Pendulum: A Reversible Computer Architecture

by

Carlin James Vieri

B.S., University of California at Berkeley (1993)

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

© Massachusetts Institute of Technology 1995

Author ____

Department of Electrical Engineering and Computer Science
May 24, 1995

Certified by ____

Thomas F. Knight, Jr.
Thesis Supervisor

Accepted by ____

Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Pendulum: A Reversible Computer Architecture

by

Carlin James Vieri

Submitted to the
Department of Electrical Engineering and Computer Science

May 24, 1995

In Partial Fulfillment of the Requirements for the Degree of
Master of Science

Abstract

Energy dissipation in modern microprocessors is rapidly becoming a primary design concern. Microprocessors containing a few million transistors and dissipating tens of watts are commonplace, limiting their usefulness in portable applications and making heat removal in dense structures difficult. An expanding market for portable devices and increasing device density will continue to encourage low energy design.

Computing engines can be designed that do not require energy dissipation, but only if the computation is logically reversible, a radical departure from both traditional logic design and traditional low energy design techniques. This thesis presents Pendulum, a logically reversible computer architecture that may operate without dissipating energy.

The novel aspect of the Pendulum reversible processor is that all computation is reversible. The processor saves enough information to invert every operation. Programs may be executed in reverse. At any point in the computation the processor direction may be reversed and any intermediate results will be "uncomputed."

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

I believe in this, and it's been tested by research...

Death or Glory, The Clash

Acknowledgments

Thanks to Raj Amirtharajah, Mike "Misha" Bolotski, Tom Simon, and Charles "Chuckles" Isbell for the endless discussions of the engineering tradeoffs involved in designing Pendulum. Thanks also for help in making the personal tradeoffs involved in writing this thesis.

Thanks to Matt DeBergalis for his tireless work in the early stages of this project.

A hearty thanks to the denizens of 7ai who have been alternately nurturing, critical, supportive, cynical, and instructive. You know who you are.

Thanks to Tom Knight, with whom I first discussed a reversible processor, for the incredible range of topics he's taught me something about, and all the valuable advice he's given me during the course of my research.

Thanks to my parents, for all the years of love and support.

Contents

1 Introduction **10**

2 Background and Previous Work **12**

2.1 Physics 12

2.2 Implementation Technology 16

2.2.1 Mechanical Reversible Logic 16

2.2.2 Circuits 17

2.3 Architecture 19

2.4 Pendulum 20

3 Reversible Architecture Design **21**

3.1 Memory Access 21

3.2 Execution Unit 22

3.2.1 Register File Access 22

3.2.2 Reversible and Irreversible Operations 23

3.2.3 Operand Specifier Format 24
3.3 Control Flow 25
3.4 Conclusion 27

4 Pendulum Processor 29
4.1 Overview 29
4.2 Pendulum Instructions 30
4.2.1 Memory Access 32
4.2.2 Special Instructions 34
4.2.3 Immediate Instructions 35
4.2.4 Control Flow Instructions 36

5 Future Work 39
5.1 Input/Output Behavior 39
5.2 Pipelining 40
5.3 Instruction Set Expansion 42
5.4 Garbage Reduction 43
5.4.1 Datapath Garbage 44
5.4.2 Control Flow Garbage 44
5.4.3 Programmer Defined Garbage 45
5.4.4 Bit Erasure 46

6 Conclusions 47

A The Pendulum Assembly Language 48

A.1 Instruction Set Encoding 49

B Detailed Datapath Schematic 73

List of Figures

2-1 Irreversible Bit Erasure Model 15

2-2 Billiard Ball Model Gates 17

2-3 Split-level Charge Recovery Logic Inverter 18

3-1 Register File Read and Write 23

3-2 Control Flow Confluence 26

4-1 Pendulum Datapath 31

4-2 Register File Connections 33

B-1 Detailed Datapath Schematic Left 74

B-2 Detailed Datapath Schematic Right 75

List of Tables

3.1 Operand Specifier Formats and Storage Requirements 25

A.1 Instruction Operation Notations 50

A.2 Instruction Formats 51

Chapter 1

Introduction

Energy dissipation in modern microprocessors is rapidly becoming a primary design concern. Microprocessors containing a few million transistors and dissipating tens of watts are commonplace, limiting their usefulness in portable applications and making heat removal in dense structures difficult. An expanding market for portable devices and increasing device density will continue to encourage low energy design.

Computing engines can be designed that do not require energy dissipation [Ben73, Lan82], but only if the computation is logically reversible. This approach is a radical departure from both traditional logic design and traditional low energy design techniques. This thesis presents Pendulum, a logically reversible computer architecture that may operate without dissipating energy.

For the computation to be physically reversible, and therefore not dissipate any energy, the computing engine must be logically reversible and implemented in a physically reversible technology [Ben82]. Any system that transitions from a state A to a state B is physically reversible if the state B uniquely determines state A, implying that the transition was logically reversible, and the energy is available to make the reverse transition, implying that the transition was made in a physically reversible technology.

Logical reversibility imposes architectural constraints not met by conventional processors. A conventional computing engine performs irreversible computations. These computations destroy information, and the second law of thermodynamics requires a minimum energy dissipation when a bit of information is discarded [Lan86]. The novel aspect of the Pendulum reversible processor is that all computation is reversible. The processor saves enough information to invert every operation. Programs may be executed in reverse. At any point in the computation the processor direction may be reversed and any intermediate results will be "uncomputed."

Previous work concerning reversible computer architecture has been either impractical or incomplete. Ressler's work [Res79, Res81] is significant in that it is the earliest work which is directly relevant to architecting fully reversible computers, but it is flawed in its exclusive use of the Fredkin [FT82] gate and its neglect of key control flow issues. Hall's work [Hal94], while correct in many high level issues, is incomplete, suggesting no mapping between instruction set architecture (ISA) and register transfer level (RTL) implementation. Indeed, Hall bases his reversible instruction set on the PDP-10, an ISA that presents a difficult mapping to an RTL datapath even for the original irreversible version.

This thesis discusses a complete instruction set and RTL datapath design for a reversible processor architecture. The ISA is based on a modern RISC processor, the MIPS R2000, and the datapath design is suitable for implementation in a VLSI technology [You94]. High level hardware description language simulations have demonstrated the functionality of the design and its ability to execute an instruction stream forward and backward.

Chapter 2 briefly describes background and previous work relating to reversible computing. Chapter 3 deals with the general issues and engineering tradeoffs that arise in the design of a reversible architecture and RTL implementation. Chapter 4 then describes the specific decisions and rationale of the Pendulum processor design. Future work is presented in Chapter 5 and Chapter 6 concludes.

Chapter 2

Background and Previous Work

Early computer researchers were interested in the physical limits of computing operations. This chapter considers research into the physical limits of energy dissipation during computation, how this research led to consideration of reversible computing systems, and what work has been done towards building a practical reversible computing engine.

This chapter examines reversible computing from the bottom up: first physics and thermodynamics, then reversible circuit and other implementation technologies, and finally, reversible computer architecture.

2.1 Physics

Maxwell's demon and Szilard's analysis [Szi29] of the demon first suggested the connection between a single degree of freedom (one bit) and a minimum quantity of entropy. In the 1950s, this connection had been popularly interpreted to mean that computation must dissipate a corresponding minimum amount of energy during every elemental act of computation. Landauer [Lan61] later recognized that energy dissipation is only unavoidable when information is destroyed. Bennett [Ben73] and Fredkin [FT78] first realized that a

reversible computation[1], in which no information is destroyed, may dissipate arbitrarily small amounts of energy.

Maxwell's Demon The limit of energy dissipation during computation is fundamentally based in the apparent thermodynamic paradox of Maxwell's demon. Maxwell described the system in [Max75]:

> For we have seen that the molecules in a vessel full of air at uniform temperature are moving with velocities by no means uniform, though the mean velocity of any great number of them, arbitrarily selected, is almost exactly uniform. Now let us suppose that such a vessel is divided into two portions, A and B, by a division in which there is a small hole, and that a being, who can see the individual molecules, opens and closes this hole, so as to allow only the swifter molecules to pass from A to B, and only the slower ones to pass from B to A. He will thus, without expenditure of work, raise the temperature of B and lower that of A, in contradiction to the second law of thermodynamics.

The demon has been depicted in various ways. Some show the demon inside the chamber with the gas, some have him outside. Any analysis must be sure to include the thermodynamic effects within the demon himself in the energy and entropy accounting. Some images give the demon a light source to aid in measurement of the particle's speed, indicating the tack taken by some authors to explain the paradox of attributing the entropy increase to dissipation during measurement.

Szilard, nearly 55 years after Maxwell first postulated the demon, attempted to resolve the paradox by arguing that the process of measurement required dissipation, although he did notice an entropy generation of $k \ln 2$ when the demon was reset. But it was not until much later [Lan61] that researchers firmly placed the source of dissipation in the erasure of information. When the demon measures a particle, he must set a bit indicating the speed of the particle. The hole between the portions of the vessel is controlled by the

[1] Landauer at first believed that logical irreversibility was required for useful computation, and therefore that reversible computation was impossible, but Bennett convincingly proved otherwise.

state of this bit. Once a particle has been directed to the correct portion, the demon must reset the bit in preparation for the next measurement value. This resetting is the logically irreversible event which saves the second law. Measurement may be performed reversibly; information *destruction*, rather than information *acquisition*, has a thermodynamic cost. In any irreversible process, entropy must increase. The required entropy increase during irreversible bit erasure is a function of the process by which it is done, the time taken for erasure, and the temperature of the system, but the increase must be at least zero. However, the required *energy dissipation* must be at least $kT \ln 2$.

Irreversible Bit Erasure As an example, consider a box, shown in Figure 2-1, with a stretchable partition that divides the box into two halves. In step A, a gas is on one side or the other of the partition. Its entropy is $k \ln 2$ because it can be in two possible states. Then, a piston is slowly pushed from one side into the box and the gas and partition are compressed isothermally (the gas is in contact with a heat reservoir) onto one side, shown in step B. The process is still reversible: any work done by the piston on the the gas and the partition may be recovered by recovering the heat from the reservoir and letting the gas and partition relax back to their original position. In step C the bit has been irreversibly erased. The partition has been removed and reinserted next to the piston, ensuring that the gas is on the right side of the partition. The entropy of the box part of the system *decreases* to $\mathrm{S} = k \ln 1 = 0$ since the system only has one possible state. To balance this decrease in entropy, the partition must dissipate at least $kT \ln 2$ of *energy* when it is removed, since $\Delta\mathrm{E} = \mathrm{T}\Delta\mathrm{S}$. Of course, the total entropy of the system has not decreased; it has increased by some amount dependent on the losses due to the details of the process.

In step D the piston has been removed and the particle can only be in one state. The net entropy of the system has increased only by some process dependent amount, but the process *requires* an energy dissipation of at least $kT \ln 2$.

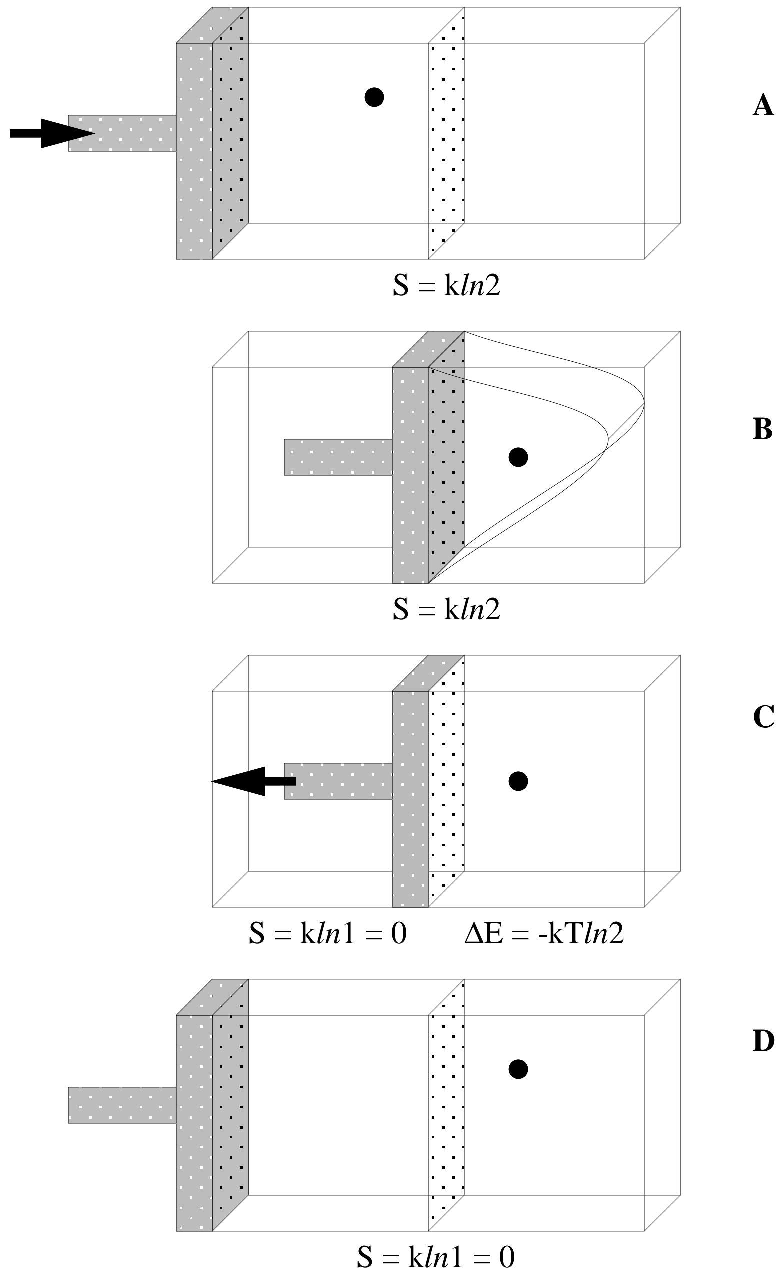

Figure 2-1: Irreversible Bit Erasure Model

2.2 Implementation Technology

For a computing system to be physically reversible, it must both avoid logically irreversible operations and be implemented in a physically reversible technology. This section discusses these physically reversible technologies; Section 2.3 deals with architectures which avoid logical irreversibilities.

2.2.1 Mechanical Reversible Logic

Once bit erasure was identified as a source of unavoidable energy dissipation, researchers investigated a number of theoretical and practical schemes to implement a reversible computing technology. The first to be proposed were a series of hypothetical mechanical constructions. Fredkin [FT82] proposed the billiard ball model which uses collisions of hard spheres and mirrors to perform reversible computations. Figure 2-2 shows a crossover gate and Feynman's two input, three output universal logic gate. Both gates are reversible and non-dissipative when isolated from imperfections. Fredkin demonstrated that such collisions are capable of simulating any logic function, but they required perfect spheres and isolation from friction, thermal noise, and other imperfections.

In this idealized environment, the presence of a ball represents (by convention) a one, the absence of a ball, a zero. The balls move in straight lines with a constant velocity and experience perfectly elastic collisions with other balls and the mirrors. All the balls are given an initial velocity with equal components in the X and Y directions and start at integral coordinates on a Cartesian plane. As the system evolves, all balls will move onto integral coordinates simultaneously. Combinations of these gates can perform any logic function, including memory and feedback, and are reversible.

Various other computing structures have been proposed. Brownian computers allow the trajectory of component particles to follow a random walk through the device, the speed of computation (and dissipation) being proportional to the gradient of an applied force. Genetic material such as DNA and RNA are cited [Ben73, Ben82] as "nature's closest ap-

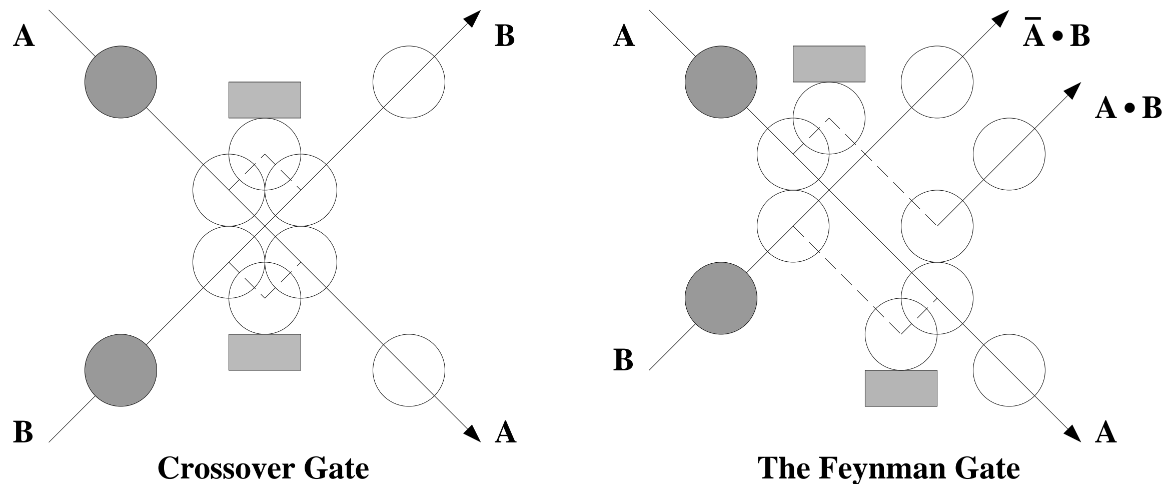

Figure 2-2: Billiard Ball Model Gates

proach to a Brownian computer" with a dissipation of between 20 and 100 kT per operation (at the cost of computation speed and random access). Bennett also describes a "baroque" reversible clockwork Turing machine which does not require the ballistic model's isolation from noise.

2.2.2 Circuits

Recently, researchers have discovered a number of energy recovering integrated circuit techniques that exploit reversibility to reduce power consumption in logic circuits. The emergence of practical reversible implementation techniques provided much of the motivation for this thesis.

Split-level Charge Recovery Logic The most highly developed energy recovering reversible logic family is probably the Split-level Charge Recovery Logic (SCRL) of Younis and Knight [YK93, YK94]. Figure 2-3, taken from [You94], shows an SCRL inverter. Instead of constant voltage rails, SCRL uses a series of clock signals which gradually swing from a midpoint voltage to either a high or low voltage. Charge that is stored on the gates of CMOS gates is recovered and the energy stored in the electric field of the circuit capacitance

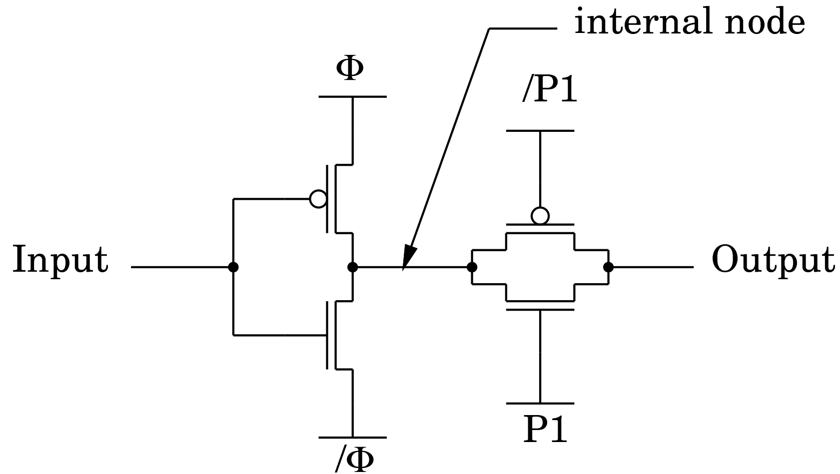

Figure 2-3: Split-level Charge Recovery Logic Inverter

is transfered into the magnetic field of an external inductor.

The energy dissipated per operation in SCRL falls linearly as the computation delay increases, as opposed to conventional CMOS circuits which have a relatively constant energy dissipation per operation. Energy dissipation in SCRL circuits falls to zero as delay increases to infinity.

The ability of SCRL circuits to recover charge requires that computations be performed reversibly. Individual devices are restricted from turning on if a potential exists across them, and voltage transitions are made to happen in a controlled manner by swinging the supply rails slowly. It is clear from Figure 2-3 that if the input value is steady at a high or low voltage value, when the power supply rails, Φ1 and /Φ1, swing from the voltage midpoint to the high and low voltage values respectively, the internal node will follow the correct rail to the proper logic level. The value of the input is computed from the output through another inverter to non-dissipatively clear the input value by bringing the rails back to the midpoint value. Younis fabricated an 8 x 8 reversible multiplier array using SCRL gates in a 2 micron technology. He measured an energy savings of over 99% of the power used in conventional CMOS implementations of the same circuits when run at speeds below 1 Megahertz.

SCRL is a promising technology for implementing a reversible computer, but a number of issues, especially the design of an appropriate ramping power supply, still must be addressed. Further details concerning SCRL appear in [You94].

Other Energy Recovering Circuits In [FT78], Fredkin and Toffoli describe a circuit implementation of a Fredkin gate. This design requires multiple large inductors per gate, and the authors admit that the concept is not appropriate for VLSI applications. They do, however, suggest that Josephson junction-based systems may provide a better platform. Likharev [Lik82] also proposed a superconducting Josephson junction-based computing engine which performed energy recovery.

Koller and Athas [KA92], using techniques similar to SCRL, have developed a method of driving highly capacitive wiring and gate loads while recovering the energy. Their work on power supply design is similar to the power supply work needed for SCRL.

Hall's [Hal92] "retractile cascade" circuits use a series of clocks and inherently pipelined primitives which are very similar in spirit to SCRL gates.

2.3 Architecture

This section addresses the previous work in developing computing paradigms that avoid logically irreversible operations.

Ressler [Res81] appears to have been the first to investigate the requirements of a reversible computer. Using only Fredkin gates, but suggesting no implementation strategy, he designed a simple accumulator-based machine. He discussed control flow issues and the concept of a garbage stack to retain extra operands from irreversible operations, but his design bears little resemblance to a modern processor model. The datapath in Ressler's work does not have explicit forward and reverse components but relies on the instruction set and reversible Fredkin gates to assure reversibility. It is difficult to interpret the datapath design decisions that distinguish his processor from a standard accumulator-based processor. Regardless,

his design is remarkable in that he was able to design an entire processor using on the order of 5000 reversible Fredkin gates.

More recently, Hall [Hal94], building on his work with retractile cascades [Hal92], discussed a reversible processor architecture and algorithms based on the PDP-10 instruction set. The decision to use a CISC instruction set allows shorter code, but for this thesis, a more straightforward RISC style makes the datapath and controller design simpler. Hall does not suggest even a block diagram level design for a processor. While he claims that intermediate results produced during some operations, such as effective address calculation, can be reversibly undone more easily in a CISC machine, he does not consider that a suitably restricted instruction set could effectively eliminate these intermediate results while using a simpler datapath. The programmer concerned with the flow of information in the processor is often better served by simple instructions with no intermediate results computed during the course of the instruction. Also, additional pipelining work and performance enhancements are possible using a RISC foundation.

Baker [Bak92] covers a wide range of topics related to reversible computing from the thermodynamics of bit erasure to garbage collection and programming subtleties. He suggests several novel physics-based architectural ideas for simulating physical systems, such as the high cost of copy operations in physical systems (mechanical metaphor) as opposed to the inexpensive copy of the traditional (writing metaphor) view of computing. He discusses implications for object oriented programming, in which each object is identifiable, as opposed to a large number of identical copies.

2.4 Pendulum

This thesis builds primarily on previous architecture work, especially that of Ressler and Hall. It assumes throughout that bit erasure must be avoided and all computation must be logically reversible.

Chapter 3

Reversible Architecture Design

This chapter presents a moderately detailed discussion of the issues and engineering tradeoffs which arise in the design of a reversible architecture and register transfer level implementation. The context is that of modifying a conventional RISC processor architecture for logically reversible operation and circuit implementation in some physically reversible CMOS technology. Chapter 4 then details the particular architectural decisions and specifications of the Pendulum Reversible Processor.

A reversible processor is motivated by the result from thermodynamics that information destruction causes an unavoidable energy dissipation. A reversible computer may not destroy information. A conventional processor destroys information in three areas: memory access, datapath operations on stored values, and control flow operations.

3.1 Memory Access

Traditional load/store memory accesses assume that copying and overwriting information are free operations. Information in a reversible machine may not be destroyed, so memory access is not allowed to erase a previously stored value. During memory access in a

load/store architecture, loading a value from the memory to the register file overwrites the previously stored value of the register, and storing a register file value to memory likewise overwrites the previously stored value of the memory location. By combining load and store into a single symmetric and reversible operation, *exchange*, the information is merely moved from one place to another rather than erased. Exchange must be used to access all architecturally visible memory elements in a reversible processor. The memory hierarchy must be adapted to avoid copying information. Cache and file coherency problems do not exist because only one copy of each data word exists. Exchange operations more closely resemble physical storage systems [Bak92] such as filing cabinets, in which accessed information is only available to one process (or person) at a time. No process may share data unless an explicit copying operation is performed.

3.2 Execution Unit

3.2.1 Register File Access

The register file must not destroy information, so access must occur as an exchange. Unlike memory access, the exchange operation is split into two separate stages because the values being read are needed early in the execution of the instruction, and the result values to be written are not available until the end of execution. Each of the stages is an exchange operation itself, but the value of interest is being exchanged with an arbitrary known constant. For processor design purposes, this known constant may conveniently be defined as zero.

If a register is to be written but not read during a particular instruction, the exchange must be split over multiple instructions. The register must contain zero before a value may be written to it, so the register is first "cleared" by exchanging the register value with a memory location which is clear, i.e. guaranteed to contain zero. The result then is exchanged with the known constant just as in the case above. A data memory which is entirely clear at the start of program execution provides the supply of clear locations for this type of register access.

Figure 3-1 diagrams both stages of a register file exchange.

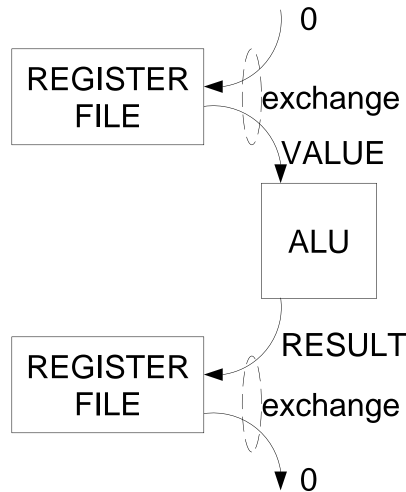

Figure 3-1: Register File Read and Write

3.2.2 Reversible and Irreversible Operations

Certain datapath functions of two operands, such as XOR, have well defined inverses which allow one operand to be reconstructed unambiguously from the result and the second operand. We call these functions reversible because they may be undone: the inputs may be reconstructed from the outputs. Other functions exhibit data-dependent reversibility. For example, summing two numbers is reversible unless the sum produces an overflow. Multiplication is also reversible unless an overflow or underflow is produced; multiplying a number by zero is reversible if the result and the non-zero operand are saved. And a few operations of interest in traditional programming languages and architectures, such as logical AND, are irreversible in the sense that the result and one operand are never sufficient to determine the second operand.

All operations performed by a reversible processor must be invertible; executing "irreversible" operations implies that additional information must be retained in order to undo the operation. The difficulty arises from the fact that a finite memory will quickly become filled if storing the results of an operation require more space than the operands used.

The processor can be structured so that manipulation of the extra information required by irreversible operations is directed by either the programmer or the processor itself. The processor may require that the programmer track the extra information and store it in memory, or the processor may store the extra information in a separate structure, a "garbage stack," (GS) automatically. If the processor uses a garbage stack, the processor controller must determine when conditionally reversible operations require that extra information must be stored. The controller must also store information which identifies the conditionally reversible instruction as having executed irreversibly. On the other hand, if the garbage information is under programmer control, the programmer may be able to perform some "garbage-collection" to re-clear memory locations. For example, if a register value has been copied, and at the end of some computation the copy is no longer needed, the original value may be subtracted from the copy and the result, zero, stored in the copy's location. Since a location which is known to be zero is defined (arbitrarily) as clear, the location has been reclaimed.

3.2.3 Operand Specifier Format

It is important for register operations to use only as many storage locations for output values as for input values. It is possible to structure the instruction set to minimize the number of datapath operations that require additional memory. This section examines general functions, denoted by $\star$, of two inputs and one output, to determine their garbage creation behavior. These register operations represent the options for a general purpose register machine.

For a general operation $\star$, four possible combinations using two sources and one destination exist. Table 3.1 details these operations and the number of word-sized storage spaces required before and after execution so that the operation is invertible when $\star$ is both reversible and irreversible.

Operations I and II require an additional storage space if $\star$ is irreversible. Operations III and IV always require one more storage space after execution than before.

Operation				before	after, ⋆ rev.	after, ⋆ irrev.
I	A	←	A ⋆ A	1	1	2
II	A	←	A ⋆ B	2	2	3
III	A	←	B ⋆ B	1	2	2
IV	A	←	B ⋆ C	2	3	3

Table 3.1: Operand Specifier Formats and Storage Requirements

Therefore, operations I and II create no garbage when performing reversible operations. Operations III and IV require that at least one extra storage location be used. For A←A⋆B operations where ⋆ is irreversible, the original value of A must be retained somehow. This suggests that the extra flexibility of the A←B⋆C operations, with the same storage requirements, might be used to execute irreversible operations rather than A←A⋆B, unless practical considerations such as datapath regularity and simplicity dictate otherwise.

3.3 Control Flow

Control flow operations pose a particularly tricky problem because they differ significantly from conventional processor operation. Jumps and branches must be invertible when running in reverse, so some program trace information must be retained. In contrast to datapath operations whose reversibility or irreversibility are independent of other instructions, the reversibility of control flow operations depends on the program structure. For example, a piece of code which may only be reached through a single unconditional jump to that location need only to store the return address of the unconditional jump. It is reversible. If multiple jumps target the same piece of code, information about which jump was taken must be stored in addition to the return address of each jump. It is irreversible, requiring that extra information be stored. Conditional branches suffer the same difficulty. The test and destination for a backward branch may not be the same as for its forward counterpart.

The essential problem is that control flow operations in the context of a conventional processor allow program execution paths to coalesce in a manner which may be irreversible. In the simplest case, two or more unconditional branches to the same location results in ambi-

guity about which path to follow in reverse. A more complex issue arises if some number of conditional branch statements all have as their destination a particular instruction address. They may have identical conditions for branching, and there is no way to compute which branch was taken (if any) to bring the program into its current state.

Consider the following instructions

```
top:  beq a,b,end       # if a = b, branch to end
      add b,a           # b = b + a
      beq a,b,end       # if a = b, branch to end
      sub a,c           # a = a − c
      beq a,b,end       # if a = b, branch to end
      add b,c           # b = b + c
end:  ...
```

Figure 3-2 diagrams this particular branch scenario. Unless information about which branch was taken is stored, the processor cannot execute the branch in reverse. A conditional branch evaluates some value(s) and changes control flow based on those values, called the branch conditional(s).

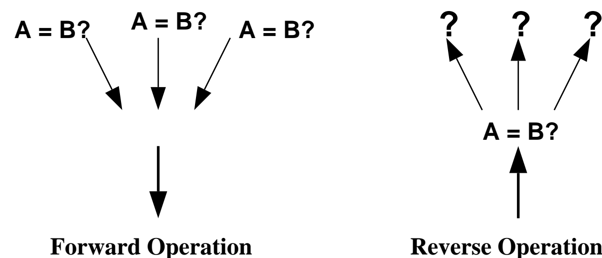

Figure 3-2: Control Flow Confluence Example

The processor should store the PC (or as much information as is required for reversibility) only when the return address cannot be calculated. The information necessary to reverse certain control flow operations may be stored in the instruction stream at compile time. This requires extra instructions and complexity beyond a standard RISC ISA. Instructions which alter control flow in one direction must have a counterpart which alters control flow in

the other direction. This does not require that time reversal symmetry be broken, however. If only reversible control flow instructions are allowed, the different instructions will be exactly symmetrical.

The amount of actual information in those PC values is fairly small. If only one branch instruction targets a particular location, only one bit need be saved to undo the branch, namely if the branch was taken or not. If this bit may be generated by evaluating the branch conditional in reverse, no garbage need be stored. An instruction which branches if a register is negative, for instance, must be paired with a backwards branch which is guaranteed to be evaluating the same register value. For a complicated branch structure this reverse evaluation may be difficult, and if the intervening code changes the register being evaluated, as in the example of section 4.2.4, the original value of the conditional needs to be copied, creating garbage.

While the program is running forward, control flow (and datapath) storage requirements monotonically increase without bound in an architecture which allows irreversible operations. The storage requirements monotonically decrease while running in reverse. This unbounded increase while running forward is undesirable. A number of optimizations are possible which encourage the use of reversible control flow operations, such as putting return addresses in the instruction stream at compile time for jumps which only go to one location. Requiring that all control flow instructions be reversible allows control flow storage requirements to remain fixed, but unlike restricting datapath operations to be exclusively reversible, restricting control flow may place too much of a burden on programmers[1].

3.4 Conclusion

Each instruction may be thought of as performing a data manipulating function and a control flow manipulating operation. Datapath instructions in a traditional RISC processor

[1]Comments referring to programmers apply to the entity which generates the assembly language code, be it a high level language optimizing compiler or a human hand-coding a routine.

may have an irreversible data manipulation effect but perform an implicit control operation, increment, which is reversible. Likewise, control instructions may have an irreversible control manipulation effect but perform an implicit data operation, read and restore, which is reversible.

A reversible architecture which only supports reversible operations in memory access, datapath operations, and control flow, does not need to retain any extra information. Since conventional processors support irreversible datapath operations and control flow, any reversible processor that is designed to resemble a conventional architecture must retain extra information. A reversible processor must retain two types of extra information that a traditional processor erases: operands used in irreversible operations and program flow information resulting from jumps and branches.

Just as the second law of thermodynamics acts as time's arrow in any irreversible process, any irreversible operation in a computing engine will break time reversal symmetry in the processor. Processor direction is distinctly identifiable, and "forward" and "reverse" execution may be discussed without ambiguity.

Chapter 4

Pendulum Processor

This chapter discusses the engineering decisions and rationale of the actual Pendulum processor design while keeping in mind the general reversible computing concepts outlined in the previous chapter. Information must not be destroyed. All memory accesses must be performed as an exchange, all data operations must retain enough information to be invertible, and enough program trace information must be retained to invert control flow operations. This design is deliberately as simple as possible, while trying to resemble the MIPS R2000 and to be technology independent. Any integrated circuit technology, reversible or not, is suitable for implementation.

The primary concern is to retain all information required so that the instruction stream may be executed in either direction. Any program may be returned to its original state by running the processor backwards.

4.1 Overview

The starting point for the Pendulum reversible processor design is a 32-bit RISC architecture, specifically that of the MIPS R2000. The simplicity of the MIPS design allows a

greater emphasis to be put on the unique features of a reversible processor. It also lets the mapping between the instruction set architecture (ISA) and the register transfer level functional architecture be simple and straightforward. The literature [PH90, PH93] contains substantial research on similar RISC architectures. Also, a RISC architecture provides a suitable starting point for implementation of a reversible pipeline for enhanced performance.

Figure 4-1 shows a functional unit level schematic of the Pendulum datapath. Control signals are not shown. Appendix B is a complete schematic generated by the CAD tool used in the design.

The current design executes instructions in five cycles; each cycle resembles an appropriate pipeline stage although the architecture is not yet pipelined. Two are dedicated to instruction fetch and decode, while the three remaining stages perform register access, operation execution or memory access, and register write back. Register access is performed in a separate stage (rather than during instruction decode, as in the MIPS R2000) so that only the registers needed during the instruction are read. When the processor changes direction, the register file read and write stages exchange functionality and the operation execute stage performs the inverse of the operation specified. Each stage performs the inverse of the function it performed running forward.

Processor direction is controlled by an external signal. The signal is synchronized with instruction execution so that the currently executing instruction completes execution before the processor direction is changed.

4.2 Pendulum Instructions

This section presents the justification for the datapath structures based on the supported Pendulum instructions. The instruction set is nearly identical to that of a conventional processor with the notable exceptions of "come-from" and "exchange."

Pendulum memory accesses are handled with the single `exchange` instruction. Pendulum

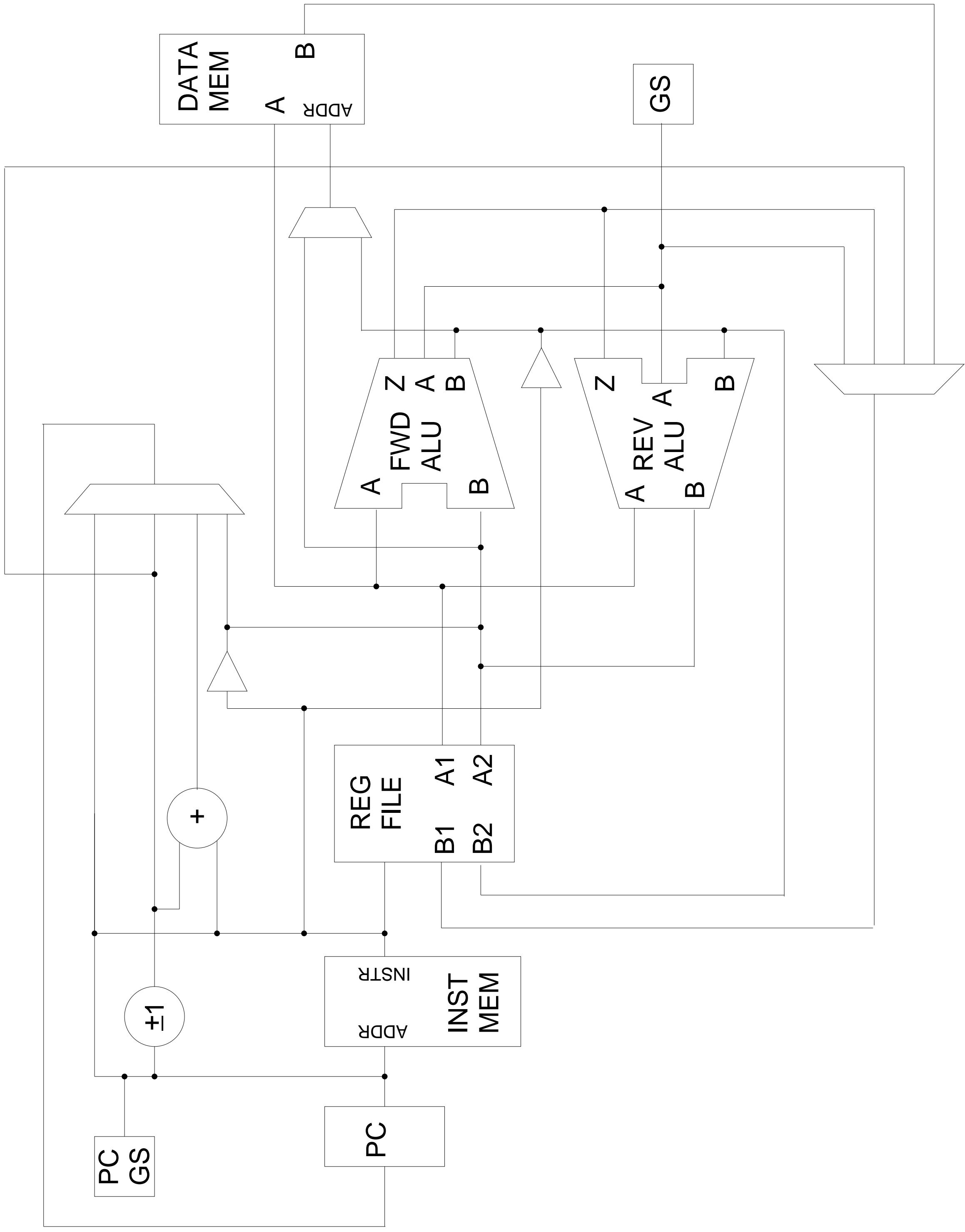

Figure 4-1: The Pendulum Datapath.

supports a full set of arithmetic and logical operations, shifts and rotates, both on register values and instruction immediates. Control flow is achieved through a number of unconditional and conditional jumps and branches, including linking jumps and jump-to-register-value.

Unlike many conventional processors, Pendulum supports rotate instructions as well as standard shift instructions. Shift is a garbage creating instruction because information is lost when bits are shifted off the end of a word. A rotate instruction transfers the bits to the other end of the word, retaining the information. If a small number is shifted left a small amount, the right (least significant) bits are filled with zeros. This is identical to a rotate operation. Likewise, a number may be shifted to the right if the low order bits are zero and the number being shifted is either non-negative or logically shifted (zero filled in the high order bits), by performing a rotate. Knowing the range of values of a number is required to replace shift with rotate, of course, but in the common case of left-shifting a small number, rotate may be used to reduce garbage.

Whenever possible, programs should be structured to make use of the non-garbage creating instructions and avoid the few irreversible operations: shifts, set-on-less-than, AND, OR, and NOR.

4.2.1 Memory Access

Memory accesses are based on a load/store system, but both a load and a store happen during each access, forming an exchange. Exchange swaps the value in a register for the value in the data memory at an address specified by another register. To operate reversibly, memory elements conform to the read/read^{-1} and write/write^{-}1 paradigm.

The Read/Read^{-1} Write/Write^{-1} Paradigm

Figure 4-2 shows the Pendulum register file and its connections. Copying information from one location to another involves losing the information which was stored at the destination.

Reading and writing information is therefore done as a swap. As mentioned in section 3.2, reading from a memory location clears it, and only memory locations which are clear[1] may be written to. This suggests that the inverse of reading a value is writing a value, and the inverse of writing a value is reading a value.

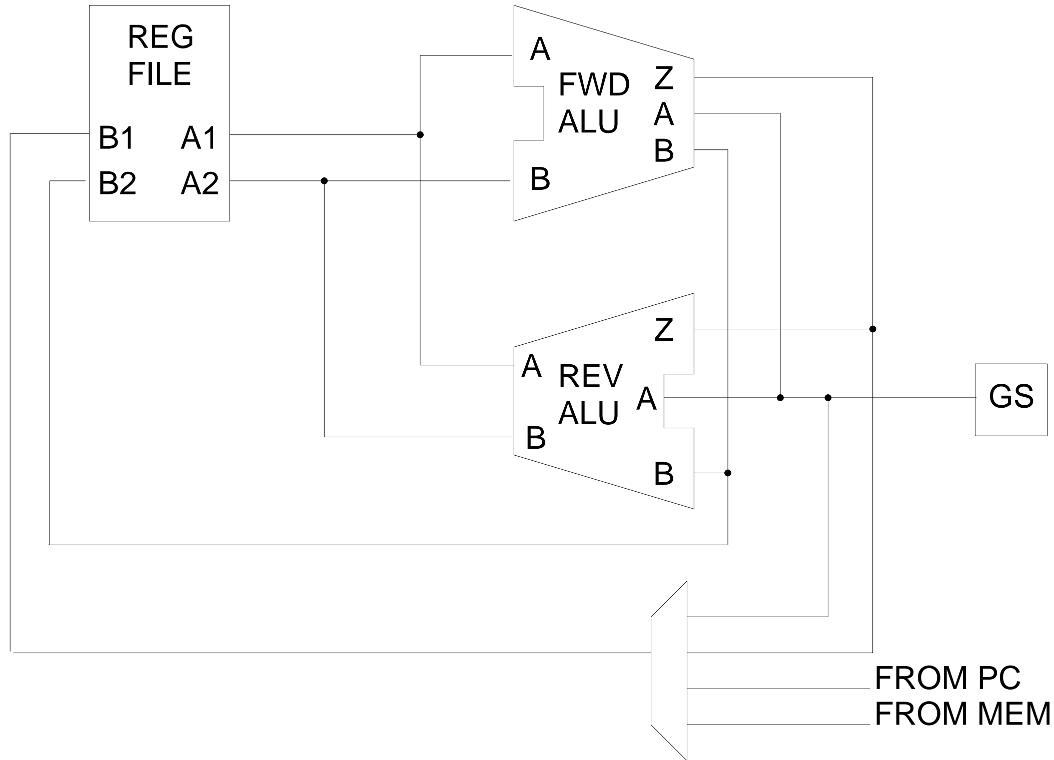

Figure 4-2: Register File Connections

Looking specifically at the register file, the processor must be able to read two and write two registers while operating in either direction, implying that the register file must have four bidirectional ports. In either direction, two are read ports and two are write ports. So when the processor is computing forward, operation proceeds as with an irreversible processor. Data is read from the ports on the right and written to the ports on the left. Then, when computation reverses, data is read (Write^{-1}) from the ports on the left and written (Read^{-1}) to the ports on the right. The terminology is used because in reverse, the processor is actually undoing the reading and writing it performed in forward execution.

[1] If a register must be cleared to receive a value being copied during program execution, the register must be exchanged with a memory location which is known to be clear.

The inverse notation more precisely describes the logical operations being performed.

This of course adds additional complexity to the register file design, requiring four bidirectional ports, but it imparts a sense of symmetry to the design and provides a reasonably clear way of thinking about memory directionality.

Address Calculation

An exchange instruction specifies two register addresses. During an exchange instruction the ALUs are inactive, and the two registers specified in the instruction are the source/destination for the swap and the memory location to be swapped. The register file B ports are read, the data address comes from B2 rather than A2, and A1 is written as B1 is swapped out. Some conventional processors calculate memory addresses by adding an offset to a base register during the memory access instruction. In Pendulum, however, any offset to be added to the address register must be specified in a separate `addi` instruction. This allows instruction execution to invert itself gracefully. If addresses were calculated during one stage and memory accessed in the next, inverse operation becomes less symmetric. The address calculation stage must still calculate an address, but the memory access stage must invert the direction of reading and writing. This mixture of inverted and non-inverted functions complicates datapath control.

4.2.2 Special Instructions

Special instructions are register to register instructions which compute arithmetic, logical, and shift-type operations. They include all instructions other than control flow, immediates, and exchange.

Register instructions of the form $A \leftarrow A \star A$ and $A \leftarrow A \star B$ require no extra storage when executing reversible operations. When executing irreversible operations, $A \leftarrow B \star C$ and $A \leftarrow A \star B$ have the same storage requirements. Supporting only $A \leftarrow A \star B$ allows the instructions to be regular, the instruction encoding to be simple, and the datapath to reflect this regularity

and simplicity.

The operands needed to reverse the computation must be retained. The irreversible operations require that the original value stored in the source/destination register be saved on the garbage stack. The reversible operations only require that the result and the second operand be stored back in the register file.

Using this instruction format dictates that the register file should have two ports in each direction. Control proceeds by first reading[2] both registers, performing the operation, and writing back the result and the second operand. In reverse, the result and second operand are read, the first operand is computed (or passed through the reverse ALU from the garbage stack to the register file) and both operands are written back to the register file. So the processor must have a multi-ported register file, two ALUs, a garbage stack, and appropriate busses connecting them.

4.2.3 Immediate Instructions

Immediate instructions face similar constraints as the special instructions. To minimize garbage creation and encourage regularity in the datapath, immediate instructions have the same format as special instructions but replace the second operand specifier with a 21-bit immediate value. Therefore, immediate-type instructions perform operations of the form $A \leftarrow A \star \text{Immediate}$.

All special instructions may be mapped to a corresponding immediate instruction by multiplexing the sign extended instruction immediate value to the ALU inputs rather than a second register value. The register address specified in the instruction is the source/destination register; the 21 bit wide immediate value replaces the value of the source register.

Just as with special instructions, the original value of A may need to be pushed onto the garbage stack. Immediate instructions do not create any garbage unless $\star$ is irreversible.

[2] Recall that reading clears the register location rather than making a copy of the value.

Running forward the register value and immediate are driven onto the forward ALU inputs, the result is computed, the original value of A is saved on the garbage stack if $\star$ is one of the irreversible operations, and the result is written to the register file. In reverse, the result is read from the register file and driven, along with the immediate value, onto the reverse ALU input. The original register value is computed or passed through from the garbage stack.

4.2.4 Control Flow Instructions

Sequentially executing code is the default control operation. Instructions which execute non-sequentially must be traced in some manner. The most direct way to achieve this is to push the program counter (PC) onto a stack whenever the PC changes non-sequentially. Pendulum treats all control flow operations as subroutine calls which must save a return address, and a single instruction, *come-from*, takes care of undoing control flow.

Come-from (CF) is actually shorthand for "Push or pop the PC garbage stack." CF executes a "push PC" during forward execution and a "pop PC" during reverse execution. This has the potential disadvantage of requiring a great deal of memory in a medium length program, since the number of executed instructions, and PC values needing to be stored, can be arbitrarily large if the dynamic execution code size is large compared to the static program size. However, future designs may store information in the instruction stream to undo control flow instructions if return addresses are known at compile time or may be computed at run time, rather than storing PC values.

Control flow instructions must target an instruction immediately following a come-from. This allows sequentially executing code to fall into a code segment which can also be jumped or branched into. If a branch was taken, the value popped when running in reverse will be the address of the branch; if not taken, the popped value will be the address of the come-from itself. Come-from instructions should only be encountered in forward execution when code is fallen into, and it should be possible to store just that single bit of information, although the current implementation stores the full PC.

The PC incrementer/decrementer lies between the PC stack and the PC itself, so when PC values are popped off the stack during reverse execution of a CF instruction, the value that is loaded into the PC is actually the value that was popped off *minus* one. This means that the jump or branch instruction which caused its address to be stored is not encountered when the instruction stream is undone. This is the symmetric case for forward execution when the jump or branch lands on the instruction at the address of the CF *plus* one.

Jump register (JR) and jump-and-link-register (JALR) must be used with great care because they give the programmer the ability to break program reversibility. The address corresponding to the value in the jump register could be anywhere. For reversibility, jumps must land on an instruction immediately following a come-from, and this condition is not guaranteed in a JR or JALR. Jump register's primary purpose is as a "return from subroutine" instruction when it is paired with a jump-and-link (JAL). But if a JR or JALR is used on its own, the programmer must be careful that the only possible destinations are instructions immediately following "come-from" instructions. The link register for the PC in a JAL or JALR must be clear before the JAL or JALR executes.

It is also interesting to note that jumps and branches have no function when encountered while running in reverse, and may therefore be treated as NOPs. If a conditional branch is reached when running backward it implicitly means that the branch was not taken and control proceeded sequentially; in other words, the branch was a NOP when it was run forward. And if a come-from is encountered in reverse operation, it implies either that the branch was taken, and the come-from is its symmetric partner, or that the CF was seen while running forward and falling into a code segment.

For example, the following pseudo-code

```
if (a > 0) a = a-10 else a = a-5;
```

compiles to

```
top:  bgtz a,mid        # branch greater than zero; if a > 0
      subi a,5          # a = a − 5
```

```
jad:  j end           # jump end
      cf              # come-from only encountered when running in reverse
mid:  subi a,10       # a = a − 10
      cf              # come from, paired with end:
end:  ...
```

Forward execution proceeds as follows: First, the comparison is made. If the branch is not taken, five is subtracted from `a` and control jumps (pushing the jump address `jad` onto the PC stack) to `end` and execution continues normally. If the branch is taken, `top` is pushed onto the PC stack, the PC is loaded with `mid`, ten is subtracted from `a`, `mid+1` is pushed onto the stack by the `cf` since control is falling into a section of code that is also a jump/branch destination, and execution continues from `end`.

Assuming the PC is beyond `end` and Pendulum is running in reverse, execution reverses as follows: The `cf` above `end` pops a PC value off the stack. This value will be either `mid+1` or `jad`. If it is `mid+1`, the PC is loaded with `mid`, ten will be added to `a`, the next `cf` will pop off another PC value, this time `top`, and the PC is loaded with `top-1`. Program reversal will continue above `top`.

If the first PC value popped off was `jad`, the PC is loaded with `jad-1`, five will be added to `a`, the `bgtz` is a NOP, and execution continues normally above `top`.

Chapter 5

Future Work

Reversible computer architecture has received scant previous consideration. The theoretical discussions of reversible computation in the literature [Ben88, Lan86] leave much work to be done before a reversible computing engine may be built. This section discusses some of the architectural issues which have yet to be resolved. Questions of implementation details, such as SCRL circuit realizations of functional blocks, clearly present substantial opportunities for future work, but are beyond the scope of this thesis.

5.1 Input/Output Behavior

Input and output operations generally occur as copy operations and must therefore occur dissipatively, assuming that a copy overwrites previously stored information. Irreversible I/O events should therefore be made infrequent compared to reversible computation events, but whenever possible, the environment should be configured to support reversible operations.

Input Program and data information are often transmitted to the processor's memory through a copy from some non-volatile memory. A program stored on a magnetic or optical device is copied many times and to many locations (assuming some sharing of resources through parallel computing or broad network access) and must be done so dissipatively, since we assume the processor must overwrite the previously stored program and data to load the new information. The energy dissipation cost of the copy must be amortized over the time that the processor is executing the program. A method of maximizing the information content of each bit, perhaps through compressing data before loading into a processor, will reduce dissipation associated with input. Whenever possible, exchange should be used to input data.

Output If output is performed as a standard memory access, an exchange, the output device must have some way of returning the information to the processor when the direction is changed. The actual act of producing output is, however, often a copy. Any irretrievable information which is output must dissipate energy.

Output to a network or a mass storage device is also usually a copy operation. If it occurs irreversibly and dissipatively, the only way to decrease the energy dissipated for some quantity of information to be output (once the output technology has been optimized) is to maximize the information content of every bit. Again, as many I/O operations as possible should be performed as an exchange to avoid dissipation.

5.2 Pipelining

Future implementations will almost certainly include some form of pipelining to enhance performance. Hazard detection is traditionally the most complex issue of pipelining, and reversible pipelining involves not only the issues of hazard detection of results, both in the forward and reverse directions, but since reading from the register file is destructive, operands must be forwarded as well. It is not obvious how this should happen.

The current Pendulum execution stages are, in order, instruction fetch (IF), instruction decode (ID), register read (REG), execute and memory access (EXM), and register write back (WB). During reverse operation, WB performs a write^{-1}, reading the result and one operand from the register file, EXM performs the inverse operation of the forward instruction, and REG performs a read^{-1}, writing back the original operands. In reverse these instruction stages invert their operation and execution reverses as IF, ID, WB, EXM, REG. These instruction execution stages may be converted into pipeline stages with the addition of pipeline registers and forwarding support.

An enhancement which slightly complicates reversible pipelining is forming addresses for the exchange operation by adding an offset, specified in the instruction, to the value in a base register. The memory access is pushed into its own execution stage, MEM, and the address calculation is performed in EX (which was EXM). This is a performance enhancement intended to speed series of memory accesses which exhibit spatial locality. Using the current exchange instruction, the memory address must be calculated in one instruction and exchanged in the next, so a series of memory accesses, such as indexing into an array, each require two instructions. Using the enhancement, each access would only require one instruction once the base address had been calculated.

If addresses are to be calculated as an offset from a register, the forward stages required are IF, ID, REG, EX, MEM, WB. Reverse execution must be IF, ID, WB, EX, MEM, REG. Bussing structures and pipeline registers are likely to be complex. Forwarding in both directions through six pipeline stages also presents a formidable challenge.

To execute such an instruction in reverse requires that the address be computed by the reverse ALU, so the outputs of both the forward and reverse ALUs must be connected to the memory address lines, and the controller must direct the reverse ALU to perform an add operation rather than an add^{-1}.

All pipeline stages must undo their actions; reads become writes and vice versa, memory accesses are undone, and inverse operations are performed on register values. The pipeline does not necessarily execute the stages in reverse order; it executes the inverse of each stage

in the same order as in forward execution. The inverse of WB is REG, but the inverse of EX is EX. Instructions are undone one after the other in reverse order, but the process of undoing an instruction consists of performing the inverse of each of the steps of instruction execution. Clearly, the instruction must be fetched and decoded when running in reverse. Then the effects of the instruction are undone by performing the inverse function of the remaining stages.

5.3 Instruction Set Expansion

This section addresses enhancements to the instruction set (additional instructions and added functionality for the currently supported instructions) which are likely to provide greater programming power and flexibility in future designs. Modifications to the instruction set which are intended to reduce garbage creation are covered in section 5.4.

The current instruction set supports very few instructions, and indeed the instruction set is deliberately very similar to the MIPS R2000. The primary distinction between standard RISC processors and Pendulum is the restriction on register specifiers, enforced to ensure that memory locations are read (and cleared) before they are written. A truly general machine should support instructions of the form $A \leftarrow B \star C$ where A, B, and C need not be three different registers. These types of general operations are apt to create garbage, and only the experience to be gained from profiling the behavior of different programs can direct garbage reduction efforts. Supporting general operations such as this may be equivalent to giving the programmer more rope with which to hang himself, and may therefore be a “misfeature.”

No binary function of two inputs and two outputs is both reversible and universal. An expansion to $A \leftarrow B \star C$ instructions opens the possibility of two input, three output operations such as the Feynman gate, which takes B and C as inputs and returns $\overline{B} \cdot C$, $B \cdot C$, and B, and is both reversible and universal. When executing the Feynman gate function, only one of the operands is restored, which may be undesirable, and register A must be clear before

the operation is executed, but two logical operations are performed at once. Reversing the Feynman gate function requires that all three registers be read, increasing bus complexity, but supporting this function maintains the logical completeness of the instruction set without requiring that any irreversible instructions be supported.

5.4 Garbage Reduction

A significant goal of instruction set expansion, beyond increasing the computational power and flexibility of the architecture, is the reduction of garbage. If clever programming techniques can be used to reduce the amount of garbage created by reusing results, it may be the correct design decision to provide programmer access to all information and eliminate the garbage stacks entirely. This may be possible by enforcing stricter requirements on control flow operations so that return addresses may be calculated rather than stored and retrieved, and by eliminating the irreversible instructions from the instruction set. Instruction set completeness must be maintained, however, through the inclusion of universal, reversible operations such as the Feynman gate function.

Conditionally reversible instructions may still require a garbage stack. If an overflow occurs during an arithmetic instruction, for example, the processor must save this information. Putting that information under programmer control may be undesirable or difficult, so the garbage stack structure may need to be retained.

Garbage is currently created through three processes: executing irreversible operations, changing the program counter non-sequentially, and computing intermediate results. The first two types are saved on the garbage stacks while the third type, programmer defined garbage, is just a reclassification of data memory values.

5.4.1 Datapath Garbage

The way to reduce, indeed eliminate, the amount of datapath garbage created during ALU operations is simply to disallow irreversible operations. Reversible rotate instructions may replace irreversible shifts for a number of cases. If logical AND, OR and NOR instructions can be eliminated by supporting reversible versions, and if the set-on-less-than instructions can be replaced by conditional branches, the datapath need not create any explicit garbage.

Since the Feynman gate function and certain other boolean functions of two inputs and three outputs are both reversible and universal, the irreversible instructions need not be supported, (after suitable datapath modifications to support mappings of two inputs to three outputs) and datapath created garbage may be eliminated. The tradeoff is that programming styles must be adapted to these instruction changes.

5.4.2 Control Flow Garbage

The reduction of control flow garbage in the standard programming idioms is crucial for the future of reversible computing. Otherwise reversible computers are an engine for creating garbage with computation as a side effect. Instructions designed to reduce control flow related garbage, especially through unrolling loops and calling and returning from subroutines, will almost certainly be added in future processors. A set of "backward-jump-register" and "backward-jump-and-link" instructions, if properly implemented, can make certain types of subroutine calls garbageless since the information required to return from the subroutine is known at compile time and can be stored in the instruction stream.

If dynamic code size is much larger than static code size, the space penalty paid in static program size for including extra instructions is small, and the primary cost is a time penalty of executing more instructions. This time/space tradeoff may be worthwhile if garbage creation is a more significant problem than execution time. The goal is to add instructions which take advantage of information already in the instruction stream and register values to reduce the number of PC values which need to be kept during execution of a potentially

large number of jumps and branches.

5.4.3 Programmer Defined Garbage

This is the most significant area in need of consideration. Effort has been made in this architecture to keep the explicit garbage stacks small, but the programmer must manage a large amount of information. In the case of summing some large set of numbers, all the numbers must be retained. Under some conditions this may be desirable; if the records are to be retained anyway, such as in banking transactions, no additional imposition is placed on the system. If the program requires only that the result be retained, the summands represent programmer defined garbage. The issue of what to do with the potentially large amount of garbage is unsolved. It may be the case that reversible computing is only a viable solution to problems which require that intermediate results be retained. Or, at some point in the future, technology advancements may make energy dissipation much more costly than memory, at which point reversible computing becomes very attractive. Or it may be that programmers are clever enough to reclaim enough of the memory locations that programmer defined garbage is rarely created. Exploration of these issues is a substantial research topic.

Another approach to all three garbage creation problems is to execute the program forward until the desired result is achieved, then dissipatively copy the result in some manner, then execute in reverse, clearing its state. A new program could then be loaded which may use the result of the previous program. This assumes that each program has some terminating condition and a clearly defined result. But if program loads are performed as an exchange, this technique cleans up its own garbage and dissipates only during output of a result. The penalty is a factor of two time penalty.

5.4.4 Bit Erasure

The current implementation keeps garbage information explicitly distinct from the program execution information, such as register file and data memory values. This extra information may be compressed or manipulated or made physically distinct from the bulk of the computing engine, or moved to a different location for erasure. Dissipating energy by destroying bits in a remote location is not useful for reducing energy consumption of a processor, but it is important if heat removal is a limiting factor in the packing density of computing elements. A processor operating at finite speed must dissipate some amount of heat due to resistive losses, and increasing processor speed increases this resistive dissipation. If resistive losses are small compared to the energy cost of bit erasure, moving bit erasure to some other location allows the computing device to run cooler, or faster for a fixed temperature. A thermostatically controlled clock set to the optimum operating temperature will keep the processor computing as fast as it can, while bits may be erased in some physically removed structure specially designed for heat removal and not actively involved in computing.

Chapter 6

Conclusions

This thesis has shown the detailed architectural and register transfer level implementation design for a reversible processor. The challenge of retaining enough information to invert an instruction stream and designing the assembly language to resemble a standard processor has been met.

Extensive and ongoing simulations of the architecture have been successful in demonstrating functionality. The processor direction signal may be changed at any point during program execution and instructions will be undone. The signal may be changed again and the processor will execute the instruction forward.

Appendix A

The Pendulum Assembly Language

Assembly language programming on the Pendulum processor greatly resembles a standard RISC architecture [KH92]. 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 is insulated from many aspects of the reversible operation going on inside the processor. The programmer must be aware of two main 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]. And second, all jumps and branches must target an instruction which is immediately preceded by a "come-from."

Pendulum supports a full set of jumps and conditional branches including linking instructions. In the current implementation, instructions are not pipelined. Instruction execution takes five clock cycles, each cycle performing operations which map onto what could become

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

pipeline stages, but to avoid the added complexity in both datapath design and instruction scheduling, instructions execute one at a time. The datapath is simplified by not having pipeline registers and because forwarding hardware is unnecessary. And in the interest of simpler programming, single instruction execution allows the elimination of delay slots and data hazards.

Special-type instructions, so named because a single, special opcode passes instruction differentiation to the `func` field, include all register to register operations, including logical, arithmetic, shift, and rotate. They take one or two register addresses and a shift amount, if necessary. For two-register instructions, such as `ADD`, the two registers must be different. This is very different from other architectures, but the inconvenience for the programmer should be small.

The following pages contain the syntax and description for forward execution of all Pendulum Assembly Language instructions. No exceptions are generated during any instructions. All jump and branch instructions push the address of the current instruction (the jump or branch) onto the program counter garbage stack if the branch is taken. Conditional branches which are not taken produce no garbage. Other irreversible instructions which produce datapath garbage are noted below.

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

A.1 Instruction Set Encoding

Since one of the earliest design decisions was to base Pendulum on a 32 bit RISC machine, the instruction word encoding strongly resembles the MIPS R2000 instruction word encoding. But, since certain types of register operations are disallowed, only two registers need to be specified: a source/destination and a source. This means the instruction word has five "extra" bits available. Rather than shorten the instruction bus width, the word size remains 32 bits long so that the architecture does not starve for address bits too quickly.

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.
$+$	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
GS	Datapath Garbage Stack
PC	Program Counter
PCGS	Program Counter Garbage Stack
MEM[x]	Memory Location x

Table A.1: Instruction Operation Notations

The opcode field is limited to six bits so that the immediate field can be large, but the special instruction `func` field, which specifies ALU and shift/rotate[2] operations, may be spread out to eleven bits and only require a slight encoding. The ALU and shifter unit support ten arithmetic and ten shift/rotate instructions, so the eleventh bit of the `func` field determines which type of special instruction is being evaluated, and the remaining ten bits are "one-hot" encoded so that the ALU need do little decoding. The instruction set is limited to 64 types of instructions with the `func` bits specifying the operation for special instructions.

Pendulum uses four types of instruction encodings, listed in Table A.2.

Special instruction types and the exchange instruction use an R-type encoding. The instruction word specifies a source/destination register, a source register, a shift or rotate amount, and type type of special operation to be performed. The `func` and `sh/rot` field

[2]The cost of having rotate as well as shift is essentially just a mux tacked on to a basic funnel shifter. A good compiler should be able to take advantage of the non-garbage-creating rotate.

are specified to be zero for exchange operations.

R-type	`op`	`rsd`	`rs`	`sh/rot`	`func`
	6 bits	5 bits	5 bits	5 bits	11 bits

J-type	`j/cf`	`target`
	6 bits	26 bits

B-type	`j/b op`	`ra`	`rb`	`offset`
	6 bits	5 bits	5 bits	16 bits

I-type	`op`	`rsd`	`immediate`
	6 bits	5 bits	21 bits

Table A.2: Instruction Formats

The unconditional jump and come-from instruction J-type encoding specifies an opcode and a target. The target field in the come-from instruction is specified as all zero.

Jump instructions other than `j` and all conditional branches use B-type instruction encoding and specify two registers and an offset.

Immediate instructions, I-type, specify one register and a 21 bit signed or unsigned, depending on opcode, immediate value.

Instruction mnemonic: ADD

Instruction name: Add

SPECIAL 000000	rsd	rs	0 00000	ADD 000 0000 0001
6 bits	5 bits	5 bits	5 bits	11 bits

Format: ADD rsd, rs

Description:

The contents of register *rsd* and register *rs* are added to form a 32-bit result. The result is placed in register *rsd*.

Operation:

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

Instruction Mnemonic: ADDI

Instruction Name: Add Immediate

ADDI 011000	rsd	immediate
6 bits	5 bits	21 bits

Format: ADDI rsd, immediate

Description:

The 21-bit *immediate* is sign extended and added to the contents of register *rsd* to form a 32-bit result. The result is placed in register *rsd*.

Operation:

$GPR[rsd] \leftarrow GPR[rsd] + (immediate_{15})^{16} \| \, immediate_{15..0}$

Instruction Mnemonic: AND

Instruction Name: And

SPECIAL 000000	rsd	rs	0 00000	AND 000 0001 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: AND rsd, rs

Description:

The contents of register *rsd* and register *rs* are combined in a bit-wise logical AND operation. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage

stack.

Operation:

GS ← GPR[rsd]

GPR[rsd] ← GPR[rsd] *and* GPR[rs]

Instruction Mnemonic: ANDI

Instruction Name: And Immediate

ANDI 011100	rsd	immediate
6 bits	5 bits	21 bits

Format: ANDI rsd, immediate

Description:

The 21-bit *immediate* is sign extended and combined with the contents of register *rsd* in a bit-wise logical AND operation. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS ← GPR[rsd]

GPR[rsd] ← GPR[rsd] *and* $(\text{immediate}_{15})^{16} \| \text{immediate}_{15..0}$

Instruction Mnemonic: BEQ

Instruction Name: Branch On Equal

BEQ 001001	ra	rb	offset
6 bits	5 bits	5 bits	16 bits

Format: BEQ ra, rb, offset

Description:

The 16-bit *offset* is sign extended and added to the address of the instruction following the current instruction to form a target address. The contents of register *ra* and *rb* are compared. If the two registers are equal, the program branches to the target address.

Operation:

if GPR[ra] = GPR[rb] then

  PC $\leftarrow$ PC + 1 + $(\text{offset}_{15})^{16} \| $ offset

endif

Instruction Mnemonic: BGEZ

Instruction Name: Branch on Greater Than or Equal to Zero

BGEZ	0	rb	offset
000110	00000		
6 bits	5 bits	5 bits	16 bits

Format: BGEZ rb, offset

Description:

The 16-bit *offset* is sign extended and added to the address of the instruction following the current instruction to form a target address. If the contents of register *rb* have the sign bit cleared, the program branches to the target address.

Operation:

if $\text{GPR[rb]}_{31} = 0$ then

  PC $\leftarrow$ PC + 1 + $(\text{offset}_{15})^{16} \| $ offset

endif

Instruction Mnemonic: BGEZAL

Instruction Name: Branch On Greater Than or Equal to Zero and Link

BGEZAL 001000	link	rb	offset
6 bits	5 bits	5 bits	16 bits

Format: BGEZAL link, rb, offset

Description:

The 16-bit *offset* is sign extended and added to the address of the instruction following the current instruction to form a target address. If the contents of register *rb* have the sign bit cleared, the program branches to the target address, and the address of the instruction following the current instruction is placed in register *link*.

Operation:

if $GPR[rb]_{31} = 0$ then

$PC \leftarrow PC + 1 + (offset_{15})^{16} \| offset$

$GPR[link] \leftarrow PC + 1$

endif

Instruction Mnemonic: BGTZ

Instruction Name: Branch On Greater Than Zero

BGTZ 001100	0 00000	rb	offset
6 bits	5 bits	5 bits	16 bits

Format: BGTZ rb, offset

Description:

The 16-bit *offset* is sign extended and added to the address of the instruction following the current instruction to form a target address. If the contents of register *rb* have the sign bit cleared and are not equal to zero, the program branches to the target address.

Operation:

if $(\mathrm{GPR[rb]}_{31} = 0)$ and $(\mathrm{GPR[ra]} \neq 0^{32})$ then

  $\mathrm{PC} \leftarrow \mathrm{PC} + 1 + (\mathrm{offset}_{15})^{16} \| \mathrm{offset}$

endif

Instruction Mnemonic: BLEZ

Instruction Name: Branch On Less Than or Equal to Zero

BLEZ 001011	0 00000	rb	offset
6 bits	5 bits	5 bits	16 bits

Format: BLEZ rb, offset

Description:

The 16-bit *offset* is sign extended and added to the address of the instruction following the current instruction to form a target address. If the contents of register *rb* have the sign bit set or are equal to zero, the program branches to the target address.

Operation:

if $(\mathrm{GPR[rb]}_{31} = 1)$ or $(\mathrm{GPR[rb]} = 0^{32})$ then

  $\mathrm{PC} \leftarrow \mathrm{PC} + 1 + (\mathrm{offset}_{15})^{16} \| \mathrm{offset}$

endif

Instruction Mnemonic: BLTZ

Instruction Name: Branch On Less Than Zero

BLTZ 000101	0 00000	rb	offset
6 bits	5 bits	5 bits	16 bits

Format: BLTZ rb, offset

Description:

The 16-bit *offset* is sign extended and added to the address of the instruction following the current instruction to form a target address. If the contents of register *rb* have the sign bit set the program branches to the target address.

Operation:

if $\text{GPR[rb]}_{31} = 1$ then

  $\text{PC} \leftarrow \text{PC} + 1 + (\text{offset}_{15})^{16} \| \text{offset}$

endif

Instruction Mnemonic: BLTZAL

Instruction Name: Branch On Less Than Zero and Link

BLTZAL 000111	link	rb	offset
6 bits	5 bits	5 bits	16 bits

Format: BLTZAL link, rb, offset

Description:

The 16-bit *offset* is sign extended and added to the address of the instruction following the current instruction to form a target address. If the contents of register *rb* have the sign bit

set the program branches to the target address, and the address of the instruction following the current instruction is placed in register *link*.

Operation:

if $GPR[rb]_{31} = 1$ then

  $PC \leftarrow PC + 1 + (offset_{15})^{16} \| offset$

  $GPR[link] \leftarrow PC + 1$

endif

Instruction Mnemonic: BNE

Instruction Name: Branch On Not Equal

BNE 001010	ra	rb	offset
6 bits	5 bits	5 bits	16 bits

Format: BNE ra, rb, offset

Description:

The 16-bit *offset* is sign extended and added to the address of the instruction following the current instruction to form a target address. The contents of register *ra* and *rb* are compared. If the two registers are not equal, the program branches to the target address.

Operation:

if $GPR[ra] \neq GPR[rb]$ then

  $PC \leftarrow PC + 1 + (offset_{15})^{16} \| offset$

endif

Instruction Mnemonic: CF

Instruction Name: Come-from

CF 001101	0 00 0000 0000 0000 0000 0000 0000
6 bits	26 bits

Format: CF

Description:

The address of the current instruction is saved on the PC garbage stack.

Operation:

PCGS $\leftarrow$ PC

Instruction Mnemonic: EXCHANGE

Instruction Name: Exchange

EXCHANGE 101000	exch	addr	0 0000 0000 0000 0000
6 bits	5 bits	5 bits	16 bits

Format: EXCHANGE 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: J

Instruction Name: Jump

JUMP 000001	target
6 bits	26 bits

Format: J target

Description:

The 26-bit target is combined with the high order six bits of the address of the current instruction. The program unconditionally jumps to this calculated address.

Operation:

PC $\leftarrow$ $PC_{31..26} ||$ target

Instruction Mnemonic: JAL

Instruction Name: Jump and Link

JAL 000011	link	0 00000	offset
6 bits	5 bits	5 bits	16 bits

Format: JAL link, offset

Description:

The 16-bit *offset* is sign extended and added to the address of the instruction following the current instruction to form a target address. The program unconditionally jumps to this calculated address, and the address of the instruction following the current instruction is placed in register *link*.

Operation:

GPR[link] $\leftarrow$ PC + 1

$PC \leftarrow PC + 1 + (offset_{15})^{16} \| offset$

Instruction Mnemonic: JALR

Instruction Name: Jump and Link Register

JALR 000100	link	jreg	0 0000 0000 0000 0000
6 bits	5 bits	5 bits	16 bits

Format: JAL link, jreg

Description:

The program unconditionally jumps to the address contained in general register *jreg*, and the address of the instruction following the current instruction is placed in register *link*. The contents of register *jreg* must specify the address of an instruction which immediately follows a CF.

Operation:

$GPR[link] \leftarrow PC + 1$

$PC \leftarrow GPR[jreg]$

Instruction Mnemonic: JR

Instruction Name: Jump Register

JR 000010	0 00000	jreg	0 0000 0000 0000 0000
6 bits	5 bits	5 bits	16 bits

Format: JR jreg

Description:

The program unconditionally jumps to the address contained in general register *jreg*. The contents of register *jreg* must specify the address of an instruction which immediately follows a CF.

Operation:

PC $\leftarrow$ GPR[jreg]

Instruction mnemonic: NOR

Instruction name: Nor

SPECIAL 000000	rsd	rs	0 00000	NOR 000 1000 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: NOR rsd, rs

Description:

The contents of register *rsd* and register *rs* are combined in a bit-wise logical NOR operation. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS $\leftarrow$ GPR[rsd]

GPR[rsd] $\leftarrow$ GPR[rsd] *nor* GPR[rs]

Instruction mnemonic: NEG

Instruction name: Two's complement negation

SPECIAL 000000	rsd	rs	0 00000	NEG 001 0000 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: NEG rsd

Description:

The contents of register *rsd* are inverted in a two's complement negation. The result is placed in register *rsd*.

Operation:

GPR[rsd] $\leftarrow$ 0 $-$ GPR[rsd]

Instruction mnemonic: OR

Instruction name: Or

SPECIAL 000000	rsd	rs	0 00000	OR 000 0010 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: OR rsd, rs

Description:

The contents of register *rsd* and register *rs* are combined in a bit-wise logical OR operation. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS $\leftarrow$ GPR[rsd]

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

Instruction Mnemonic: ORI

Instruction Name: Or Immediate

ORI 011101	rsd	immediate
6 bits	5 bits	21 bits

Format: ORI rsd, immediate

Description:

The 21-bit *immediate* is sign extended and combined with the contents of register *rsd* in a bit-wise logical OR operation. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

$GS \leftarrow GPR[rsd]$

$GPR[rsd] \leftarrow GPR[rsd]$ *or* $(\text{immediate}_{15})^{16} \| \text{ immediate}_{15..0}$

Instruction mnemonic: RL

Instruction name: Rotate Left

special 000000	rsd	0 00000	amt	RL 100 0100 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: RL rsd, amt

Description:

The contents of register *rsd* are rotated left by *amt* bits. The result is placed in register *rsd*.

Operation:

$GPR[rsd] \leftarrow GPR[rsd]_{31-amt..0} \| \text{ } GPR[rs]_{31..(31-amt+1)}$

Instruction mnemonic: RLV

Instruction name: Rotate Left Variable

special	rsd	rs	0	RLV
000000			00000	101 0000 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: RLV rsd, rs

Description:

The contents of register *rsd* are rotated left by the number of bits specified by the low order five bits of the contents of register *rs*. The result is placed in register *rsd*.

Operation:

amt $\leftarrow$ GPR[rs]$_{4..0}$

GPR[rsd] $\leftarrow$ GPR[rsd]$_{31-amt..0}$|| GPR[rs]$_{31..(31-amt+1)}$

Instruction mnemonic: RR

Instruction name: Rotate Right

special	rsd	0	amt	RR
000000		00000		100 1000 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: RR rsd, amt

Description:

The contents of register *rsd* are rotated right by *amt* bits. The result is placed in register *rsd*.

Operation:

GPR[rsd] $\leftarrow$ GPR[rsd]$_{amt-1..0}$|| GPR[rs]$_{31..amt}$

Instruction mnemonic: RRV

Instruction name: Rotate Right Variable

special	rsd	rs	0	RRV
000000			00000	110 0000 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: RRV rsd, rs

Description:

The contents of register *rsd* are rotated right by the number of bits specified by the low order five bits of the contents of register *rs*. The result is placed in register *rsd*.

Operation:

amt $\leftarrow$ GPR[rs]$_{4..0}$

GPR[rsd] $\leftarrow$ GPR[rsd]$_{amt-1..0}$|| GPR[rs]$_{31..amt}$

Instruction mnemonic: SLL

Instruction name: Shift Left Logical

special	rsd	0	amt	SLL
000000		00000		100 0000 0001
6 bits	5 bits	5 bits	5 bits	11 bits

Format: SLL rsd, amt

Description:

The contents of register *rsd* are shifted left by *amt* bits, inserting zeros into the low order bits. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS $\leftarrow$ GPR[rsd]

GPR[rsd] $\leftarrow$ GPR[rsd]$_{31-amt..0}$|| 0^{amt}

Instruction mnemonic: SLLV

Instruction name: Shift Left Logical Variable

special 000000	rsd	rs	0 00000	SLLV 100 0000 1000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: SLLV rsd, rs

Description:

The contents of register *rsd* are shifted left by the number of bits specified by the low order five bits of the contents of register *rs*, inserting zeros into the low order bits. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS $\leftarrow$ GPR[rsd]

amt $\leftarrow$ $\text{GPR[rs]}_{4..0}$

GPR[rsd] $\leftarrow$ $\text{GPR[rsd]}_{31-amt..0} \| \; 0^{amt}$

Instruction mnemonic: SLT

Instruction name: Set on Less Than

SPECIAL 000000	rsd	rs	0 00000	SLT 010 0000 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: SLT rsd, rs

Description:

The contents of register *rsd* and *rs* are compared. Considering both quantities as signed 32-bit integers, if the contents of register *rsd* are less than the contents of register *rs*, the result is set to one. Otherwise the result is set to zero. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS $\leftarrow$ GPR[rsd]

if GPR[rsd] $<$ GPR[rs] then

GPR[rsd] = $0^{31} \| 1$

else

GPR[rsd] = 0^{32}

endif

Instruction mnemonic: SLTI

Instruction name: Set on Less Than Immediate

SLTI 011010	rsd	immediate
6 bits	5 bits	21 bits

Format: SLTI rsd, immediate

Description:

The 21-bit *immediate* is sign extended and compared to the contents of register *rsd*. Considering both quantities as signed 32-bit integers, if the contents of register *rsd* are less than the sign-extended immediate, the result is set to one. Otherwise the result is set to zero. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS $\leftarrow$ GPR[rsd]

if GPR[rsd] $<$ $(\text{immediate}_{15})^{16} \| \text{immediate}_{15..0}$ then

GPR[rsd] = $0^{31} \| 1$

else

GPR[rsd] = 0^{32}

endif

Instruction mnemonic: SRA

Instruction name: Shift Right Arithmetic

special	rsd	0	amt	SRA
000000		00000		100 0000 0100
6 bits	5 bits	5 bits	5 bits	11 bits

Format: SRA rsd, amt

Description:

The contents of register *rsd* are shifted right by *amt* bits, sign extending the high order bits. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS ← GPR[rsd]

GPR[rsd] ← $(\mathrm{GPR[rsd]}_{31})^{amt} \| \mathrm{GPR[rsd]}_{31..amt}$

Instruction mnemonic: SRAV

Instruction name: Shift Right Arithmetic Variable

special	rsd	rs	0	SRAV
000000			00000	100 0010 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: SRAV rsd, rs

Description:

The contents of register *rsd* are shifted right by the number of bits specified by the low order five bits of the contents of register *rs*, sign extending the high order bits. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS $\leftarrow$ GPR[rsd]

amt $\leftarrow$ GPR[rs]$_{4..0}$

GPR[rsd] $\leftarrow$ (GPR[rsd]$_{31}$)^{amt}|| GPR[rsd]$_{31..amt}$

Instruction mnemonic: SRL

Instruction name: Shift Right Logical

special	rsd	0	amt	SRL
000000		00000		100 0000 0010
6 bits	5 bits	5 bits	5 bits	11 bits

Format: SRL rsd, amt

Description:

The contents of register *rsd* are shifted right by *amt* bits, inserting zeros into the high order bits. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS $\leftarrow$ GPR[rsd]

GPR[rsd] $\leftarrow$ 0^{amt}|| GPR[rsd]$_{31..amt}$

Instruction mnemonic: SRLV

Instruction name: Shift Right Logical Variable

special	rsd	rs	0	SRLV
000000			00000	100 0001 0000
6 bits	5 bits	5 bits	5 bits	11 bits

Format: SRLV rsd, rs

Description:

The contents of register *rsd* are shifted right by the number of bits specified by the low order five bits of the contents of register *rs*, inserting zeros into the high order bits. The result is placed in register *rsd*. The original value of register *rsd* is stored on the garbage stack.

Operation:

GS $\leftarrow$ GPR[rsd]

amt $\leftarrow$ GPR[rs]$_{4..0}$

GPR[rsd] $\leftarrow 0^{amt}||$ GPR[rsd]$_{31..amt}$

Instruction mnemonic: SUB

Instruction name: Subtract

SPECIAL	rsd	rs	0	SUB
000000			00000	000 0000 0100
6 bits	5 bits	5 bits	5 bits	11 bits

Format: SUB rsd, rs

Description:

The contents of register *rs* are subtracted from the contents of register *rsd* to form a 32-bit result. The result is placed in register *rsd*.

Operation:

GPR[rsd] $\leftarrow$ GPR[rsd] $-$ GPR[rs]

Instruction mnemonic: XOR

Instruction name: Exclusive Or

SPECIAL 000000	rsd	rs	0 00000	XOR 000 0100 0000
6 bits	5 bits	5 bits	5 bits	11 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] ← GPR[rsd] *xor* GPR[rs]

Instruction Mnemonic: XORI

Instruction Name: Exclusive Or Immediate

XORI 011110	rsd	immediate
6 bits	5 bits	21 bits

Format: XORI rsd, immediate

Description:

The 21-bit *immediate* is sign extended and combined with the contents of register *rsd* in a bit-wise logical exclusive OR operation. The result is placed in register *rsd*.

Operation:

GPR[rsd] ← GPR[rsd] *xor* $(\text{immediate}_{15})^{16} \| \text{immediate}_{15..0}$

Appendix B

Detailed Datapath Schematic

The following figure is a printout of the Pendulum datapath showing all control signals, data busses, and functional units. It is taken from the CAD package used to design Pendulum. Each block is described by a Verilog HDL module.

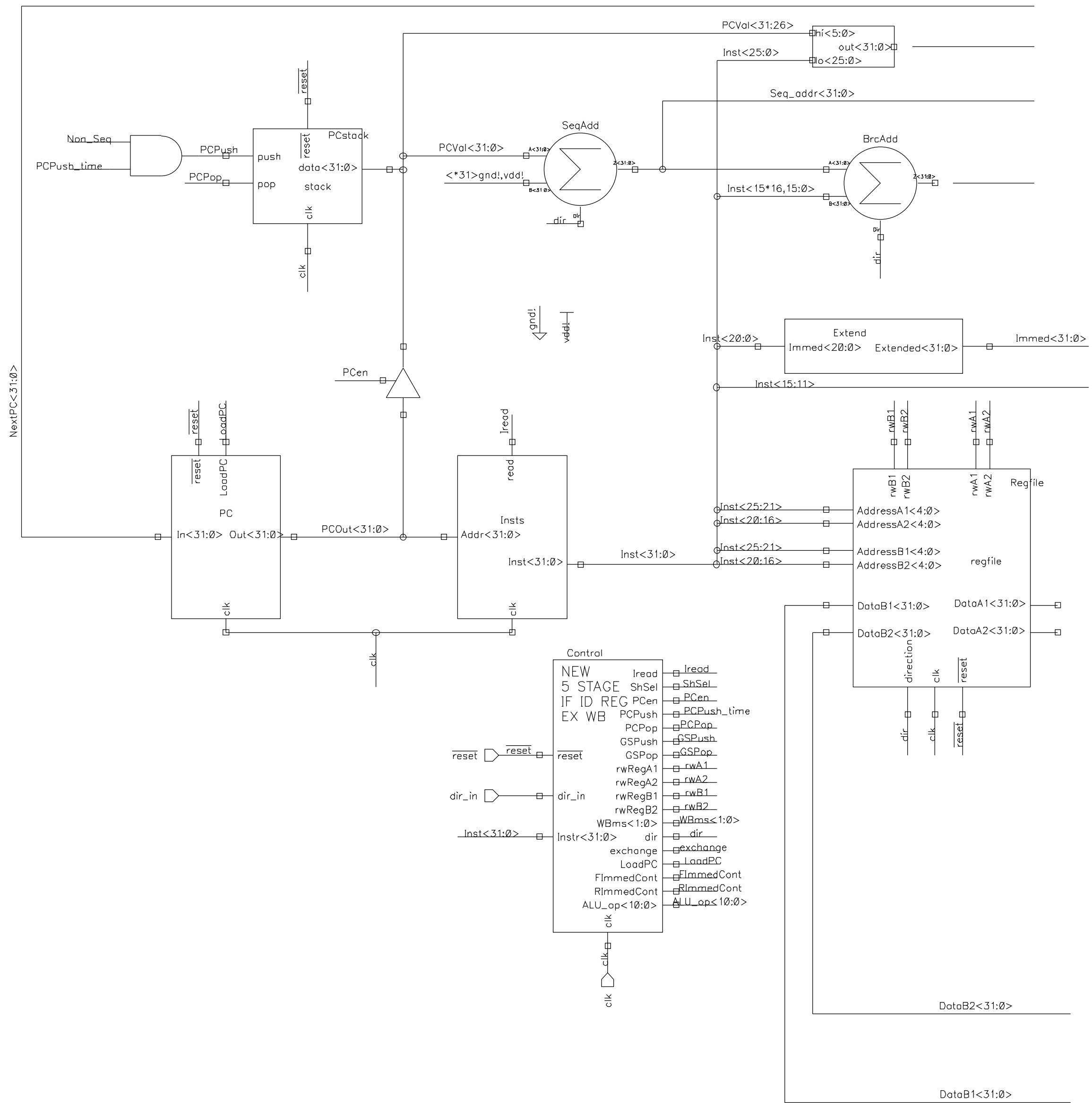

Figure B-1: Detailed Datapath Schematic

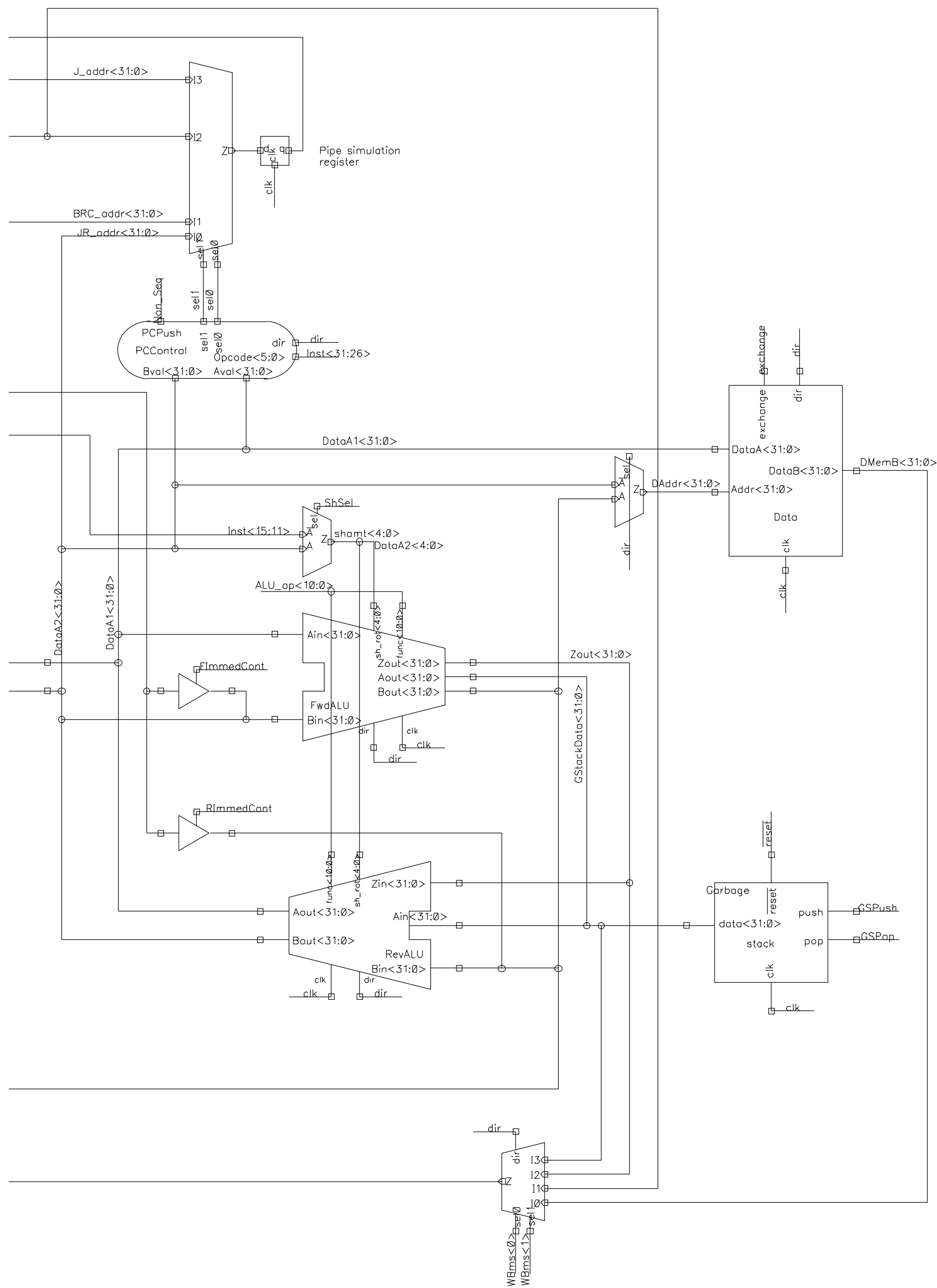

Figure B-2: Detailed Datapath Schematic

Bibliography

[Bak92] H. G. Baker. Nreversal of fortune — the thermodynamics of garbage collection. In Y. Bekkers, editor, *International Workshop on Memory Management*, pages 507–524. Springer-Verlog, 1992.

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

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

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

[FT78] E. F. Fredkin and T. Toffoli. Design principles for achieving high-performance submicron digitial technologies. DARPA Proposal, November 1978.

[FT82] E. F. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3/4):219–253, 1982.

[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 W. C. Athas. Adiabatic switching, low energy compputing, 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] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[Lan82] R. Landauer. Uncertainty principle and minimal energy dissipation in the computer. *International Journal of Theoretical Physics*, 21(3/4):283–297, 1982.

[Lan86] R. Landauer. Computation: A fundamental physical view. *Found. Physics*, 16:260–266, 1986.

[Lik82] K. K. Likharev. Classical and quantum limitations on energy consumption in computation. *Innnnternational Journal of Theoretical Physics*, 21(3/4):311–325, 1982.

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

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

[PH93] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, San Mateo, beta edition, 1993. From Uncorrected Preliminary Manuscript.

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

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

[Szi29] L. Szilard. On the decrease of entropy in a thermodynamic system by the inter-

vention of intelligent beings. *Zeitschrift für Physik*, 53:840–852, 1929. English translation in *Behavioral Science*, 9:301-310, 1964.

[YK93] S. G. Younis and T. 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] S. G. Younis and T. F. Knight, Jr. Asymptotically zero energy split-level charge recovery logic. In *International Workshop on Low Power Design*, pages 177–182, 1994.

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