# Reversible Computer Engineering and Architecture

by

Carlin James Vieri

B.S., University of California at Berkeley (1993)
S.M., Massachusetts Institute of Technology (1995)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999



Author_____
Department of Electrical Engineering and Computer Science
May 14, 1999

Certified by_____
Thomas F. Knight, Jr.
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Committee on Graduate Students

# Reversible Computer Engineering and Architecture

by

Carlin James Vieri

Submitted to the
Department of Electrical Engineering and Computer Science

May 14, 1999

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

## Abstract

There exists a set of reversible primitive modules that may be combined according to a set of simple rules to create larger systems, which are themselves reversible primitive modules. These modules may be combined to create a fully reversible, universal computer.

I describe techniques for creating reversible primitive modules from simple switches, such as conventional CMOS transistors, and the rules for combining these modules to maintain reversibility. As an example of a complex system that can be constructed from these modules, a fully reversible microprocessor, Pendulum, is presented. This universal microprocessor has been fabricated in a 0.5 $\mu$m CMOS process and tested. Pendulum has eighteen instructions, including memory access, branching instructions, and direction changing instructions. The datapath supports twelve-bit wide data and address values.

The motivation for reversible computing research comes from an investigation of fundamental limits of energy dissipation during computation. A minimum energy dissipation of $k_{\mathrm{B}}T\ln 2$, where $k_{\mathrm{B}}$ is Boltzmann's constant, is required when a bit is erased. Erasing a bit is a logically irreversible operation with a physically irreversible effect. A reversible computer avoids bit erasure.

Fully reversible in the case of the Pendulum processor means reversible at the instruction set architecture (ISA) level and the circuit level. The primary contributions of this thesis are, first, the discoveries about how to design and build reversible switching circuits, including memory and control structures, and notational inventions that ease reversible system design, and second, the architectural innovations brought about by the development of the Pendulum reversible processor.

Thesis Supervisor: Thomas F. Knight, Jr.
Title: Senior Research Scientist, MIT AI Lab

*I didn't spend six years in Evil medical school to be called "Mister," thank you very much.*

*Dr. Evil in* Austin Powers: International Man of Mystery

## Acknowledgments

While I could never hope to write acknowledgments as cleverly as Dr. Olin Shivers [SC95], and I try to do without the "Jack 'n Zac", I would like to thank a few people. The length of this list is testament to the fact that I didn't do it all by myself. I have had an astounding series of opportunities to work with and for some fabulously intelligent people.

First is Mike "Misha" Bolotski, my officemate for two years, friend for six years, brewing partner, coffee buddy, fellow hockey addict, boss, and all around good guy. He taught me a lot about how to think and what it really means to be a Canadian.

I'd like to thank all my friends and colleagues. These are the people that have not only provided a social support structure over the far-too-many-years I spent working on this dissertation, but have helped me to be a better person: Raj Amirtharajah, Tom Simon, Saed Younis, for the use of numerous early SCRL documents and design data, Jim Goodman, Tony Gray, Jill Fekete, Graeme B. Shaw, Dana Stuart, Ray Sedwick, Nicole Casey, Phillip Alvelda, Matt "Maddog" Marjanovic, Vadim Gutnik, Halting Problem & Execution Time, Holly Yanco & Oded Maron, Rodney Daughtrey, Polina "Kiddo" Golland, Raquel Romano, Liana Lorigo, Charles Isbell, Carl "Don Juan" de Marcken, Gideon Stein, the "Scoring Line" of Jeremy deBonet, Chris Stauffer, & Erik Miller, Eric Sommers, Mark Foltz, Hugh Herr, Carlo Maley, Michael Leventon, Jeremy Brown, Rich Roberts and Maja Mataric, William "Sandy" Wells, Ian Eslick, Ethan Mirsky, and Silicon Spice for helping with funding, anyone who now has, ever has had, or ever will have, an office on 7ai, all the support staff in the AI lab and the EECS graduate office, especially Lisa Inglese, Eileen Nielsen, Monica Bell, and Marilyn Pierce, Anantha Chandrakasan, Dimitri Antoniadis, and Isabel Yang from the MIT MTL, and William Athas, Lars "Johnny" Svensson and the folks at ISI.

I am deeply indebted to the Pendulum Group at MIT: Matt deBergalis, Mike Frank, Norm Margolus, and Josie Ammer. They provided most of the inspiration for the work in this thesis and the motivation to stick with it. Thanks to all of you.

Josie Ammer and Amory Wakefield provided invaluable assistance with the XRAM chip.

Thanks to Lisa Dron McIlrath and Northeastern University, for helping with funding.

Thanks to everyone at The MicroDisplay Corporation for help during the final year of my graduate school career: Chris "Boo" Foley, Dave Huffman, Ya-Chieh Lai, Jean-Jacques Drolet, and the rest of the electronics group at The MicroDisplay Corporation, thanks to Rand Stadman, Todd Stiers, Chris Mullin, and special thanks to Mike O'Brien, Tim Wayda, and Terri Ross for service above and beyond in the development of the Pendulum test board.

Thanks to my thesis committee: Tom Knight, Gerry Sussman, and Gill Pratt. Their comments and suggestions have been tremendously helpful, and if I had listened more attentively to them this would have been a much better dissertation.

I can't thank Britte enough, but I'll try.

Thanks to my parents for supporting me for the last 27 years, and for encouraging me to spend 23 of them in school. You mean the world to me.

This work was supported by DARPA contract DABT63-95-C-0130.

# Contents

# List of Figures

# List of Tables

# Part I

# Introduction to Reversible Computing

# Chapter 1

# Introduction

## 1.1 My Thesis

There exists a set of reversible primitive modules that may be combined according to a set of simple rules to create larger systems, which are themselves reversible primitive modules. These modules may be combined to create a fully reversible, universal computer.

It is possible to design and fabricate a fully reversible processor using resources which are comparable to a conventional microprocessor. Existing CAD tools and silicon technology are sufficient for reversible computer design. I have demonstrated this by designing such a processor and fabricating and testing it in a commercially available CMOS process.

Fully reversible in this case means reversible at the instruction set architecture level and the circuit level. Programming the processor is only briefly treated in this dissertation, as are the underlying physics of bit erasure and reversibility. The primary contributions of this thesis are, first, the discoveries about how to design and build reversible switching circuits, including memory and control structures, and notational inventions that ease reversible system design, and second, the architectural innovations brought about by the development of the Pendulum reversible processor.

This thesis, while originally motivated by the promise of asymptotically zero energy com-

puting using reversible circuits, is relatively unconcerned with actual energy and power dissipation of the processor. The development, implementation, and testing of a fully reversible general purpose processor is a substantial advancement of the field.

Power and energy dissipation in modern microprocessors is obviously a concern in a great number of applications. Riding the technology curve to deep submicron devices, multi-gigahertz operating frequencies, and low supply voltages provides high performance and some reduction in dissipation if appropriate design styles are used, but for applications with a more strict dissipation budget or technology constraints, more unusual techniques may be necessary in the future.

Adiabatic or energy recovery circuit styles have begun to show promise in this regard. Motivated by the result from thermodynamics that bit erasure is the only computing operation associated with required energy dissipation, various techniques that either try to avoid bit erasure or try to bring the cost of bit erasure closer to the theoretical minimum have been implemented. To truly avoid bit erasure, and therefore perform computation that has no theoretical minimum dissipation, the computing engine must be reversible. Losses not associated with bit erasure are essentially frictional, such as the non-zero "on" resistance of transistors, and may be reduced through circuit design techniques such as optimally sized transistors, silicon processing techniques such as silicided conductors, and by moving charge through the circuit quasistatically as with constant current ramps in adiabatic circuits.

## 1.2 A Few Words About Words

Much discussion about reversible computing is hampered by imprecision and confusion about terminology. In the context of this dissertation, terms will hopefully be used consistently according to the definitions in this section.

The overall field of reversible computing encompasses anything that performs some operation on some number of inputs, producing some outputs that are uniquely determined by and uniquely determine the inputs. While this could include analog computation, this dissertation is only concerned with digital computation. Reversible computation only implies

that the process is deterministic, that the inputs determine the outputs, and that those outputs are sufficient to determine the inputs. Every state of the computing system uniquely determines the next state and its previous state.

A particular design may be reversible at one or more levels and irreversible at other levels. For example, an instruction set may be reversible and yet be implemented in an irreversible technology. The test chip "Tick" was an eight-bit implementation of an early version of the reversible Pendulum instruction set in an irreversible standard CMOS logic family.

The specific implementation of a reversible computing system may vary greatly. Numerous mechanical structures have been proposed, including Drexler's "Rod Logic" and Merkle's clockwork schemes [Mer93, Mer92]. Younis and Knight's SCRL is a reversible electronic circuit style suitable for use in the implementation of a reversible computer.

Many papers refer to "dissipationless" computing. This is a misnomer and generally may be interpreted as shorthand for "*asymptotically* dissipationless" computing. This means that if the system could be run arbitrarily slowly, and if some unaccounted-for $N^{th}$ order effect does not interfere, the dissipation asymptotically approaches zero in the limit of infinite time per computation. Only reversible computations may be performed asymptotically dissipationlessly because bit erasure requires a dissipation of $k_{\mathrm{B}}T \ln 2$ regardless of operation speed. Any $N^{th}$ order effects that prevent a reversible system from dissipating zero energy are essentially frictional in nature, rather than fixed, fundamental barriers.

One of the most abused terms is "adiabatic computing." While technically "adiabatic" means "without transfer of heat," it is generally incorrectly used to mean "quasistatic." Current flow in "adiabatic" circuits is externally controlled and ideally constant, and reducing the current flow allows the voltage drop across the on-resistance of a transistor to be arbitrarily small. In the limit, charge flows quasistatically, and the energy dissipation in the transistors falls to zero.

SCRL also has a confusing name, split-level charge recovery logic. All systems recover charge, the feature of SCRL is that the charge is recovered at a higher energy than in a conventional system. It could more accurately be called split-level *energy* recovery logic, because the

energy is recovered rather than dissipated as heat.

Reversible architecture designs also have a number of confusing terms. Section 8.5 mentions the production and management of "garbage" information. In Ressler's early reversible computer work [Res81] and Vieri's reversible instruction set design [Vie95], intermediate information produced during computation is consigned to a "garbage stack." The pejorative term "garbage" is undeserved, and the results referred to are only garbage in the sense that they are not explicitly required in the computation at the time; they are necessary for reversibility and may be useful or necessary later. The programmer may be clever enough to re-use the values or "recycle" them to reclaim the storage space without erasing information.

In the context of CMOS-based circuits, we may talk of the energy on a node being "reclaimed" or the voltage may be "restored" to some value. The implication is that the node is being quasistatically reset to a known value. The node may be reset from ground or from $V_{dd}$ to an intermediate value, so the energy stored on the node may be increasing or decreasing. A logic stage using one set of signal values to compute another set of values and restore the second set of nodes to a known state is generally referred to as a "reverse" node. The imposition of direction on the energy flow, "forward" referring to energy flowing from the power supply to the internal nodes of the system, and "reverse" referring to energy being returned to the power supply, is somewhat arbitrary, although hopefully intuitive.

## 1.3 Structure of this Dissertation

The thermodynamic background and the motivation behind reversible computing research are discussed in Chapter 2. The various implementation techniques, from mechanical logic to biological systems, are briefly addressed. The bulk of this dissertation is devoted to the lessons learned from developing the Pendulum Reversible Processor. This processor is a none too distant descendent of traditional, indeed early, RISC architectures. Part II is a step-by-step examination of general reversible logic design. The specifics of the Pendulum processor, including datapath design, instruction set choices, and processor control, are found in Part III.

It has been suggested that a reversible circuit technology might find a more appropriate application in a more radical architecture. The Flattop billiard ball computer design exercise, discussed in Chapter 10, was an investigation of an alternative processor architecture. The inherently reversible cellular automata used in the Flattop chip maps well into a reversible energy recovery logic implementation. Other architectures that take advantage of the structure of an underlying reversible implementation might also be powerful. Open questions such as this are addressed in Part IV.

## 1.4 Contributions of this Work

This dissertation presents two substantial contributions. The first is a set of techniques for creating reversible logic modules and connecting them to create larger reversible modules. As such, it is a primer for future reversible logic designs. The second contribution is the Pendulum Reversible Processor, a fully-reversible twelve-bit microprocessor. It serves both as a design example for the reversible logic design techniques mentioned above, but also contributes to the understanding of reversible computer architecture, including instruction set design, logic block design, and datapath organization.

The Pendulum processor is currently the largest fully reversible system ever implemented. Other than the designs described in this dissertation, Younis's eight-bit multiplier [You94] was the largest fully reversible system. It contained ten levels of logic and was fully combinational.

Developing the Pendulum processor required invention of a notation and methodology for constructing and describing reversible logic. Using those techniques, a cell library, including memory elements, had to be constructed. Those cells were used to implement large datapath modules, such as the register file, the ALU, and the program counter update unit. An instruction set and datapath architecture were also designed, determining the datapath module interconnections. Some techniques for assembly language programming were also developed.

# Chapter 2

# History of and Motivation for Reversible Computing

## 2.1 Relationship of Reversibility to Energy Dissipation

Reversibility requires that each state of a computing system uniquely determine both the next state and the previous state. This implies that no information is erased in the transition from one state to the next.

The link between entropy in the information science sense and entropy in the thermodynamics sense is demonstrated by Maxwell's demon [Max75] and was discussed by Szilard [Szi29]. Landauer [Lan61] showed that erasing a bit requires dissipation of $k_{\mathrm{B}}T\ln 2$ energy units, where $k_{\mathrm{B}}$ is Boltzmann's constant. Erasing a bit is a logically irreversible operation with a physically irreversible effect. Bennett [Ben73] then showed how a reversible computation could be performed.

By avoiding information erasure, the compute engine avoids this energy dissipation. The other, essentially frictional, sources of dissipation may be reduced arbitrarily by quasistatically transitioning the compute engine from state to state. A system that avoids bit erasure may therefore be operated with asymptotically zero energy dissipation. An irreversible system has a fundamental lower limit to the energy dissipated during a computation, pro-

portional to the number of bits erased.

Any system with asymptotically zero energy dissipation may not erase information, and a system that does not erase information is fully reversible. An irreversible system may store information produced during a computation, rather than erasing it, but not provide a path from each state to its previous state. This type of system merely delays the energy dissipation due to bit erasure, since the information stored during computation represents an unrecoverable quantity of energy added to the system during computation. The energy used to store this information is unrecoverable unless the system is reversible.

To be fully reversible, a compute engine must be reversible at all levels of its design. Reversible electronic computers must have a reversible power supply, which power reversible circuits, which implement a reversible architecture, which runs reversible programs.

It may seem obvious why reversible circuits and a reversible power supply are required to avoid energy dissipation, since these are the parts of the system actually shuttling the energy through the system, but it is less obvious why a reversible architecture and reversible code are required. The reason is that if a program or instruction set erase information, it is impossible to build a reversible circuit that performs that operation. Bottom up pressure from the circuit and power supply level dictates that irreversible operations are prohibited at all levels of the system if dissipation due to bit erasure is to be avoided. The reversibility may be "hidden" or made more palatable, but each part of the system must be reversible to avoid the energy dissipation from bit erasure.

## 2.2 Implementation Techniques

A logically reversible circuit may be implemented in a variety of different physical structures. During the twenty years between Bennett's 1973 description of how a reversible computation might be performed and the 1993 discovery of SCRL by Younis and Knight [YK93], a number of mechanical "thought-experiment" style implementations were proposed. This section discusses some of the schemes that have been proposed.

Various nanotechnology researchers have become interested in mechanical reversible logic. The macroscopic models they envision involve sliding rods, turning gears, "locks" and "keys," and various other parts that would be at home in a Babbage engine. Fredkin's billiard ball model [FT82] became popular for describing reversible systems. Merkle's mechanical logic [Mer92] was motivated by nanotechnology.

The focus of this work is on digital electronic computation, primarily using the SCRL logic family. Invented by Younis and Knight [YK93], it has proven to be a useful logic style for constructing reversible computation structures in standard industrial silicon processes. Athas's Adiabatic CMOS group at USC's Information Sciences Institute, has applied selective reversibility to reduce the energy dissipation of practical systems [TA96, ATS+97, ASK+94, KA92, AS94], including energy recovering power supplies [AST96].

Moving electrons at a finite rate through a resistive element dissipates energy whether or not information is erased. Eliminating the resistive elements eliminates this source of energy dissipation. Superconducting switching elements, such as Josephson Junctions [Lik82], avoid resistive losses, but all AC superconducting circuits dissipate some energy.

Computing with a quantum system, in which so-called q-bits may be in a superposition of both one and zero states, *requires* reversible logic. If a quantum system is coupled to the environment, the superposition is lost, resulting in dissipation, but more importantly, the computational advantage of the quantum system disappears [Sho94, Fey86].

The replication of DNA in living cells is very efficient, with only a few tens of $kT$ dissipated during each operation [Ben82]. Recently it has been proposed that existing cellular machinery may be manipulated to perform computations, with the results being expressed by the production of various detectable proteins.

## 2.3 Reversible Architecture and Programming

Ressler's work, designing and implementing a reversible computer using entirely the conservative Fredkin gate [Res79, Res81], was the first attempt at a fully reversible computer.

The physical implementation suggested was Fredkin's billiard ball model. Hall's work with retractile cascades [Hal92, Hal94] was motivated by nanotechnology, and his proposal for a reversible architecture was based on a PDP-10. No implementation technology was proposed.

Baker proposed various reversible programming techniques [Bak92b], including a reversible lisp [Bak92a], and Frank addressed a number of issues with algorithms and programs for reversible computers [Fra99].

Vieri's Master's thesis suggested a RISC-like architecture and programming style [Vie95]. This work is a direct extension of that research. The following parts of this dissertation cover building digital, electronic, SCRL logic to implement the Pendulum reversible microprocessor.

# Part II

# Building Reversible Circuits

# Chapter 3

# Fundamentals of Reversible Logic

## 3.1 Reversible Logic Design Styles

A number of reversible circuit styles proposed to date, from charge steering logic discussed in Section 5.1.3, to Hall's retractile cascades [Hal92], to SCRL, have in common the distribution of power[1] and timing information on the same set of wires. These "power clocks" may be driven by a resonant, energy recovering power supply, or by more conventional means.

The SCRL logic family requires between sixteen (for two-phase logic using only inverting stages) and thirty-six clock rails (for four-phase logic using non-inverting stages). SCRL logic uses both rail polarities for each timing phase, so the number of different phase clocks is half the total number of clocks. It may be possible to eliminate the full swing t-gate control clocks since they do not actually power the logic.

By contrast, retractile designs require one clock phase for each level of logic, and are therefore not regularly scalable. The Pendulum processor would require 73 power clocks if designed using retractile logic, a factor of more than two over the thirty SCRL clocks used. The logic would have been much simpler, however, requiring half as many transistors. The differences between retractile systems and pipelined systems such as SCRL are discussed

[1] Most reversible CMOS-based logic styles also require distribution of DC valued supply rails for well and substrate taps. Ideally no energy is dissipated from these supplies.

below.

### 3.1.1 Retractile and Pipelined Design Styles

The three fabricated integrated circuits fabricated in the course of this research, the billiard-ball-model cellular automaton chip Flattop, the eXchange RAM chip XRAM, and the fully reversible Pendulum processor, were all designed using three-phase SCRL logic. The transistor-level discussions are often specific to SCRL, but the notational and hierarchical lessons are generally applicable to any pipelined reversible logic family using switching elements like CMOS transistors.

Hall's "retractile cascade" [Hal92] is one of the earliest forms of reversible switch-based logic proposed. The distinguishing feature between reversible pipelined logic and retractile cascades is that in a retractile cascade the energy enters and exits the system through a single path. A pipelined system has separate paths for the energy to enter and exit. In a pipelined system, input signals need not be held for the entire duration of the computation. A retractile cascade requires that the first signal to be asserted be held until the all subsequent signals have been asserted and retracted. The number of power rails for a retractile cascade circuit therefore depends on the depth of the computation. A pipelined circuit may have an arbitrarily deep computation path with a constant number of rails. For most variations of SCRL this number is approximately twenty.

Some parts of the design styles discussed below are applicable to retractile designs. Retractile designs are not particularly scalable, and a pipelined design style can perform the same operations as a retractile design, but with potentially fewer clock signals. However, since energy flows into and out of the system through the same path in a retractile cascade, it does not require the top to bottom symmetry discussed below in Section 3.2. The logic must still be symmetric left to right. The single symmetry of a retractile cascade suggests that perhaps a pipelined logic family could be developed that only requires one axis of symmetry.

The focus on SCRL in this dissertation is purely pragmatic; other superior reversible logic

families are sure to be developed and the design techniques below refined, but the general strategies of using abstraction and hierarchy and careful logic design are applicable to design of any complex system.

### 3.1.2 Younis's Pipelined SCRL

Detailed coverage of Younis's SCRL reversible logic style may be found in the literature [You94, YK93, YK94]; this section serves only as review so that Chapter 4 may be more intelligible. Readers interested in the development and energetics of SCRL are encouraged to consult the literature.

The development of SCRL resulted in two rules for constructing reversible CMOS logic. The first is that a transistor may only be turned on when no potential difference exists between its source and drain. The second is that charge flow between any two nodes in the circuit occurs gradually and under external control. These two rules may only be obeyed for every node in the circuit if reversible logic is employed. By using reversible logic and following these rules, the circuit will not dissipate any energy due to bit erasure. The circuit will suffer resistive losses from the finite on resistance of transistors, and losses in the power supply may be substantial depending on its design, but none of the dissipation will be due to bit erasure, and the losses are therefore asymptotically zero.

Section 4.1 treats standard CMOS energetics and how to convert a CMOS inverter into an SCRL inverter. SCRL does not use DC supply rails to power the logic. Swinging rails are used to ensure that when a gate's input values change, the potential on both sides of the transistors are the same. When the input values are stable, the rails can swing, gradually and under external control so as to obey the second rule, and the output nodes will transition along with the swinging power supply. The slowly swinging rails ensure that the potential drop across the transistor is small, reducing $V^2/R$ dissipation in the device. In the limit, charge flows quasistatically through the circuit. If reversibility is broken, a device's input will change with a potential across the device, and the sudden current flow across this large potential drop dissipates the energy associated with the bit being erased.

In the discussions that follow, it is helpful to remember these two rules. All the complexities of reversible logic, swinging rails, and signal timing are endured to ensure that these two rules are obeyed. SCRL is just one possible logic family that obeys these rules, but other CMOS-based reversible logic is subject to the rules as well.

## 3.2 Symmetry

Symmetry is an important aspect of reversible system design. For the most part, wherever symmetry is broken, reversibility is likely to be broken. The reversible systems analyzed in this dissertation are all excruciatingly symmetric. The symmetry is perhaps overdone. Each reversible module, as will be shown in Chapter 4, contains a part that drives a value onto the output and a part that uses the output value(s) to restore the input(s) to a known value. The computation proceeds from one logic stage to the next in a pipelined fashion. Each module is symmetric in having a "forward" function and "reverse" function that computes the inverse of the forward part. If the computation is proceeding from left to right, the module is symmetric top to bottom.

The logic is also generally symmetric from left to right, along the computation path. Signals that are decoded are later encoded. Values computed are later uncomputed. Every signal fan-out has a counterpart fan-in somewhere downstream. Chapter 6 analyzes in detail a reversible adder; the adder modules contain both the internal top to bottom symmetry, but the adder as a whole performs an add and an "unadd."

Pipelined design considerations prompt the top to bottom symmetry consideration. An SCRL logic gate, or any other reversible pipelined logic gate, has separate paths for adding and removing energy from a capacitive node. These separate paths form the top to bottom symmetry mentioned above. A retractile design uses the same path for adding and removing energy, and therefore does not require top to bottom symmetry. The cost is that every stage of logic must have its own power clock. A pipelined design reuses a single set of power clocks, and is therefore more easily scalable.

The left to right symmetry arises from a desire to recover the energy stored in intermediate

values produced during the course of a computation. During modulo addition, for instance, a ripple-carry adder computes a multi-bit sum $S$ from two multi-bit operands $A$ and $B$. When the final bit of $S$ is computed, the output of the system, if every stage has been performed reversibly, is $S$, $A$, $B$, and the final carry out bit. These three multi-bit values and one extra bit are a redundant encoding of the desired information. Modulo addition of two input values need only retain $S$ and $B$; the $A$ and carry out values are merely intermediate "garbage" values, retained to ensure that each stage of the ripple add may be performed reversibly. These intermediate values may be uncomputed by using an "unadd" chain. This is symmetric left to right to the forward adder. This symmetry is unnecessary if the intermediate values are desirable, or even if the modulo addition is performed in some other way so as to not produce the intermediate values at all. However, reversible computing often exhibits this compute-uncompute symmetry, either spatially in hardware or temporally during execution [Ben88, Ben89].

## 3.3 Conclusion

Reversible logic may be constructed in a variety of logic styles. Pipelining the logic allows a fixed set of clocks to power the entire, scalable system. This requires that the logic be more complex, supporting symmetric paths for energy to enter and exit the computing system. The next chapter presents techniques for constructing and composing reversible logic modules, concentrating on a notation and methodology for SCRL circuits.

# Chapter 4

# Pipelined Reversible Logic Design

The focus of this dissertation is the development of techniques for designing complex reversible logic systems. The Pendulum Reversible Processor, presented in detail in Part III, is a suitably complicated example, and many of the lessons in this chapter were learned during the development of the Pendulum processor. Because of this background, the techniques below are slanted towards the use of three-phase SCRL CMOS logic. It must be emphasized however that they are, for the most part, generally applicable to any switch-based implementation technology, and even to any pipelined reversible logic style. Three-phase SCRL was chosen over other variants of SCRL because it requires the fewest power rails while always actively driving all circuit nodes.

This chapter very deliberately follows much of the structure and style of the seminal book by Weste and Eshraghian, *Principles of* CMOS VLSI *Design*. I hope that imitation is the sincerest form of flattery.

## 4.1 The Inverter

The standard CMOS inverter is the traditional introductory logic gate in conventional textbooks. This section analyzes the CMOS inverter and the necessary modifications to create an SCRL inverter. The basics of a conventional CMOS inverter and the flow of energy through

the inverter are examined, followed by the construction of a "clocked inverter" and a shift register. Then the conventional inverter DC power supplies are replaced with SCRL's swinging rails. Finally, the complete SCRL inverter is examined in detail. Younis [You94] is responsible for the development of SCRL logic, so this section may be considered essentially a review of the state of the art with respect to SCRL's novelty. The design techniques, symbology, and notation, however, are contributions of this dissertation.

This and the following sections assume the reader has a fairly intimate understanding of conventional CMOS circuits and Boolean logic design.

A standard CMOS inverter, shown in Figure 4-1, uses two voltage-controlled, resistive switches to charge and discharge a load capacitance. The input to the switches is modeled as a step. When the pmos switch is on, current flows from the positive supply through the on-resistance of the switch, pulling the load capacitance to the upper rail voltage. The load capacitance charges to a value $V$ with a time constant $RC$ where $R$ is the on resistance of the pfet device. During charging, $\frac{1}{2}CV^2$ units of energy are dissipated in the resistive pfet device and an equal amount of energy is stored on the load capacitor.

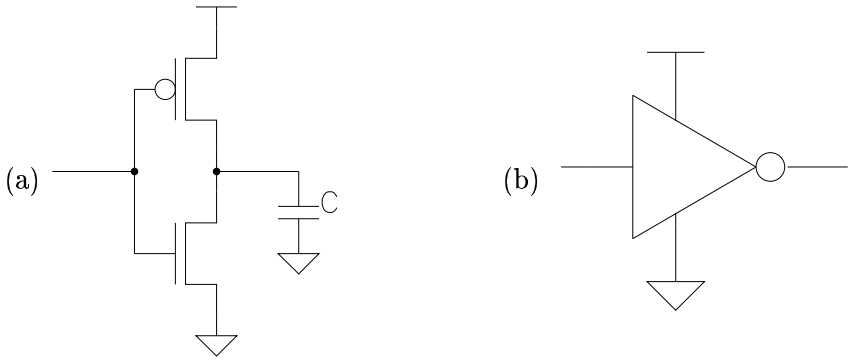


Figure 4-1: The conventional CMOS inverter (a) schematic showing parasitic capacitance and (b) symbol.

When the input value changes, the charge stored on the load capacitor is discharged through the on resistance of the nmos switch to ground. To first approximation, energy is only dissipated during transitions. More accurately, the switches are slightly "leaky," and a small current flows when the input is stable. This leakage may be reduced through a

number of techniques, including process modification and by increasing the gate length of the devices. In the current discussion leakage will be ignored.

The inverter output changes some small delay after its input changes. It is often desirable to regulate the time at which the output node changes, so we now add a clocked output stage to the simple inverter, shown in Figure 4-2. The output stage is merely a simple

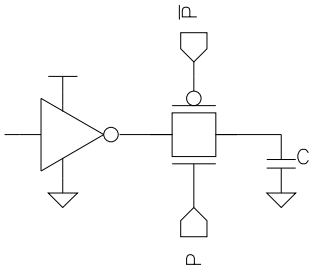


Figure 4-2: A clocked CMOS inverter.

complementary transmission gate. The nmos side is clocked with clock signal $P$ and the pmos side is clocked with its inverse, $\overline{P}$. When the t-gate is on, the input value is propagated to the output, and when the t-gate is off, the output is isolated from changes of the inverter. This "clocked inverter" will become our first primitive building block. Fan-out is a simple matter; the output of one clocked inverter can drive multiple clocked inverter inputs. Conventional concerns of $RC$ delays and transistor sizing apply.

Any number of clocked inverters can be connected together into a clocked shift register using two pairs non-overlapping clock signals, $P_1$, $\overline{P_1}$, $P_2$, and $\overline{P_2}$, as shown in Figure 4-3. The clock signals and waveforms at the labeled nodes are shown in Figure 4-4. The shift register could be clocked using only one pair of clock signals by connecting the gates of alternating t-gates to the opposite polarity clock signal, as shown in Figure 4-5, if the clock edges are sharp enough to prohibit the signal from passing through more than one clocked inverter at a time.

An obvious shortcoming of this clocked inverter shift register is that the input nodes to each inverter, nodes A, C, and E in Figure 4-3, are not actively driven and rely on capacitive charge storage for half the time. The next modification to the clocked inverter is the addition

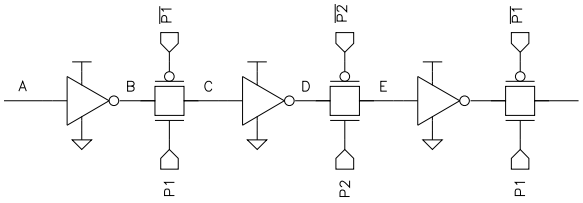


Figure 4-3: A clocked CMOS inverter shift register.

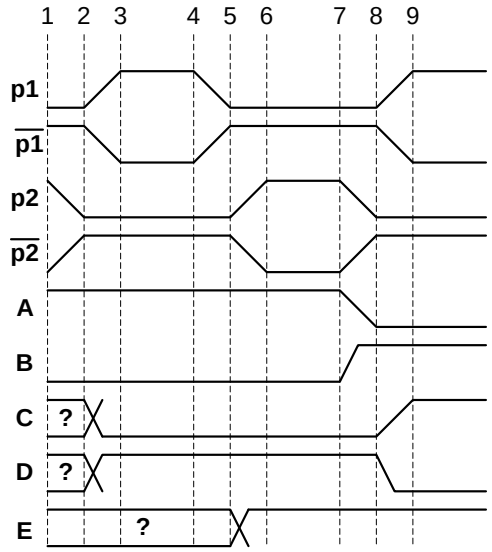


Figure 4-4: Clocked CMOS inverter shift register timing. Note that the edges of signals B and D are sharpened by the inverters. Nodes C and E only change when the pass gates they are attached to are turned on. Nodes C, D, and E have unknown values until the shift register input, A, has been clocked to those nodes.

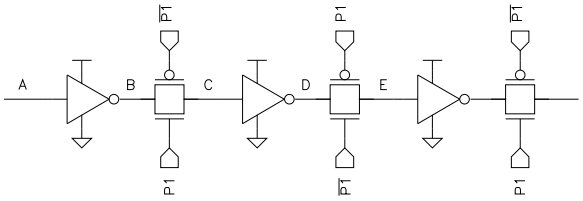


Figure 4-5: Alternate clocking of the clocked CMOS inverter shift register.

of a "keeper" gate to ensure that all nodes are actively driven all the time. Conveniently, another clocked inverter will serve as a suitable keeper gate. Figure 4-6 shows three clocked inverters added as keeper gates to the shift register. The keepers' t-gates are clocked with the opposite polarity clocks as the t-gates in the clocked inverter above and to the left of the keeper.

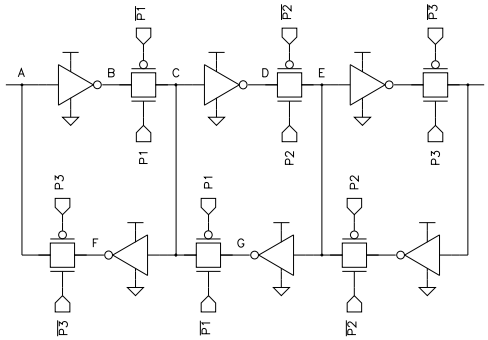


Figure 4-6: A clocked CMOS inverter shift register with keeper gates.

This approach to using a clocked inverter as keeper gate is in some sense more complicated than necessary. The addition of a pfet, as shown in Figure 4-7, will hold the input high when the output is low, but the input still floats when the output is high. This scheme also relies on the keeper devices being weak compared to the inverters. An unclocked inverter could be put in the feedback path, creating something resembling a conventional SRAM cell but requiring careful transistor sizing to ensure proper operation. The clocked inverter keeper is a step towards a fully reversible SCRL shift register.

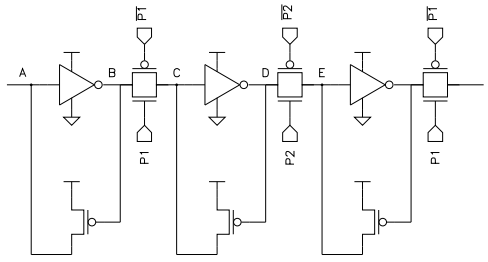


Figure 4-7: A clocked CMOS inverter shift register with pfet keepers.

This clocked inverter keeper is not applicable to a string of arbitrary gates. Gates such as

NAND or NOR have multiple inputs and only one output and cannot, in general, use a set of keepers to hold the input values stable based only on the output value. At this point it should be clear that the notion of reversibility is creeping into the discussion. The keeper gates used in the clocked inverter shift register use the output value of a clocked inverter to generate the input value to the clocked inverter. The notion of using the output of a gate to generate the input to the gate is fundamental to building reversible logic.

The clocked inverter shift register requires one additional modification to turn it into an SCRL shift register: the DC supply rails powering each inverter must be converted to swinging "power clocks." This is to obey the SCRL constraint, brought up in Section 3.1.2, that charge flow quasistatically through the resistive transistors. It is fundamentally important to note that ensuring quasistatic charge flow at all times in a circuit, and therefore potentially asymptotically zero energy dissipation, *requires* the use of reversible logic. The dissipation of $\frac{1}{2}CV^2$ in the on-resistance of a conventional inverter's transistors during switching is due to the large potential drop across the switch when it is turned on. By keeping the potential drop across the transistor small, the associated dissipation may be made arbitrarily small. Rather than charge the load capacitance with a constant voltage, SCRL and other adiabatic CMOS logic use a constant current. For a linear capacitance, that implies a voltage ramp. The energy dissipation, from Seitz [SFM+85] in Athas [AS94], when charging a load capacitance $C$ to a voltage $V$ through a resistance $R$ using a constant current $I$ is

$$E_{diss} = P \cdot T = I^2 R \cdot T = \left(\frac{CV}{T}\right)^2 RT = \left(\frac{RC}{T}\right) CV^2$$

Increasing the charging time $T$ therefore, to first order, reduces the energy dissipated during switching.

Proper SCRL circuit design ensures that no transistor is turned on with a potential drop across it. Node voltages only change when following the controlled ramp of the supply voltage. Proper SCRL power supply implementation ensures that the current through transistors flows in a controlled manner. The energy-recovering details of an appropriate power supply are beyond the scope of this dissertation, but various designs can be found in the literature [BK98, AST96, SK94].

Beginning with the clocked inverter from Figure 4-2, the $V_{dd}$ and ground connections are replaced with two complementary swinging power clocks, $\phi$ and $\overline{\phi}$. These clocks each ramp from $V_{dd}/2$ to $V_{dd}$ and ground, respectively, as shown in Figure 4-8. The signal $\phi$ and its

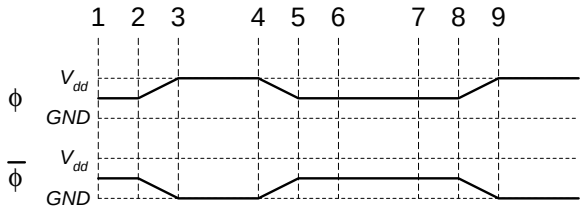


Figure 4-8: The SCRL split-rails $\phi$ and $\overline{\phi}$.

complement are always used together, and they may be referred to as the $\phi$ rail pair, or sometimes just as the $\phi$ rails. The slope of these ramps determines the dissipation in each transistor since the charging time varies, and the dissipation falls according to the equation above. The DC powered clocked inverter, on the other hand, drives the output node, C in Figure 4-3, with a step as soon as the t-gate is switched on. The clocked inverter charges the capacitive node with time constant $RC$ where $R$ is the on-resistance of the charging (or discharging) transistor plus the on-resistance of the t-gate. By using swinging power clocks, node C may be gradually ramped from $V_{dd}/2$ to $V_{dd}$ or ground with a time constant determined by the slope of the rail. A single clocked inverter with swinging power rails and their timing are shown in Figure 4-9. This structure is an SCRL "half-inverter."

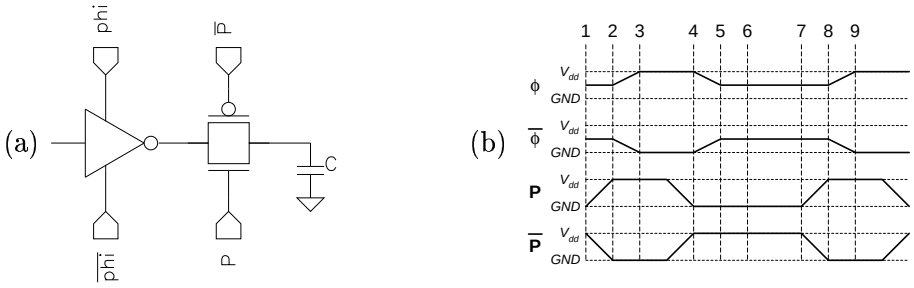


Figure 4-9: The SCRL half-inverter (a) schematic and (b) supply clock timing. The load capacitor is parasitic, not an explicitly added element.

The astute reader may now be asking how the output node returns to $V_{dd}/2$ rather than staying at $V_{dd}$ or ground. The answer is that the keeper gate connected to the output

node, itself an SCRL half-inverter, is responsible for restoring the voltage to $V_{dd}/2$. This topology is shown in Figure 4-10. The keeper gate has become a "keeper/restorer." Four half-inverters are shown; the two half-inverters of interest are in the upper left and lower right.

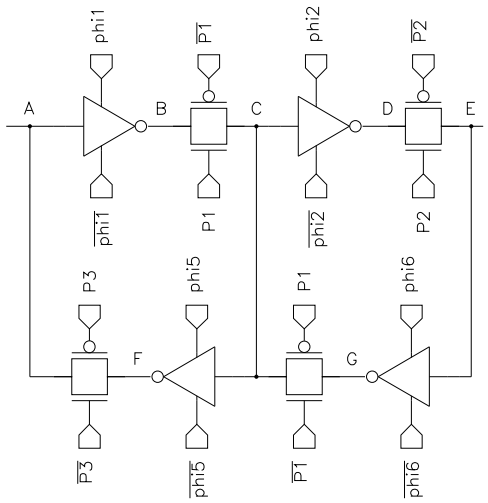


Figure 4-10: Four SCRL half-inverters.

The number of swinging rails has increased, and the rail timing for Figure 4-10 is shown in Figure 4-11. Note that the keeper half-inverter on the lower right must be able to generate the value at node C from the value at node E in order to restore node C to $V_{dd}/2$. The half-inverter on the upper left sets node C from a known intermediate value to a valid logic one or zero. The keeper gate on the lower right then resets node C to the known intermediate value. When charging node C to $V_{dd}$, energy from the power supply flows in to node C through one half-inverter then back to the power supply through the other half-inverter. An SCRL node is always set to a valid value by the upper half-gate and is restored to $V_{dd}/2$ by the lower half-gate. This is in contrast to a conventional logic gate in which the energy flows from $V_{dd}$ to the output node through the pmos devices and out to ground through the nmos device of the same gate.

The number of devices and clocks used so far has grown enough that some hierarchy is useful. The SCRL full inverter symbol is shown in Figure 4-12 (a). Note that the power

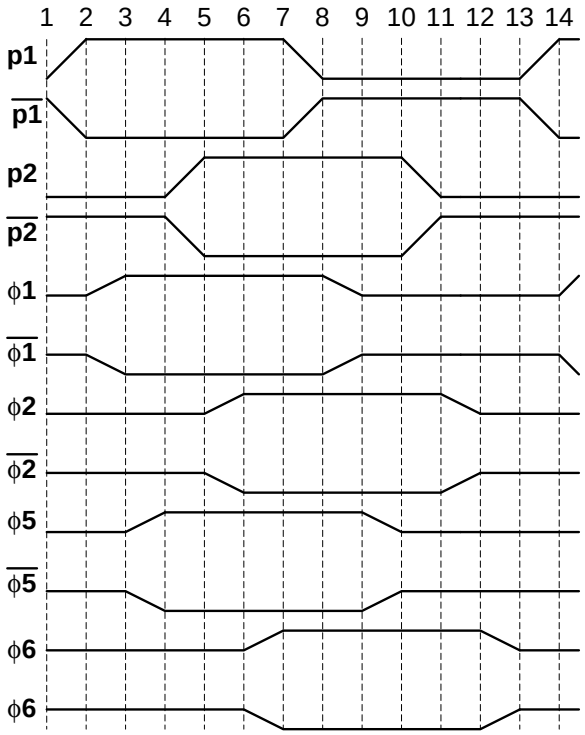


Figure 4-11: Timing signals for a shift register of four half-inverters.

clocks are bundled together in the "rails" pin. It may at first seem that the upper left and lower right half-inverters in Figure 4-10 should make up a full SCRL inverter because of their energy-flow relationship. However, the keeper half-inverter uses the output of the inverter above it to restore node C to $V_{dd}/2$. The keeper must know what logical function the gate above it is performing to return its output node to $V_{dd}/2$. Therefore, an SCRL gate in general consists of a clocked gate and the keeper gate directly below. A properly designed full SCRL gate is able to drive its output node to a valid logic level and restore its input node to $V_{dd}/2$. The schematic for the SCRL full inverter is shown in Figure 4-12 (b). This full inverter abstraction allows the designer to use the gate for its logical behavior and not worry about the rail timing or energy flow of the gate. Designing a gate requires only that the inputs to the gate be restored to $V_{dd}/2$ by the keeper gate.

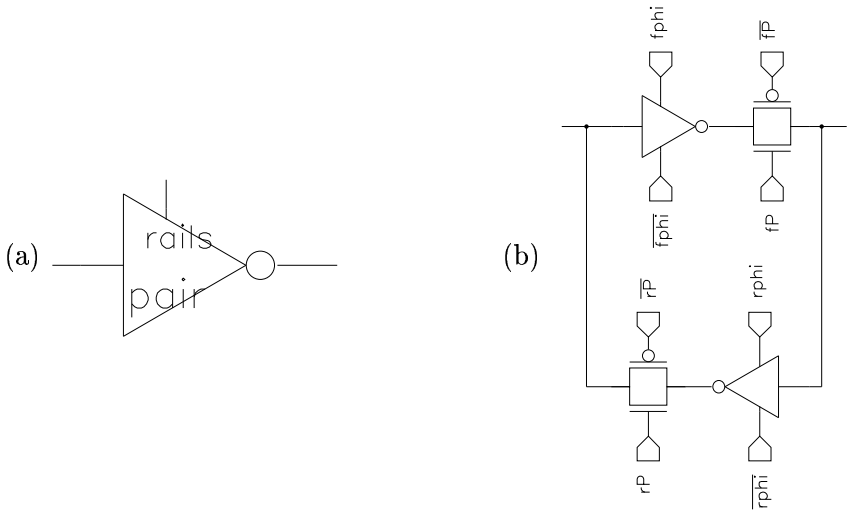


Figure 4-12: The SCRL full inverter (a) symbol and (b) schematic.

For convenience, we refer to the upper half-gate as the forward gate and the keeper gate as the reverse gate. The power clocks and t-gate control signals in Figure 4-12 (b) are prefixed "f" and "r" for forward and reverse, following this convention. This choice of label is for the most part arbitrary. Digital designers may find it convenient to think of the computation proceeding in a particular direction. The symbol for an SCRL full inverter implicitly suggests a direction for computation despite being completely symmetric internally. The following section details the development of an SCRL full NAND gate, which is not fully symmetric.

It is presented as an example of how to construct the schematic and hierarchical symbolic representation for a general combinational logic gate.

## 4.2 Combinational Logic

### 4.2.1 The NAND Gate

The inverter is the classic starting point for understanding a logic family; the NAND gate is the classic starting point for building computational systems. The truth tables for a conventional NAND gate and the SCRL NAND gate to be developed are shown in Table 4.1. The most obvious thing to notice is that the SCRL NAND gate has three outputs: $\overline{A}$, $\overline{B}$, and $\overline{A \cdot B}$. Because it has more outputs than inputs it is called an *expanding* operation. In contrast, the inverter is a *non-expanding* operation having one input and one output. Chapter 6 presents similar, higher level examples of expanding and non-expanding operations in a microprocessor ALU.

| conventional NAND | | | SCRL NAND | | | | |
|---|---|---|---|---|---|---|---|
| in | | out | in | | out | | |
| $A$ | $B$ | $\overline{A \cdot B}$ | $A$ | $B$ | $\overline{A \cdot B}$ | $\overline{A}$ | $\overline{B}$ |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

Table 4.1: Conventional and SCRL NAND truth tables.

The three outputs of the SCRL NAND gate are relatively easy to produce. The $\overline{A \cdot B}$ output is generated from an SCRL half-NAND, shown in Figure 4-13 (a). As with the SCRL half-inverter, the t-gate is clocked with full-swing signals $P$ and $\overline{P}$ and the output is driven by half-swing power clocks $\phi$ and $\overline{\phi}$.

Note that a generalized SCRL half-gate is composed of a logic portion and a t-gate portion. The $\phi$ signals drive the logic portion and the $P$ signals control the connection between the logic portion and the output node. The logic portion may be an arbitrarily complex

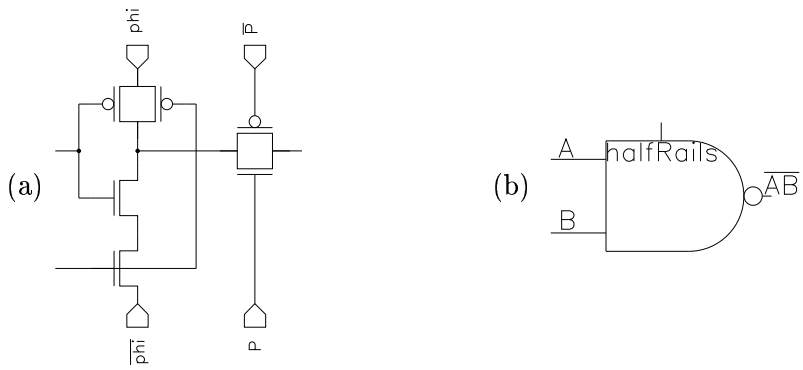


Figure 4-13: SCRL half-NAND gate (a) schematic and (b) symbol.

single level of logic. Traditional transistor sizing rules apply, however, so a large gate may quickly become unwieldy. Logic minimization rules also apply, so Karnaugh maps or other minimization tools may be used to simplify the pull-up and pull-down networks in the logic portion of any SCRL half-gate. The only constraint is that it must be single-level logic, *e.g.*, no NAND blocks driving NOR blocks in the logic portion of a single SCRL gate are allowed. Relatively complex pull-up and pull-down networks are common in reversible logic designs. Each pipeline phase is constructed to perform as much computation as possible so as to amortize the SCRL power clock overhead.

The SCRL NAND gate requires three outputs so that the reverse part of the gate, the keeper, may restore the input nodes to $V_{dd}/2$. The two outputs $\overline{A \cdot B}$ and any one other are insufficient to uniquely determine the inputs and thereby quasistatically restore them to $V_{dd}/2$. The two additional outputs, $\overline{A}$ and $\overline{B}$, are chosen somewhat arbitrarily. Non-inverted values of $A$ and $B$ would also be sufficient to determine the input values, but inverting gates are simpler to implement in CMOS than non-inverting gates. The $\overline{A}$ and $\overline{B}$ outputs are each generated using SCRL half-inverters.

The two input values can be restored merely by adding a pair of half-inverters as keepers for the two half-inverters that generate $\overline{A}$ and $\overline{B}$. The output of the half-NAND is not needed nor used to restore the inputs. The full SCRL NAND gate, shown in Figure 4-14 (a), is composed of a half-NAND and two full inverters. All three components are clocked in parallel by the same set of swinging rails. The hierarchical symbol for the SCRL NAND is

shown in Figure 4-14 (b).

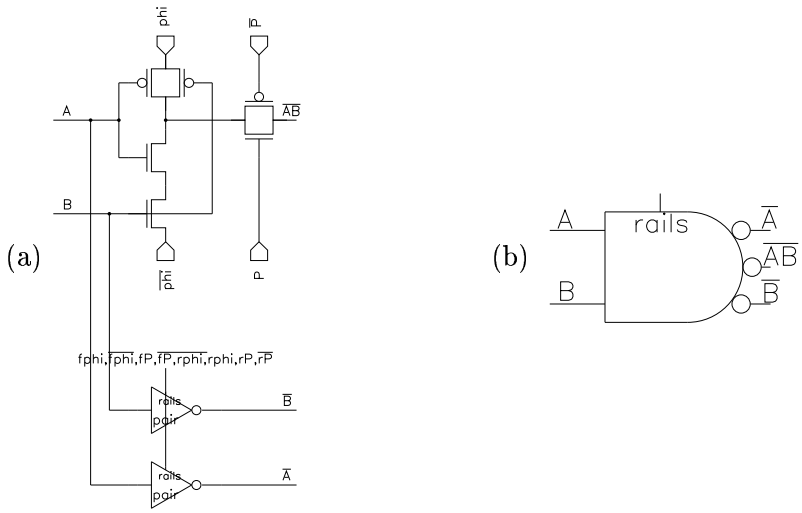


Figure 4-14: SCRL full NAND gate (a) schematic and (b) symbol.

In practice, it is often more convenient to use the half-NAND from Figure 4-13 (b), since the $A$ and $B$ signals may pass through another section of the system. In this case, the designer must be careful to make sure that signals $A$ and $B$ are indeed restored to $V_{dd}/2$ by another gate. "Breaking the rules" in the sense of creating a gate that does not itself restore its inputs to $V_{dd}/2$ is possible but requires additional designer diligence, namely that another part of the system properly restore the input values to $V_{dd}/2$.

Note that NAND gates with more than two inputs may be constructed similarly to the two input NAND gate. The logic portion of the half-NAND must be changed to a three-input NAND, and each additional input must be connected to a full inverter feedthrough gate. A two-input SCRL full NAND gate uses twenty-two transistors: six in the half-NAND and eight in each full inverter. A three-input SCRL full NAND gate uses thirty-two transistors: eight in the three-input half-NAND and eight in each full inverter.

### 4.2.2 The Un-NAND Gate

Since reversible computing requires that information not be discarded, the use of expanding operations, in this case the NAND gate discussed above, burdens the designer with additional

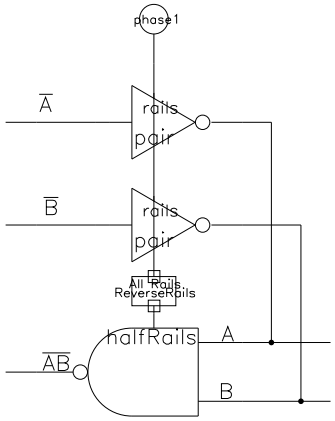


Figure 4-15: SCRL un-NAND operation using a half-NAND gate and two full inverters.

signals to feed through the system. At his or her earliest convenience, the designer may want to perform a symmetric un-expanding operation to eliminate this additional signal. In a sense, the three outputs of the SCRL NAND gate are a redundant encoding of the two inputs $A$ and $B$. Performing an "un-NAND" operation, shown in Figure 4-15, is merely a more efficient encoding of the three signals. This practice of decoding a set of signals, in this case from $A$ and $B$ to $\overline{A}$, $\overline{B}$, and $\overline{A \cdot B}$, at the last possible moment, using the decoded signals for some computation, then re-encoding the signals as soon as possible, is a powerful technique in reversible system design. This technique of "just in time" decoding is discussed from a system point of view in Section 8.3.

We have now constructed a full inverter and multiple-input half-NAND gates. This is enough to construct as complex a computing system as desired. Section 4.3 discusses proper technique for interconnecting gates and a simple bookkeeping device to keep track of the plethora of swinging power clocks. The next two subsections cover additional combinational logic gates.

### 4.2.3 The XOR Gate

The XOR gate in conventional CMOS is a sort of "black sheep" of the family. It is not a universal gate[1]; NAND and NOR are more powerful in that sense. It is not a conveniently implementable gate; it requires far more transistors than NAND or NOR. It is dutifully treated in textbooks because of its importance in the half-adder, but it is never accorded much respect. Designing reversible systems engenders a certain affection for the XOR gate. Not only is it easily reversible, it is its own inverse. It is a non-expanding operation, able to take two inputs $A$ and $B$ and produce two outputs, $A \oplus B$ and $B$, which are sufficient to determine the value $A$. A dual-rail implementation of the XOR gate is fairly straightforward. Generation of $Z = A \oplus B$ and $\overline{Z} = \overline{A \oplus B}$ from $A$, $\overline{A}$, $B$, and $\overline{B}$ is shown in Figure 4-16 (a). Note the similarity to the gates which restore $A$ and $\overline{A}$ to $V_{dd}/2$, shown in Figure 4-16 (b). The $B$ and $\overline{B}$ signals are passed through two full inverters, not shown.

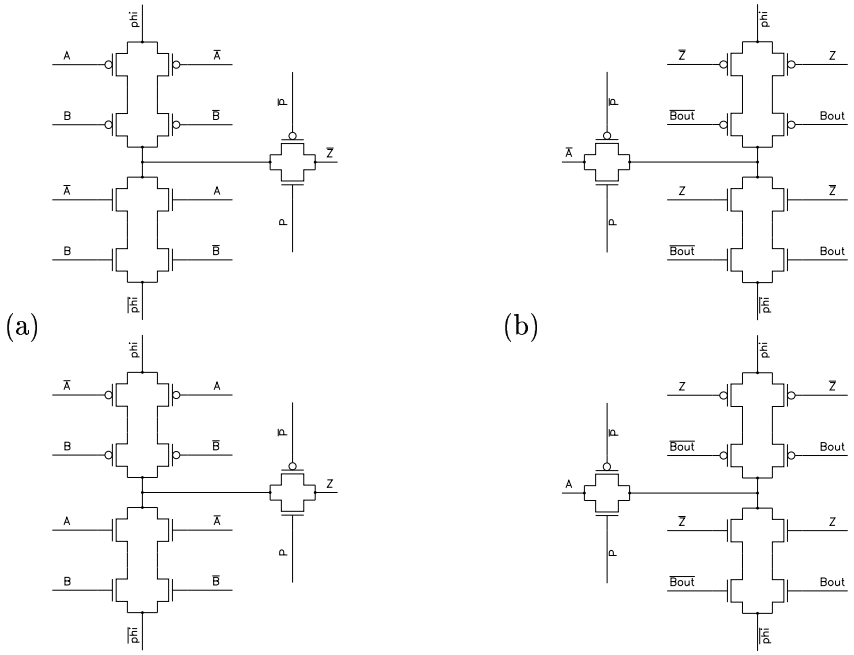


Figure 4-16: SCRL XOR gate (a) forward and (b) reverse parts.

An SCRL XOR gate is still not universal, of course, but it is still very useful. Schematic

[1] Interestingly, quantum computing XOR gates are both universal and reversible.

design and layout need only be done for one 10-transistor gate which is then replicated four times to create the forward and reverse path for four signals. The other two signals are sent through the ubiquitous full inverter. Three-input XOR gates are also relatively simple to construct.

The XOR operation, because it is easily invertible, is occasionally appended to the logic equation for some other function. For example, an expanding operation, such as logical AND, may impose undesirable constraints on a system in order to maintain reversibility. Rather than implement AND as an expanding, two input/three output function, a designer may prefer to implement a non-expanding, three input/three output "ANDX" gate which performs $Z^+ = (A \cdot B) \oplus Z$. The inputs are $A$, $B$, and $Z$; the $A$ and $B$ values pass through unchanged. The ANDX operation is its own inverse, since it is essentially an XOR operation between the two operands $A \cdot B$ and $Z$. Rather than retain the compact, but lossy, encoding $A \cdot B$, both values $A$ and $B$ are retained. Combining an expanding operation with XOR is a powerful technique for ensuring reversible operation while avoiding expanding logic. The drawback is the use of three input/three output gates and their associated complexity. The Pendulum processor ALU uses ANDX to simplify the constraints on the programmer to maintain reversibility. The details are covered in Chapter 8.

### 4.2.4 Other Logic

Using the techniques above, a designer is now free to concoct any combinational logic gate desired. The logic portion of a gate is made up of an appropriate pull-up network of pfets and the complementary nfet pull-down network. The t-gate portion of all non-tri-state gates is the same, and with appropriate fan-out for expanding operations, any logic function may be turned into a full SCRL gate, driving its outputs at the correct time and restoring its inputs to $V_{dd}/2$. The following section treats interconnection of these low-level modules.

The formula to create any reversible gate is to first create the static CMOS version of that gate, the pfet pull-up and nfet pull-down networks. The addition of a t-gate on the output creates a clocked gate.

### 4.2.5 Constants

Generation and reclamation of constants in a conventional machine is trivial: inputs may be wired to one of the supply rails to produce a constant value, and constant outputs may be ignored or wired to the appropriate supply. Because the information content is very low, constants are also relatively easy to deal with in a reversible system. In SCRL circuits, the prime concern is to maintain proper timing behavior. A constant SCRL value does not have a fixed voltage level. Rather, it always swings from $V_{dd}/2$ to one of the rails. The designer knows to which rail the output will swing but still must construct a suitable circuit so the signal will swing at the correct time.

Likewise, to reclaim a constant value, a circuit must be constructed to swing from the rail to $V_{dd}/2$ at the correct time. The SCRL circuits for a "one-source" and "one-sink" are shown in Figure 4-17. Note that the source and sink are SCRL half-gates without a logic portion. Rather than connect the t-gate to a logic portion, it connects directly to the appropriate swinging supply rail. A source and sink for a zero value simply connect the t-gate to the opposite polarity rail.

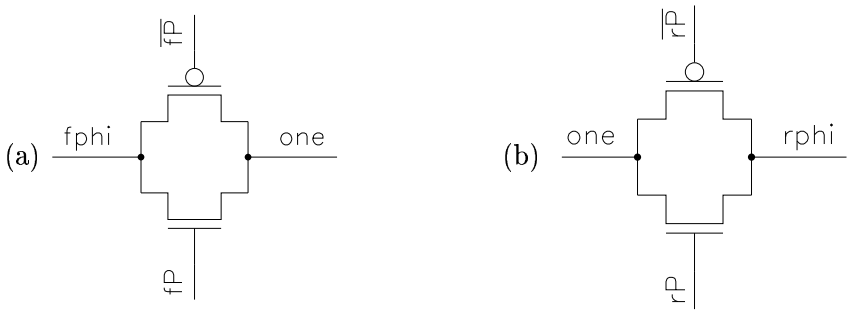


Figure 4-17: SCRL constant (a) source and (b) sink.

## 4.3 Circuit and System Representations

The examples above have shown that it is possible to abstract away the low-level constraints of SCRL logic from the system designer. The rules for symbol construction may now be

formalized. Modules internally connected to one set of power clocks are treated first, *i.e.*, blocks like the full inverter but not one full inverter driving another.

First, and perhaps obviously, a module's symbol must somehow indicate its logical behavior. Conventional shapes for NAND NOR, XOR, and so on, are perfectly appropriate. Larger blocks, such as half-adders, ALU's, and decoders, may have more complex shapes or markings. It is surprising how many designs use inadequately descriptive schematic symbols.

Second, a module must drive its output(s) according to SCRL timing constrains. This merely implies that the t-gate at the output of the logic portion be connected to a pair of swinging $P_i$ and $\overline{P_i}$ signals and that the logic portion be connected to $\phi_j$ and $\overline{\phi_j}$.

Third, a full module must restore the voltage on its input nodes to $V_{dd}/2$ according to SCRL timing constraints. A half module is exempt from this constraint. This rule is the key to hiding circuit-level reversibility from the logic designer. The implication of this rule is that a full module must have a reverse gate, connected to appropriate $P_k$ and $\phi_l$ signals, that can compute the module's input value(s) using only the module's output value(s). Note also that choosing a value of $i$ or $j$ for the forward gate sets the value for $k$ and $l$ for the reverse gate, and *vice versa*.

Fourth, a module must have a single power-bus connection. In all the SCRL gate figures above, each symbol has a single bus connection for all the swinging power clocks. This is made very powerful by the conventions of "lollipop notation," treated below.

Modules created following these four rules may be easily combined to form more complicated logic blocks. Additional rules for combining modules will be added in Section 4.3.2. The complex blocks may themselves be treated as primitive modules and further combined. The powerful tools of abstraction and hierarchy may thereby be brought to bear on reversible system design.

### 4.3.1 Lollipop Notation

As mentioned above, each simple module must have a single bus connection to the power rails. Early SCRL designs required super-human effort to keep track of supply connections.

A non-inverting stage requires twelve distinct power clocks, and keeping track of which clocks go where can quickly bring a designer to a grinding halt.

Just as circuit complexity can be hidden using hierarchy and a symbolic representation, so too the power rail complexity can be hidden using a "lollipop" symbol to represent both polarities of both the forward and reverse $\phi$ and $P$ signals for a full SCRL logic phase.

Figure 4-18 shows the three-phase SCRL rail connections to five stages of an SCRL pipeline in explicit detail. Each square represents a half-gate. The logic portion of the half-gate is powered by the $\phi$ rails indicated in the lower left corner of the square. The t-gate portion of the half-gate is controlled by the $P$ clocks indicated at the lower right. Each $\phi$ and $P$ represents the positive and negative polarities of that rail phase.

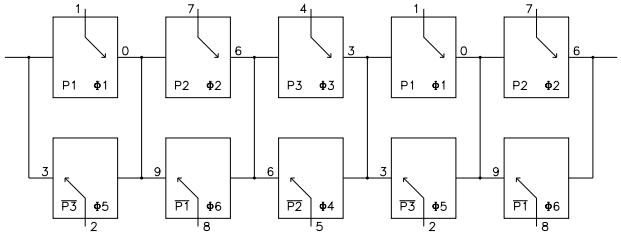


Figure 4-18: SCRL pipeline showing rail connections. This figure is based on work by Younis [You94].

The numbers on the right sides of the upper squares and left sides of the lower squares correspond to the times at which the $P$ rails transition. The numbers on the top of the upper squares and bottom of the lower squares correspond to transitions of the $\phi$ rails. The timing diagram is shown in Figure 4-19. A complete period of transitions for all nine distinct clock pairs takes eighteen clock ticks.

If non-inverting stages are required, the situation becomes even more complex. Each non-inverting stage is powered by its own clock pair, so if all three stages may be non-inverting, fifteen clock pairs swing with a thirty tick period. This timing diagram is shown in Figure 4-20. See Section 4.5 below for more details.

Figure 4-20 may also be described in table form, shown below. It may seem that I am belaboring the point, describing the SCRL power clock timing in multiple formats, but

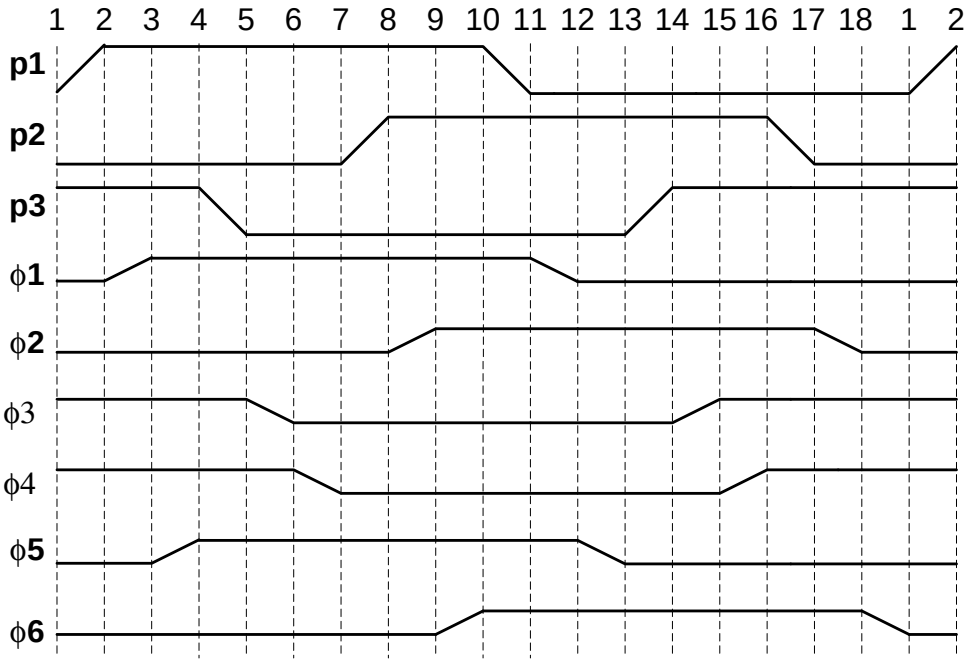


Figure 4-19: SCRL power rail timing. This figure is based on work by Younis [You94].

understanding the timing behavior of these rails in all their complexity is important to the design of the abstraction used below to hide this complexity.

To simplify rail interconnection, first notice that the rails may be grouped into three sets. Two- or four-phase SCRL rails may be grouped into two or four sets, respectively. Each set is composed of the dual-rail pair of forward $P_i$ rails, the dual-rail pair of forward $\phi_j$ rails, and the corresponding reverse $P_k$ and $\phi_l$ rails, for a total of four dual-rail pairs of rails in a set. Each $P_i$ pair of signals is used as forward controls in one set and inverted and used as reverse controls in one other set. For example, the forward t-gate in the first pipeline stage in Figure 4-18 is on when $P_1$ is high, while the reverse t-gate in the second pipeline stage is on when $P_1$ is low. The t-gate control is denoted as $\overline{P_1}$ in this case.

The $\phi$ rails always appear in groups of two pairs, one forward and one reverse. For example, in three phase SCRL, $\phi_1$ and $\phi_5$ share a set, $\phi_2$ and $\phi_6$ share a set, and $\phi_3$ and $\phi_4$ share a set.

The three sets are therefore:

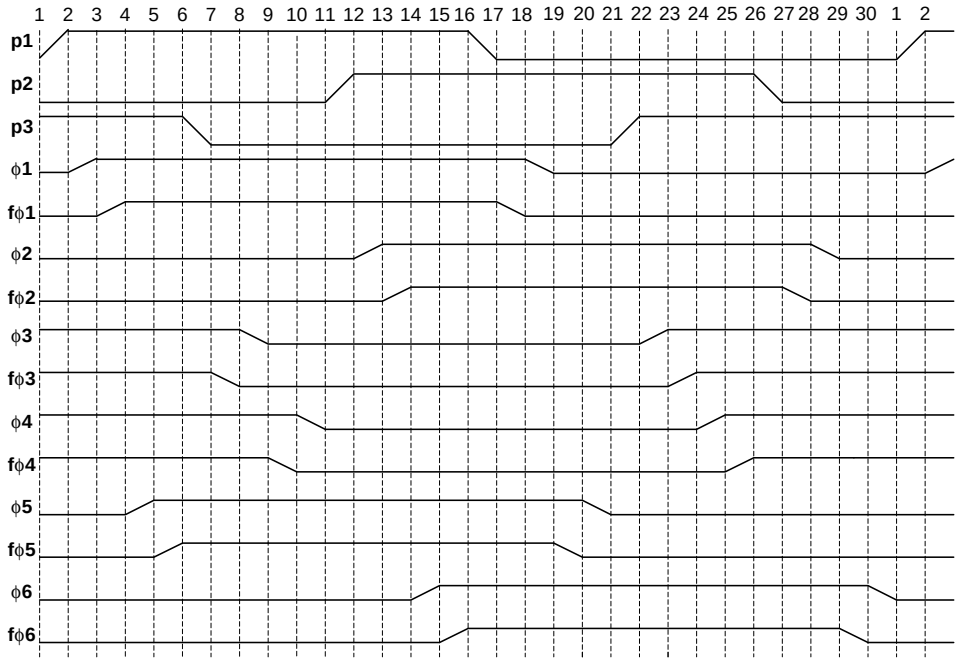


Figure 4-20: SCRL power rail timing, including "fast" rails. This figure is based on work by Younis [You94].

---

- $\phi_1$, $P_1$, $\phi_5$, $\overline{P_3}$
- $\phi_2$, $P_2$, $\phi_6$, $\overline{P_1}$
- $\phi_3$, $P_3$, $\phi_4$, $\overline{P_2}$

If each symbol expects connections to the power bus to be in a pre-established order, say forward-$\phi$, forward-$P$, reverse-$\phi$, reverse-$P$, a unit containing the appropriate rails for one of the three sets, in the correct order, can be attached to the bus port, providing a simple symbolic power connection to the symbol. The rail unit used is shown in Figure 4-21, and because of its shape it has become known as a lollipop.

Connecting power rails to a schematic symbol is then as simple as connecting the appropriate phase's lollipop symbol to the power port of the gate. Three full SCRL inverters connected in a ring oscillator topology are shown in Figure 4-22. The phase one, two, and three lollipop symbols are connected to each full inverter's power port. Further issues involved in connecting multiple SCRL gates together are discussed in the next section.

| Rail | Tick | | Tick | |
|---|---|---|---|---|
| $P1$ | 1 | $\uparrow$ | 16 | $\downarrow$ |
| $\phi 1$ | 2 | $\uparrow$ | 18 | $\downarrow$ |
| $f\phi 1$ | 3 | $\uparrow$ | 17 | $\downarrow$ |
| $\phi 5$ | 4 | $\uparrow$ | 20 | $\downarrow$ |
| $f\phi 5$ | 5 | $\uparrow$ | 19 | $\downarrow$ |
| $P3$ | 6 | $\downarrow$ | 21 | $\uparrow$ |
| $f\phi 3$ | 7 | $\downarrow$ | 23 | $\uparrow$ |
| $\phi 3$ | 8 | $\downarrow$ | 22 | $\uparrow$ |
| $f\phi 4$ | 9 | $\downarrow$ | 25 | $\uparrow$ |
| $\phi 4$ | 10 | $\downarrow$ | 24 | $\uparrow$ |
| $P2$ | 11 | $\uparrow$ | 26 | $\downarrow$ |
| $\phi 2$ | 12 | $\uparrow$ | 28 | $\downarrow$ |
| $f\phi 2$ | 13 | $\uparrow$ | 27 | $\downarrow$ |
| $\phi 6$ | 14 | $\uparrow$ | 30 | $\downarrow$ |
| $f\phi 6$ | 15 | $\uparrow$ | 29 | $\downarrow$ |

Table 4.2: SCRL rail timing transition table. A $\uparrow$ represents splitting $\phi$ rails or $P$ and $\overline{P}$ rails going high and low, respectively. A $\downarrow$ represents $\phi$ rails converging to $V_{dd}/2$ or $P$ and $\overline{P}$ rails going low and high, respectively. The numbers correspond to the points in Figure 4-20. An $f\phi$ labels a "fast" rail.

### 4.3.2 Combining Gates

Connecting multiple SCRL gates in sequence is as simple as counting 1-2-3 when they are powered by lollipop symbols. Every gate powered by phase $i$ must be driven by the output of a gate powered by phase $i-1$ and must drive only gates powered by phase $i+1$. A simple example of the output of a full NAND gate driving another full NAND gate is shown in Figure 4-23. Note that inputs and outputs are not allowed to float; every wire must be

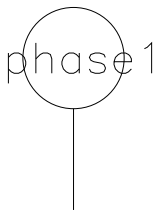


Figure 4-21: The phase 1 lollipop rail symbol. This symbol contains the $\phi_1$, $\overline{\phi_1}$, $P_1$, $\overline{P_1}$, $\phi_5$, $\overline{\phi_5}$, $\overline{P_3}$, and $P_3$ rails.

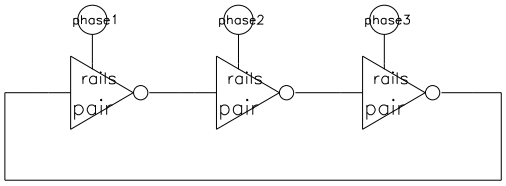


Figure 4-22: An SCRL ring oscillator.

connected to the input and output of a gate. The phase one powered gates drive phase two powered gates, and every level of logic is a pipeline stage, requiring delay stages for signals such as C, Aout, and Bout, to maintain proper timing relationships.

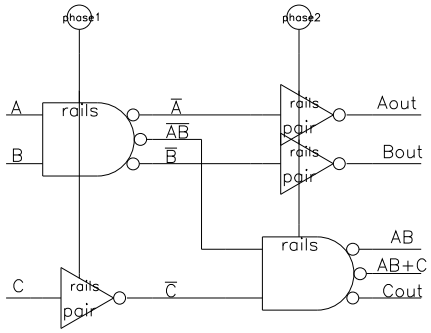


Figure 4-23: An SCRL full NAND gate driving a second full NAND gate.

The rail timing sequence is circular, so a phase three gate drives a phase one gate. One implication of this is that the shortest feedback paths possible requires a minimum of three stages of logic. The ring oscillator mentioned above is a case in point.

A signal being driven from phase one may be some integer multiple of three too early or late relative to other signals driven from phase one. To keep track of this, it is useful to assign letters to each logic stage in a large, complex logic block. An N-bit ripple-adder, for example, has at least N stages of logic, and while stages one, four, seven, and so on, are all driven by the same phase rails, unless all signals pass through the same number of logic stages, they will not arrive at the output edge of the block at the same time. Figure 4-27 shows this error. To correct for potential offsets of this nature, delay elements are frequently required. Imagine a pipelined processor with on the order of 100 pipeline stages. Other

than explicit data forwarding, each stage's output is an input to the stage immediately following. If one output from stage 63 is not needed until stage 87, delay elements must be inserted in stages 64 through 86. An SCRL circuit must obey this constraint as well as the 1-2-3 counting constraint imposed by the rail timing.

As with Ressler's Fredkin gate computer [Res81], the designer must consider the schematic to be cut into a set of "time slices." Signals must pass through a logic gate to move from one time slice to the next. A NAND gate is a single slice; an ALU may have twenty or thirty slices.

Each clock phase may be considered to be a fraction of a full clock cycle, and the schematic for a full clock cycle delay element is shown in Figure 4-24. The rail connections are shown explicitly for each of the six half-inverters in the cycle delay element. This module was developed in the early stages of this work and therefore does not take advantage of later hierarchical developments. Lollipop notation, however, is not appropriate in this case as the rails are meant to be connected at a higher level of the hierarchy.

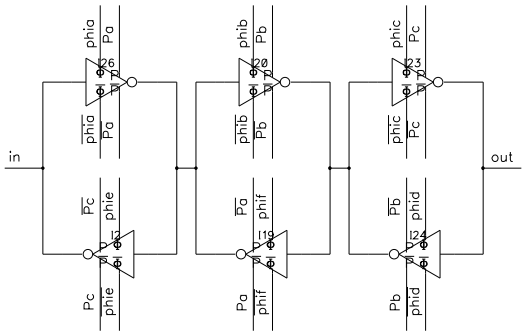


Figure 4-24: A full clock cycle delay element.

Fan-out of a signal is a simple matter; a single gate can drive any number of gates as long as they are powered by the appropriate phase rails. In conventional terms, fan-out is a copy operation: the output of one gate drives more than one input of the gates downstream. A gate takes some inputs, computes an output value, and fans out multiple copies of that value to the inputs of other gates.

In a reversible computer, an expanding operation may be thought of as a fan-out that is

more complex than a mere copy. The expanding reversible gate takes some inputs, computes multiple outputs, and may fan out multiple copies of these multiple outputs to the inputs of other gates. The NAND gate above performs a fan-out of $A$ and $B$ into something other than copies of $A$ and $B$.

Section 4.2.1 showed how multiple-input gate fan-in is constructed. More complex, from a logic point of view, is the fan-in combination of two or more signals. The SCRL NAND gate creates a redundant encoding of the information in the two input signals; the un-NAND gate eliminates the additional signal. Recall that the NAND gate produces $\overline{A}$, $\overline{B}$, and $\overline{A \cdot B}$. At some point, the $\overline{A \cdot B}$ signal may be unnecessary. The symmetry imposed by most reversible constructions requires that for each fan-out, copy or otherwise, there be a fan-in.

Eliminating the $\overline{A \cdot B}$ signal is simple. A feedback gate with inputs $A$ and $B$ and appropriate power clock connections will restore a node to $V_{dd}/2$. This is simply the half-NAND gate from Figure 4-13 turned backwards and connected to the appropriate rails. Figure 4-25 shows a half-NAND gate and two full inverters driving three full inverters on the left half, and three full inverters driving a backwards half-NAND gate and two full inverters on the right half. Each vertical column is driven by a single lollipop signal, ensuring proper rail timing. Note that the half-NAND and half-un-NAND gates are only driven by the forward or reverse set of rails, respectively. The "forward rails" and "reverse rails" symbols merely pass through the appropriate $\phi$ and $P$ signals.

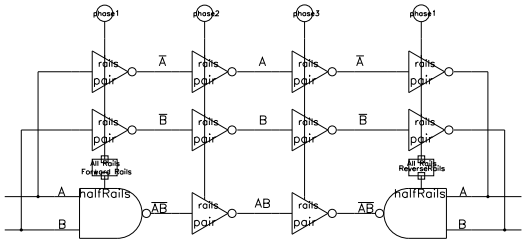


Figure 4-25: An SCRL NAND and un-NAND logic block.

The signal fan-out of the expanding NAND operation is undone by the fan-in of the un-NAND gate. Computation is performed, creating one intermediate signal, and once the intermediate value is no longer needed, the computation is undone. Reversible systems, from

the circuit level to the software level, exhibit this characteristic. This particular example performs a trivial computation using the $\overline{A \cdot B}$ signal, namely delay, but the inverters in the middle of Figure 4-25, powered by the phase 2 and 3 lollipops, could be replaced by more interesting computing elements, and more than just two logic stages.

### Dual-rail Signaling and Signal Minimization

The topology shown in Figure 4-26, while common in conventional systems, is illegal in an SCRL system. The inverter powered by phase one rails can not feed its output forward to drive the inverter powered by phase three rails. This serves to illustrate the desirability of both using dual-rail signaling conventions and of reducing the number of signals being fed through a block. Dual-rail signaling reduces the need for inverters to generate the correct logic polarity and reduces reliance on non-inverting SCRL gates with their associated additional power rail complexity. Reducing the number of signals passing through a block reduces the number of gates that are used only to re-time a signal.

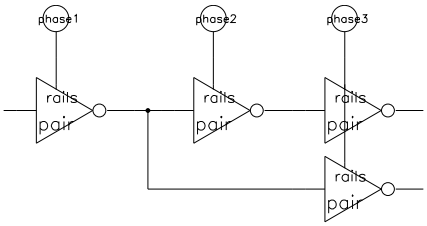


Figure 4-26: Incorrect SCRL fanout.

While the circuit in Figure 4-27 may seem a simple feedthrough, remember that each logic stage contains a built-in pipeline register. Signals driving in on the left edge at the same time will not arrive at the NAND gate on the right edge at the same time. Additional retiming gates, generally inverters, are required so that every signal is equally delayed. Just as the Cray-1 used 20 or 30% its gates for delaying and terminating signal lines, so too SCRL circuits use a substantial number of gates to properly time signals.

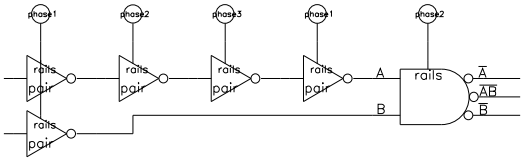


Figure 4-27: An incorrect SCRL feedthrough schematic. The B input to the NAND gate should have been delayed using a full cycle delay element.

### 4.3.3 Schematic Hierarchy

The reader now has at his or her disposal techniques for composing arbitrary logic gates, attaching them to one another, including feedback, fan-out, and logic fan-in, and connecting them to the appropriate power clocks. The next powerful tool for dealing with complexity is hierarchy.

Connecting lollipops to the gates at each logic stage makes module re-use difficult. Conversely, passing supply pins up through levels of the hierarchy so that the lollipops may be connected at a higher level increases module complexity. Proper system hierarchy design requires a delicate balance between these two components. Note that fixing supply rail connection hierarchy also fixes signal hierarchy. If a module has four lollipop connections, for example, the signals coming in and going out of the module must fit in with the rest of the system. The 1-2-3 rail phase ordering must be respected for lollipops and signals.

Most of the modules in the Pendulum processor occupy a single time slice. Rail connections in this case are made simply through a single lollipop. An example of the Pendulum hierarchical structure is shown in Figure 4-28. The lollipops are generally connected at the "small modules" or "large modules" level. The "large modules" must be carefully designed to fit in with each other. Delay elements may need to be added to ensure proper 1-2-3 rail ordering.

A technique that was considered but not used was to pass the rails up through the hierarchy using a generic A-B-C counting. These nodes could be connected to the proper 1-2-3 lollipops at the highest level. This was not done for two reasons. The first is that is was largely unnecessary. The only two levels of hierarchy above the lollipop connections were

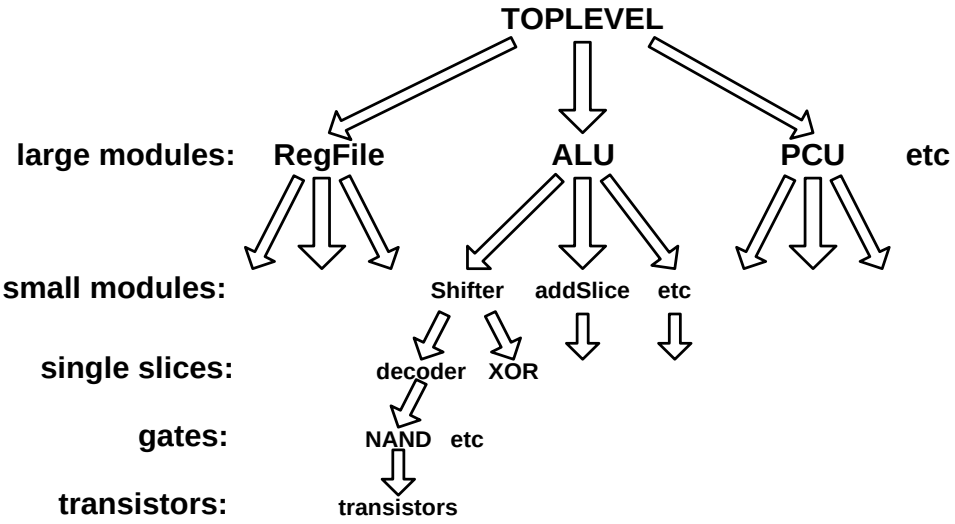


Figure 4-28: A portion of the Pendulum processor hierarchy.

simple enough to understand that no extra effort was required. The second reason is that the details of the lollipop timing are a detriment to understanding the toplevel data flow. By putting the lollipop connections at a lower level, the datapath block diagram is more easily understood. Only when the rail phases are relevant should they be explicitly shown. It is, however, useful to know how many stages of logic are in a module and what phase the modules inputs expect and outputs are driven on. This information may be added as a note on the module's symbol.

The Pendulum processor has seventy-four stages of logic. The longest of the large modules were the ALU and PCU, at thirty-two stages each. This level of complexity is by no means beyond a competent designer.

## 4.4 Multiplexing and Shared Busses

Signal multiplexing is a very common tool in conventional circuits. This "many to one" mapping is not allowed in a reversible computer because the single output contains no information about the inputs that are not selected. A shared bus is also a type of multiplexing; a number of signals connected to one set of wires through tri-state buffers may drive the

bus one at a time. This too is a many to one mapping, and in fact multiplexers are often implemented using tri-state gates. Shared busses are powerful and useful constructs for the digital designer; a reversible version would be equally useful. The problem is keeping the features of the many to one mapping while retaining all information.

### 4.4.1 The SCRL Tri-state Inverter

The first component needed is an SCRL tri-state gate. This enhancement to SCRL was not envisioned by Younis and Knight [YK93, YK94, You94], and it is a contribution of this research. A properly designed standard SCRL gate will always drive the output node to a zero or one value when the supply rails $\phi$ and $\overline{\phi}$ split to $V_{dd}$ and ground. This is because the pass gate at the output is always turned on before the rails split. By controlling whether or not the pass gate turns on, the logic portion may be isolated from the output node. The pass gate portion of an SCRL gate must be modified for tri-state operation.

To construct a tri-state inverter, we begin with the standard SCRL half-inverter from Figure 4-9. The desired behavior is shown in Table 4.3. The output value "hiZ" represents a high impedance state in which the gate is isolated from the output node. The select, or enable, signal Sel is passed through some other part of the system. It must have the same SCRL timing as the input A.

| in | | out |
|---|---|---|
| *A* | *Sel* | *Zout* |
| 0 | 0 | hiZ |
| 0 | 1 | 1 |
| 1 | 0 | hiZ |
| 1 | 1 | 0 |

Table 4.3: The SCRL tri-state inverter truth table.

To make a tri-state full inverter, the input node A must be conditionally restored to $V_{dd}/2$. Unlike the standard SCRL full inverter, the tri-state full inverter must only restore its input if it drove its output. If the output node is a shared bus, some other gate will have driven a value on it. That value need bear no relation to the input node of any gates that do

not drive the output bus. This is the whole point of a shared bus. Therefore, the tri-state full inverter can not use the shared bus value to restore its input. It must therefore have a tri-state reverse path. This implies that the reverse path must have its own select signal. When not selected, a reversible tri-state not only does not drive its output, but it also "ignores" its input.

Both polarities of the select signals are generally required. The symbol, therefore, of an SCRL tri-state gate is shown in Figure 4-29. Note the separate forward and reverse select signals, and the single lollipop-ready rail connection pin.

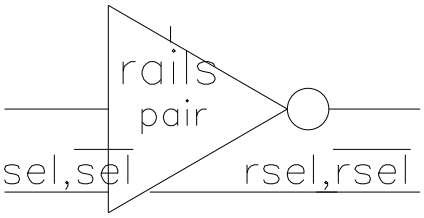


Figure 4-29: The SCRL tri-state inverter symbol.

The modification to the pass gate portion of the SCRL inverter is straightforward. The select signal controls a pair of t-gates, inserted to control whether the rail signals $P$ and $\overline{P}$ can connect to the t-gate between the internal node and the output node. The logic portion remains unchanged; tri-state gates other than the inverter may be constructed by adding the tri-state t-gate portion to a standard SCRL logic portion. In a proper SCRL tri-state full inverter, the select signals are passed through the gate with a pair of standard full inverters. The entire SCRL tri-state full inverter schematic is shown in Figure 4-30.

In the Pendulum processor, many of the tri-state gates used were modified from a library of conventional tri-state gates. A standard CMOS tri-state gate is shown in Figure 4-31 (a). The transistors controlled by the select signals are connected to the supply rails. An SCRL tri-state logic portion may use this topology along with the tri-state t-gate portion without changing the logical behavior of the gate. A power advantage may be realized because the total capacitance attached to the swinging rails is reduced since the internal nodes of the logic portion are disconnected from the rails when the select signal is not asserted. However,

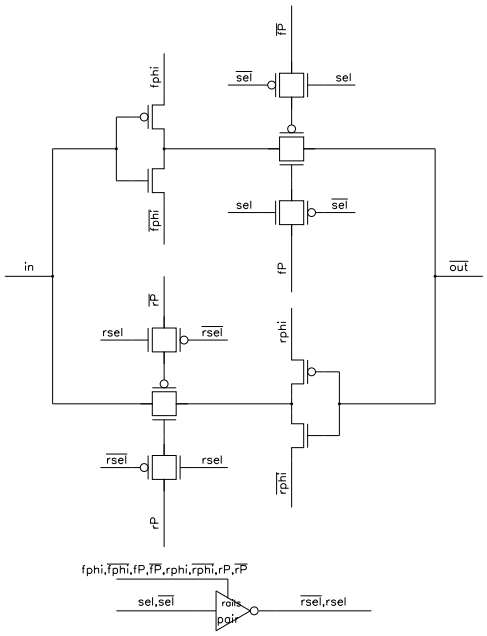


Figure 4-30: The SCRL tri-state full inverter schematic.

the additional loading on the select signals may offset this gain. The actual tri-state inverter schematic used in the Pendulum processor is shown in Figure 4-31 (b). The select signals must be passed through an external gate.

### 4.4.2 Using Reversible Tri-state Gates

To use the tri-state full inverter, or any other reversible tri-state gate in a logic block, the designer must be careful to make sure bus contention is avoided and that one of the gates restores each of the input nodes to $V_{dd}/2$. Conventional design requires that bus contention be avoided, but the constraint that the input nodes be restored is, of course, unique to reversible design. This constraint leads to the general rules for using tri-state gates in a reversible system. The first is that no information may be lost. This outlaws the use of conventional-style many-to-one multiplexer mappings. Rather, tri-state gates are used in a reversible system to "steer" information down one of many possible paths. The other paths are entirely shut down, neither driving their outputs or restoring their inputs. A very simple

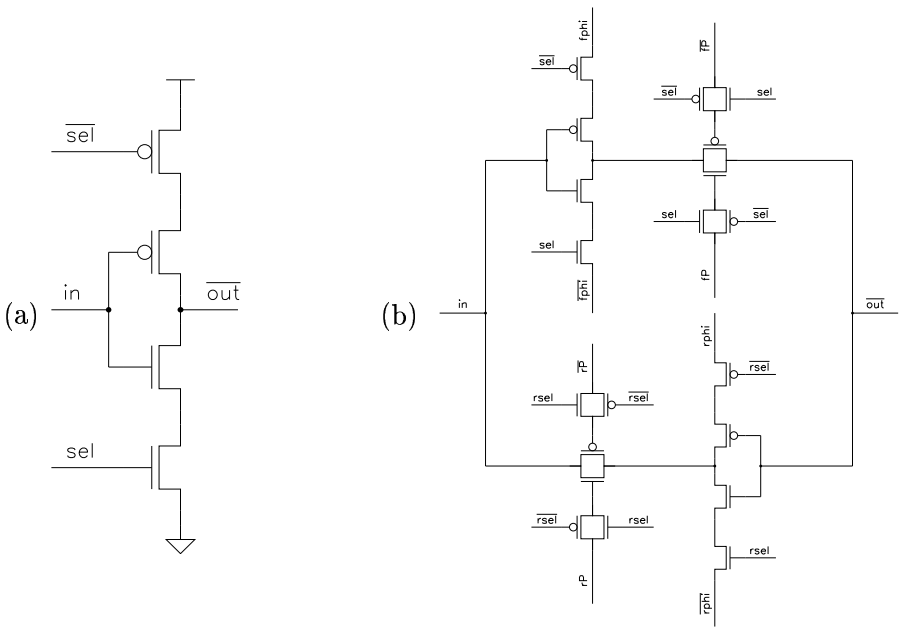


Figure 4-31: (a) A conventional CMOS tri-state inverter schematic and (b) the SCRL tri-state inverter used in the Pendulum processor.

case is shown in Figure 4-32. The output node has the same logical state for either value of the select signal, but the output is driven by the top inverter if the select signal is asserted and by the bottom inverter otherwise. Note that since each tri-state full inverter contains a path for the select signal, an optimization is possible by removing the select signal path from the tri-state gate itself.

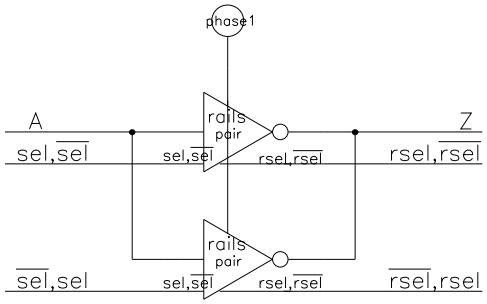


Figure 4-32: Simple SCRL tri-state application.

A somewhat more interesting case is shown in Figure 4-33 (a). The output node $Z$ is driven

to be either $\overline{A}$ or $\overline{B}$. If the select signal is asserted, node $A$ is inverted and drives the output with $\overline{A}$. Node $A$ is also restored to $V_{dd}/2$ by the upper inverter. The lower inverter in this case, however, neither drives the output nor restores node $B$. To maintain reversibility, node B must not be driven by the previous stage. If it were, charge would be "stranded" on node $B$, and energy would be dissipated at node $B$'s next transition. Therefore, the logic from Figure 4-33 (a) must be used as in Figure 4-33 (b). Again we see the symmetry of reversible logic. The fan out on the left side of Figure 4-33 (b) has a matching fan-in on the right side. Of course, Figure 4-33 (b) is a trivial example, but if the computation performed on the upper and lower paths differ, the select signals select what function is executed.

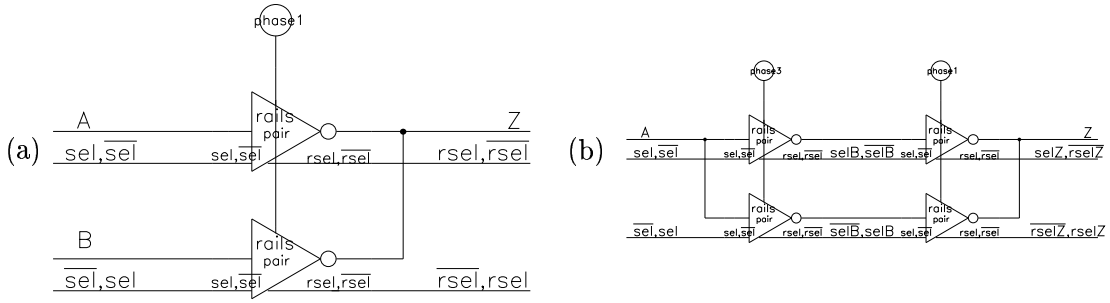


Figure 4-33: Multiple signals A and B driving a shared bus shown in (a), and the pairing of fan-out and fan-in is shown in (b).

This "information steering" is a powerful tool for building reversible systems, and is the reversible equivalent of a multiplexer. Reversibility requires that the multiplexer have a de-multiplexer upstream from it. Using tri-state gates allows the designer to chose a computation path for a signal and disable all other potential paths for that signal. Reversible memory uses this technique; the address lines select either the recirculating storage path or the output node for each bit in the memory. Only one bit may drive the output node at at time; all others recirculate. See Section 5.2.4 for more details.

The Pendulum datapath also uses this technique. Each operand is passed along a tri-state path, only selecting the logic that is being used to compute the function for that particular opcode. For a one operand instruction, only one of the three register file output paths is active. The ALU also has multiple paths, only one of which is active at at time. For example, during the `add` operation, all other paths, such as for computing ANDX, XOR, and

so on, are disabled. The disabled paths do not drive their outputs or restore their inputs. All input and output nodes in a disabled path remain at $V_{dd}/2$.

### 4.4.3 A Tri-state Optimization

If a signal passes through a long tri-state delay chain, an optimization may be possible. Rather than pass the data and select signals through two long chains of delay elements, the circuit may rely on charge storage of the data values at the beginning of the chain and delay only the select signal. The data passes through a tri-state gate at the beginning and end of the chain. The before and after topologies are shown in Figure 4-34. This design was considered for the Pendulum processor but considered too risky for implementation. Further investigation of this technique's usefulness is required.

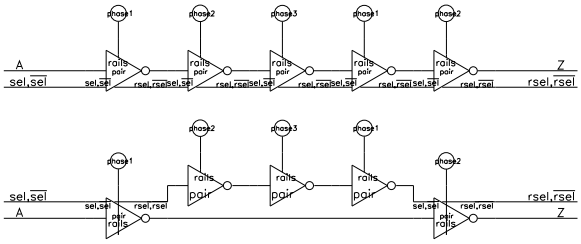


Figure 4-34: A potential optimization for long tri-state delay chains. The top chain of tri-state inverters may be optimized by eliminating the tri-state portion and keeping only the delay chain for the select signals.

## 4.5 Non-inverting Gates

All of the logic gates discussed to this point have used inverting logic. Both polarities of a signal may be required to compute certain functions, necessitating either that all signals be run dual-rail or that some gates be non-inverting. This section covers the techniques for creating non-inverting SCRL logic.

Non-inverting SCRL gates impose a penalty on the logic designer: substantially more power clock rails are required. In three-phase SCRL, two additional pairs of rails are required for

each logic phase that is to be able to perform non-inverting operations. Three-phase SCRL requires eighteen rails in inverting form; thirty rails are required if all three phases are to be able not to invert. These additional rails are generally called "fast" rails because they split and collapse entirely within a standard splitting rail pair. The timing of a rail and its fast complement is shown in Figure 4-35. The fast and slow rail are a retractile pair, a version in miniature of Hall's retractile cascade [Hal92]. The full timing of all thirty rails for three-phase SCRL is shown in Figure 4-20.

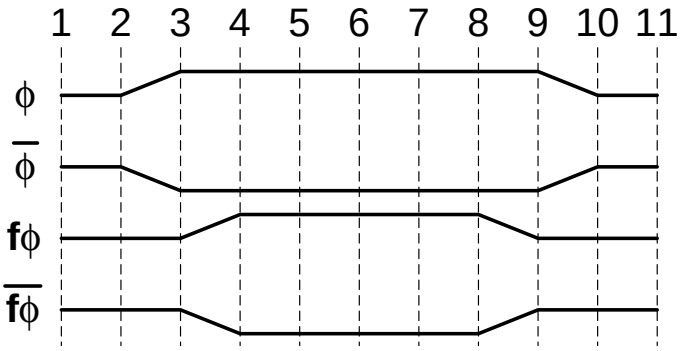


Figure 4-35: One slow and fast rail pair timing. The fast rail is a retractile version of the slow rail.

The fast rails power an additional logic portion between the original logic portion and t-gate portion of an inverting SCRL gate. An SCRL half-buffer is shown in Figure 4-36. After the input value is stable, the t-gate turns on and the "slow" rails split, just as in the standard SCRL inverter. The input to the internal "fast" inverter is then stable and the fast rails are split, driving the output node to a valid logic level.

The slow and the fast logic portions may be replaced with any static CMOS gate. To make a half-AND gate, for example, a NAND gate may be followed by a fast inverter stage, as in Figure 4-37. Or, to make a more complex detector gate, such as the Pendulum exchange opcode detector shown in Figure 4-38, fast inverters on some or all of the inputs drive a logic portion powered by the fast rails. The SCRL pipeline design ensures that all the inputs to the fast logic portion are stable before the fast rails split, whether or not they are driven by the slow logic portion.

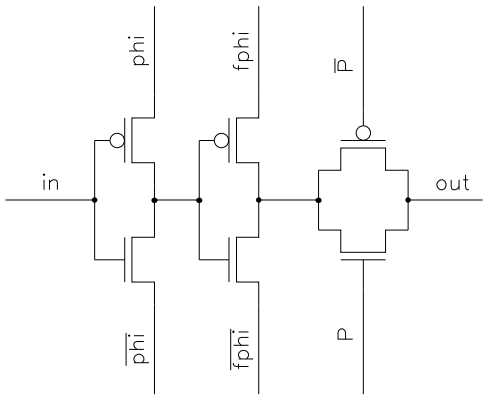


Figure 4-36: SCRL half-buffer schematic. The $fphi$ and $\overline{fphi}$ rails are as shown in Figure 4-35.

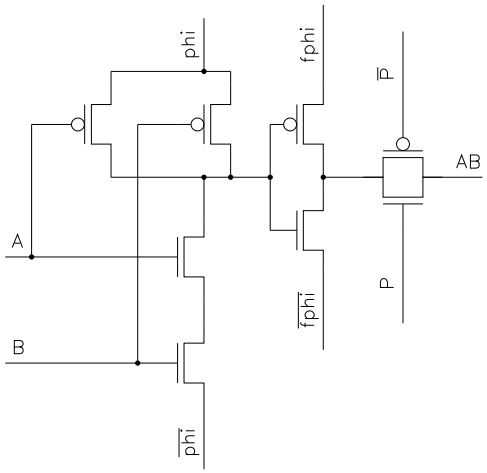


Figure 4-37: An SCRL half-and gate. The inverting NAND portion is followed by an inverter powered by fast rails.

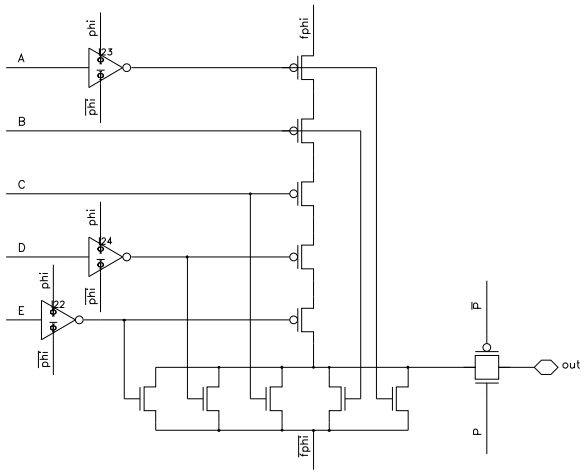


Figure 4-38: A pattern detector from the Pendulum processor using inverters on the input signals. The inverters are simple two-transistor devices, not containing a t-gate. The inverters are powered by the slow rails and the five-input OR gate is powered by the fast rails.

---

Non-inverting tri-state gates may be constructed by adding the tri-state t-gate portion to the two logic portions of non-inverting logic. Clearly, SCRL gate construction is a modular process: a variety of inverting or non-inverting logic portions may be connected to a standard or tri-state t-gate portion, creating a wide variety of logic gates.

### 4.5.1 Generating and Killing Dual-Rail Signals

Non-inverting logic stages may be combined with inverting logic to create a gate that generates both polarities of an input signal. A dual-rail generating gate is shown in Figure 4-39. The gate contains two forward portions, one inverting and one non-inverting, and one inverting reverse portion. The $\overline{Aout}$ signal is used to restore the input, $A$, to $V_{dd}/2$.

An optimization is possible, but was not used in the Pendulum processor, to reduce the transistor count in the forward portions. Note that the input to the forward fast portion is identical to the output node of the inverting logic portion. The inverter driving the fast inverter may be eliminated, and the inverting logic portion may drive both the t-gate connected to $\overline{Aout}$ and the fast inverter, as in Figure 4-40.

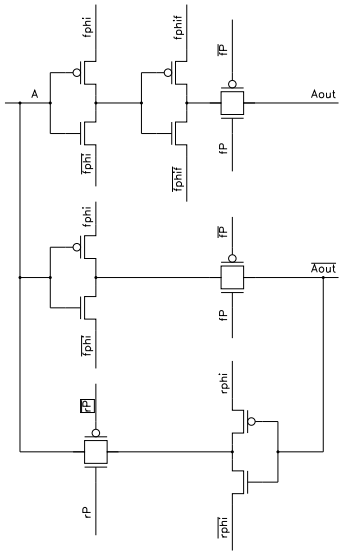


Figure 4-39: An SCRL full dual-rail generation gate. Note that in this figure and other full gates using fast rails, a leading f in a rail name indicates "forward," while a trailing f indicates "fast." For the reverse rails, likewise a leading r indicates "reverse," so $f\phi f$ is a forward fast rail, and $r\phi f$ is a reverse fast rail.

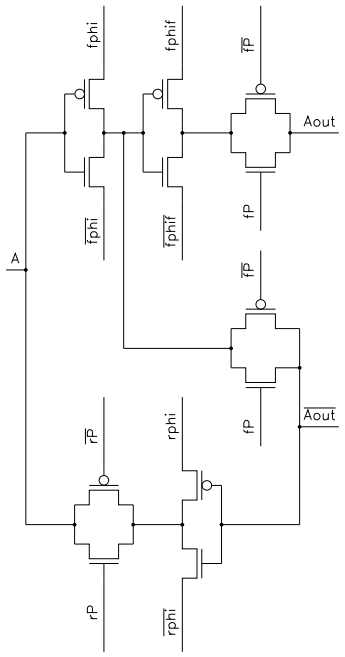


Figure 4-40: An SCRL full dual-rail generation gate, with one inverter unit eliminated.

Likewise, a "dual-rail-undo" gate may be constructed from a single inverting forward half-inverter and two reverse gates, an inverter and a buffer. This two input, one output gate is shown in Figure 4-41. Note that the reverse portion of this gate has the optimization mentioned above, eliminating the additional slow inverter stage. The dual-rail circuits here are simply a full inverter with an added half-inverter stage, either forward or reverse, connected between the logic portion and t-gate portion of one of the half-inverters in the full inverter. Care must of course be exercised when using this fan-in gate to ensure that the two inputs signals really are always complementary.

Both the dual-rail generation and undo gates may be used as tri-state gates with the addition of a tri-state t-gate portion on the forward output(s) and the reverse portion(s).

### 4.5.2 Connecting Non-Inverting Gates

During testing of the Pendulum chip, an interconnection error was found, shown in Figure 4-42. The error arises from the rail timing difference between a signals driven by fast and

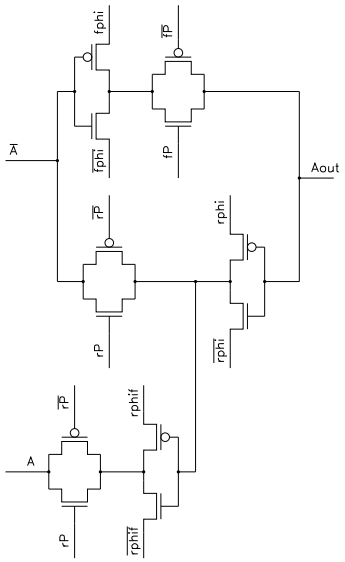


Figure 4-41: An SCRL full dual-rail "undo" gate. Note the use of a leading f or r to indicate "forward" or "reverse," and a trailing f to indicate a fast rail.

slow rails.

The output of a dual-rail-undo block and a buffer are connected together, performing fan-in of two identical signals. The dual-rail signals $wBrB$ and $\overline{wBrB}$ were generated from the $ALUopD$<3> signal many logic stages ago. The signals $ALUopD$<3> and $wBrB$ are identical, and could have been merged at the outputs of the previous stage, potentially avoiding this error. As it is, the output node $InstrE$<18> is driven by an inverting logic stage and a "hidden" non-inverting stage. To reduce the complexity of the dual-rail-undo gate, the output is driven by an inverter connected to the negative input, as shown above in Figure 4-41. The buffer driven by $ALUopD$<3> has a non-inverting output stage, as shown above in Figure 4-36. If the node $InstrE$<18> were driven either by two fast stages or two slow stages, no error would occur.

Observe the sequence of events as the output node is driven. First, the input values are driven to stable values, turning on one of the transistors in each of the two slow inverter stages. Then the t-gate portions are turned on, connecting the output node to the internal node of the inverting gate and through one of the slow inverter's transistors to a slow rail.

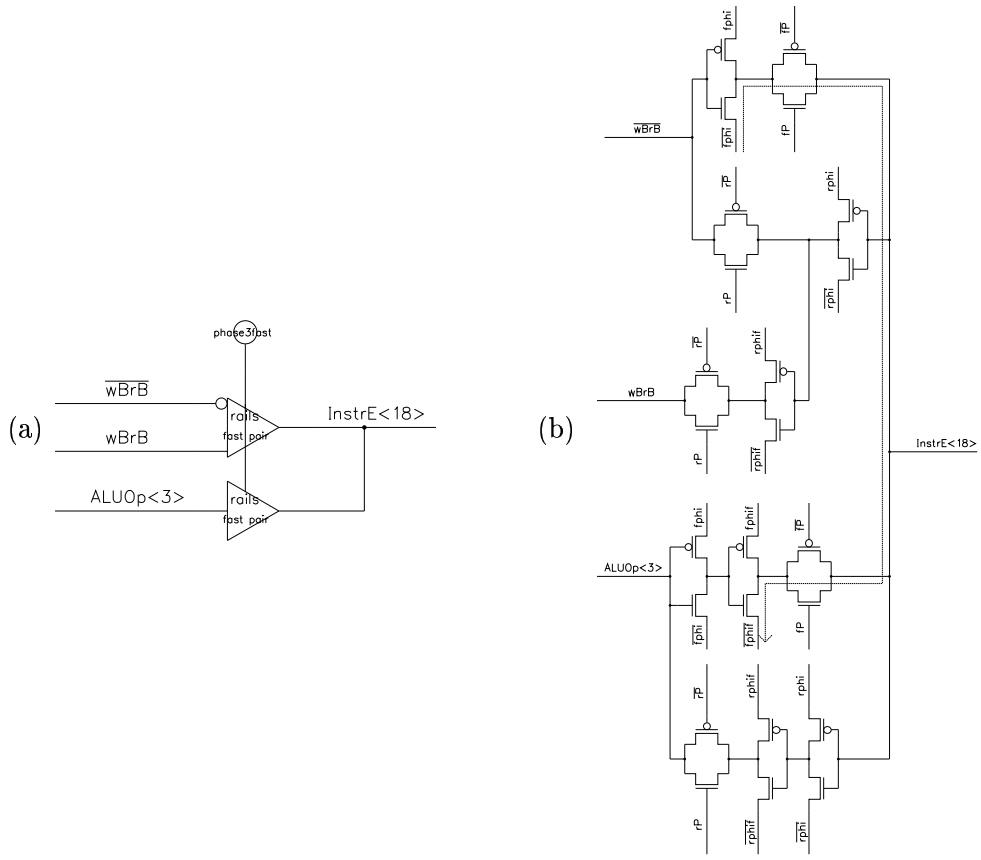


Figure 4-42: An error due to improper output connection of a dual-rail undo gate and a buffer. Note that even though the two output values are logically identical, the timing problem of connecting an output driven by a fast rail and one driven by a slow rail creates a temporary short circuit. The gate-level schematic, which obeys all the SCRL connection rules listed above, is shown in (a), and the underlying transistor-level schematic is in (b). The dotted line indicates the short circuit path.

As the slow rails split, the output node and the input to the fast inverter are driven to a valid value. One of the transistors in the fast inverter is turned on, connecting the output node to one of the fast rails. Observe that the output is at a valid logic level and the fast rails are still at $V_{dd}/2$. A short circuit is momentarily created between one of the slow rails and one of the fast rails at $V_{dd}/2$. The path of the short circuit is from a slow rail, through one of the transistors of the inverting gate, the inverting gate's t-gate, the non-inverting gate's t-gate, one of the transistors of the non-inverting gate, to one of the slow rails. As soon as the slow rails split, the output node is driven to the same rail and the short circuit is between two nodes at the same potential. Depending on how quickly the fast rail splits after the slow rail, enough current may flow to damage the circuit.

This error is disturbing not only because it may cause a circuit malfunction, but also because it violates one of the SCRL constraints, namely that charge flow in a controlled manner through the circuit. The short between two supply rails at different potentials through a resistive element is an uncontrolled energy dissipation. The implication for the design techniques presented in this dissertation is that even if two output signals have the same value, they may not be connected together unless both are driven by either slow or fast rails. This is an "ugly" constraint, requiring the designer know the implementation details of two gates whose outputs are connected. Breaking an abstraction barrier this way is distasteful, although the "workaround" is simple.

A better workaround is to use the "fast" rails as the default $\phi$ connections in a gate, and only use the slow rails for a non-inverting gate. This ensures that the outputs of all gates will transition at the same time and that the inputs to all gates will be restored at the same time, whether they are inverting or non-inverting.

## 4.6 Layout

Layout of reversible logic is very similar to conventional VLSI layout. The primary difference is the large number of power supplies. For connecting individual gates to the supplies, regularity makes layout easier. By using a standardized ordering for the supplies regardless

of rail phase, a single gate layout can be used in any logic stage. The rails run in strips along the chip, and the desired logic is inserted under the appropriate phase. The cells may abut as in conventional standard cell layout. The power supply rails run horizontally in the top metal layer and abut from cell to cell. Logic signals run vertically from one routing channel to the next. To connect the rails for a series connection of logic levels, a vertical bus of all supply rails runs below the top metal layer on the edge of the logic block. Appropriate vias connect the top metal, horizontal rail connections to the vertical bus.

At the chip boundary, the rails encircle the chip core in a "ring of fire" made up of concentric rings in the top metal layer. Each logic module's supply bus wires run to this ring and connect up to the appropriate rail. Each of the concentric supply rings connects to a pad.

## 4.7 Chip Testing

The goal of this thesis is to demonstrate a viable design methodology for building complex reversible systems. As such, it is relatively unconcerned with the actual energy dissipation of the reversible systems constructed. To test the functionality of the integrated circuits fabricated during this research, a suitable test board was needed.

Modern programmable logic devices, such as the Altera Flex 10K series parts used in the Pendulum test board, contain substantial memory capacity, large numbers of output pins, and a great deal of programming flexibility. One of the primary challenges of reversible logic testing is generation of the numerous power clock rails. The tri-state output pin feature of the Altera part was used to help generate the power clocks for testing XRAM and the Pendulum processor. A simple resistive voltage divider, using pairs of 10k$\Omega$ resistor packs, was wired to the 3.3V supply on one end, to ground on the other, and to both the Altera part and the reversible logic sockets. When the Altera output was set to high impedance, the voltage divider pulled the middle node to $V_{dd}/2$. At the appropriate time, the Altera part drives the node to one rail or another, with a time constant determined by the $RC$ loading on that particular rail wire. The time constant could be changed by using a different value resistor pack. The full swing $P$ rails were simply driven by the Altera outputs and

not attached to a voltage divider.

The signal pins on the reversible logic parts, both inputs and outputs, were wired to the Altera I/O pins, providing maximum flexibility. The Flex 10K series contains enough internal SRAM that it was used as the instruction memory for testing the Pendulum processor. Testing revealed a bug, discussed in Section 4.5.2, in the instruction encoder. Other systems were functional. The register file operation was verified, the ALU adder and both PCU adders were operational, and a small (10 instruction) program of sequential instructions was run correctly.

The board contained a socket for testing Pendulum and a socket for separately testing the XRAM chip. While Pendulum did not function perfectly, the XRAM chip did. When presented with an address, an enable signal, and eight bits of data, the memory chip was able to perform the exchange operation at any desired location.

The power dissipation of the Pendulum processor chip was measured by noting the difference in current draw when the Pendulum chip was not socketed versus when the chip was socketed. The Altera part and all supply rails were running identically in both cases. The board used a 4 MHz crystal oscillator to clock the Altera part, so the rail ticks occurred at that rate. The measured current draw was 610 $\mu$A, for a power dissipation of 2.0 mW at 4 MHz, or 0.5 mW/MHz. By comparison, the purpose built, low power adiabatic processor AC-1 [ATS+97] dissipated approximately 0.1 to 0.2 mW/MHz when running in the tens of MHz range. Conventional mainstream microprocessors dissipate approximately 50 mW/MHz.

## 4.8 Conclusion

The old standby tools of abstraction, hierarchy, and clear symbolic notation, are just as powerful when applied to reversible circuits as to conventional circuits. This chapter presented a suite of tools and rules that have successfully been used to design three fully reversible integrated circuits.

The valuable lessons learned from these design experiences are that reversible modules are fairly easy to construct. The reversibility of each gate and supply rail connections to each gate can be hidden with appropriate abstractions. Interconnecting modules is as easy as counting to three, and layout is comparable to conventional CMOS standard cell layout.

# Chapter 5

# Reversible Memory

This chapter discusses the design of fully reversible memory systems. It covers the basics of reversible memory design, as well as the details of a number of different implementation styles, including SCRL circuits. The complete memory system design presented in Section 5.2 is a suitable component for a reversible computing system.

The literature contains a number of adiabatic and partially-reversible memory system designs. Hall's description of a reversible instruction set [Hal94] touched on the idea of exchange-based reversible memory operations but did not discuss implementation issues. Tzartanis and Athas [TA96], Somasekhar, *et al.* [SYR95] and Avery and Jabri [AJ97] have proposed memory systems which use some form of energy recovery. It must be emphasized, however, that these are *not* reversible memory systems. They are merely traditional memory designs with some degree of energy recovery performed during operation.

## 5.1 Reversible Memory Design Styles

From a system point of view, the only additional requirement of a reversible memory, beyond a traditional memory system's function, is that it not erase bits when it is read from and written to. The memory must of course perform as a random access memory, allowing bits to be stored and retrieved. Erasing a bit means that at some point in the operation of

the memory the value stored on some node is lost. This occurs when an operation such as Landauer's RESTORE TO ONE [Lan61] is executed.

An exchange-based memory is reversible, and if implemented in a reversible logic style, does not erase bits. Bit erasure can happen as a fundamental side effect of the operation of the memory or as a function of the particular implementation. For example, one can imagine a memory in which the externally visible values being stored and retrieved are never lost but the implementation of the memory is such that intermediate bits are erased internally. The next subsection deals primarily with the higher level issues of avoiding bit erasure. Avoiding bit erasure in the lower-level switching circuit implementation is discussed in the subsequent subsections.

### 5.1.1 Irreversibility of Conventional Memory Architectures

The architecture of a memory is the high-level specification of how the memory is accessed. A traditional SRAM architecture is based on read/write operations. An address is presented to the memory and, based on a read/write and possibly an enable signal, a word of data is read from or written to the memory array. Data may be read from any location an arbitrary number of times, and data written to a location overwrites that location's previously stored data.

Reading a value does not at first seem to pose any difficulty to a reversible computing system. Reading from a standard memory creates a copy of the stored value and sends it to another part of the computing system. An arbitrary number of copies may be created this way. If, in a reversible system, the overall system can properly manage these copies, the memory need not be concerned with them. The larger system will, however, probably exhaust its ability to store or recover the bits generated by the production of an arbitrary number of copies. So it is a desirable feature of a reversible memory not to be a limitless source of bits when used in a larger system. It must be emphasized, however, that copy itself is *not* an irreversible operation.

An irreversible memory performs explicit bit erasure during writes because the previously

stored value is overwritten and lost. A reversible memory must save those bits somehow. The specific mechanism for this may vary.

### 5.1.2 Access through Exchange

Mechanisms to rule out bit erasure may be realized in a number of ways. For example, reads may be performed destructively, as in a DRAM, to avoid producing copies of the data. The information is moved out of the memory rather than being copied from it.

During writes, the value which would be overwritten could be pushed off to a separate location. This only postpones the problem until later since any finite capacity storage will be filled eventually.

If the separate location is accessible to the programmer, that data may either be useful or it may be possible to recover the space by undoing earlier operations. So if a write is preceded by a destructive read, the old information is moved out of the memory and into the rest of the system, and the new information replaces it in the memory. The old value has been *exchanged* for the new value. This type of eXchange RAM memory architecture, or XRAM, is the primary focus of this chapter.

The initial condition of the memory may be specified to be entirely clear, so that the first writes do not overwrite any information, or the system may tolerate some bit erasure at startup until the memory is filled. If the memory starts up in some unknown state, the results of computations performed using those unknown values will also be unknown.

The use of exchange operations avoids the need for a "garbage stack" [Res81]. The actual exchange operation is not generally useful, in that programs most often interact with the memory as if it were performing loads and stores, loading previously stored values by inserting place-holders in their locations and storing values to locations known to be clear. Certain techniques can take advantage of the exchange beyond this load/store usage. Hall [Hal94] has written a sorting routine which uses exchange effectively.

The essential insight of the XRAM is that performing a read and then a write to the same memory location does not lose any information. One data word is moved out of the memory,

leaving an empty slot for a new value to be moved in. In general, *moving* data rather than *copying* is a valid technique in reversible computing for avoiding bit erasure on the one hand and avoiding producing large amounts of garbage information on the other.

If controlled information erasure is desired, a system could contain a "memory mapped bit bucket." An exchange with this location could always produce a zero value out. No explicit exchange instruction is needed, simply a place to send information from which it never returns.

The details of the memory interface and implementation may vary. The following subsections discuss in some detail three different exchange-based memory architectures. All of them are completely reversible and do not erase information in any stage of their operation.

The first, using charge-steering logic, was developed during the summer of 1996. Charge-steering logic is a design style based on using complementary transmission gates to direct the path of a logic signal. Some number of adiabatic clock phases, generally around six, power the system. These power clock signals are conditionally steered to the inputs of other transmission gates.

The second style uses Fredkin gates [FT82] to perform a series of bit exchanges to push the desired value out. Bits are repeatedly exchanged in the memory array in a butterfly-style network, and the amount of shuffling is determined by the XOR of the desired address and the previously accessed address. This approach to reversible memory design is not particularly successful because of its scaling behavior and wiring complexity.

The third style, SRAM-style reversible memory, is based on a traditional memory array flanked by address decoders. The bit cell resembles a Fredkin gate but uses reversible tri-state gates so that the data can be written and read on a set of shared busses. This shared bus design improves the scaling and wiring behavior of the memory over the Fredkin gate memory.

The XRAM chip, discussed in detail in Section 5.2, was designed in the SRAM-style. It has been fabricated in a 0.5 $\mu$m CMOS process. It uses Younis and Knight's three-phase SCRL [YK94] logic family to build an eight $\times$ eight memory with three-bit address decoders.

Continuing research has uncovered various improvements to the design which reduce the number of transistors in the bit cell. The XRAM memory block was used with minor modifications as the register file of the Pendulum reversible processor analyzed in Part III.

It is worth noting here that the simplest way to perform a reversible exchange is to connect two previously isolated nodes of known capacitance to each other through a known inductance. For equal-value capacitances, the period of oscillation is $\pi\sqrt{2LC}$. This technique uses lossless elements to physically exchange the charge stored on one node to the other. Insofar as CMOS circuits use lossy elements and do not use inductors, this technique is not useful.

In circuits using lossy switches, an exchange is performed as a series of steps. One technique is to measure the charge on one node and conditionally change its value based on the value of the other node, executing a conditional toggle. Another technique is to copy the stored value out of the memory and clear the storage location. The clear location is then available to receive a new value. This second multi-step technique of copy-clear-set, rather than conditional toggle, is the exchange technique used in the charge-steering logic memory. The SRAM-style memory performs a similar exchange. The Fredkin gate memory does not have bit cells in the traditional sense.

### 5.1.3 Charge-Steering Logic

Charge-steering logic (CSL) is a design style based on using complementary transmission gates to direct the path of a logic signal. It is similar to pass gate logic, but it uses a number of resonant clock phases to power the system. These power clock signals are conditionally steered to the inputs of other transmission gates. Energy flows into and out of the system through these non-dissipative, resonant power clocks.

Circuits constructed with charge-steering logic are quite different from SCRL circuits. Use of charge-steering logic for reversible circuits is a novel technique and is discussed in some detail here. Both CSL and SCRL logic styles use a set of resonant power clocks to power the circuits, but CSL rails are all full swing, as opposed to the set of half-swing SCRL rails, and

generally fewer in number than SCRL rails. Both techniques are pipelined, with a separate path for injecting and removing energy into and from the circuit. SCRL circuits do this through a more formal set of forward and backward logic blocks while CSL circuits are at this point more purpose built and *ad hoc*.

It is also useful to remember in the discussion of reversible memory that the various implementation strategies may be decoupled from the basic reversible operations, such as using exchange to access memory. Reversible operations may be implemented in a non-adiabatic circuit family, although generally only for pedagogic reasons or simulations. The XRAM chip and the Fredkin gate memory were both designed using SCRL because of designer familiarity and because it is known to be useful for designing integrated circuits. Charge-steering logic was investigated as part of a search for the simplest style of memory implementation (essentially by measure of transistor count), and so the implementation style in this case drove the memory topology.

The basic structure in charge-steering logic is a transmission gate (t-gate), which can be modeled as a bidirectional switch controlled by a single signal (actual implementations are dual-rail). If the control input b is wired to a capacitive node and one of the bidirectional nodes is wired to a clock that swings from low to high, the value of the other bidirectional node will be a copy of the capacitive node, as shown in Figure 5-1(a).

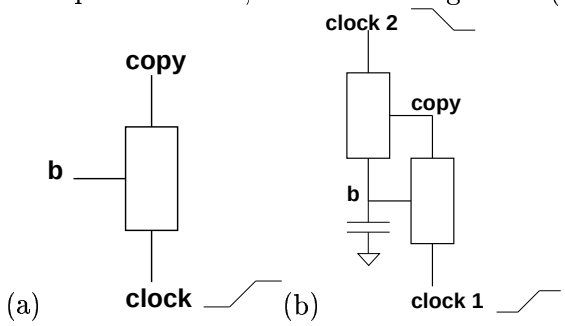


Figure 5-1: (a) Charge-steering logic copy. (b) Charge-steering logic clear.

---

This structure can be used to perform a non-destructive read (copy) of the bit stored at b. This is the first stage of a copy-clear-set exchange access. The copy is at a valid value while the clock is high.

The next stage is to clear node b, by recovering the charge stored there, before the new

value is written. Using copy as input to a second t-gate, clock 2 is swung from high to low and b tracks it from high to low if it was high (the t-gate is on), and left untouched if it was low initially. The requisite structure is shown in Figure 5-1(b). When the copy is made, if b was high, the second t-gate is switched on. Both sides of the gate are known to be high. When clock 2 swings low, the charge stored at b is returned to the power supply. Had b been low, the first t-gate would have been off and copy would have remained low, so b would not be connected to clock 2 and b would have remained low.

Note that at this point, copy is a copy of the original value of b and the capacitive node b is known to be zero. No information has been erased and the entire operation has been reversible.

Putting a pair of t-gates in parallel creates a dual-input gate (a dual-rail version would have four inputs). Replacing the single input gates of Figure 5-1(b) with these dual-input gates allows signals to be passed forward. Connecting a number of clearing structures in series creates a shift register, in which values are passed along, copied from b to copy1 to copy2 and so on, shown in Figure 5-2 [ASK+94]. Each node is then cleared non-dissipatively, so the values are moved along the shift register.

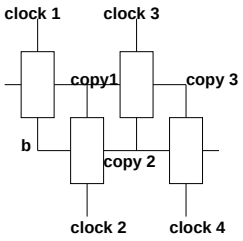


Figure 5-2: Charge-steering logic shift register.

To use the structure of Figure 5-1(b) as part of a memory cell, a t-gate must be added to write the new value. The new value is effectively copied to b from Din, as shown in Figure 5-3. The value on node Din is cleared by another t-gate, and the copy created is sent on to a later circuit.

The timing of clocks one, two, and three are not shown in these figures, only their initial

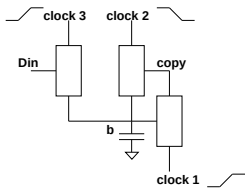


Figure 5-3: Charge-steering logic write.

and final levels. In Figure 5-3, the clocks swing sequentially: one, two, three.

Figure 5-3 is essentially a single bit cell for a charge-steering logic memory. The operation is fully reversible and non-dissipative. No information is lost. A value is moved from the bit cell storage node, b, to copy where it may be sent along and used for other computations. A new value is written to the storage node when Din is moved into b.

The next challenge is to gate the clocks to address one of some number of bit cells. Addressing is performed by steering another clock through a tree of t-gates. The control for the t-gates is provided by the address bits. A two bit decoder for a single clock is shown in Figure 5-4.

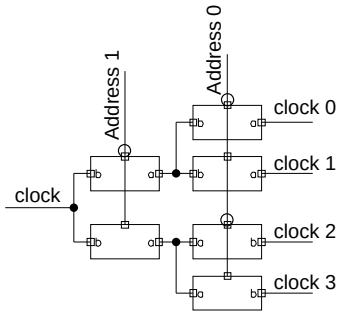


Figure 5-4: Charge-steering logic two bit decoder.

With a pair of valid address bits, clock swings from low to high. Exactly one of the four output clocks is driven high and may be used to gate a value into a bit cell. Once the exchange is performed, the input clock is then retracted, then the address bits are retracted. This operation is similar to Hall's "retractile cascade" [Hal92] and is fully reversible and non-dissipative.

A full schematic of the address decoders, four bits of memory, and a recirculating shift register path, is shown in Figure 5-5. The address inputs are assumed to be generated by

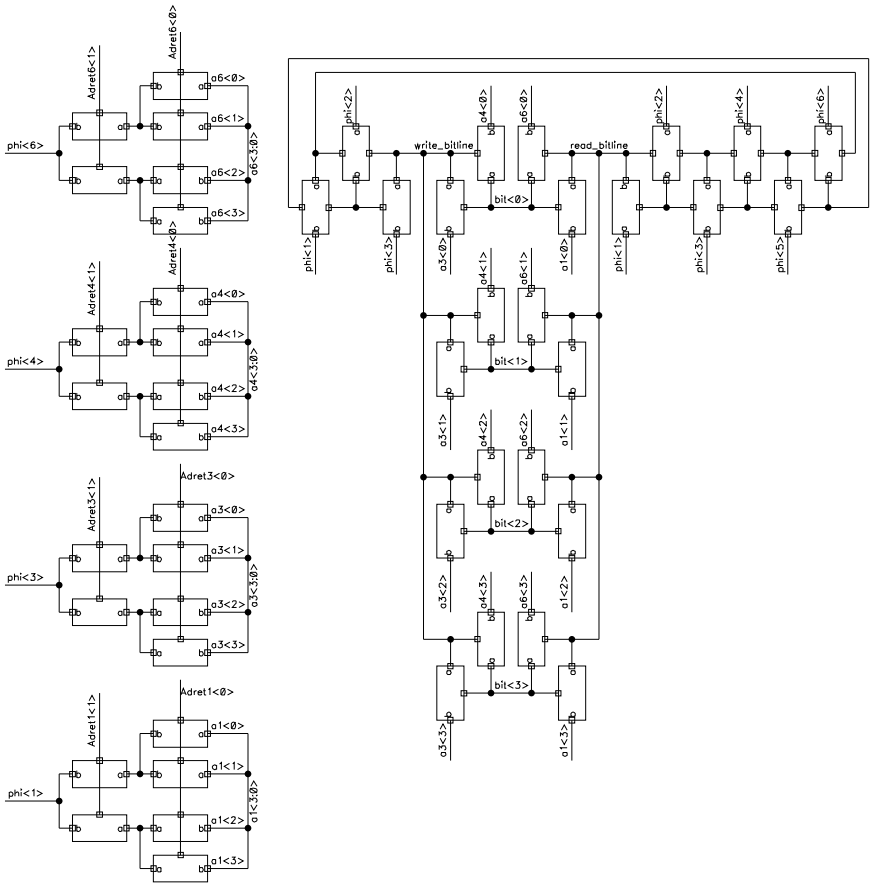


Figure 5-5: Charge-steering logic four bit memory schematic.

some other external, reversible control circuitry. For simulation simplicity, values read out are passed through a simple shift register and recirculated back to the memory. This models the computation system to which the memory is assumed to be attached. The only other inputs to the memory are resonant clocks. The design is fully reversible and asymptotically non-dissipative.

### 5.1.4 Fredkin Gate Memory

This memory design was developed primarily to examine the difficulty of building a memory out of Fredkin gates [FT82]. The Fredkin gate, shown in Figure 5-6, has three inputs and three outputs. It is both universal and reversible. The gate swaps the B and C inputs

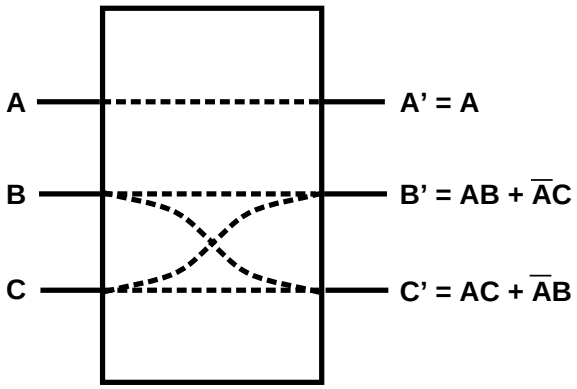


Figure 5-6: The Fredkin Gate

if the A input is zero; otherwise all three inputs are passed unchanged to the output. It is therefore its own inverse, as passing a set of signals through two series Fredkin gates always restores the original values. In most discussions of Fredkin gates, a unit delay for each signal path and a fanout limit of one is also assumed. For larger fanouts, buffers can be constructed from additional gates, at a delay penalty.

As the purpose of this design was to investigate gate-netlist-level issues, the underlying implementation of the logic gates was deemed unimportant, other than that it be reversible. The Fredkin gates used in the memory were implemented with dual-rail signaling and SCRL logic in these simulations. This is not an especially efficient way to build Fredkin gates: each gate is comprised of 96 transistors. An existing cell library infrastructure and designer familiarity strongly influenced the decision to use SCRL. It is important to note that Fredkin gates could also be implemented in other logic styles, such as the charge-steering logic of Section 5.1.3.

A properly wired Fredkin gate is a one-bit exchange RAM. The A input is the (decoded) address bit. It is passed through to the A' output. The B input, wired to the B' output,

forms a recirculation path for the stored bit. The C input and the C' output are the data input and output, respectively. If the single bit location is not being addressed, a bit entering the memory at C is passed though to C', and the stored value is recirculated from B' to B. If the bit location is addressed, the stored value is sent from B to C' and the new value is sent from C via B' into the recirculation path. This "conditional-exchange bit cell" forms the basis for the all-Fredkin-gate memory described in this section, as well as for the bit cell of the XRAM design described in Section 5.2.

Some of the more "obvious" ways to extend a one-bit memory to larger capacities must be ruled out when not only the system, but also its constituent gates, are to be reversible. For example, it would appear that $n$ one-bit memories connected in parallel could be connected to single input and output lines by a 1-to-$n$ buffer and an $n$-to-1 multiplexer; but multiplexers are not reversible, since all input bits but one are lost. Another idea is to connect $n$ bit cells in series, pass the incoming data bit down the chain, and perform an exchange operation at the proper bit. The latency of such a memory would clearly be proportional to $n$, which would limit its usefulness.

This design instead extends the one-bit memory by expanding each recirculation path to encompass several Fredkin gates. The gates constitute a shuffling network, which connects one of the recirculation paths to the I/O paths, in the same manner as for the one-bit memory. A seven-bit example of such a memory is shown in Figure 5-7. All signals are dual-rail. Seven feedback paths connect the output of the shuffling network to its input. Three address bits, and therefore three columns of Fredkin gates, are needed to select one of the seven paths for exchange; the all-zero address causes the input datum to be passed through to the output.

The shuffling network can select any bit for exchange, but as a side effect, it shuffles the remaining bits in the memory core. To compensate for this shuffling, it is necessary to modify the address for the next access. With the butterfly network shown in Figure 5-7, this pre-computation is simple: the address is XORed with the previous address before being sent to the memory.

Accesses with the address zero cause no perturbation of data in the memory and so must

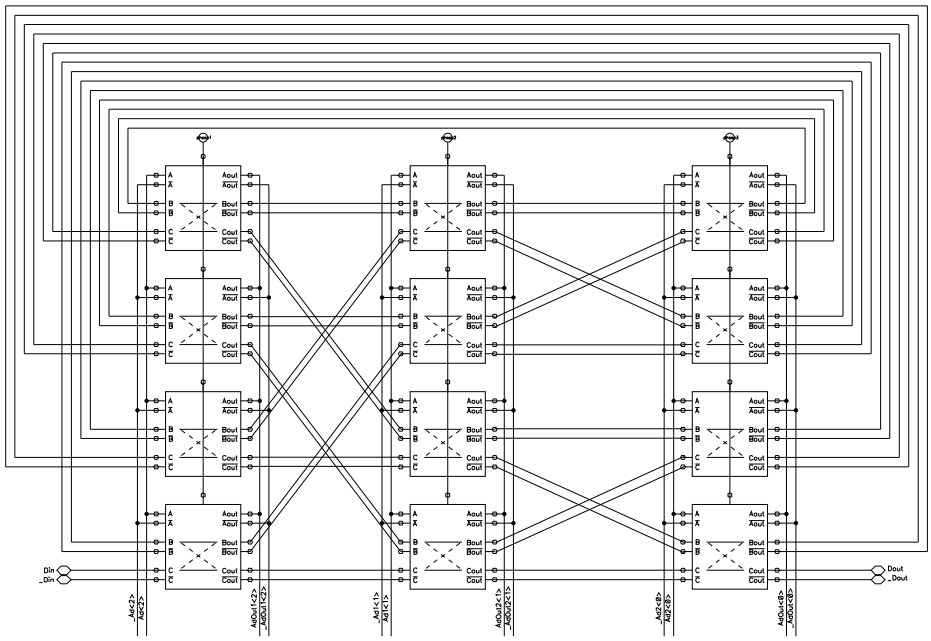

Figure 5-7: Seven-bit butterfly memory.

be excluded from pre-computation. Addressing the same location twice always produces an XOR of zero, so must also be excluded from pre-computation. These exceptions are to account for the addition and removal of data from the memory through the I/O ports.

The latency of this memory grows as $\log(n+1)$, where $n$ is the number of bits. However, the number of Fredkin gates grows as $\frac{n+1}{2} \times \log(n+1)$. This scaling behavior is clearly undesirable. Wiring complexity also becomes unmanageable as the number of bits increases. Note, however, that networks of other topologies could be used in place of the butterfly network. It would also be possible to trade off wiring complexity for logic complexity by introducing extra columns in the memory array. These possibilities have not been investigated further. The rich theory of shuffle networks suggests that further study might be rewarding.

In summary, the Fredkin-gate-based memory described here is intriguing in its use of a perfect-shuffle network. At present, however, wiring and logic complexity limit its appeal.

### 5.1.5 SRAM-style Reversible Memory

Conventional SRAM bit cells are based on the notion of a recirculating data path. In a CMOS memory cell, the data is held by a recirculating path made up of two cross-coupled inverters. When a bit cell is written and changed, the inverters "fight" with the bit lines and lose, toggling the value stored in the cell. When the cell is read, the cross-coupled inverters drive the stored value, non-destructively, onto the bit lines where it is amplified and sent on. All the bit cells in a single column share, through time multiplexing, single bit line. This time multiplexed single wire driven by multiple signal sources is known as a shared bus. Cells that are not being read or written have a high-impedance connection to this shared bus. A cell stores both polarities of the bit, and, when selected, reads or writes one polarity to each of the bit lines.

The distinguishing feature of an SRAM-style memory is that it has an array of bit cells that grows linearly with the number of bits in the memory, arranged in a number of multi-bit word rows, and a set of shared bit line buses. Address decoders drive word-lines along the rows to select one of the words. The basic architecture of a reversible SRAM-style memory is not fundamentally different from a traditional irreversible SRAM.

Making this type of memory reversible presents a number of challenges. The use of a shared bus requires the use of a multiplexer or tri-state gate made in a reversible logic family. The address must be decoded and sent to the array with the correct timing. And the bit cell must be able to hold a bit and perform a reversible exchange, while interfacing correctly to the shared bus. Some of these challenges also apply to the charge-steering logic and Fredkin gate memories. The next section describes in detail the design of the XRAM chip, an SRAM-style memory.

## 5.2 The XRAM Chip

The XRAM chip is a fully reversible implementation of a reversible memory architecture. The chip has been fabricated in a 0.5 $\mu$m CMOS process. Instead of reading and writing from and

to the memory separately, memory locations are accessed with an exchange operation. The value stored is read out and a new value replaces it. No information is lost in this process and it is therefore reversible.

The design was intended to be a proof of concept with an eye to reusing the XRAM design as the register file and memory of the Pendulum processor. The modifications to the XRAM chip are discussed in Section 8.2.2. The primary difficulty was the addition of "glue logic" to use the single-port XRAM as a triple-ported regfile.

### 5.2.1 Top Level Architecture

The memory is arranged much like a traditional SRAM. A block of address decoders at the input drives dual-rail word lines across the array of eight 8-bit words. Each bit cell in a selected word drives a bit line. Since this is a reversible memory, however, all the input signals must be passed through the chip to be sent back to the part of the system that generated them. In particular, the addresses have to be re-encoded at the output.

The memory's inputs are an enable signal (EnIn) and its complement, three address bits (Ain) and their complements, and eight bits of data (DataIn). It outputs the same enable signal (EnOut=EnIn), the same address bits (Aout=Ain), and eight new bits of data (DataOut). DataOut is the value previously stored at the addressed location, *i.e.,*the value being read from the memory. The value of DataIn replaces DataOut as the value stored in the memory at address Ain. Figure 5-8 is a diagram of the XRAM input and output signals.

This memory architecture performs trivially a reversible operation. To reverse the process, EnOut, DataOut, and Aout are used as inputs to the memory, which then outputs EnIn, DataIn and Ain, the values with which the whole function started.

The exchange operation in a single bit cell is conditional on the select bit. Therefore, there are two modes of the bit cell: holding and exchanging. Holding describes the mode in which the cell is not being selected and is therefore holding its current value. Exchanging describes the selected mode in which a value is read in while the previous value is read out.

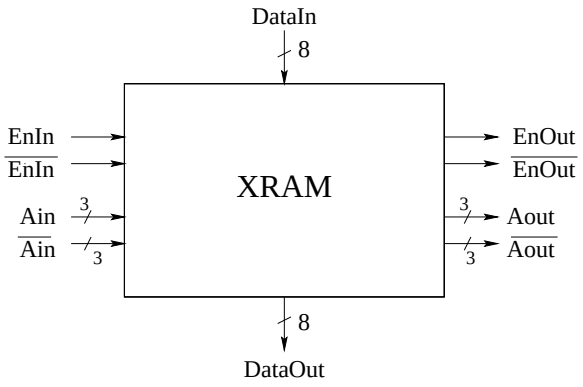


Figure 5-8: Top level design of the XRAM chip.

Because non-inverting stages require additional power-clock rails in SCRL, it is advantageous to have both a signal and its complement available as inputs to logic functions. Thus, dual-rail logic was used in most of the circuit design.

The memory has a latency of five clock phases because of the inherently pipelined nature of SCRL circuits. The address decoding is done in one phase, the memory-cell latency (from data in to data out) is three phases, and re-encoding the address takes one phase.

The top level schematic of the XRAM chip is shown in Figure 5-9.

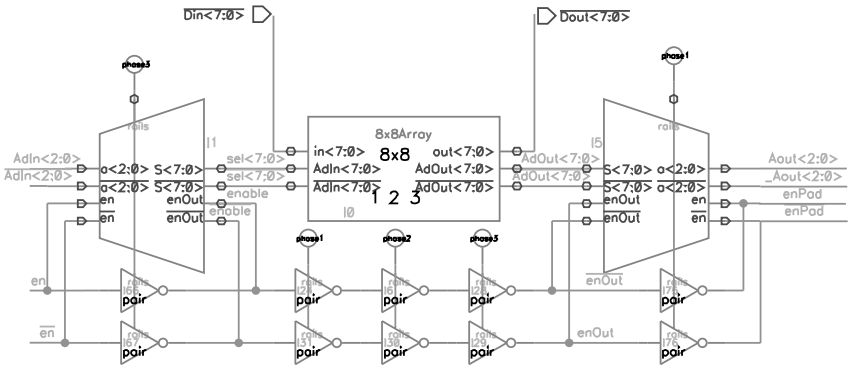


Figure 5-9: Top level schematic of the XRAM chip.

### 5.2.2 XRAM Bit Cell

The XRAM bit cell is a three-input three-output logic gate. The block diagram is shown in Figure 5-10. Like the Fredkin gate memory cell described earlier, the B input is wired to the B' output to form a recirculating path. The A input is tied to a word select line, the C input is tied to the input bit line DataIn, and the C' output is tied to the output bit line DataOut. The bit cell behaves like a Fredkin gate memory cell when A is high: the values on B and C are exchanged. The previously recirculating value is passed to DataOut, and the new value at DataIn is passed to the recirculating path.

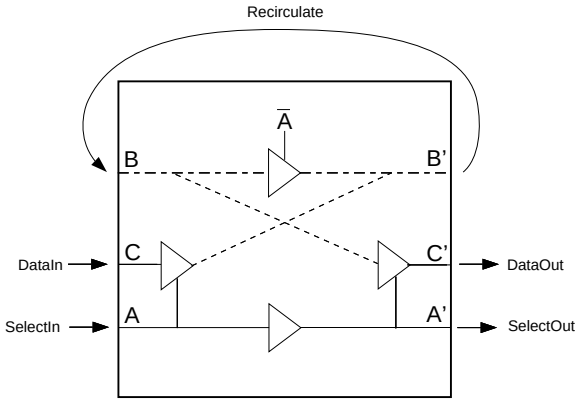


Figure 5-10: Block diagram of a bit cell.

When the A bit is low, the gate behaves differently than a Fredkin gate memory cell. The B input is still passed to the B' output, but the C input is ignored and the C' output is set to high impedance. It is the use of tri-state buffers that allows the bit cells to occupy a shared bus. This design gives the XRAM memory better scaling behavior than the Fredkin gate memory described above.

### 5.2.3 Address Decoders

SCRL design style dictates that each logic function be cross-coupled with its logical inverse for charge recovery. Therefore, an SCRL decoder is implemented by cross-coupling a standard decoder and encoder. The decoder is the forward path, and the encoder is the return path.

The top-level design of the XRAM chip requires the address bits to be re-encoded at the output. Similar to the decoder, an SCRL encoder uses a standard encoder as the forward path cross-coupled with a standard decoder as the return path. Hence, an SCRL encoder is the same circuit as an SCRL decoder, the only difference being which terminals are used as the input and which are used as the output.

The decoder/encoder design illustrates an important property of SCRL circuit design: once a logic function has been designed, the inverse of that function is often a trivial extension. Using the input as the output and the output as the input inverts the logic block. This property of SCRL design can be exploited to ease design and layout.

The decoder also incorporates an enable bit. When the XRAM is enabled, the address bits are decoded as usual, selecting one word line. However, when the XRAM is disabled, all word lines are deselected, so all the memory cells hold their state.

The reversibility of the decoder/encoder circuit is complicated by the addition of the enable bit. When the memory is deselected, the address bits are ignored. The state of the word lines does not depend on the particular address at the input. It may seem that information is lost and reversibility is destroyed by this operation. When the encoder is presented with no selected word line, which address should it generate for output? The solution is that the encoder sets all the output address bits to high-impedance. Reversibility is not broken with this procedure if the XRAM is incorporated into a reversible, shared bus structure as described below. This scheme also preserves the complementary design of the decoder and encoder; they remain inverses of each other.

The particular implementation of XRAM described here is an eight-bit by eight-word array of memory. The address decoder is a 3:8-bit operation, and the encoder is an 8:3-bit operation.

### 5.2.4 XRAM Shared Bus Design

As mentioned above, the XRAM chip uses a shared bus architecture for the bit lines. All the bit cells in a single column share, through time multiplexing, a single bit line. This time multiplexed single wire driven by multiple signal sources is known as a shared bus.

Tri-state SCRL gates are also discussed in Section 4.4.

The difficulty in constructing a reversible bus arises from the inherent irreversibility of a multiplexer. A simple two-to-one multiplexer takes two inputs and a select bit and outputs the value of one of the inputs. The second input is ignored. This is clearly irreversible since the value of the non-selected input can not be determined from the value of the output and the select bit.

One conventional implementation of a shared bus or multiplexer is a set of tri-state (high, low, and high-impedance) gates whos outputs are wired together. Exactly one of the tri-state gates is allowed to drive the shared wire. Likewise, a decoder may operate by having an input signal passed through to exactly one of many output wires.

Using an SCRL tri-state gate as part of a multiplexer driving a shared bus introduces a problem in the energy recovery at the input node of a tri-state gate that is not driving the bus. Since the reverse path is also in a high-impedance state, the input node is isolated from the energy recovering reverse path. This problem is solved by guaranteeing that the input node was not driven by the previous stage. How then are any signals ever driven? The answer is that a shared output bus must have a symmetric input bus earlier in the pipeline. The shared input bus, a decoder (the dual of the multiplexing shared output bus), drives exactly one of the outputs. The other paths in the pipeline are in a high impedance state from that point in the pipeline until the shared output bus. A block diagram of the fan-out, memory elements, and fan-in is shown in Figure 5-11.

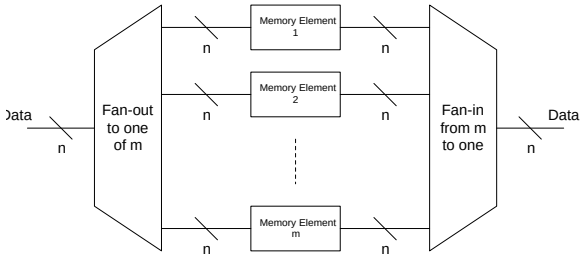


Figure 5-11: Memory system fan-out and fan-in.

The system must guarantee that the gate that is driven by the input shared bus will recover

the value stored on the bus, and that the one pipeline path that passes a value to the output shared bus will use that value to recover the energy back along that path.

## 5.3 Conclusion

Since reversible circuits are generally fairly large and complex, it may be advantageous either to design a circuit which takes advantage of reversibility at some low level, such as the Fredkin gate-based memory, or to design a reversible circuit based on a very simple conventional circuit. The Fredkin gate memory presented in this chapter is elegant but not particularly suited to a traditional CMOS implementation. The other memory designs are more reasonable, but are still large since they are static memories.

One solution to the size and complexity of reversible SRAM is may be to use dynamic memory. A conventional dynamic RAM is very simple, needing only one transistor and a capacitor to store a bit. Leakage is of course irreversible and dissipative, but does not have a thermodynamic lower bound. A reversible DRAM, especially one constructed in SCRL, will be substantially different and more complex than a conventional DRAM. The problems with overwriting a stored value still exist, but may be addressed by incorporating writing into the refresh logic.

This work investigating various possible designs of reversible memory will undoubtedly lead to improved techniques, and the experience of designing complete integrated circuits in different reversible logic families has led to better understanding of the design tool requirements and methods specific to reversible computing.

# Chapter 6

# Design Example: A Bitsliced ALU

The Pendulum processor's bitsliced ALU is a good example of a fairly complex reversible module built using the techniques from Chapter 4. It is a fully combinational module; memory issues are covered in Chapter 5. This chapter covers the low-level details of performing reversible arithmetic and the timing difficulties arising from a module with twenty-six pipeline stages and six extra delay stages to match the ALU output to the PCU output. Details of how the ALU fits into the Pendulum datapath are covered in Part III.

## 6.1 Functional Description

The Pendulum ALU performs four 12-bit arithmetic and logic operations: `add`, `andx`, `orx`, and `xor`. The `andx` and `orx` instructions are logical AND and OR followed by an XOR, respectively. These two instructions require three operands, performing $Z^+ = (A \cdot B) \oplus Z$ and $Z^+ = (A+B) \oplus Z$. The addition and XOR instructions require two operands, performing $A^+ = A + B$ and $A^+ = A \oplus B$.

The ALU also encompasses a shifter which performs left and right logical shift, right arithmetic shift, and left and right rotate. All shifts and rotates are by one bit position. The shift operations are followed by XOR, *e.g.*, producing $Z^+ = (A \gg 1) \oplus Z$. Shift operations take two operands while rotates take a single operand.

Addition is performed with a straightforward ripple-carry. No carry lookahead or other fast arithmetic operations are performed.

The ALU symbol is shown in Figure 6-1. It takes in three twelve-bit operands, a dual-rail direction signal, two decoded control buses, and the dual-rail five-bit opcode. One control bus is for the shift operations, the other is for the arithmetic and logic operations. Each bus is fourteen wires wide, encoding seven signals in dual-rail pairs. The three operands are single polarity; all other signals are dual-rail, for a total of seventy-six wires entering the ALU.

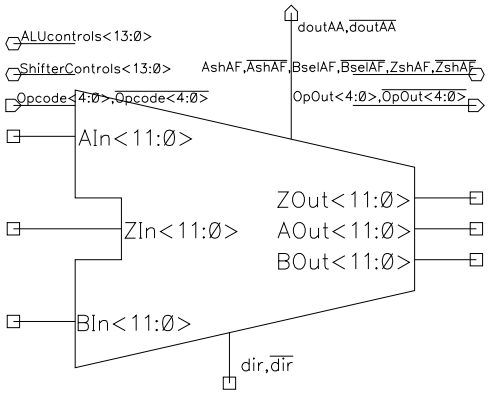


Figure 6-1: The Pendulum ALU symbol.

During the course of execution, some of the fully-decoded control signals are re-encoded. The total of twenty-eight input control signals were originally decoded from the five dual-rail opcode signal pairs, so the opcode value can be used to re-encode the control signals when they are no longer needed. At the output of the ALU, only three dual-rail signal pairs remain. The direction signal, the opcode, and three result signals are also output, for a total of fifty-four wires.

From input to output, each signal passes through thirty-two levels of logic. The first logic level generates both polarities of the three operands and computes two more control signals. The next 24 levels perform the arithmetic and logic operations. Twelve levels are necessary for the 12-bit ripple carry add. The shift operation is performed in the first four ALU stages. The operands are then sent through nine delay stages to synchronize the shifter

output with the arithmetic and logic output.

The next twelve levels perform an "unadd" operation; the need for unadd is covered in Section 6.2.5. Operands and results from instructions other than add are sent through 12 delay stages to synchronize the values with the unadd output.

Stage twenty-six combines the dual-rail operand signals back into a single polarity. It also encodes a control signal. The final six stages are delay elements to match the execution time of the ALU to that of the PCU. The block diagram is shown in Figure 6-2.

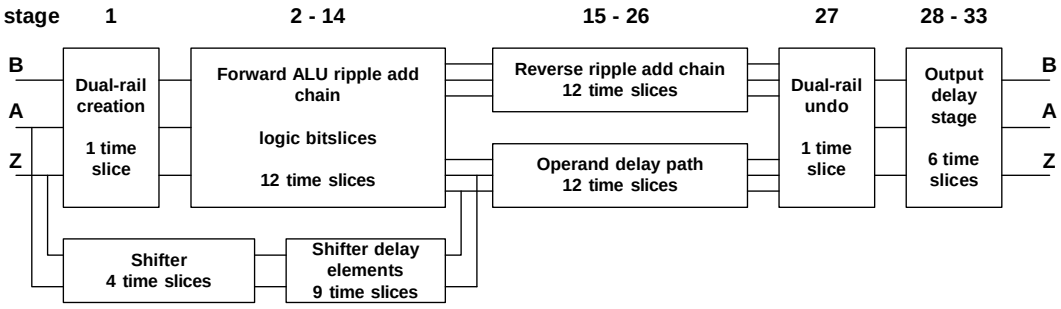


Figure 6-2: The Pendulum ALU block diagram.

The control signals that pass through the ALU are encoded as soon as they have served their function. The processor's instruction decoder computes some ALU control signals from the opcode; since the opcode is passed through the ALU, the information necessary to uncompute the control signals is readily available. One of the challenges of reversible system design is not only to ensure that information is not discarded but that it be available and in a suitable form in the right place at the right time. The technique of encoding information as soon as possible is an efficiency optimization intended to simplify the designer's task. The primary benefit of re-encoding quickly is that fewer signals must be routed through the downstream logic. The drawback is that quite a few small encoding blocks must be specially designed.

The adder computes modulo addition, with no overflow. To keep the processor simple exceptions and interrupts are not handled; computing an overflow bit turns addition from a non-expanding operation to a more complex conditionally expanding operation. The ALU

also does not support a programmer-visible subtraction operation. It is capable of performing subtraction but only as an addition performed in reverse. That is, when the direction bit is set to reverse, the `add` opcode causes a subtraction to be performed. Subtraction was excluded merely to simplify instruction decoding. Subtraction is implemented by inverting the $B$ operand and setting the least significant carry-in bit to one, thereby computing $A + \overline{B} + 1$ or twos complement subtraction.

## 6.2 Forward and Reverse Bitslices

A simple design for a conventional ALU is shown in Figure 6-3. The input operands fan-out to the various functional units, and the outputs from each unit are multiplexed to the ALU output. Each functional unit performs a computation and all but one result is "thrown away." This is clearly not feasible for a reversible processor.

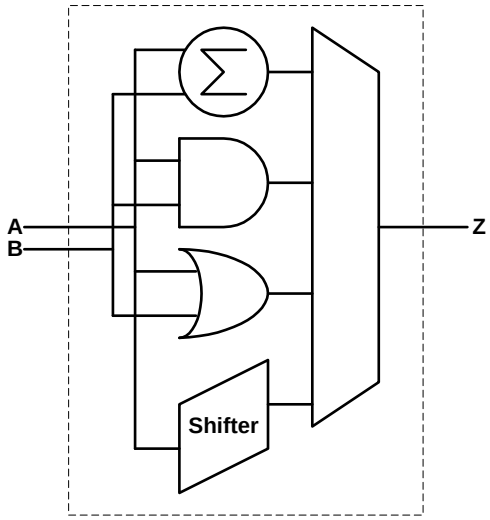


Figure 6-3: A conventional ALU design.

The Pendulum processor does have a similar structure, with multiple functional units. However, the shared input bus and shared output bus connects to these units through tri-state gates, as designed in Section 4.4. At most one path through the bitslice is ever selected at a time. The input operands are steered through one functional unit and all the other

units are disconnected from the shared busses. When the ALU is idle, all of the tri-state paths are in the high-impedance state.

Conventional ALU's generally take two operand inputs and generate one output value. A reversible ALU must output enough information to reconstruct the inputs, including possibly passing the operands through the ALU unchanged. The Pendulum ALU passes expanding operations' input(s) through to the output. Two-operand non-expanding operations generate a result and one of the operands. One-operand non-expanding operations, rotate left and right, only generate a result. Recall from Section 4.2.1 that expanding operations, such as NAND have more outputs than inputs, while non-expanding operations, such as modulo addition, have the same number of inputs as outputs.

The most striking aspect of a reversible ALU is the "unadd" path which "uncomputes" the carries generated by each bitslice. The details of reversible addition are in Section 6.2.5 below.

The Pendulum bitslice schematic is shown in Figure 6-4. The following sections address the four components of the bitslice from simplest to most complex.

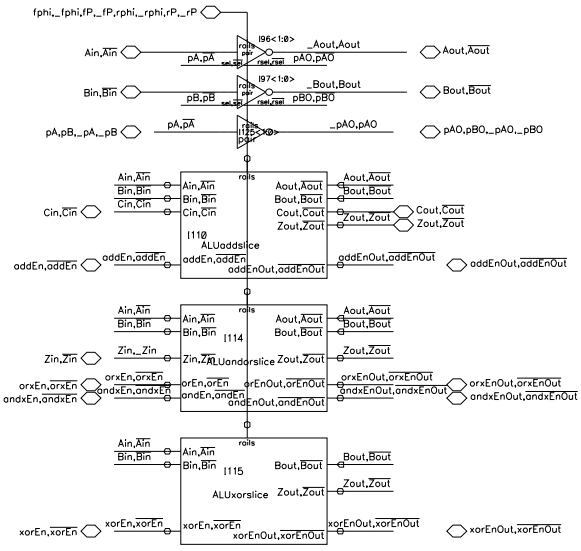


Figure 6-4: The Pendulum ALU bitslice schematic. The inverters at the top are operand feedthroughs. The three other blocks perform the XOR, ANDX, ORX, and addition operations.

### 6.2.1 Operand Feedthroughs

Two tri-state full inverters at the top of Figure 6-4 pass the $A$ and $B$ operands through the bitslice. The pA and pB signals, which enable the inverters, stand for "pass A" and "pass B." They are asserted based on which operation is being performed. For example, `and` is a two-operand expanding operation and passes both $A$ and $B$ through the ALU. The `xor` operation is a two-operand non-expanding operation and only passes $B$ through the ALU; the $A$ operand is unambiguously determined by the output value $Z$ and operand $B$.

The non-tri-state inverter passes the pA and $\overline{\text{pA}}$ control signals through the bitslice. The pB and $\overline{\text{pB}}$ signals are, because of a style error by the designer, passed through the ALU at a higher level of the hierarchy.

The operand feedthroughs inside the bitslice are different from the operand feedthroughs external to the bitslice, known as The Wedge. The internal feedthroughs pass only the one bit of each operand being used for computation in that particular bitslice. The Wedge is a feedthrough for all the other bits in the operands and results. The Wedge is large and, although composed entirely of wires and tri-state inverters, complex relative to the bitslice. It is covered in Section 6.3.

### 6.2.2 XOR

XOR is a non-expanding operation. The two inputs $Ain$ and $Bin$ fully determine and are fully determined by the two outputs $Zout$ and $Bout$. Figure 6-5 shows the tri-state dual-rail XOR gate and a full inverter for the enable signals. The XOR gate is made up of tri-state versions of Figures 4-16 and 4-16 (b). Note that the $Bout$ signals are actually reverse inputs to the XOR gate. The $Bout$ operand is passed through the bitslice one level up the hierarchy, in Figure 6-4. Rather than have each block in the bitslice pass its operands individually, replicating the tri-state inverters in each block, the operand feedthrough path is shared by all three blocks. The savings of two sets of four tri-state inverters comes at the cost of the pA and pB control signals; a worthwhile trade. The note "B not rec" on the XOR symbol indicates to the designer that the $Bin$ and $\overline{Bin}$ signals are not restored to $V_{dd}/2$ by the

XOR gate used here. Unlike an SCRL full-gate, all inputs are not restored to $V_{dd}/2$. Unlike an SCRL half-gate, one of the inputs is restored to $V_{dd}/2$. The XOR gate in Figure 6-5 is a sort of "three-quarters" SCRL gate.

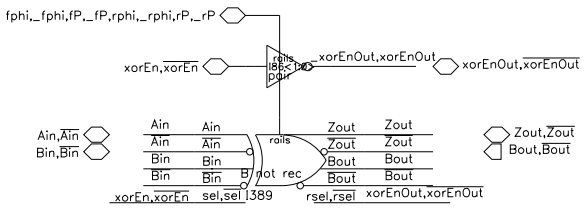


Figure 6-5: The Pendulum ALU XOR bitslice.

The XOR block restores the $A$ inputs to $V_{dd}/2$ and drives the $Z$ outputs. The $B$ inputs are restored by the tri-state inverters. This is a very simple block.

### 6.2.3 ANDX and ORX

Computing ANDX and ORX is slightly more complex. Since logical AND and OR are expanding operations, the block must have three outputs. To simplify interactions with the rest of the Pendulum datapath, the ALU executes the three-operand functions ANDX and ORX. The ANDX function is defined as $Z^+ = (A \cdot B) \oplus Z$. Similarly, ORX is $Z^+ = (A + B) \oplus Z$. Each bitslice is constructed using only one level of logic, *i.e.,* one pipeline stage. Therefore, the ANDX and ORX blocks must satisfy the truth table in Table 6.1.

| in | | | out | | | |
|---|---|---|---|---|---|---|
| $A$ | $B$ | $Z$ | $A$ | $B$ | $Z^+ = (A \cdot B) \oplus Z$ | $Z^+ = (A + B) \oplus Z$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |

Table 6.1: ANDX and ORX truth tables.

The schematic for a tri-state half-ANDX gate is shown in Figure 6-6. Four of these gates are required for a dual-rail full ANDX gate: one forward gate for each polarity of the output $Zout$ ($Z^+$ above) and one reverse gate for each polarity of $Zin$ ($Z$ above). The $A$ and $B$ operands are passed through the tri-state inverters one level up the hierarchy. Likewise, four tri-state half-ORX gates are required to generate the $Zout$ values and restore the $Zin$ signals to $V_{dd}/2$.

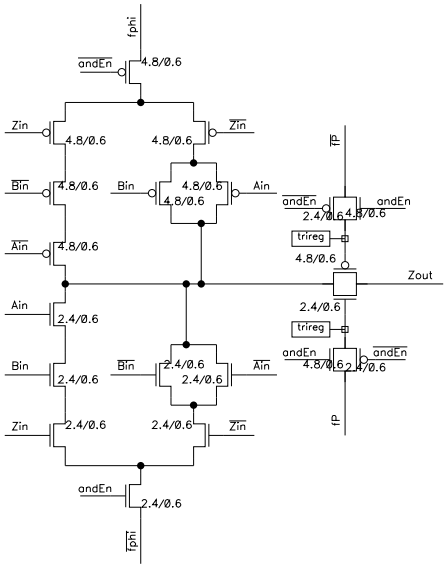


Figure 6-6: An SCRL half-ANDX gate.

The enable signals for these two tri-state gates are passed through a pair of full-inverters. As with the XOR gate, the $Aout$ and $Bout$ signals needed to restore $Zin$ to $V_{dd}/2$ are inputs to the ANDX and ORX gates.

The moderate complexity of these gates relative to the simple XOR arises from the more complex logic function described in Table 6.1. The basic operation of this gate is otherwise very similar to the XOR gate.

### 6.2.4 Addition

Modulo addition is a non-expanding operation; given two operands $A$ and $B$, the sum $Z = A + B$ and $B$ are sufficient to exactly determine $A$. The truth table for modulo

addition of two two-bit numbers is shown in Table 6.2. Note that the sixteen distinct input combinations produce sixteen distinct output combinations.

| in | | | | out | | | |
|---|---|---|---|---|---|---|---|
| $A_1$ | $A_0$ | $B_1$ | $B_0$ | $Bout_1$ | $Bout_0$ | $S_1$ | $S_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Table 6.2: Two-bit summand modulo addition truth table.

However, a full adder performs an expanding operation. The truth table for a full adder is shown in Table 6.3.

The carry-sum pairs of bits `01` and `10` can each be generated with three different input combinations. Two other bits are required to distinguish between these three possible input combinations, for a total of four output bits. The full adder has only three input bits, so it is an expanding operation.

The difficulty with computing multiple-bit addition is that addition is not a bit-parallel operation; adding two $N$ bit numbers requires a large function of $2N$ inputs. The difficulties of computing carry lookahead values for large $N$ are well known [PH90]. As will be seen in the next section, however, a bit-serial reversible ripple carry adder incurs substantial overhead from operand distribution. Use of conventional fast adder techniques to minimize the logic required by a reversible adder is still an open research question.

The current discussion is restricted to performing modulo addition using an expanding

| in | | | out | | | |
|---|---|---|---|---|---|---|
| $A$ | $B$ | $Cin$ | $A$ | $B$ | $Cout$ | $S = A + B$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 6.3: Full adder truth table.

ripple carry block. The block takes dual-rail pair inputs of $A$, $B$, and $Cin$. The dual-rail sum, labeled $Zout$ and $\overline{Zout}$ in Figure 6-7, and carry out signals are generated from the three dual-rail inputs. The carry in signal can be computed from the $A$ and $B$ operand values and the sum according to the equation

$$Cin = Cout \cdot (\overline{B} + \overline{A} + Zout) + \overline{A} \cdot \overline{B} \cdot Zout \tag{6.1}$$

The reversible ripple carry full adder computes the output values from Table 6.3; the two operands $A$ and $B$ are passed through the block using tri-state inverters. The inverters restore the input values of $A$ and $B$ to $V_{dd}/2$. A circuit implementing the equation above restores $Cin$ to $V_{dd}/2$. The sum bits computed at each stage of the ripple carry chain are passed through tri-state inverters for retiming.

### 6.2.5 Un-Addition

Note that at the end of the chain, the values of $A$, $B$ and $Cout$ are retained, as well as the desired sum. Rather than performing the desired mapping of $(A, B) \leftrightarrow (S, B)$ where $S$ is the modulo sum, the adder has so far computed $(A, B) \rightarrow (Cout, S, A, B)$. The intermediate, "garbage," values of $Cout$ and $A$ must now be "uncomputed." A second ripple adder chain uncomputes the carry out and $A$ values using the sum, $B$, and $Cout$. This is the opposite of using $A$, $B$, and $Cin$ to compute the sum and $Cout$. The ALU therefore computes

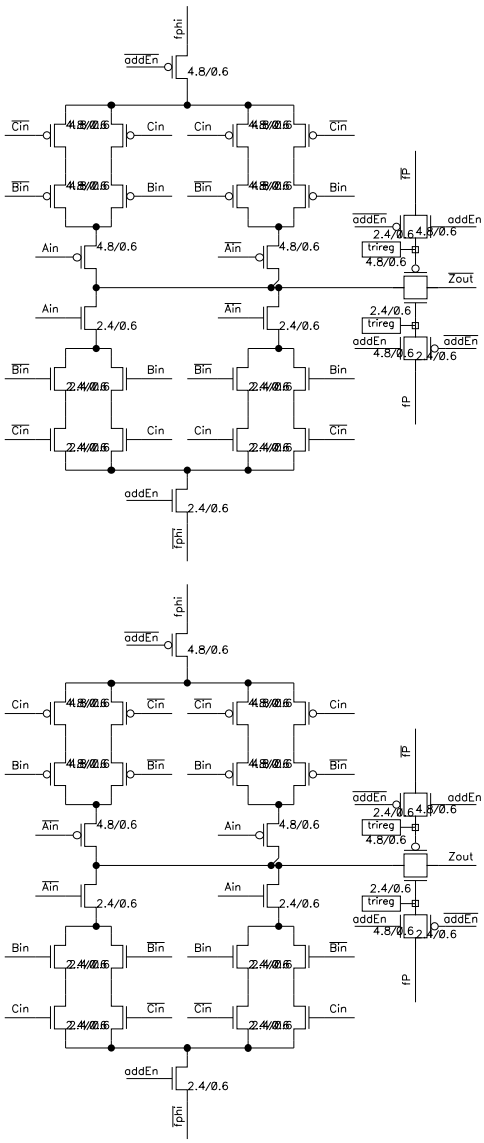


Figure 6-7: The Pendulum ALU ripple carry add slice.

$(A, B) \leftrightarrow (Cout, S, A, B) \leftrightarrow (S, B)$. The truth table for the uncomputation is shown in Table 6.4.

| in | | | | out | | |
|---|---|---|---|---|---|---|
| $A$ | $B$ | $Cout$ | $S$ | $B$ | $S$ | $Cin$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 6.4: Reverse adder chain truth table.

Note that addition is the only operation that requires an uncompute path. No intermediate values are computed during `andx`, `orx` or `xor`, so no uncomputation is necessary. The unaddition bitslice path has two parts: an unadd block to restore the $A$ and $Cout$ signals, and a set of inverters to delay the other operations' results appropriately.

In the uncomputation table, not all combinations of $A$, $B$, $Cout$, and $S$ are listed. Because $Cout$ and $S$ are themselves functions of $A$ and $B$, the cases for which $A$ and $Cin$ must be restored are limited. Specifically, the four input signals have only eight valid input combinations. Three output signals are therefore sufficient to uniquely determine each input combination. This is an unexpanding operation similar to the un-NAND operation from Section 4.2.2: a sparse, decoded representation is compressed into a more compact encoding.

The unadd computation may only proceed after the forward add has completed. Normal ripple carry add proceeds from the least significant bits to the most significant bits, generating and propagating carries from the the LSB to the MSB. The final carry out from the MSB is the first carry out to be uncomputed by the unadd chain. Uncomputation then proceeds back towards the LSB. By the time the unadd chain has uncomputed all the carry out values and the $A$ values, the only signals left are $B$, the sum, and the original carry in. For addition, this carry is a zero; for subtraction, it is a one. Constants are trivially

uncomputed.

As is common in reversible systems, a computation produces some set of intermediate values on the way to computing the desired final result. The intermediate values are then uncomputed, leaving only the result.

During both the forward computation and reverse uncomputation, the multi-bit $A$, $B$, and $S$ values are passed through the system. Each stage of the ripple carry add and unadd must have the proper bit signals input at the proper time. A delay chain for the operands and results must be constructed to feed the bitslices operand values and remove result values. The next section details this procedure.

## 6.3 The Wedge: Operand Distribution

In terms of silicon area, wire count, and transistor count, the operand distribution network is by far the largest component of the ALU. From a schematic point of view, it is just a tedious set of full inverters.

Figure 6-8 shows the first two bitslices of the ALU and the delay elements for the operands and results. At the start of the ALU, when the bitslice is computing the LSB sum of $Ain_0$ and $Bin_0$, the other $N-1$ bits of $Ain$ and $Bin$ are passed through a delay element, in this case a pair of inverters for each bit, one for each polarity. Recall that all signals going into the ALU bitslice are dual-rail. The first bitslice executes the expanding full adder operation, driving the sum bit on the output line $Zmid_0$ and carry $Cout_0$ as well as passing the $Ain_0$ and $Bin_0$ values through to the output, where they are renamed $Amid_0$ and $Bmid_0$. These output values are inputs to the lower set of delay elements. The next bitslice takes in $Ain_1$ and $Bin_1$ from the upper delay elements and produces $Zmid_1$ and $Cout_1$, passing $Amid_1$ and $Bmid_1$ through.

Each bitslice occupies a single time slice. SCRL timing constraints require that signals pass from one time slice to another through a logic gate. The inverters feeding the operand and result signals through the ALU serve to control the information flow from one time slice

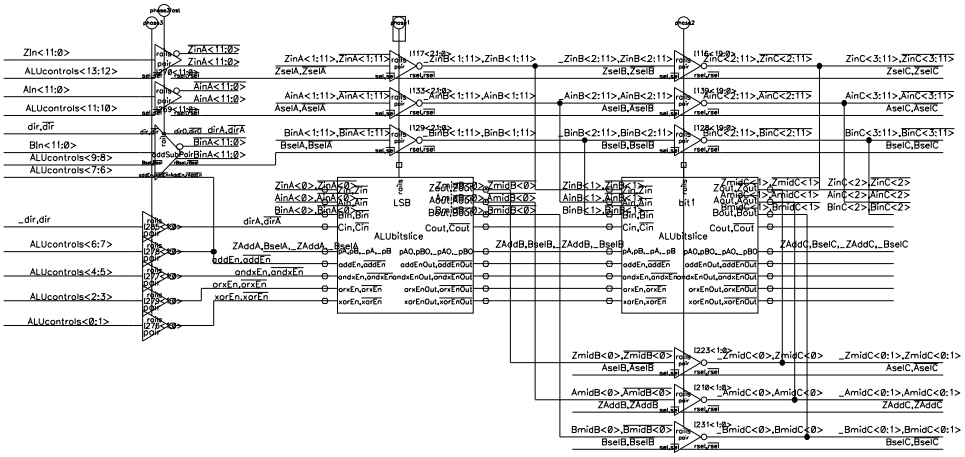


Figure 6-8: The first two bitslices from the Pendulum ALU.

to the next. It is useful to label each time slice with a different letter. The wires contain signals such as $A_{1A}$ and $S_{8B}$. As with a conventional vector, the number represents the bit's position in the vector. The subscript letter labels the vector's temporal position in the block. A bitslice module may add $A_{1B}$ and $B_{1B}$ to produce $Cout_{1C}$ and $S_{1C}$. The CAD tool used to produce Figure 6-8 denotes vectors using an angle-bracket notation, so signals in a time slice are labeled as AinA<0>, ZinB<1:11>, and so on. The labels Ain and Zin denote the signal name, the suffix A or B is the time slice, and the trailing <0> or <1:11> is the set of bits from the vector that are part of the labeled net. For example, as the signals AinA<1:11> and $\overline{\text{AinA<1:11>}}$ pass through a set of twenty-two inverters, the output is labeled as _AinB<1:11> and AinB<1:11>. The leading underscore is due to a limitation of the CAD tool and is equivalent to an overbar.

The operands entering the ALU are Ain, Bin, and Zin. The Zin operand is used in expanding operations such as `andx` and `sllx`, which perform $Zmid = (Ain \cdot Bin) \oplus Zin$ and $Zmid = (Ain \ll 1) \oplus Zin$, respectively. The outputs of the bitslices are Amid, Bmid, and Zmid. The "mid" denotes their position as the middle results of the ALU. The input values are obviously labeled "in," the outputs are labeled "out." The middle values are those that are produced by the forward carry chain and are used as inputs to the unadd chain.

The reason the operand distribution inverters are known as The Wedge[1] is that as the operands are used by the bitslices, fewer and fewer delay elements are needed for the input operands, and more and more delay elements are needed for the middle operands. The operand LSB's are used in the first bitslice, so $3 * (N - 1)$ full inverter pairs are needed for the set of dual-rail input operand values. The second bitslice uses bit 1 of each operand, so $3 * (N - 2)$ inverter pairs are in the upper delay chain. Meanwhile, the first bitslice has produced a result bit and passed the operands through. So the lower delay chain requires three inverter pairs. As the computation progresses, the number of delay elements above the bitslice is decreasing while those below are increasing. Each time slice has a total of $N$ delay gates.

When the computation reaches the unadd chain, the Amid operand values are being uncomputed at each unadd bitslice. The net number of delay elements is therefore decreasing. The middle values are passed through the bitslice and become the output values. The first two unadd bitslices are shown in Figure 6-9.

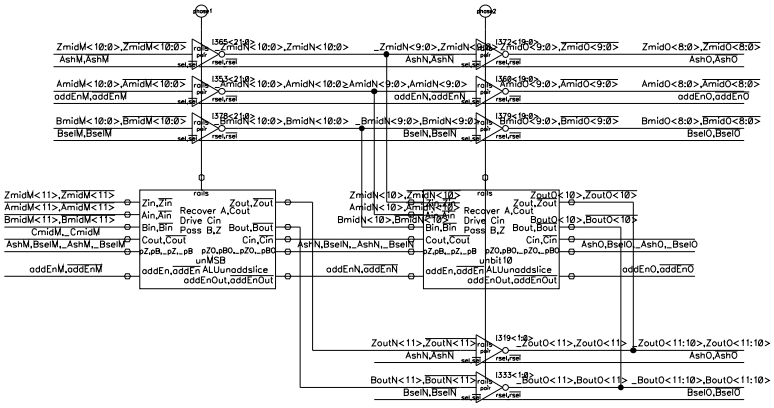

Figure 6-9: The first two bitslices from the unadd ripple carry add chain of the Pendulum ALU.

In layout, the forward addition upper wedge and lower wedge fit together to make a square array. The Wedges for each operand may be stacked on top of each other and the operand values and bitslice outputs fed up and down to and from the Wedges through wiring channels

[1]The term was coined by Athas and Svensson in 1994 [AS94].

between each bitslice and delay element. The uncomputed Amid values form a decreasing wedge and do not have a complementary increasing wedge. Figure 6-10 shows a block diagram of the increasing and decreasing wedges for the A, B, and Z operands.

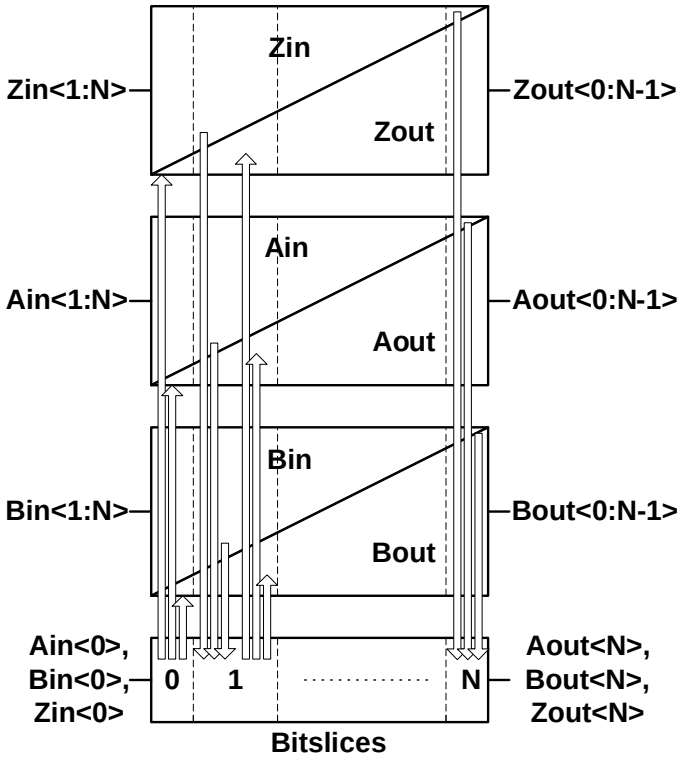


Figure 6-10: Block diagram of The Wedge layout. $N-1$ bits of three operands enter on the left side and $N-1$ bits of two operands and the result exit on the right side. Single bits of each operand are sent down to the bitslices at the bottom, and the output bits of each bitslice are sent up to the output wedge. The $0^{th}$ and $N^{th}$ bits of the input and output vectors enter and leave the bitslice directly. The silicon area consumed by each block is in similar scale to this diagram.

---

For all operations other than add, the unadd section of the ALU is disabled and the operands and results are simply fed through for retiming. Since expanding operations need to output the Ain operand unchanged, an extra feedthrough path for the Ain operand is needed. This path is separate from the Amid unadd wedge of Figure 6-10.

The Wedges consume a substantial fraction of the ALU chip area. The wires running up and down between the Wedges and the bitslices are long, on the order of 1 mm in the

Pendulum chip's 0.5 $\mu$m technology. Wire engineering and transistor sizing efforts are well spent here. Also, conventional CAD tools are ill-equipped for schematic design of such a structure. The Pendulum ALU schematic is a Byzantine construction with hundreds of wires, dozens of symbols, and thirty-two time slices. Some of the complexity is due to designer error, but much of the complexity is due to CAD tool constraints. However, even with all its complexity, the Pendulum ALU is still a highly regular structure, easily verifiable by automated simulation and verification tools.

Adding to the complexity of operand distribution is the integration of the shifter inputs and outputs with the bitsliced operations. The next section covers shifter design and how it fits in with the other ALU operations.

## 6.4 Shifter

The Pendulum shifter performs left and right logical shifts, right arithmetic shift, and left and right rotate. All shifts and rotates are by one bit position. The shift operations are followed by XOR, *e.g.*, producing $Z^+ = (A \gg 1) \oplus Z$. Rotates do not perform XOR. Shift operations take two operands while rotates are single operand operations. The shift and rotate operations require four time slices. The first logic stage conditionally computes the dual rail versions of the operands to be used, either $Ain$ or $Ain$ and $Zin$. Recall that rotates are one operand instructions while shifts perform a shift followed by a two operand XOR. A control signal is also computed in the first stage.

The second logic stage actually performs the shift or rotate. Figure 6-11 is the entire shifter schematic. The inputs to the XOR gates and the tri-state inverters are shifted or rotated appropriately. The control signals enable at most one of the five possible paths through the shifter.

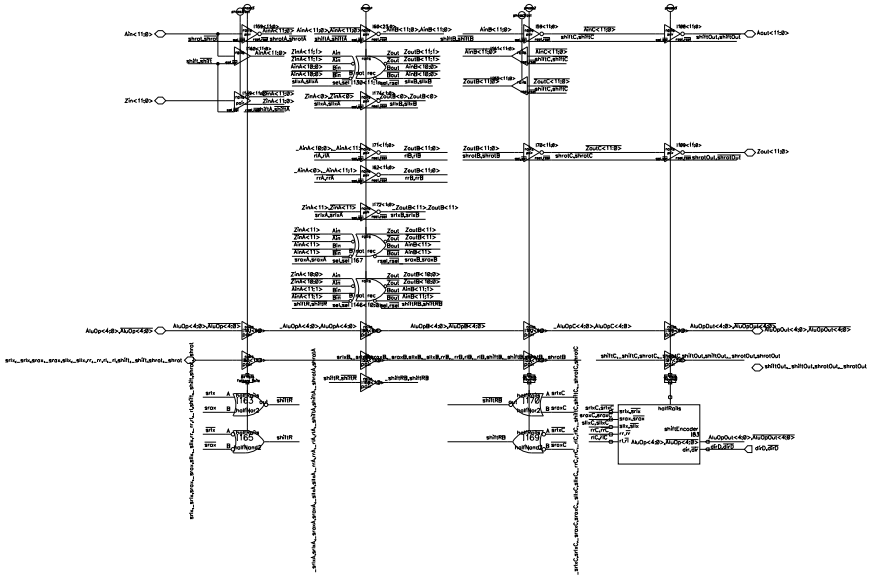

Figure 6-11: The Pendulum shifter schematic.

### 6.4.1 Rotation

Performing the actual single-bit rotate is very simple. Essentially the input operand wires are switched around and passed through a logic stage. Figure 6-12 shows how the rotates are performed. When the appropriate control signal enables one of the two tri-state inverters, the output value is rotated relative to the input bus ordering.

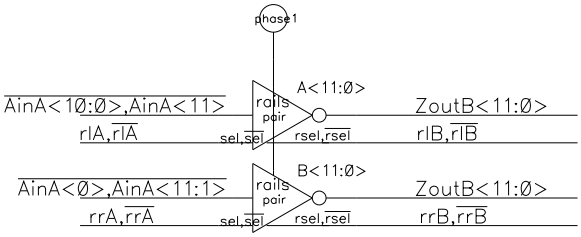


Figure 6-12: The two tri-state inverters that actually perform the rotation in the Pendulum shifter.

The difficulty associated with performing the rotate operations is that rotate is not its own inverse. The processor must know what direction it is running in to know whether to rotate

left or right. This complicates the control logic. If the processor is running forward, the rotate left opcode means the shifter should perform a left rotate. If the processor is running in reverse, a rotate left instruction means the shifter should perform an "un-left rotate," or a right rotate. The Pendulum datapath is so symmetric that very few blocks perform different operations based on the processor direction. The two exceptions in the ALU are rotates and addition. Just as one direction of rotate is not its own inverse, addition is not its own inverse, so the adder must know if it should perform add or "unadd," *i.e.,* subtract.

Figure 6-13 is the logic gate that generates the rotate left control signal. The rotate left and rotate right opcodes differ only in their LSBs, so the output is asserted when the processor direction bit is low, indicating forward operation, and the opcode is rotate left, or when the dirction bit is high and the opcode is rotate right.

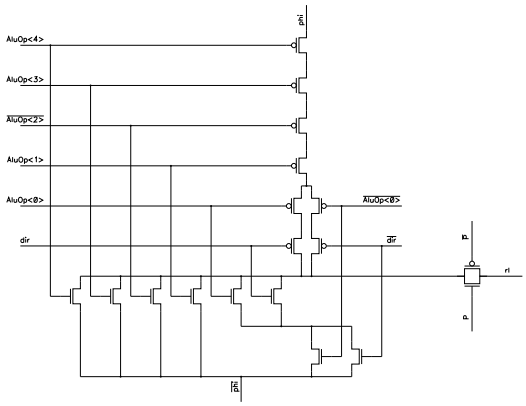


Figure 6-13: Left-rotation (lr) generation logic. Note that the lr signal depends on the processor direction.

This lesson is worth restating: an operation that is its own inverse may be controlled and implemented without using the direction bit. A reversible processor must contain some representation of its direction, but making as many operations as possible be their own inverses will reduce the use of that representation and simplify control signal generation and operand decoding. Using operations that are their own inverses may complicate those operations, such as using ANDX rather than AND, but that complexity is generally offset by the simplified control.

### 6.4.2 Shifts

By performing shifts followed by XOR, the expanding shift operations are their own inverses. Actual implementation of the shift operation is very similar to the rotation, save that instead of an inverter, the inputs to an XOR gate are wired to perform the single bit shift. Since shift is an expanding operation, the operand must be retained and used to restore one of the XOR gate's inputs. The shifter supports logical and arithmetic shifts by including support for generating a constant zero or for replicating the sign bit for the MSB to be XORed with the *Zin* value.

The paths through the rotate logic, shiftx logic, and delay elements, are all constructed of tri-state gates. Only the path for the operation performed is active at any time.

# Chapter 7

# SCRL Scorecard

This chapter summarizes the "take-home" messages of Part II. At the end of their first chapter, Weste and Eshraghian summarize the characteristics of CMOS circuits relative to other integrated circuit technologies, such as GaAs and bipolar devices. Again following their lead, the bulleted list below describes the main features of SCRL and compares SCRL to conventional CMOS.

- Fully restored logic levels; *i.e.,* output settles at $V_{dd}$ or ground.
- Transition times–the rise and fall times of a gate are set by the external power clock ramp slope. The system's energy dissipation is directly tied to the transition time.
- Memories are implemented with very low density relative to standard CMOS. A static RAM cell requires 25 times as many devices as a conventional 6-T SRAMcell.
- Transmission gates are used exclusively at the output of logic gates. They are not used on their own to perform computations.
- Power dissipation–Asymptotically zero dissipation. While standard CMOS, or any irreversible logic family, must dissipate a minimum quantity of energy for every bit erased, as well as dissipation due to "frictional" losses, SCRL circuits have no theoretical minimum dissipation and only incur frictional energy losses.
- Precharging characteristics–Node charging and discharging happens quasistatically, controlled by the external power clocks.

- Power supply–DC power and ground supplies are needed for well and substrate taps. Twenty to thirty additional external, AC power clocks control the flow of energy into and out of the circuit. Design of an efficient, energy-recovering power supply is an unsolved problem. This is a substantial stumbling block in the use of reversible systems.
- Packing density–Requires at least $2n + 2$ devices for $n$ input gates. Requires $2n + 8$ devices for tri-state gates. Also requires routing of eight power clocks and the two DC supplies to each gate. For gates of even low complexity, the chip area required for supplies is comparable to or smaller than the area required for transistors and local interconnect. As in conventional circuit design, clock, supply, and block-to-block signal routing consume the bulk of the chip area.
- Layout–Standard cell-style layout is possible using SCRL gates. The area penalty for SCRL vs. conventional CMOS circuits for a large system is approximately an order of magnitude.

The primary reason a designer would choose SCRL over standard CMOS is the asymptotically zero power dissipation of reversible circuits. As silicon technology continues to improve, the area penalty of reversible circuits, already less than about an order of magnitude, becomes less significant. The applications and motivations for low power/low energy computing engines are well known, and in certain contexts, the additional cost of silicon area may be negligible relative to other system costs. Also, just as solid state devices replaced vacuum tubes and high-performance CMOS circuits replaced bipolar devices, "slower" reversible circuits may eventually be faster than their irreversible counterparts because their lower energy dissipation allows them to be packed more densely without overheating the system [Fra99].

# Part III

# The Pendulum Reversible Processor

# Chapter 8

# Reversible Computer Architecture

This chapter discusses the engineering challenges of architecting a fully reversible processor, specifically the lessons learned in developing and implementing the Pendulum processor instruction set, detailed in Appendix A. As implemented in SCRL, the Pendulum processor contains approximately 200,000 transistors. It was fabricated in a 0.5 $\mu$m CMOS process. The chip has 180 pins, of which 30 are used for the swinging SCRL power clocks and two are for DC $V_{dd}$ and ground. Including the pads, the chip is approximately 6 mm by 8 mm. After the instruction set had been designed, schematic design, simulation, and chip layout required approximately two person-years. No automated layout generation or place and route software was used, although the regularity of SCRL gates suggests that place and route tools would be useful in the layout of future chips.

## 8.1 Why Design a Reversible Architecture

A fully reversible processor must implement a reversible instruction set in a reversible circuit implementation. A reversible instruction set is one in which both the previous and next instructions are known for each instruction in a correctly written reversible assembly language program. The processor may have varying degrees of protection guaranteeing reversible operation in the face of incorrectly written code, as discusssed in Section 8.5. The

dynamic instruction stream of a reversible processor may be executed both forwards and backwards. The instruction set described here is a modification [Fra97] of the one designed in Vieri's master's thesis [Vie95]. A detailed trace of a sample program, a Fibonacci number calculating routine, written in the Pendulum Assembly Language (PAL) is in Chapter 9.

Adiabatic operation requires that the circuits perform logically reversible operations. If one attempts to implement a conventional instruction set in a reversible logic family, reversibility will be broken at the circuit level when the instruction set specifies an irreversible operation. This break in reversibility translates to a required energy dissipation. To ensure reversibility at the circuit level, the instruction set must always be performing reversible operations.

## 8.2 The Pendulum Instruction Set

The Pendulum processor was originally inspired by the elegantly simple MIPS RISC architecture [KH92]. The register-to-register operations, fixed instruction length, and simple memory access instructions make it a good starting point for a radically different approach to instruction set design. Reversibility makes Pendulum novel enough; a more unusual architecture is covered in Chapter 10.

For ease of implementation, and of course to maintain reversibility, the instruction set has been substantially modified. It retains the general purpose register structure, fixed length instructions, and spartan instruction set principles. The instruction set includes conventional register to register operations such as addition and logical AND, shift and rotate operations, operations on immediate values such as add immediate and OR immediate, conditional branches such as branch on equal to zero and branch on less than zero, and a single memory access operation, exchange. The direction of the processor is changed using conditional branch-and-change-direction instructions.

The Pendulum datapath was developed in conjunction with the instruction set, first as a set of Verilog HDL modules which were eventually designed at the transistor level. The Verilog simulations allowed flexibility to experiment with changes in the instruction set. As the datapath modules were implemented at lower levels, nailing down the exact instruction

encoding was necessary, as were minor modifications to the instructions supported by the datapath.

### 8.2.1 Register to Register Operations

Conventional general purpose register processors read two operands, stored in two possibly but not necessarily different registers, and perform some operation to produce a result. The result may be stored either in the location of one of the operands, overwriting that operand, or some other location, overwriting whatever value was previously stored there. This produces two difficulties for a reversible processor. First, writing a result over a previously stored value is irreversible since the information stored there is lost. Second, mapping from the information space of two operands to the space of two operands and a result will quickly fill the available memory if values are not allowed to be overwritten. However, if the result is stored in the location originally used for one of the operands, the computation takes two operands as input and outputs one operand and a result. This space optimization is not required for reversibility if the processor can always store the result without overwriting some other value, but it is a useful convention for managing resources.

An astutely designed instruction set will inherently avoid producing "garbage" information while retaining as much flexibility and power for the programmer. This leads to the distinction between expanding and non-expanding operations. Both types of instruction are reversible; the distinction is made only in how much memory these instructions consume when executed.

All non-expanding two operand instructions in the Pendulum instruction set take the form:

$$R_{sd} \leftarrow \mathcal{F}(R_{sd}, R_s) \tag{8.1}$$

where $R_{sd}$ is the source of one operand and the destination for the result, and $R_s$ is the source of the second operand.

By contrast, logical AND is not reversible if only the result and one operand are retained. Except for the special case of the saved operand having every bit position set to one, the

second operand can not be recovered accurately and must be stored separately. Since operations like AND, including logical OR and shift operations require additional memory space after execution, they are termed expanding operations. The problem then arises of how store that extra information. If the two operands continue to be stored in their original location, the result must be stored in a new location. It is still not permissible for the result to overwrite a previously stored value, so the result may either be stored in a location that is known to be clear or combined with the previously stored value in a reversible way. The logical XOR operation is reversible, so the Pendulum processor stores the result by combining it in a logical XOR with the value stored in the destination register, forming ANDX, ORX and so on. Logical ANDX and all other expanding two operand instructions take the form:

$$R_d \leftarrow \mathcal{F}(R_s, R_t) \oplus P \tag{8.2}$$

where $R_s$ and $R_t$ are the two operand registers, $R_d$ is the destination register, and $P$ is the value originally stored in $R_d$. Note that if $P = 0$ then $R_d \leftarrow \mathcal{F}(R_s, R_t)$.

Constructing a datapath capable of executing an instruction stream forwards and backwards is simplified if instructions perform the same operation in both directions. Addition, which appears simple, is complicated by the fact that the ALU performs addition when executing in one direction and subtraction when reversing. The expanding operations are their own inverses, since

$$P = \mathcal{F}(R_s, R_t) \oplus R_d \tag{8.3}$$

Non-expanding operations could take the same form as the expanding operations, performing `addx` and so on, simplifying the ALU, but programming then becomes fairly difficult. Only expanding operations, which require additional storage space, are implemented to consume that space.

Converting a conventional register to register instruction execution scheme to reversible operation is relatively simple. The restriction on which registers can be operands is a minor inconvenience for the programmer.

### 8.2.2 Memory Access

Chapter 5 covers the issues in designing reversible memory. The challenge of adapting the XRAM memory architecture involved its conversion from a single-ported memory to a triple-ported register file. The solution chosen was to time-multiplex access to the memory.

Each operation may read up to three different operands from the memory. Instructions are required not to read any register file location more than once, *i.e.,* the $N$ operands used by an instruction must be read from $N$ distinct register file locations. The register file performs exchange operations, but since the processor reads values from the register file, performs some operation on those values, and writes values back to the register file, the exchange operation must be split into two temporally distinct parts.

When the operands for an instruction are read, an exchange is performed, but the value inserted in the register file locations is a constant zero. When the new values are written at the end of the instruction execution, the constant zeros are moved out of the register file and sent to a zero sink.

The multiple reads and writes to the register file that occur during instruction execution happen sequentially. If an instruction reads three operands, $A$, $B$, and $Z$, the enable and address values for $A$ are sent to the register file inputs while the enable and address values for $B$ and $Z$ are sent through one and two levels of delay elements, respectively. When the $A$ value has been read, the control signals for $B$ arrive at the register file inputs. The $A$ output value is delayed. When the $B$ output value has been read, the $Z$ control signals arrive at the register file inputs, the $A$ output value is again delayed, and the $B$ output value is delayed. When the $Z$ output value is read, all three output values, $A$, $B$, and $Z$, are sent on to the next section of the datapath. During each read, a constant zero is inserted into each register file location.

After the instruction execution stage has completed, the write operations happen similarly to the read operations. Each operand is written, pushing out the constant zeros stored during the read, one at a time while the other values are delayed. The address and control signals are appropriately delayed so as to exit the register file simultaneously.

The delay modules and wiring for the register file consume more silicon area than the memory array itself.

### 8.2.3 Control Flow Operations

Conditional branches are crucial to creating useful programs. The processor must be able to follow arbitrary loops, subroutine calls, and recursion during forward and reverse operation. A great deal of information is lost in conventional processors during branches, and adding structures to retain this information is very difficult.

Any instruction in a conventional machine implicitly or explicitly designates the next instruction in the program. Branch instructions specify if a branch is to be taken, and if so, what the target is. Non-branch instructions implicitly specify the instruction at the next instruction memory address location. To follow a series of sequential instructions backwards is trivial; merely decrement the program counter rather than incrementing it. Following a series of arbitrary jumps and branches backwards in a traditional processor is impossible: the information necessary to follow a jump or branch backwards is lost when the branch is taken. A reversible computer must store enough information, either explicitly in the instruction stream or elsewhere, to retrace program execution backwards.

The literature contains a number of techniques for performing jumps and branches reversibly that differ from the scheme presented here [Fra97, Vie95, Hal94]. The discussion below refers only to the particular scheme used in the Pendulum processor.

Pendulum branch instructions specify the condition to be evaluated, either equal to zero or less than zero, the register containing the value to be evaluated, and a register containing the target address. The instruction at the target address must be able to point back to the branch address and know if the branch was taken or if the target location was reached through sequential operation. For proper operation, each branch instruction must target an identical copy of itself.

When a branch condition is true, an internal branch bit is toggled. If the branch bit is initially false and the branch condition is true, the program counter update (PCU) unit

exchanges the value of the program counter and the target address, and then the branch bit is toggled. The target address must hold an identical branch instruction which toggles the branch bit back to zero, and sequential operation resumes. The address of the first branch instruction is stored in the register file so that during reverse operation the branch can be executed properly in reverse.

## 8.3 Instruction Fetch and Decode

Reading from the instruction memory has the same constraints as reading from the data memory or register file. Each copy created when an instruction is read must be "uncopied." If instruction fetch operations are performed by moving instructions rather than copying them, they must be returned to the instruction memory when the instruction has finished executing.

After the instruction is read (or moved) from the instruction memory, the opcode is decoded and a number of datapath control signals are generated. Just before the instruction is moved back to the instruction memory, these control signals must be "ungenerated" by encoding the instruction.

A certain symmetry is therefore enforced with respect to instruction fetch and decode. An instruction is moved from the instruction memory to the instruction decode unit. The resulting datapath control signals direct operation of the execution and memory access units. After execution, the control signals are used to restore the original instruction encoding, and the instruction may then be returned to the instruction memory.

The processor must be able to return the instruction to its original location, so its address must be passed through the datapath and made available at the end of instruction execution. Since the address of the next instruction must also be available at the end of instruction execution, the Pendulum processor has two instruction address paths. One path contains the address of the instruction being executed, and the program counter update unit uses it to compute the address of the next instruction. The second path contains the address of the previous instruction and the PCU uses it to compute the address of the current instruction.

The output of the PCU is then the next instruction address and the current instruction address, which are the values required to return the current instruction to the instruction memory and read out the next instruction.

The processor has two bits to keep track of what type of control flow operations are being performed, a "branch" bit and a "flip" bit. The branch bit indicates if the processor is in the middle of a branch, *i.e.,* if a branch conditional evaluated to true. The flip bit is also asserted if the branch conditional of a direction changing branch instruction evaluates to true. The two tables below indicate the action the PCU takes to update the two program counter values for both direction changing branch instructions and other types of instructions. Remember that during branches, the PC value is exchanged with a register file value, denoted rb in the tables below; rb' is the value written back to the register file during a branch instruction. Br is the branch bit state when instruction execution begins. Sel is the evaluation of the branch conditional. The PC' column indicates how the previous PC value (PPC) is updated to become the current PC, and the NextPC column indicates how the current PC value is updated to compute the address of the next instruction. The previous PC and PC are incremented by the value d, representing the direction bit. If the processor is running forward, d = 1; if reverse, d = -1.

| Beginning State | | Next State | | | |
|---|---|---|---|---|---|
| Br | Sel | Br' | PC' | NextPC | rb' |
| 0 | 0 | 0 | PPC + d | PC + d | rb |
| 0 | 1 | 1 | PPC + d | rb | PC |
| 1 | 0 | disallowed | | | |
| 1 | 1 | 0 | rb | PC + d | rb |

Table 8.1: Current PC and Previous PC update behavior for non-direction-changing instructions.

Performing these computations during branching instructions is very difficult, especially when the processor is changing direction. Traditional processors throw away every instruction executed, and ensuring that the instructions are returned to the instruction memory is difficult.

| Beginning State | | | Next State | | | | | |
|---|---|---|---|---|---|---|---|---|
| Br | Sel | Flip | Br' | Flip' | d' | PC' | NextPC | rb' |
| 0 | 0 | 0 | 0 | 0 | d | PPC + d | PC + d | rb |
| 0 | 0 | 1 | | | | disallowed | | |
| 0 | 1 | 0 | 1 | 1 | $\overline{\text{d}}$ | PPC + d | rb | PC |
| 0 | 1 | 1 | | | | disallowed | | |
| 1 | 0 | 0 | | | | disallowed | | |
| 1 | 0 | 1 | | | | disallowed | | |
| 1 | 1 | 0 | 0 | 0 | d | rb | PC + d | rb |
| 1 | 1 | 1 | 0 | 0 | d | rb | PC + d | rb |

Table 8.2: Current PC and Previous PC update behavior for direction-changing branch instructions.

## 8.4 Distributed Instruction Encoding and Decoding

The Pendulum processor uses a technique of "just in time" decoding and encoding of the instruction into control signals. Because of the high cost of passing signals through the datapath delay elements, reversible logic encourages the designer to pass as compact a representation of the control signals as possible through the datapath. The compact representation should only be decoded at the last possible moment, expanding the encoded signals into the desired set of decoded signals. After the control signals have done their duty, the complementary unexpansion should be performed.

The Pendulum processor has an explicit instruction decoder and encoder, but these blocks perform only a fraction of the signal decoding and encoding in the processor. The ALU contains a number of purpose-build encoding and decoding blocks, as does the program counter update unit and the register file. One block, primarily used to delay control signals while the register file reads are occurring, also performs a substantial amount of control signal encoding and decoding. This encoding and decoding breakdown reduces the number of wires needed to distribute the control signals but increases the number of special logic blocks required.

## 8.5 Reversibility Protection in Hardware

The balance between hardware control and software control of various aspects of the reversible processor is frequently problematic. Early designs using a hardware-controlled garbage stack were found to be inefficient and abandoned for more programmer-accessible garbage management. Likewise, the instruction set itself may be simple, flexible, and reversible only if programs are carefully constructed. Or, the hardware may incorporate some level of protection for reversibility. For example, any instruction set that requires branch instructions to be paired with programmer-inserted reverse branch instructions is susceptible to broken reversibility if the reverse branch is left out or incorrect. Also, storing values in the register file may be irreversible unless either the programmer is careful to only store to clear locations or "stores" are actually an XOR of the previously stored value and the new value.

If conditionally expanding instructions, such as overflow during addition, are supported, the additional information may be difficult to handle in hardware, so guaranteeing reversibility in the face of such exceptions may be left to software. While this may compromise the overall reversibility of the processor, a design may not make any claims regarding guaranteed reversibility for any program, only that a properly written program will be fully reversible.

## 8.6 Conclusion

All the primary blocks of a traditional RISC processor erase bits, and retaining those bits presents varying levels of difficulty to the designer. Making control flow operations reversible is difficult, and a reversible adder is surprisingly difficult. Distributing the control signal encoding and decoding is a helpful design tool for simplifying datapath wiring. When actually implementing a reversible instruction set, frequently the "devil is in the details." Generating and sinking constants, matching pipeline delays, constructing shared busses, and so on, require most of the designers effort.

These challenges present the opportunity to reexamine conventional RISC architecture in

terms of bit erasure during operation. Register to register operations and memory access are relatively easy to convert to reversibility, but control flow and, surprisingly, instruction fetch and decode, are decidedly non-trivial. This knowledge may be used in traditional processor design to target datapath blocks for energy dissipation reduction.

# Chapter 9

# Detailed Code Trace

This chapter follows the execution of a small program written in Pendulum Assembly Language, PAL. The program computes Fibonacci numbers using an iterative algorithm. The program is compact enough that it does not need to store values in the external memory; all operands fit in the eight local registers. The PAL code is shown below. The `start` and `finish` instructions are not implemented in the Pendulum processor but were useful for simulation purposes.

```
main:   start                   ; Compute Nth Fibonacci number

        addi $1 1               ; $0=0, $1=1, base case
        addi $2 18              ; hold N, loop counter
        neg $2                  ; shorthand for (xori $2 0x1ff) + 1
        addi $2 1               ; setup $2 for loop count
        addi $3 bottom          ; loop target is top; swapped with bottom
        addi $4 t1              ; entry point
        addi $5 exit            ; exit address
call:   bez $4 $7               ; call fib routine

        rbez $5, $7             ; return from rcall
        addi $5 -1
rexit:  bez $4 $7

top:    bltz $3 $2              ; paired loop branch, could be J
t1:     bez  $4 $7
```

```
        addi $7 1
        add $0 $1
        xor $0 $1               ; swap $0, $1
        xor $1 $0
        xor $0 $1
        addi $2 1               ; increment counter
        addi $3 -10             ; correct for exchange
b1:     bez $5 $2               ; exit condition
bottom: bltz $3 $2              ; loop until done

exit:   bez $5 $2               ; exit point. top and bottom could
                                ; have an unconditional jump
;;; Copy output and call the fib generator in reverse.
        add $6 $1               ; Result to $6
        addi $4 3               ; $4 points to call, adjust to rexit
rcall:  rbez $5 $2              ; call backwards; goto b1
        bez $5 $7               ; return from rcall
;; reclaim other space
;; we know $0=0, $1=1, $2=-17, $3=bottom,
           $4=t1, $5=call+1, $6=fib(18), $7=0
        ;; clear out those values
        addi $1 -1
        addi $2 17
        addi $3 -22
        addi $4 -13
        addi $5 -9

end:    finish
```

The algorithm overview is as follows. A few constants are set, including Fib(0) and Fib(1). The inner loop is called unconditionally. The inner loop increments a loop counter, adds Fib(i) and Fib(i-1), storing Fib(i+1) in the register previously containing Fib(i-1). The two register locations are then swapped. Another loop counter is incremented, and the target address for the top of the inner loop is calculated. This calculation is needed because of the way Pendulum executes branch instructions and is explained in detail below. An exit condition is tested, and if it fails, the PC jumps to the top of the loop. When the exit contition is true, the program leaves the inner loop, and copies the result to an unused register.

When exiting the loop, the program has computed Fib(N) but also contains the ending

values of the loop counters and Fib(N-1). These unwanted intermediate values may now be "cleaned up." The loop counter values are known at compile time, but Fib(N-1) must be computed by calling the routine in reverse. After the uncomputation, the loop counters and other constants are cleared.

The computation is set up between the instructions at `main` and `call`. All registers are assumed to contain zero at the start of the program. Registers $0 and $1 are allocated to Fib(i-1) and Fib(i), respectively. The base case for the iterative algorithm has Fib(0) = 0 in $0 and Fib(1) = 1 in $1. Register $2 contains N, the input to the program. In this case it is hardcoded to be 18, since Fib(18) is the largest Fibonacci nubmer expressible in twelve bits. It is negated with the `neg` instruction, which is actually an assembler macro for an XOR immediate with all ones followed by add one, and serves as a loop counter. Register $2 is incremented by one because the counter counts from -17 to 0, not -18 to 1. During program execution, register $3 will alternately contain `top` and `bottom`; it is initially set to `bottom`. Register $4 alternately contains `call` and the loop entry point, `t1`. Register $5 is initialized to the loop exit address, `exit`. Register $7 is used as a reverse loop counter and starts at zero.

The register allocation is:

- $0 Fib(N-1)
- $1 Fib(N)
- $2 forward loop counter
- $3 loop address register
- $4 loop entry address
- $5 loop exit address
- $6 result register
- $7 reverse loop counter

The instruction labeled `call` is a jump, since $7 contains zero at this point, and the program jumps to the address specified in $4, `t1`. The PC value `call` is exchanged with the contents of $4. At `t1`, the `bez` instruction toggles the processor branch bit back to zero. The reverse

loop counter in $7 is incremented from zero to one. Registers $0 and $1 are added and stored in $0, computing Fib(2). Registers $0 and $1 are swapped using three XOR instructions, so $0 contains Fib(1) and $1 contains Fib(2). The forward loop counter $2 is incremented by one, from -17 to -16. Register $3, containing the address `bottom`, is corrected for the offset between `bottom` and `top`, and now contains `top`. The version of the assembler used with this code was not capable of taking an immediate value of the form `bottom - top`, so that calculation was made by the programmer and included as a constant.

The exit condition at `b1`, `bez $5 $2`, evaluates to false, so the loop is called again by the instruction at `bottom`. Register $3 and the PC exchange values, jumping to `top` and putting `bottom` in $3. The paired conditional branch at `top` toggles the branch bit back to zero. The entry condition at `t1` is no longer true, and the loop executes again, incrementing the reverse loop counter in $7, computing Fib(3) and exchanging the values of $0 and $1, incrementing the forward loop counter $2, correcting the loop offset in $3, and testing the exit and loop conditions again.

Registers $3, $4, and $5, are used for control flow and contain branch targets. Pendulum processor control flow is discussed in Section 8.2.3. The values stored in these registers are exchanged with the program counter when the branch condition evaluates to true. After the exchange, the registers contain the "come-from" address.

Once Fib(18) has been computed, the forward loop counter in $2 has been incremented from -18 to zero. The exit condition at `b1` evaluates to true, so the values of the PC and $5 are exchanged. The program branches to `exit` and `b1` is stored in $5. The paired `bez` instruction at `exit` toggles the branch bit back to zero. Fib(18), stored in $1, is copied to the previously-empty $6.

Register $0 now contains Fib(17), so the inner loop is called in reverse to uncompute that value. Register $4 still contains the address `call` from the original loop entry. Adding three, or `rexit - call`, allows $4 to be used for the reverse exit location. Since the loop exited, $2 must be zero, and it is used for the reverse call condition. Register $5 contains `b1` since the exit condition evaluated true. In reverse, starting from `b1`, the inner loop uncomputes each Fibonacci number and loop counter. Register $7 is decremented from seventeen to zero,

and when all the Fibonacci numbers have been uncomputed, the reverse exit condition at `t1` evaluates true and the program branches to `rexit`. Register $5 is changed from `rcall` to `rcall + 1` by performing a reverse add of -1, and the `rbez` instruction below `call` puts the processor back into forward operation and branches to the instruction below `rcall`. At this point all the registers other than $6 have been restored to their original values, so the bottom five `addi` instructions clean up the remaining constants: Fib(1) in $1, N-1 in $2, `bottom` in $3, `t1` in $4, and `call + 1` in $5. Had this been called as a subroutine of another program, N would not be known at compile time, and the output of the routine would be N and Fib(N).

At `end`, all registers contain zero except for $6, which contains Fib(18), or 2584.

# Part IV

# Open Questions

# Chapter 10

# Other Reversible Computer Architectures

One of the early test chips built using the design methodology from Part II was an implementation of Margolus's Billiard Ball Model Cellular Automaton (BBMCA) [Mar87, FVA+98]. The chip, known as Flattop, is not a very convenient computer to program, but it is simple, universal, reversible, and scalable. Though the BBMCA model itself is only a two-dimensional cellular automaton, Flattop could in principle be wired in a 3-D mesh as well, for scalably executing reversible 3-D algorithms.

The following sections describe the BBMCA and the Flattop circuit design. Flattop is named after the local pool hall, Flattop Johnny's. The Flattop design is obviously substantially different from a conventional RISC processor, but it is a universal, reversible computing engine. Future reversible computers may lie somewhere in the design space between Flattop and Pendulum, not as conventional[1] as the RISC-like Pendulum, but not as unusual as Flattop.

[1] Only in relation to something as odd as a BBMCA could Pendulum be described as "conventional!"

## 10.1 The BBMCA

In the billiard-ball model (BBM) of computation, logic values are modeled by the presence or absence of "billiard balls" moving along predetermined paths in a grid. These classical, elastic, hard spheres are constrained to travel at constant speed along straight lines. All the balls in the system are synchronized so that all collisions happen at well defined grid points. A logic gate is a point where two potential ball trajectories interact. Fixed walls may be introduced to bounce signals around. Figure 10-1 shows two possible BBM logic gates. The rectangles are fixed walls. The output is represented by the presence or absence of a ball at the designated output points of the gate. Any reversible logic circuit can be implemented in this model; it can be as universal as NAND.

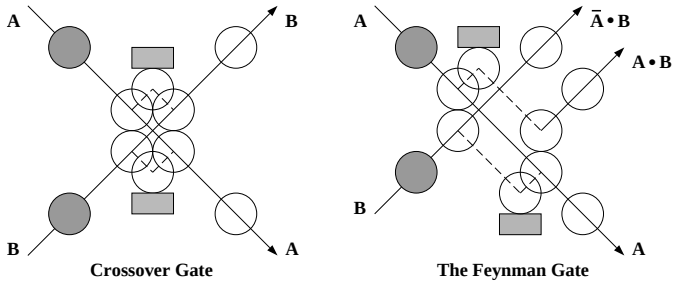


Figure 10-1: Two examples of BBM logic gates.

The BBM cellular automaton is a universal, reversible computing element based on these interactions of hard spheres. Each cell is a four input, four output logic block. Balls that enter the block alone continue along their original path, out the other side of the block. If exactly two balls enter opposite corners of the block, they "bounce" off each other and exit the block through the other two corners. For example, two balls entering on the left and right of a block bounce and leave through the top and bottom of the block.

An electronic circuit implementation of the BBM cellular automaton simulates the billiard ball model by representing balls and walls using high voltage values and represents empty space using low voltage values. The logic blocks are arranged in a grid with nearest neighbor communications. Because of the timing of the signals moving through the cells, the grid

behaves as though it were actually two grids, offset from and overlapping each other. So the grid is effectively subdivided into 2×2 blocks, in two different overlapping meshes, offset from each other by one cell in each of the x and y directions, as in Figure 10-2 (a). These two meshes reversibly transform each of their 2×2 blocks.

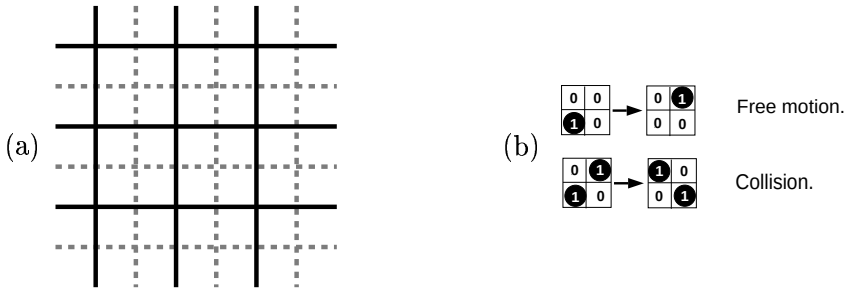


Figure 10-2: Billiard ball model meshes (a) and update rules (b).

On each cycle, the contents of each block in the array are modified according to the simple reversible transformation mentioned above. Figure 10-2 (b) shows the BBMCA update rules. The rules are applied independent of block orientation. All multiple-ball input configurations other than those shown result in the input being passed to the output. Essentially, if balls enter in a configuration other than the two shown, they bounce off each other and return the way they came in. Fixed features can be constructed out of several adjacent bits that are all one. Bits that are isolated move diagonally through the array.

The CA is implemented by placing an SCRL processing element at the center of every 2x2 block on each of the two alignments. The processor takes its inputs from the block's four cells (which are the wires coming from the processors in the centers of the four overlapping blocks) and produces its outputs in those same cells, but shifted to an alternate bit-plane (another set of four wires going back out to the neighboring processors). See Figure 10-3. The wires going off the edges of the array can be folded back into the array to connect the two bit-planes, or fed out to neighboring chips in a larger array.

A computational structure can be created by initializing the array with appropriate initial conditions. When set into motion by clocking the circuits, the signals propagate according

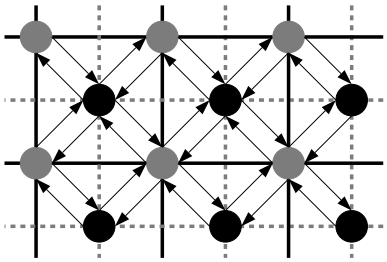

Figure 10-3: The 2×2 blocks of cells are updated by processing elements (circles) at the center of each block. The state of a cell is stored on a wire (arrow) connecting two PEs.

to the rules of the CA and perform any desired computation. Programming the initial conditions is somewhat difficult and is not treated here.

## 10.2 Circuit Design

This section describes the design of the Flattop circuitry and gives examples of some of the more important components. Since this design was an early exercise, some of the notational and methodological advancements are not used here.

The processing element consists of three levels of SCRL gates. The inputs to level one are $A, B, C$, and $D$, and the outputs of each level are shown below.

- Level 1: $\overline{A}$, $\overline{B}$, $\overline{C}$, $\overline{D}$, and $\overline{S} = \overline{(A+C)(B+D)}$.
- Level 2: $A$, $B$, $C$, $D$, $S$, $\overline{S}$, and $\overline{A_{out}} = \overline{SA + \overline{S}\overline{A}(C+BD)}$, and similarly for $B_{out}$, $C_{out}$, and $D_{out}$.
- Level 3: $A_{out}$, $B_{out}$, $C_{out}$, and $D_{out}$.

In this logic, level one is mainly a buffer but also generates the $S$ signal used in level two. Level two computes the update function, and level three is another buffer. Level two must produce $S$ at its output so that $S$ will be available for use by the reverse half of level two to compute the inverse update function.

### 10.2.1 Block Diagram

Figure 10-4 shows a schematic block diagram of a single processing element. Note the three blocks, one for each of the three levels in the 3-phase SCRL pipeline. Each level shown below contains both the forward and reverse SCRL circuitry.

Lollipop notation is used in these schematics for rail connections. The cell has four signal inputs: A, B, C, and D, which come from the four neighboring cells, and the cell has four corresponding outputs which go back out to those cells.

The SHIFT in and out signals are global signals shared by all cells; they tell the cells whether to operate in initialization mode or normal mode. In initialization mode the array of cells behaves as a shift register, and the array contents may be shifted in and out; in normal mode, the array obeys the BBMCA update rules.

The boxes attached to the input are for setting initial conditions during HSPICE simulation of the circuit. In layout, each cell measures approximately 170 $\mu$m $\times$ 90 $\mu$m.

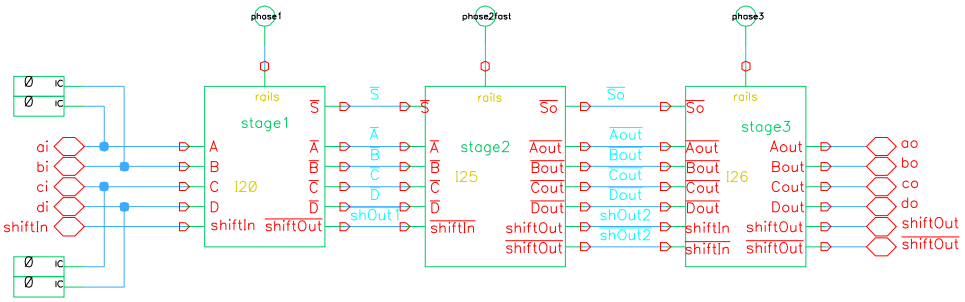


Figure 10-4: Block diagram of PE cell.

Figure 10-5 shows the schematic icon that represents an entire processing element. The icon portrays the 2×2 block of BBMCA cells which the PE is updating, with the PE inputs and outputs placed in the appropriate cells. The cell grid is rotated 45 degrees from the representation in Figure 10-3 to show how the CA array is oriented with respect to the edges of the chip. With this orientation, the array of processing elements can communicate along pathways that run parallel to the chip edges, making layout easier.

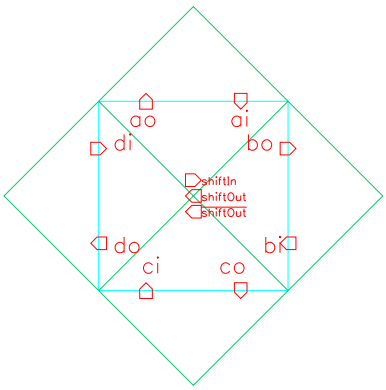


Figure 10-5: Icon for a single PE cell.

Figure 10-6 shows one corner of an array of these processing elements. In the upper left corners are pathways used for initialization and reading out the whole array when used as a shift-register. Along the edges of the array are edge cells which provide connections to pins, allowing chip to chip communication during normal operation. There are not enough pins to allow communication everywhere along the edge, so in other places the wires at the edge loop around to feed back into the array. Every PE receives the global shift signals, which run horizontally.

### 10.2.2 Detailed Schematics

Below are the schematics for the logic in the three levels of each cell. Figure 10-7 shows the logic in level one which computes the inverses of the inputs, together with the inverse of the S signal used in level two. Note that S is turned on when in shift mode. This logic supports array initialization.

Figure 10-8(a) shows the gate used for computing each second-level output. The forward part of level two includes four instances of this gate, differing as to which inputs are fed to the $A, B, C$, and $D$ pins, plus six "fast" inverters for generating the $A, B, C, D, S$, and $shift$ signals from their inverses using the $f\phi_2$ and $\overline{f\phi_2}$ rails, plus one other inverter for generating $\overline{S}$ on the level two output.

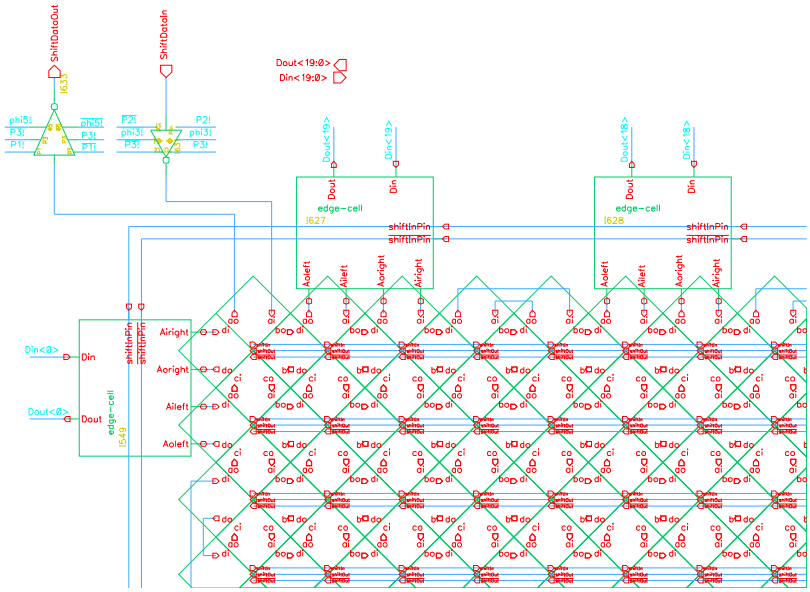


Figure 10-6: One corner of an array of processing elements.

Finally, Figure 10-8(b) shows the gate that is repeated four times (with different pin assignments) in the forward part of stage three. This gate selects either $A$ or the bit in the opposite corner of the block for passing through to the output, depending on $sh$. If $sh$ is on, input bits are passed to the opposite output bits, enabling the array as a whole to act as one large shift register, which can be used to initialize the array contents.

## 10.3 Conclusion

This chapter describes the circuit design for a reversible, universal computer that is significantly different from the Pendulum reversible processor. The architecture is parallel and scalable to arbitrarily large arrays, assuming power supply inputs are repeated periodically. Global timing skew is not an issue since all data interconnections are local. Future reversible computer architecture designs may lie somewhere in the designn space between this compute engine and the Pendulum processor.

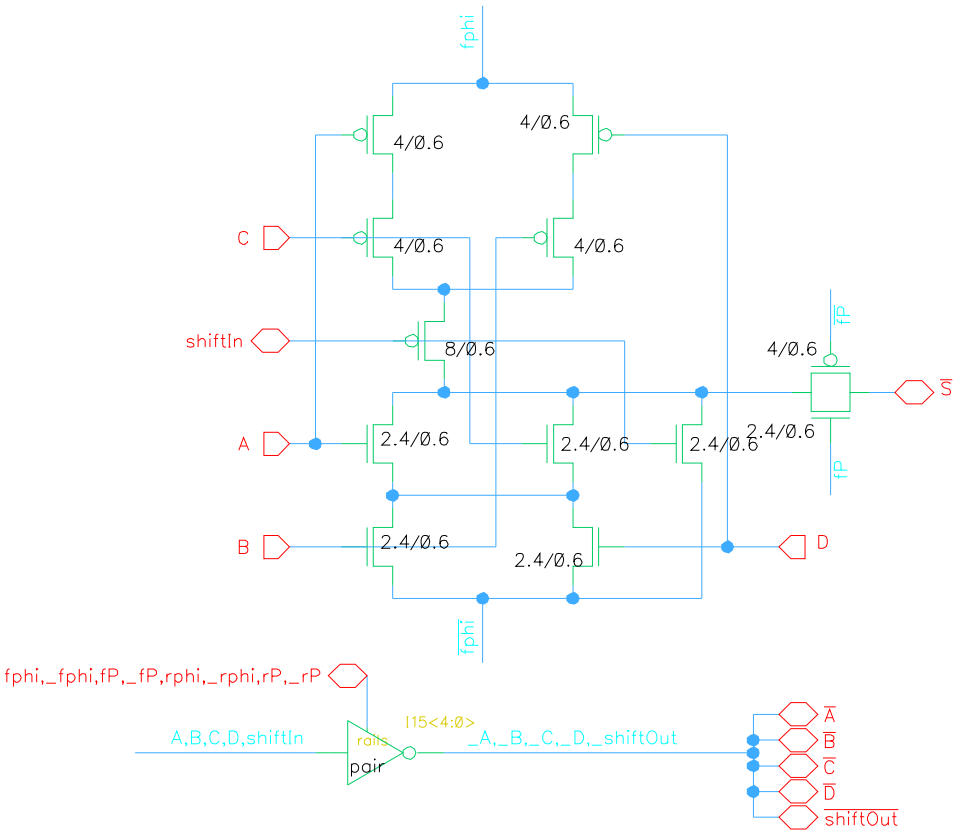


Figure 10-7: Level one logic, $\overline{S} = \overline{sh + (A + C)(B + D)}$.

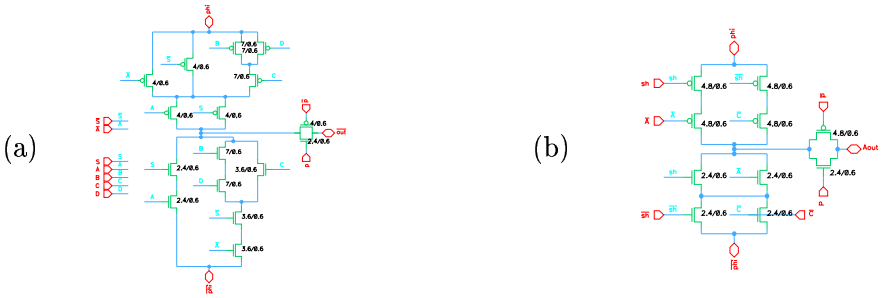


Figure 10-8: Gates that computes (a) $\overline{A_{out}}$ in level two and (b) $A' = \overline{\overline{sh}\ \overline{A} + sh\ \overline{C}}$ in stage three.

# Chapter 11

# Future Work

The Pendulum processor and design style presented here are a significant demonstration of the possibilities of reversible logic design. The design process has unearthed a number of opportunities for further study, including improved architectures, power supply design, other logic families, logic complexity costs of reversible logic design and reversible logic minimization, and software applications.

## 11.1 Other Architectures

The Pendulum processor architecture was chosen because it is a simple instruction set, and it is similar to conventional machines. Mapping these somewhat conventional instructions onto reversible logic is perhaps more difficult than necessary. A more inherently reversible architecture, perhaps similar to the Flattop BBMCA, may map more simply to reversible logic.

And while other architectures are being explored, the logic family used to implement them may be modified. SCRL is most likely not the optimal reversible logic family, but much of the notation and methodology developed in this dissertation is applicable to other logic families and switching element technologies. For example, it is possible to build a fully-reversible six-clock-rail shift register. Using fewer clocks may be possible. A stronger theoretical basis

for the number of clocks required for full reversibility in various applications could be useful.

## 11.2 Power Supplies

Much discussion in the adiabatic computing literature is devoted to the generation of the gradually swinging rails needed for various adiabatic logic families. While this dissertation has assumed the existence of the needed supply rails, development of an appropriate energy-recycling, constant-current power supply is still an open question. The ISI "blip circuit" [AST96], stepwise capacitor charging [SK94], and MIT's transmission line clock drivers [BK98], are steps toward this end.

## 11.3 Reversible Computer Analysis

This dissertation has described the synthesis steps needed to build a reversible computer. The analysis of such a machine, an exploration of the tradeoffs between erasing bits and maintaining full reversibility, is needed if reversible computing is to be a practical tool for designing compute engines. It would be useful to study conventional machines further to determine a hierarchy of "hot spots" of energy dissipation due to bit erasure. Energy dissipation could be reduced by selective application of reversible techniques to the components that both erase a large number of bits and that may easily be converted to reversible operation.

## 11.4 Reversible Hardware Theory

This dissertation has occasionally mentioned the engineering tradeoffs involved in instruction set design relating to ensuring that the instruction set is "guaranteed" to be reversible. In cases such as whether or not branch instructions are required to have a paired reverse branch instruction, no particular proof is given and the instruction set merely appears to be

"more" reversible in the sense that fewer demands are put on the compiler or programmer to generate code that will run forwards and backwards properly.

This poses the interesting question of how to rigorously prove or guarantee that an arbitrary stream of bits will be executed as a fully reversible program. At a lower level, it is also interesting to ask how to prove that a set of circuits will perform microscopically reversible operations on arbitrary bit strings. For an SCRL block, containing a forward and reverse stage, the inputs and outputs must conform to the SCRL rules in order to maintain reversibility. For a simple block, reversibility may be proven by exhaustively testing the set of inputs and outputs that satisfy the rules. Guaranteeing that a group of such blocks will still be reversible when wired together relies on the fact that each block will generate as output a set of signals that respect the SCRL rules. In general, however, it is difficult to guarantee that a large system composed of smaller blocks is reversible unless each block is known to be reversible and each interface between blocks follows SCRL rules. This is hardly a rigorous testing scheme, and a more thorough or general technique would be useful in designing reversible circuits.

## 11.5 Reversible Adder Logic Minimization

The number of inverters needed for operand distribution in The Wedge suggests that some adder other than a straightforward ripple carry might require less logic. Techniques traditionally used to speed up addition may be useful in reducing the quantity of logic required by a reversible adder. I first considered performing addition two bits at a time. The logic seemed too complex at the time, so I designed the relatively simple ripple carry adder. After seeing how big The Wedge layout actually is, multi-bit addition logic may make more sense. The trouble is that unless the entire $N$ bit addition is performed with one big non-expanding operation, the adder must have an unadd path to uncompute the carry bits. Since the Pendulum ALU and PCU adders required the most area on the chip, improvements to the adders may save substantial silicon area.

## 11.6 Software

The schematic design and layout for XRAM, Flattop, and Pendulum were all done by hand. The structures involved exhibit substantial regularity and automatic generation may be possible. Most conventional integrated circuit designs are automated to some degree, and these automation techniques seem applicable to reversible logic design. The regularity of SCRL layout especially resembles conventional standard cell layout.

Also, at a higher level, reversible compiler technology is very young. An assembler was developed for the Pendulum processor, and Frank's work [Fra99] suggests some directions for compiler design, but more work is clearly needed.

# Chapter 12

# Conclusion

Recall my thesis from the introduction:

> There exists a set of reversible primitive modules that may be combined according to a set of simple rules to create larger systems, which are themselves reversible primitive modules. These modules may be combined to create a fully reversible, universal computer.

I have shown in this dissertation the rules for creating primitive modules and connecting them together. I developed a notation for reversible logic schematics. I have demonstrated this thesis buy designing, fabricating, and testing a fully reversible microprocessor.

Building reversible systems using this methodology uses resources which are comparable to a conventional microprocessor. Existing CAD tools and silicon technology are sufficient for reversible computer design. The layout was performed by hand, without automated routing or layout synthesis, although the regularity of SCRL layout suggests that automation may be used in the future. Power clock routing is regular and does not consume unreasonable chip area. Logic layout is also regular and comparable to conventional layout. The 12-bit Pendulum processor consumed less than a square centimeter of silicon in a no-longer-modern process technology.

On the architectural side, I have shown the issues involved in saving all the information in a microprocessor. The way conventional programs are written is oriented very much towards throwing away information, but it is possible to avoid this information erasure.

SCRL CMOS logic is probably not going to replace conventional CMOS logic in high performance systems in the near term, but this work has demonstrated that building a reversible system is not only possible, but can be performed systematically and on a substantial scale.

# Appendix A

# The Pendulum Instruction Set Architecture (PISA)

Assembly language programming on the Pendulum processor resembles a cross between a standard RISC architecture [KH92] and a PDP-8, with a few modifications to support reversibility. The Pendulum instruction set is decidedly "RISC-y". Instructions are not very powerful but are deliberately simple so that the programmer can keep track of the information flow. The programmer must be aware of a few constraints. First, memory accesses are always an exchange. To copy a value of a register, ensure that one register is clear (by exchanging it with an empty memory location if necessary) and add the other register to it in `copy←copy+original` where `copy` is initially clear[1]. Second, control flow operations must appear in identical pairs.

Register-type instructions include all register to register operations, including logical, arithmetic, shift, and rotate. They take one, two, or three register addresses. For non-expanding instructions, such as `ADD`, the destination register `rsd` must be the same as the first source register. Expanding instructions must specify three different registers. This is very different from other architectures, but the inconvenience for the programmer should be small.

[1] "Clear" may be different from "zero." A cleared location is known to be zero. A location may contain zero as a product of computation and not be cleared.

The following pages contain the syntax and description for execution of all Pendulum Assembly Language instructions. No exceptions are generated during any instructions.

The format and notation of the instruction set details is taken from the MIPS R2000 architecture reference manual [KH92].

| Symbol | Meaning |
|---|---|
| $\leftarrow$ | Assignment |
| $\|\|$ | Bit string concatenation |
| $x^y$ | Replication of bit value $x$ into a $y$-bit string. Note that $x$ is always a single-bit value |
| $x_{y..z}$ | Selection of bits $y$ through $z$ of bit string $x$. Little endian bit notation is always used. If $y$ is less than $z$, this expression is an empty (zero length) bit string. |
| ! | Logical inversion |
| $+$ | Two's complement addition |
| $-$ | Two's complement subtraction |
| $<$ | Two's complement less than comparison |
| *neg* | Two's complement negation |
| *and* | Bitwise logic AND |
| *or* | Bitwise logic OR |
| *xor* | Bitwise logic XOR |
| *nor* | Bitwise logic NOR |
| GPR[x] | General Register $x$ |
| PC | Program Counter |
| MEM[x] | Memory Location $x$ |

Table A.1: Instruction Operation Notations

Since one of the earliest design decisions was to base Pendulum on a 32 bit RISC machine, the instruction word encoding resembles the MIPS R2000 instruction word encoding. Later reduction of the Pendulum datapath to 8 bits shrunk a number of instruction bit fields. Increase in the datapath width to 12 bits made the instruction set somewhat resemble that of the PDP-8.

The opcode field is limited to five bits so that the immediate field can be large. The instruction set is limited to 32 types of instructions. Pendulum uses three types of instruction encodings, listed in Table A.2.

Register instruction types and the exchange instruction use an R-type encoding. The in-

struction word specifies a destination register and two source registers. Non-expanding instructions must have `rd` and `rs` identical and different from `rt`, and expanding instructions must have all three registers different. To emphasize this point, the instruction encodings for non-expanding operations identify the three registers as `rsd`, `rsd`, and `rt`. The second source register is specified to be zero for exchange operations.

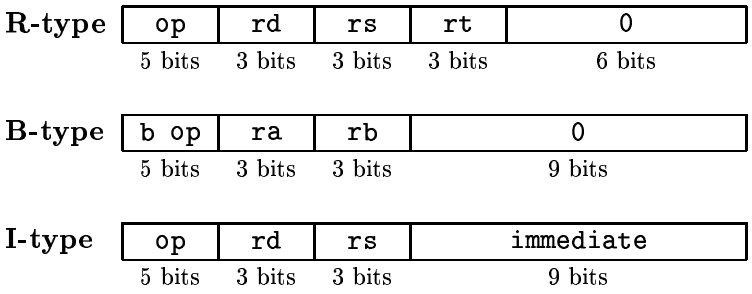


Table A.2: Instruction Formats

---

Branches use B-type instruction encoding and specify two registers, one containing the address to be jumped to and the other containing the value on which the branch is conditional. The two registers must be different.

All non-expanding R-type operations must have `rd` and `rs` equal. All expanding R-type operations must have all three registers different.

The immediate instructions, I-type, are identical to R-type encodings with respect to expanding and non-expanding operations, except that instead of an `rt` register, immediate instructions specify a 9-bit sign extended immediate value.

---

**Instruction mnemonic: ADD**

**Instruction name: Add**

| ADD | rsd | rsd | rt | 0 |
|---|---|---|---|---|
| 01001 | | | | 00 0000 |
| 5 bits | 3 bits | 3 bits | 3 bits | 6 bits |

**Format:** ADD rsd, rt

**Description:**

The contents of register *rsd* and register *rt* are added to form a 12-bit result. The result is placed in register *rsd*.

**Operation:**

GPR[rsd] $\leftarrow$ GPR[rsd] + GPR[rt]

---

**Instruction Mnemonic: ADDI**

**Instruction Name: Add Immediate**

| ADDI | rsd | rsd | immediate |
|---|---|---|---|
| 00001 | | | |
| 5 bits | 3 bits | 3 bits | 9 bits |

**Format:** ADDI rsd, immediate

**Description:**

The 9-bit *immediate* is sign extended and added to the contents of register *rsd*. The result is placed in register *rsd*.

**Operation:**

GPR[rsd] $\leftarrow$ GPR[rsd] + $(\text{immediate}_8)^3 \| \text{immediate}_{8..0}$

---

**Instruction Mnemonic: ANDIX**

**Instruction Name: And Immediate-Xor**

| ANDIX | rd | rs | immediate |
|---|---|---|---|
| 10000 | | | |
| 5 bits | 3 bits | 3 bits | 9 bits |

**Format:** ANDIX rd, rs, immediate

**Description:**

The 9-bit *immediate* is sign extended and combined with the contents of register *rs* in a bit-wise logical AND operation. The result is combined in a logical XOR with the contents of register *rd* and is placed in register *rd*.

**Operation:**

GPR[rd] $\leftarrow$ (GPR[rs] *and* (immediate$_8$)$^3$|| immediate$_{8..0}$) *xor* GPR[rd]

---

**Instruction Mnemonic: ANDX**

**Instruction Name: And-Xor**

| ANDX | rd | rs | rt | 0 |
|---|---|---|---|---|
| 11000 | | | | 00 0000 |
| 5 bits | 3 bits | 3 bits | 3 bits | 6 bits |

**Format:** ANDX rd, rs, rt

**Description:**

The contents of register *rs* and register *rt* are combined in a bit-wise logical AND operation. The result is combined in a logical XOR with the contents of register *rd* and is placed in register *rd*.

**Operation:**

GPR[rd] $\leftarrow$ (GPR[rs] *and* GPR[rt]) *xor* GPR[rd]

---

**Instruction Mnemonic: BEZ**

**Instruction Name: Branch On Equal to Zero**

| BEZ | ra | rb | 0 |
|---|---|---|---|
| 10100 | | | 0 0000 0000 |
| 5 bits | 3 bits | 3 bits | 9 bits |

**Format:** BEZ ra, rb

**Description:**

The contents of register *rb* are compared to zero. If the contents of the register equal zero, the program branches to the address specified in register *ra* by exchanging the contents of *ra* with the value of the program counter.

**Operation:**

if GPR[rb] = $0^{12}$ then

  PC ← GPR[ra]

  GPR[ra] ← PC

endif

---

**Instruction Mnemonic: BLTZ**

**Instruction Name: Branch On Less Than Zero**

| BLTZ | ra | rb | 0 |
|---|---|---|---|
| 10101 | | | 0 0000 0000 |
| 5 bits | 3 bits | 3 bits | 9 bits |

**Format:** BLTZ ra, rb

**Description:**

If the contents of register *rb* have the sign bit set the program branches to the address specified in register *ra* by exchanging the contents of *ra* with the value of the program counter.

**Operation:**

if $GPR[rb]_{11} = 1$ then

  PC ← GPR[ra]

  GPR[ra] ← PC

endif

---

**Instruction Mnemonic: EXCH**

**Instruction Name: Exchange**

| EXCH | exch | addr | 0 |
|---|---|---|---|
| 10011 | | | 0000 0000 |
| 5 bits | 3 bits | 3 bits | 9 bits |

**Format:** EXCH exch, addr

**Description:**

The contents of register *exch* are placed at the data memory location specified by the contents of register *addr*. The contents of the data memory location specified by the contents of register *addr* are placed in register *exch*.

**Operation:**

GPR[exch] $\leftarrow$ MEM[addr]

MEM[addr] $\leftarrow$ GPR[exch]

---

**Instruction Mnemonic: ORIX**

**Instruction Name: Or Immediate-Xor**

| ORIX | rd | rs | immediate |
|---|---|---|---|
| 10001 | | | |
| 5 bits | 3 bits | 3 bits | 9 bits |

**Format:** ORIX rd, rs, immediate

**Description:**

The 9-bit *immediate* is sign extended and combined with the contents of register *rs* in a bit-wise logical OR operation. The result is combined in a logical XOR with the contents of register *rd* and is placed in register *rd*.

**Operation:**

GPR[rd] $\leftarrow$ (GPR[rs] *or* $(\text{immediate}_8)^3 \| \text{immediate}_{8..0}$) *xor* GPR[rd]

---

**Instruction mnemonic: ORX**

**Instruction name: Or-Xor**

| ORX | rd | rs | rt | 0 |
|---|---|---|---|---|
| 11001 | | | | 00 0000 |
| 5 bits | 3 bits | 3 bits | 3 bits | 6 bits |

**Format:** ORX rd, rs, rt

**Description:**

The contents of register *rs* and register *rt* are combined in a bit-wise logical OR operation. The result is combined in a logical XOR with the contents of register *rd* and is placed in register *rd*.

**Operation:**

GPR[rd] $\leftarrow$ (GPR[rs] *or* GPR[rt]) *xor* GPR[rd]

---

**Instruction Mnemonic: RBEZ**

**Instruction Name: Reverse Direction, Branch On Equal to Zero**

| RBEZ | ra | rb | 0 |
|---|---|---|---|
| 10110 | | | 0 0000 0000 |
| 5 bits | 3 bits | 3 bits | 9 bits |

**Format:** RBEZ ra, rb

**Description:**

The contents of register *rb* are compared to zero. If the contents of the register equal zero, the program branches to the address specified in register *ra* by exchanging the contents of *ra* with the value of the program counter. The global direction bit is also toggled when this instruction is executed.

**Operation:**

if GPR[rb] = $0^{12}$ then

  PC $\leftarrow$ GPR[ra]

  GPR[ra] $\leftarrow$ PC

  direction $\leftarrow$ !direction

endif

---

**Instruction Mnemonic: RBLTZ**

**Instruction Name: Reverse Direction, Branch On Less Than Zero**

| RBLTZ | ra | rb | 0 |
|---|---|---|---|
| 10111 | | | 0 0000 0000 |
| 5 bits | 3 bits | 3 bits | 9 bits |

**Format:** RBLTZ ra, rb

**Description:**

If the contents of register *rb* have the sign bit set the program branches to the address specified in register *ra* by exchanging the contents of *ra* with the value of the program counter. The global direction bit is also toggled when this instruction is executed.

**Operation:**

if $\text{GPR[rb]}_{11} = 1$ then

  PC $\leftarrow$ GPR[ra]

  GPR[ra] $\leftarrow$ PC

  direction $\leftarrow$ !direction

endif

---

**Instruction mnemonic: RL**

**Instruction name: Rotate Left**

| RL | rsd | rsd | 0 | 0 |
|---|---|---|---|---|
| 00100 | | | 000 | 00 0000 |
| 5 bits | 3 bits | 3 bits | 3 bits | 6 bits |

**Format:** RL rsd

**Description:**

The contents of register *rsd* are rotated left one bit. The result is placed in register *rsd*.

**Operation:**

$\text{GPR[rsd]} \leftarrow \text{GPR[rsd]}_{10..0} \| \text{ GPR[rsd]}_{11}$

---

**Instruction mnemonic: RR**

**Instruction name: Rotate Right**

| RR | rsd | rsd | 0 | 0 |
|---|---|---|---|---|
| 00101 | | | 000 | 00 0000 |
| 5 bits | 3 bits | 3 bits | 3 bits | 6 bits |

**Format:** RR rsd

**Description:**

The contents of register *rsd* are rotated right one bit. The result is placed in register *rsd*.

**Operation:**

GPR[rsd] $\leftarrow$ GPR[rsd]$_0$|| GPR[rsd]$_{11..1}$

---

**Instruction mnemonic: SLLX**

**Instruction name: Shift Left Logical-Xor**

| SLLX | rd | rs | 0 | 0 |
|---|---|---|---|---|
| 11100 | | | 0000 | 00 0000 |
| 5 bits | 3 bits | 3 bits | 3 bits | 6 bits |

**Format:** SLLX rd, rs

**Description:**

The contents of register *rs* are shifted left by one bit, inserting zero into the low order bit. The result is combined in a logical XOR with the contents of register *rd* and is placed in register *rd*.

**Operation:**

GPR[rd] $\leftarrow$ (GPR[rs]$_{10..0}$|| 0) *xor* GPR[rd]

---

**Instruction mnemonic: SRAX**

**Instruction name: Shift Right Arithmetic-Xor**

| SRAX | rd | rs | 0 | 0 |
|---|---|---|---|---|
| 11110 | | | 0000 | 00 0000 |
| 5 bits | 3 bits | 3 bits | 3 bits | 6 bits |

**Format:** SRAX rd, rs

**Description:**

The contents of register *rs* are shifted right by one bit, sign extending the high order bit. The result is combined in a logical XOR with the contents of register *rd* and is placed in register *rd*.

**Operation:**

GPR[rd] $\leftarrow$ (GPR[rs]$_{11}$ $\|$ GPR[rs]$_{11..1}$) *xor* GPR[rd]

---

**Instruction mnemonic: SRLX**

**Instruction name: Shift Right Logical-Xor**

| SRLX | rd | rs | 0 | 0 |
|---|---|---|---|---|
| 11101 | | | 0000 | 00 0000 |
| 5 bits | 3 bits | 3 bits | 3 bits | 6 bits |

**Format:** SRLX rd, rs

**Description:**

The contents of register *rs* are shifted right by one bit, inserting zero into the high order bit. The result is combined in a logical XOR with the contents of register *rd* and is placed in register *rd*.

**Operation:**

GPR[rd] $\leftarrow$ (0 $\|$ GPR[rs]$_{11..1}$) *xor* GPR[rd]

---

**Instruction mnemonic: XOR**

**Instruction name: Exclusive Or**

| XOR | rsd | rsd | rt | 0 |
|---|---|---|---|---|
| 01010 | | | | 00 0000 |
| 5 bits | 3 bits | 3 bits | 3 bits | 6 bits |

**Format:** XOR rsd, rs

**Description:**

The contents of register *rsd* and register *rs* are combined in a bit-wise logical exclusive OR operation. The result is placed in register *rsd*.

**Operation:**

GPR[rsd] $\leftarrow$ GPR[rsd] *xor* GPR[rs]

---

**Instruction Mnemonic: XORI**

**Instruction Name: Xor Immediate**

| XORI | rsd | rsd | immediate |
|---|---|---|---|
| 00010 | | | |
| 5 bits | 3 bits | 3 bits | 9 bits |

**Format:** XORI rsd, immediate

**Description:**

The 9-bit *immediate* is sign extended and combined with the contents of register *rsd* in a bit-wise logical XOR operation. The result is placed in register *rsd*.

**Operation:**

GPR[rsd] $\leftarrow$ GPR[rsd] *xor* $(\text{immediate}_8)^3 \| \text{immediate}_{8..0}$

# Bibliography


[AJ97] Stephen Avery and Marwan Jabri. Design of a register file using adiabatic logic. Technical Report SCA-06/06/97, University of Sydney SEDAL, June 1997.

[AS94] William C. Athas and Lars "J." Svensson. Reversible logic issues in adiabatic CMOS. In *IEEE Workshop on Physics and Computing*, 1994.

[ASK+94] William C. Athas, Lars "J." Svensson, J. G. Koller, Nestoras Tzartzanis, and E. Chou. A framework for practical low-power digital CMOS systems using adiabatic-switching principles. In *Proc. of the Int'l Workshop on Low-Power Design*, pages 189–194, 1994.

[AST96] William C. Athas, Lars "J." Svensson, and Nestoras Tzartzanis. A resonant signal driver for two-phase, almost-non-overlapping clocks. In *International Symposium on Circuits and Systems*, 1996.

[ATS+97] William C. Athas, Nestoras Tzartzanis, Lars "J." Svensson, Lena Peterson, H. Li, X. Jiang, P. Wang, and W-C. Liu. AC-1: A clock-powered microprocessor. In *International Symposium on Low Power Electronics and Design*, pages 18–20, 1997.

[Bak92a] Henry G. Baker. Lively linear lisp—'look ma, no garbage!'. *ACM Sigplan Notices*, 27(8):89–98, August 1992.

[Bak92b] Henry G. Baker. NREVERSAL of fortune — the thermodynamics of garbage collection. In Y. Bekkers, editor, *International Workshop on Memory Management*, pages 507–524. Springer-Verlag, 1992.

[Ben73] C. H. Bennett. Logical reversibility of computation. *IBM J. Research and Development*, 6:525–532, 1973.

[Ben82] C. H. Bennett. The thermodynamics of computation, a review. *Int'l J. Theoretical Physics*, 21(12):905–940, 1982.

[Ben88] C. H. Bennett. Notes on the history of reversible computation. *IBM J. Research and Development*, 32(1):281–288, 1988.

[Ben89] C. H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Computing*, 18(4):766–776, 1989.

[BK98] Matthew E. Becker and Thomas F. Knight, Jr. Transmission line clock driver. In *Proceedings of the 1998 ISCA Power Driven Microarchitecture Workshop*, 1998.

[CCD98] C. S. Calude, J. Casti, and M. J. Dinneen, editors. *Unconventional Models of Computation*. Springer-Verlag, 1998.

[Fey86] Richard P. Feynman. Quantum mechanical computers. *Foundations of Physics*, 16(6), 1986.

[Fra97] Michael P. Frank. Modifications to PISA architecture to support guaranteed reversibility and other fetures. Online draft memo, July 1997. `http://www.ai.mit.edu/~mpf/rc/memos/M07/M07_revarch.html`.

[Fra99] Michael P. Frank. *Reversibility for Efficient Computing*. PhD thesis, MIT Artificial Intelligence Laboratory, 1999.

[FT82] Edward F. Fredkin and Tommaso Toffoli. Conservative logic. *Int'l J. Theoretical Physics*, 21(3/4):219–253, 1982.

[FVA+98] Michael P. Frank, Carlin J. Vieri, M. Josephine Ammer, Nicole Love, Norman H. Margolus, and Thomas F. Knight, Jr. A scalable reversible computer in silicon. In Calude et al. [CCD98], pages 183–200.

[Hal92] J. Storrs Hall. An electroid switching model for reversible computer architectures. In *Physics and Computation*, pages 237–247, October 1992.

[Hal94] J. Storrs Hall. A reversible instruction set architecture and algorithms. In *Physics and Computation*, pages 128–134, November 1994.

[KA92] J. G. Koller and William C. Athas. Adiabatic switching, low energy computing, and the physics of storing and erasing information. In *Physics of Computation Workshop*, 1992.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

[Lan61] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM J. Research and Development*, 5:183–191, 1961.

[Lik82] K. K. Likharev. Classical and quantum limitations on energy consumption in computation. *Int'l J. Theoretical Physics*, 21(3/4):311–326, 1982.

[Mar87] Norman H. Margolus. *Physics and Computation.* PhD thesis, Massachusettes Institute of Technoloogy, 1987.

[Max75] James C. Maxwell. *Theory of Heat.* Longmans, Green & Co., London, 4th edition, 1875.

[Mer92] Ralph C. Merkle. Towards practical reversible logic. In *Physics and Computation*, pages 227–228, October 1992.

[Mer93] Ralph C. Merkle. Two types of mechanical reversible logic. *Nanotechnology*, 4:114–131, 1993.

[PH90] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, 1990.

[Res79] Andrew L. Ressler. Practical circuits using conservative reversible logic. Bachelor's thesis, MIT, 1979.

[Res81] Andrew L. Ressler. The design of a conservative logic computer and a graphical editor simulator. Master's thesis, MIT Artificial Intelligence Laboratory, 1981.

[SC95] Olin Shivers and Brian D. Carlstrom. Scsh reference manual. Draft, November 1995.

[SFM+85] C. L. Seitz, A. H. Frey, S. Mattisson, S. D. Rabin, and D. A. Speck. Hot-clock NMOS. In Henry Fuchs, editor, *Chapel Hill Conference on VLSI*, 1985.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete log and factoring. In *Foundations of Computer Science, Proc. 35th Ann. Symp.*, pages 124–134. IEEE Computer Society Press, 1994.

[SK94] Lars "J." Svensson and J. G. Koller. Adiabatic charging without inductors. Technical Report ACMOS-TR-3a, USC Information Sciences Institute, February 1994.

[SYR95] Dinesh Somasekhar, Yibin Ye, and Kaushik Roy. An energy recovery static RAM memory core. In *Symposium on Low Power Electronics*, pages 62–63, 1995.

[Szi29] Leo Szilard. On the decrease of entropy in a thermodynamic system by the intervention of intelligent beings. *Zeitschrift für Physik*, 53:840–852, 1929. English translation in *Behavioral Science*, 9:301-310, 1964.

[TA96] Nestoras Tzartzanis and William C. Athas. Energy recovery for the design of high-speed, low power static RAMs. In *International Symposium on Low Power Electronics and Design*, pages 55–60, 1996.

[Vie95] Carlin J. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.

[YK93] Saed G. Younis and Thomas F. Knight, Jr. Practical implementation of charge recovering asymptotically zero power CMOS. In *Proceedings of the 1993 Symposium in Integrated Systems*, pages 234–250. MIT Press, 1993.

[YK94] Saed G. Younis and Thomas F. Knight, Jr. Asymptotically zero energy split-level charge recovery logic. In *International Workshop on Low Power Design*, pages 177–182, 1994.

[You94] Saed G. Younis. *Asymptotically Zero Energy Computing Using Split-Level Charge Recovery Logic.* PhD thesis, MIT Artificial Intelligence Laboratory, 1994.