT.M8_07_H XDUT.M8_06_H XDUT.M8_05_H XDUT.M8_04_H
+XDUT.M8_03_H XDUT.M8_02_H XDUT.M8_01_H XDUT.M8_00_H
.SETBUS M8L
+XDUT.M8_15_L XDUT.M8_14_L XDUT.M8_13_L XDUT.M8_12_L
+XDUT.M8_11_L XDUT.M8_10_L XDUT.M8_09_L XDUT.M8_08_L
+XDUT.M8_07_L XDUT.M8_06_L XDUT.M8_05_L XDUT.M8_04_L
+XDUT.M8_03_L XDUT.M8_02_L XDUT.M8_01_L XDUT.M8_00_L
.SETBUS S8H
+XDUT.S8_15_H XDUT.S8_14_H XDUT.S8_13_H XDUT.S8_12_H
+XDUT.S8_11_H XDUT.S8_10_H XDUT.S8_09_H XDUT.S8_08_H
+XDUT.S8_07_H XDUT.S8_06_H XDUT.S8_05_H XDUT.S8_04_H
+XDUT.S8_03_H XDUT.S8_02_H XDUT.S8_01_H XDUT.S8_00_H
.SETBUS S8L
+XDUT.S8_15_L XDUT.S8_14_L XDUT.S8_13_L XDUT.S8_12_L
+XDUT.S8_11_L XDUT.S8_10_L XDUT.S8_09_L XDUT.S8_08_L
+XDUT.S8_07_L XDUT.S8_06_L XDUT.S8_05_L XDUT.S8_04_L
+XDUT.S8_03_L XDUT.S8_02_L XDUT.S8_01_L XDUT.S8_00_L

VREQI REQIN 0 DC 0 PULSE 0 VVDD 75N 0.1N 0.1N 50N 12075N

.PLOTBUS AH VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS AL VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS BH VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS BL VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS ZH VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS ZL VTH1=0.2*VVDD VTH2=0.8*VVDD

.PLOTBUS A0H VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS A0L VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS B0H VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS B0L VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS M8H VTH1=0.2*VVDD VTH2=0.8*VVDD
```

```
.PLOTBUS M8L VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS S8H VTH1=0.2*VVDD VTH2=0.8*VVDD
.PLOTBUS S8L VTH1=0.2*VVDD VTH2=0.8*VVDD

.PARAM STOPTIME = 24000n

.OPTION AEX
.OPTION ALIGNEXT
.OPTION EXTMKSA
.OPTION NOASCII

.EXTRACT TRAN LABEL=XReqIn1   XUP(V(reqIn),  0.5*vvdd, 1)
.EXTRACT TRAN LABEL=XReqOut1
+ XUP(V(reqOut), 0.5*vvdd, MEAS(XReqIn), stoptime, 1)
.EXTRACT TRAN LABEL=CalcTime1
+ TPDUU(V(reqIn),  V(reqOut), VTH=0.5*vvdd, OCCUR=1)

.EXTRACT TRAN LABEL=XReqIn    XUP(V(reqIn),  0.5*vvdd, 2)
.EXTRACT TRAN LABEL=XReqOut
+ XUP(V(reqOut), 0.5*vvdd, MEAS(XReqIn), stoptime, 2)
.EXTRACT TRAN LABEL=CalcTime
+ TPDUU(V(reqIn),  V(reqOut), VTH=0.5*vvdd, OCCUR=2)

.EXTRACT TRAN LABEL=IntVdd1
+ INTEG(I(Vdd),    MEAS(XReqIn1), MEAS(XReqOut1))
.EXTRACT TRAN LABEL=IntVddSwc1
+ INTEG(I(VddSwc), MEAS(XReqIn1), MEAS(XReqOut1))

.EXTRACT TRAN LABEL=VddPwr1    vvdd*MEAS(IntVdd1)
.EXTRACT TRAN LABEL=VddSwcPwr1 vvdd*MEAS(IntVddSwc1)

.EXTRACT TRAN LABEL=IntVdd
+ INTEG(I(Vdd),    MEAS(XReqIn), MEAS(XReqOut))
.EXTRACT TRAN LABEL=IntVddSwc
+ INTEG(I(VddSwc), MEAS(XReqIn), MEAS(XReqOut))

.EXTRACT TRAN LABEL=VddPwr    vvdd*MEAS(IntVdd)
.EXTRACT TRAN LABEL=VddSwcPwr vvdd*MEAS(IntVddSwc)

.EXTRACT TRAN LABEL=Cap3_1 YVAL(V(vc3),MEAS(XReqOut1))
.EXTRACT TRAN LABEL=Cap2_1 YVAL(V(vc2),MEAS(XReqOut1))
.EXTRACT TRAN LABEL=Cap1_1 YVAL(V(vc1),MEAS(XReqOut1))
.EXTRACT TRAN LABEL=Cap3   YVAL(V(vc3),MEAS(XReqOut))
.EXTRACT TRAN LABEL=Cap2   YVAL(V(vc2),MEAS(XReqOut))
.EXTRACT TRAN LABEL=Cap1   YVAL(V(vc1),MEAS(XReqOut))

.TRAN 1N 'STOPTIME' UIC
*.PROBE TRAN VTOP
.PROBE TRAN V(REQIN) V(REQOUT) V(ACKIN) V(ACKOUT)
.PROBE TRAN V(VC3) V(VC2) V(VC1) V(RST0)
.PROBE TRAN V(XDUT.ASWC0)
.PROBE TRAN W
.PROBE TRAN I(VDD) I(VDDSWC)

*END

.END
```

UNIVERSITY OF WESTMINSTER

Applied DSP and VLSI Research Group

Applied Research for Industry

Expertise:

- Reduced complexity FIR/IIR filter design techniques and their efficient realizations.
- Fixed/adaptive FIR/IIR filter techniques for transmission/reception paths of future mobile communication systems.
- Design and implementation of comms novel adaptive schemes for non-linear distortion compensation and frequency estimation.
- Sigma-Delta modulator based systems.
- Sigma-Delta based data acquisition and conversion systems with their associated DSP.
- Sigma-Delta based fractional frequency synthesis systems and compression techniques.
- Ultra-low-power algorithms for real-time biomedical, comms systems applications including hearing aids and mathematical morphology.
- Reduced complexity ultra-low-power algorithms and architectures for fixed-point custom and FPGA based arithmetic circuits.
- Power/space/performance-efficient implementation of image/audio/video processing algorithms on custom/FPGA platforms.
- Ultra-low-power reconfigurable full-custom mixed Analog/Digital processor development and design.
- The MPEG standard and its custom silicon/FPGA implementations.
- Imaging techniques for the analysis of peripheral blood films for blood parasite detection, e.g. malaria
- Interpretation and analysis of images in the presence of speckle noise, e.g. Ultrasound, x-ray microscopy.
- Applied optical computer vision solutions for routine and time-consuming tasks.
- Global Navigation Satellite Systems, including software configurable receiver designs, as well as reduced complexity receivers for GPS, Galileo, GLONASS and other emerging standards.

PhD Theses from the Group:

- Artur Krukowski, “Flexible IIR Digital Filter Design and Multipath Realization”, 1999.
- Lorenzo Pasquato, “Adaptive Filtering with Balanced Model Truncation”, 2000.
- Mohammed Al-Janabi, “Design, Analysis and Investigation of Bandpass Sigma-Delta Modulators”, 2000.
- Jeremi Gryka, “Extension of Balanced Model Reduction Techniques for Flexible Digital Filter Design & Apps.”, 2000.
- Mucahit Kozak, “ Oversampled Delta-Sigma Modulators: Analysis, Applications and Novel Topologies”, 2001.
- Robert Beck, “An Investigation of Finite-Precision Digital Resonators”, 2002.
- Ediz Cetin, “Unsupervised Adaptive Signal Processing Techniques for Wireless Receivers”, 2002.
- Suleyman Sirri Demirsoy, “Complexity Reduction in Digital Filters and Filter Banks”, 2003.
- K.N.R. M. Rao, “Application of mathematical morphology to biomedical image processing”, 2003.
- Izzet Ozcelik, “Adaptive System Identification Algorithms & Sequence Estimation in Telecommunication Channels”, 2005.
- Mohamed Rezki, “Cramer-Rao Bounds and Frequency Offset Estimation in Wireless Telecommunication Systems”, 2005.
- Taoufik Bourdi, “Fast frequency hopping synthesizers for wireless communications”, 2006.
- Jaswinder Lota, “A Comprehensive Design Methodology for the Design of Discrete-Time Sigma Delta Modulators”, 2007.
- F. Boray Tek, “Computerised diagnosis of Malaria”, 2007.
- H. Zare-Hoseini, “Continuous-Time Delta Sigma Modulators With Immunity to Clock Jitter”, 2008.

Applied DSP and VLSI Research Group
Department of Electronic, Communication & Software Engineering
115 New Cavendish Street, London W1W 6UW
ENGLAND, UK.
Tel: +44-(0)20-7911-5157 Fax: +44-(0)20-7911-5089
http://www.adrvg.wmin.ac.uk