View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

A general-purpose pulse sequencer for quantum computing

by

Paul Tan The Pham

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2005

© Paul Tan The Pham, MMV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author . . . .
Department of Electrical Engineering and Computer Science
January 28, 2005

Certified by . . . .
Isaac L. Chuang
Associate Professor of Physics
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A general-purpose pulse sequencer for quantum computing

by

Paul Tan The Pham

Submitted to the Department of Electrical Engineering and Computer Science
on January 28, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Quantum mechanics presents a more general and potentially more powerful model of computation than classical systems. Quantum bits have many physically different representations which nonetheless share a common need for modulating pulses of electromagnetic waves. This thesis presents the design and evaluates the implementation of a general-purpose sequencer which supports fast, programmable pulses; a flexible, open design; and feedback operation for adaptive algorithms.

The sequencer achieves a timing resolution, minimum pulse duration, and minimum delay of 10 nanoseconds; it has 64 simultaneously-switching, independent digital outputs and 8 digital inputs for triggering or feedback. Multiple devices can operate in a daisy chain to facilitate adding and removing channels. An FPGA is used to implement a firmware network stack and a specialized pulse processor core whose modules are all interconnected using the Wishbone bus standard. Users can write pulse programs in an assembly language and control the device from a host computer over an Ethernet network. An embedded web server provides an intuitive, graphical user interface, while a non-interactive, efficient UDP protocol provides programmatic access to third-party software.

The performance characteristics, tolerances, and cost of the device are measured and compared with those of contemporary research and commercial offerings. Future improvements and extensions are suggested. All circuit schematics, PCB layouts, source code, and design documents are released under an open source license.

Thesis Supervisor: Isaac L. Chuang
Title: Associate Professor of Physics

Acknowledgments

Isaac Chuang, my thesis adviser, provided me with a challenging project and a chance to actively contribute to quantum computing research. He has been generous with his knowledge and patience, and his scientific drive has always provided me with ample guidance and motivation. My research collaborators at NIST Boulder, John Martinis and Steve Waltman, are both extraordinary engineers from whom I learned much about digital electronics. My labmates in the quanta group taught me about physics and physicists, and Ken Brown and Andrew Houck both provided helpful feedback on early thesis drafts. Ken helped me become a better go player through constant defeat, and Andrew's posters and presentations remain my exemplars. I have also been fortunate to learn from several excellent computer scientists in my career thus far. Michael Ernst is a steadfast teacher and academic role model. Michael Sipser makes complexity theory simple, and Silvio Micali is a cryptographic superhero. Johannes Helander was a resourceful mentor both during and after my forays into industry.

My awesome housemates during my MEng year, Malima Wolf and Blake Stacey, tolerated my chalkboard graffiti and subterranean habitat. Alena Tansey made sure I left lab, MIT, and Boston on a regular basis. Leigh Stringfellow gives me encouragement and inspiration. I continue to regard my undergraduate years in a leaky box of LEDs, loud music, and suspended furniture at TEΦ, Xi Chapter, as somewhere between mythological and apocryphal. Evidence to the contrary is often supplied by my roommates for life, Adrian Bischoff and Jason Rolfe. Thanks to Tool, A Perfect Circle, Radiohead, Autechre, Opeth, Juno Reactor, Finntroll, and Ted Leo for making good music and to Gonzo Digimation for producing fine anime. Finally, my family has always been supportive of my education. I owe a tremendous debt to my parents, Phiet and Minh, and my sisters: Mai, Ann, and Tracie. My new niece Nadia gets honorable mention for making Christmas extra special this year.

Contents

1 Introduction **19**

1.1 Pulse Programming 20

1.1.1 Multiple-Bit Pulses 21

1.1.2 Figures of Merit 22

1.2 Goals 24

1.3 Approach 25

1.4 Organization 27

2 Background **29**

2.1 Physical Representation of Qubits 29

2.1.1 Nuclear Magnetic Resonance 30

2.1.2 Ion Traps 31

2.1.3 Superconducting Current Phase 31

2.2 Related Work 32

2.2.1 Microprocessor Approaches 33

2.2.2 Programmable Logic Approaches 35

3 Hardware Design **37**

3.1 PCB Design 38

3.1.1 Transmission Line Parameters 39

3.1.2 Minimizing Reflection 40

3.1.3 Minimizing Crosstalk 41

3.1.4 Power Filtering 42

3.2 I/O Interfaces . . . 43
3.2.1 Ethernet . . . 44
3.2.2 LVDS . . . 45
3.2.3 I^{2}C, VersaLink, and SMA . . . 47
3.3 Logic and Storage . . . 47
3.3.1 SRAM . . . 48
3.3.2 FPGA . . . 48
3.3.3 Clock Multiplexing . . . 50
3.4 Implementation . . . 51
3.5 Contributions . . . 52
3.6 References . . . 52

4 Firmware Design **53**
4.1 Bus Primitives . . . 54
4.1.1 Sizers . . . 55
4.1.2 Proxies . . . 56
4.1.3 Arbiters . . . 57
4.2 Communication Stacks . . . 58
4.2.1 Daisy-Chain Controller . . . 59
4.2.2 Network Controller . . . 64
4.3 Processor Cores . . . 65
4.3.1 AVR Controller . . . 65
4.3.2 Pulse Control Processor . . . 67
4.4 Implementation . . . 71
4.5 Contributions . . . 72
4.6 References . . . 73

5 Software Design **75**
5.1 Pulse Programs . . . 76
5.1.1 Encoding Pulses . . . 77
5.1.2 Pulse Events . . . 77

5.1.3 PCP Assembly Syntax 79
5.2 Development Tools 82
5.2.1 PTP Client 83
5.2.2 GNU Assembler 84
5.3 Common Gateway Interface 85
5.3.1 Selection Mode 86
5.3.2 Operation Mode 87
5.4 Implementation 87
5.5 Contributions 89
5.6 References 89

6 Measurements and Results **91**
6.1 Prerequisites 92
6.2 Output Skew Measurements 93
6.3 Clock Speed Measurements 95
6.4 Duration Measurements 97
6.4.1 Error 99
6.4.2 Jitter 100
6.5 Delay Measurements 100
6.5.1 Error 101
6.5.2 Jitter 102
6.6 Latency Measurements 103
6.6.1 Trigger Latency 104
6.6.2 Feedback Latency 106
6.7 Noise Measurements 107
6.7.1 Reflection Noise 108
6.7.2 Crosstalk Noise 110
6.8 Power Consumption 110
6.8.1 Firmware Power Consumption 111
6.8.2 Task-Dependent Power Consumption 112

6.8.3 Clock-Dependent Power Consumption 112

7 Conclusion **115**

7.1 Evaluation 115

7.1.1 Performance 116

7.1.2 Feedback 116

7.1.3 Programmability 117

7.1.4 Ease-of-Use 117

7.1.5 Flexibility 118

7.1.6 Comparison 118

7.2 Future Work 119

7.2.1 Hardware Extensions 120

7.2.2 Firmware Extensions 120

7.2.3 Software Extensions 122

7.2.4 Methodology 123

A Hardware Design Documents **125**

A.1 Transmission Line Model of a Microstrip 125

A.2 Layer Stackup 127

A.3 Circuit Schematics, Layouts, and Drawings 129

B Firmware Design Documents **143**

B.1 Power-On Reset Circuit 143

B.2 Memory Segments 144

B.3 Pulse Transfer Protocol Application Layer 145

B.3.1 Null Opcode 145

B.3.2 Status Opcodes 146

B.3.3 Memory Opcodes 146

B.3.4 Start Opcodes 148

B.3.5 Trigger Opcodes 149

B.3.6 I^{2}C Opcodes 150

B.3.7 Debug Opcodes 151
B.3.8 Discover Opcodes 152

C Programming Reference for the PCP 153

C.1 Architectural Parameters 153
C.2 Instruction Set for the `pcp0` 154
C.2.1 Load 64-bit Immediate Instruction 155
C.2.2 Jump Instruction 156
C.2.3 Branch on Trigger Instruction 157
C.2.4 Halt Instruction 158
C.2.5 Pulse Immediate Instruction 159
C.2.6 Pulse Register Instruction 160

D Software Design Files 161

D.1 AVR Maps 161
D.2 CGI Variables 163

List of Figures

1-1 Where does a pulse sequencer fit into quantum computing? 20
1-2 Square (binary) pulse output. 21
1-3 Gaussian pulse output. 22
1-4 Figures of merit for pulse sequencer output. 23
1-5 The three layers of the pulse sequencer. 25

3-1 The assembled printed circuit board for the pulse sequencer. 38
3-2 A PCB trace modeled as a microstrip transmission line. 39
3-3 Lumped element approximation of transmission lines 39
3-4 Three different termination schemes. 41
3-5 Split power planes provide high-quality bypass capacitance. 43
3-6 The Samtec edgemount connector. 46

4-1 Top-level functional blocks of the firmware. 54
4-2 A chain of firmware modules acting as masters and slaves. 55
4-3 A sizer converts from one data width to another. 55
4-4 A proxy can convert a master into a slave and a slave into a master. . 56
4-5 An arbiter can serialize transfers from many masters (one master) to one slave (many slaves). 57
4-6 Three sequencer devices connected in a daisy-chain. 59
4-7 Daisy-chain wire protocol transferring one byte. 60
4-8 Fields of a Pulse Transfer Protocol frame 61
4-9 PTP routing layer transmit and receive chains. 62
4-10 Interconnections within the daisy-chain controller stack. 63

4-11 The receive chain of the network controller. 64
4-12 Machine model of the AVR controller adapted for the sequencer . . . 66
4-13 Machine model of the `pcp0`. 69
4-14 Maximum delays for various timer widths at 10 nanosecond resolution. 70

5-1 Execution platforms for the software layer. 76
5-2 Encoding pulse outputs 77
5-3 Pulse events and overlapping instructions. 78
5-4 An example pulse program in PCP assembly language. 80
5-5 The software development process for pulse programs. 82
5-6 An example of command-line syntax for the `ptpclient` tool. 84
5-7 An example of command-line syntax for the `pcp-elf` tools. 85
5-8 Modes of operation in the web interface 86
5-9 The device selection page of the web interface. 87
5-10 The device operation page of the web interface. 88

6-1 The LED test board constructed at NIST Boulder. 94
6-2 Test pulse program for measuring output skew and clock frequency-dependence. 94
6-3 Plot of output skew. 95
6-4 Plot of clock frequency-dependent output jitter. 97
6-5 Test pulse program for measuring duration error and jitter. 98
6-6 Plot of duration error. 99
6-7 Plot of duration jitter. 100
6-8 Test pulse program for measuring delay error and jitter. 101
6-9 Plot of delay error. 102
6-10 Plot of delay jitter. 102
6-11 Test pulse program for measuring LVDS latency. 103
6-12 Histogram of LVDS rising edge latency. 104
6-13 Histogram of LVDS falling edge latency. 105
6-14 Test pulse program for measuring trigger latency. 105

6-15 Histogram of trigger latency. 106
6-16 Test pulse program for measuring feedback latency. 107
6-17 Histogram of feedback latency. 108
6-18 Test pulse program for measuring reflection noise. 109
6-19 Plot of signal-to-reflection noise ratio. 109
6-20 Plot of signal-to-crosstalk noise ratio. 110
6-21 Plot of clock frequency-dependent power consumption. 113

A-1 Schematic for clock switch and clock sources. 130
A-2 Schematic for power supplies and safety devices. 131
A-3 Schematic for FPGA, SRAM, LED driver, and fiberoptic connectors. 132
A-4 Schematic for Ethernet controller and connectors. 133
A-5 Schematic for LVDS drivers and receivers. 134
A-6 Schematic for daisy-chain connectors and isolation transformers. . . . 135
A-7 Schematic for Samtec edgemount connector. 136
A-8 Assembly drawing for top side (pads and silkscreen). 137
A-9 Assembly drawing for bottom side (pads and silkscreen). 138
A-10 PCB layout for top routing layer (traces and pads). 139
A-11 PCB layout for bottom routing layer (traces and pads). 140
A-12 PCB layout for inner ground planes (keepout regions). 141
A-13 PCB layout for inner power planes (keepout regions). 142

B-1 Power-on reset circuit for bootstrapping digital logic. 143

List of Tables

1.1 Hierarchy of abstraction layers. 27

2.1 Pulse requirements for different qubits (adapted from [Chu04]). . . . 30

2.2 Figures of merit for the Varian ^{UNITY}INOVA spectrometer. 34

2.3 Figures of merit for SpinCore's pulse programmers. 36

3.1 I/O bus standards, logical interfaces, and physical connectors. 44

4.1 Layers in the network and daisy-chain protocol stacks. 58

4.2 ID address space for the Pulse Transfer Protocol. 62

4.3 Parameters of the PCP64 architecture. 68

5.1 Timing parameters of the PCP. 81

6.1 Maximum clock speed constraints. 96

6.2 Timing error and jitter for the on-board 100 MHz oscillator. 97

6.3 Linear fit parameters for duration error and jitter. 98

6.4 Linear fit parameters for delay error and jitter. 101

6.5 Gaussian fit parameters for LVDS edge latency. 104

6.6 Gaussian fit parameters for trigger latency. 106

6.7 Gaussian fit parameters for feedback latency. 107

6.8 Linear fit parameters for signal-to-reflection-noise ratio. 109

6.9 Linear fit parameters for signal-to-crosstalk noise ratio. 110

6.10 Idle and peak power consumption of sequencer device. 111

6.11 Power consumption of various tasks at +5.4V 112

6.12 Linear fit parameters for clock-dependent power consumption. 113

7.1 Feature comparison between pulse sequencer and competing devices. . 119

A.1 The six layer, controlled dielectric stackup for the PCB. 127

B.1 Data, address, and segment prefix widths for SRAM masters. 144
B.2 Payload description for the Status Reply opcode. 146
B.3 Payload description for the Memory Request opcode. 147
B.4 Payload description for the Memory Reply opcode. 147
B.5 Payload description for the Start Request opcode. 148
B.6 Payload description for the Start Reply opcode. 148
B.7 Payload description for the Trigger Request opcode. 149
B.8 Payload description for the Trigger Reply opcode. 150
B.9 Payload description for the I^2C Request opcode. 150
B.10 Payload description for the I^2C Reply opcode. 151
B.11 Payload description for the Debug Request opcode. 151
B.12 Payload description for the Debug LED Request subopcode. 151
B.13 Payload description for the Debug Reply opcode. 152
B.14 Payload description for the Discover Request opcode. 152
B.15 Payload description for the Discover Reply opcode. 152

C.1 Machine parameters for the pcp0 in the PCP64 architecture. 153
C.2 Example of a PCP64 instruction format table. 154
C.3 Notation for the PCP64 instruction set. 155
C.4 Interpretation of the tr operand in the btr instruction. 157
C.5 Interpretation of the sel_{32} operand in the p instruction. 159

D.1 The AVR linking map for the web server binary on the host. 161
D.2 The AVR memory map for the web server binary on the sequencer. . 162
D.3 CGI variables and allowed values. 163

Chapter 1

Introduction

In the 1930s, the Turing machine and the lambda calculus were proven to be the most general model of classical computation. Problems were characterized by their computability and complexity assuming only time and space resources for any real machine. Early computer scientists tried many different representations for a classical bit, but in realizing their theoretical model, they implicitly assumed that only deterministic automata were physically possible.

The formulation of quantum mechanics during the same time period opened up the possibility of building machines with non-classical resources, such as entangled states. The field of quantum computing is still in its infancy today, and there are many viable representations for a quantum bit (qubit). In each case, the qubit can be manipulated by the selective excitation of certain energy transitions. The transitions themselves are addressed by a carrier wave, and quantum gates can be implemented by modulating the amplitude and phase of this carrier with precisely timed pulses.

Like early classical computers, quantum computers are currently unable to bootstrap themselves. In contrast, they can benefit from the advances of their classical predecessors, including high-speed digital logic, programming languages, and network protocols. A general-purpose sequencer device is presented in this thesis which uses the above classical techniques to achieve fast, configurable pulse sequencing for quantum computing.

1.1 Pulse Programming

In a typical quantum computing setting, the user would like to perform operations on a qubit. These operations can be reduced to a sequence of pulsed electromagnetic waves with certain amplitudes, phases, carrier frequencies, and time durations. A recurring problem in these experiments is the transfer of desired pulse sequences from the user to the qubit, known as *pulse programming*, using minimal resources and introducing as few errors as possible. This thesis describes a solution involving a language for specifying arbitrary pulse sequences (*pulse programs*) and a *pulse sequencer* device for translating these programs into the desired digital outputs efficiently and accurately. An abstract model of this approach is depicted in Figure 1-1.

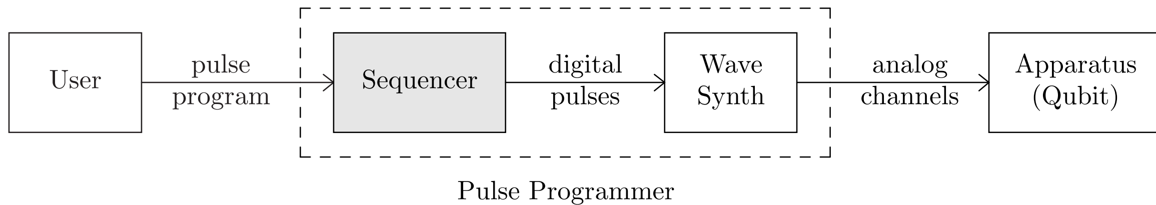

Figure 1-1: Where does a pulse sequencer fit into quantum computing?

By itself, the sequencer only controls the timing of digital outputs (bits) and is agnostic to how they are used. These bits are interpreted by a *waveform synthesizer*, which combines them with a carrier wave to produce modulated analog output. Each analog signal, known as a *channel*, feeds into an apparatus which is qubit-specific. Multiple bits from the sequencer can be assigned to a single channel, and one sequencer can control multiple channels at once.

Traditionally, the sequencer and synthesizer have been combined into a single device called a *pulse programmer*. By decoupling these functions into two devices, the sequencer can accommodate changes in user requirements while the waveform synthesizer can remain optimized for particular experiments.

To motivate the sequencer's design and construction more concretely, its behavior must be quantified with figures of merit and goals must be defined. Because terminology for digital pulse programming is inconsistent in the literature, it is helpful to begin with the preliminary definitions below.

1.1.1 Multiple-Bit Pulses

Originally, the term "pulse" implied a square wave such as the one shown in Figure 1-2. These so-called binary pulses can modulate a carrier wave with two levels, corresponding to On and Off states; they can be generated by a variety of means, both analog and digital. Binary pulses can be characterized by the duration of their On states, called *pulse widths*, and the duration of the intervening Off states, called *delays*.

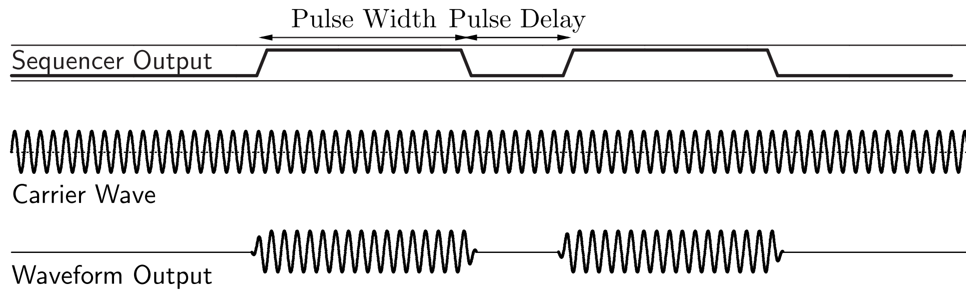

Figure 1-2: Square (binary) pulse output.

However, sometimes a more complicated shape is desirable. For example, Gaussian pulses have a narrower distribution of spectral power density and can excite fewer unwanted transitions than a square wave [SMC03]. Using analog circuitry, continuously-variable levels can be achieved at the cost of lower noise tolerance and the inflexibility of hard-wiring pulse sequences. Thus, digital devices are preferred for approximating arbitrary waveforms in a repeatable and programmable manner. Digital circuits discretize both the pulse width (by a clock cycle) and the output level (by the number of output bits), as shown in Figure 1-3. Greater fidelity to the desired shape can be achieved by increasing the number of output bits, decreasing the clock cycle, or both.

Pulses can now be described digitally with multiple-bit binary numbers; however, with this change in representation from a single-bit square pulse, some terms require clarification. "Width" can refer to the number of output bits, so a pulse's temporal dimension will be called a *duration*. Since arbitrary waveforms may make use of an all-zero value, there is no well-defined Off state; any multiple-bit state is a valid

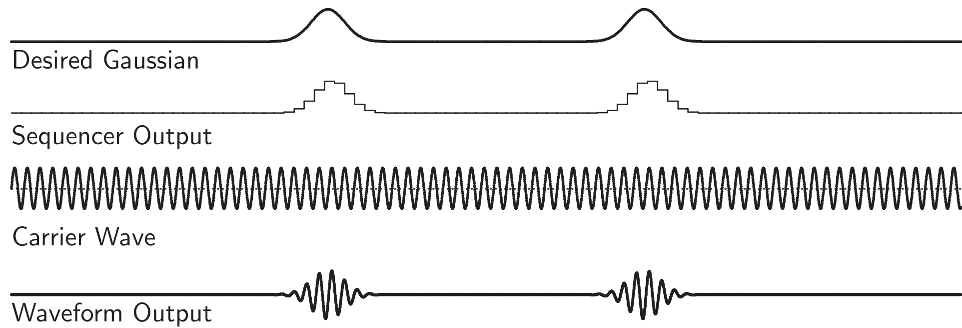

Figure 1-3: Gaussian pulse output.

output value. A pulse then consists of a duration and an output value. "Delay" now refers to decoding overhead between a pulse's desired duration and its actual duration. Using these terms, figures of merit can be defined to characterize the sequencer.

1.1.2 Figures of Merit

The sequencer possesses the following output capabilities: range of pulse durations, minimum delay between two pulse values, resolution (precision) of pulse durations, feedback latency, and number of parallel digital outputs. These figures of merit describe the interface with the waveform synthesizer and other apparatus and are useful for evaluating whether the sequencer can meet experimental requirements As a visual aid, they are overlaid on the example pulse outputs of Figure 1-4.[1]

First, the user must determine whether the desired pulse durations fall within the sequencer's range. The *maximum duration* is rarely a limiting factor since it is easy to increase durations by repeating them, but the *minimum duration* determines which qubits the sequencer can control; by inspection, this latter parameter is 3 cycles in the figure, for the output value of 0x6. The *minimum delay* is the timing error at the end of one pulse before the next pulse appears. It is always less than or equal to the minimum duration; in the figure, it is the overshoot of 1 clock cycle for the output value 0x5.

Even when the above parameters are satisfied, one must be able to vary durations with a fine enough *resolution* to perform reliable operations on qubits. This figure,

[1]The prefix `0x` indicates that a hexadecimal value follows.

also called the *cycle time*, is effectively equal to the period of the system clock in digital systems. It represents the horizontal scale in the figure and does not affect the relative timings of other events. As a consequence, the minimum duration and minimum delay are multiples of the resolution. The minimum delay is ideally zero, corresponding to perfect accuracy within digital switching tolerances.

To interface pulse programmers with other devices, pulse programs should be able to wait for or respond to external events, called triggers or feedback. Because they are asynchronous to the sequencer, there is some overhead for "rounding" the feedback to the next clock cycle, which manifests itself as *feedback latency* between an event and its response. Its value is uniformly distributed between minimum and maximum clock multiples but is not generally a clock multiple itself; in the figure it appears to be 1.5 cycles.

While the preceding parameters all deal with a pulse's duration, the final figure of merit is simply the number of bits of the pulse's output value. A linear increase in number of bits allows the user to encode an exponentially greater number of output values. The example pulse outputs above have 3 bits, each of which can be treated as a square wave pulse running in parallel and synchronized with the others; they can be varied independently by choosing appropriate output values, allowing for one to three channels. Bits can also be assigned to different channels by the waveform synthesizer. Thus, the number of bits is also the upper limit of the number of channels that can be controlled simultaneously.

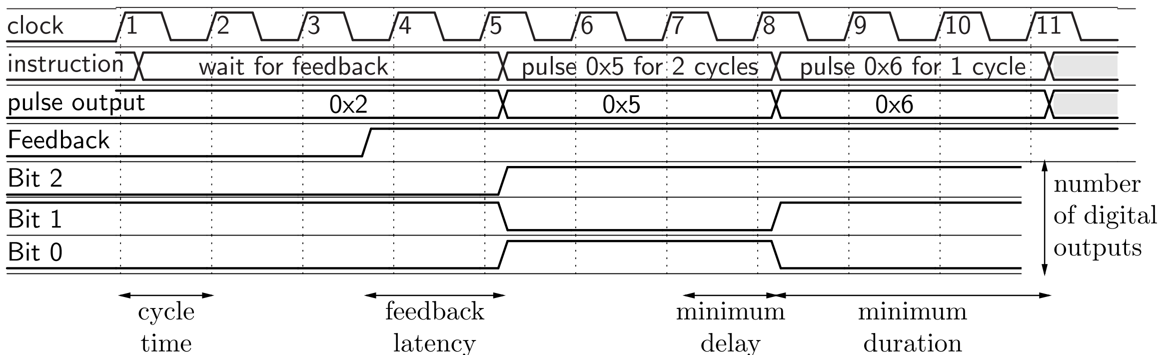

Figure 1-4: Figures of merit for pulse sequencer output.

1.2 Goals

Building on the context of these figures of merit and the previous definitions, overall goals for the sequencer can now be presented.

Performance: The sequencer should have a timing resolution and minimum duration of 10 nanoseconds or less. Consequently, its minimum delay should be less than or equal to 10 nanoseconds and as close to zero as possible. It should also be able to switch tens of digital signals simultaneously. This is sufficient to control one of the most demanding qubits, superconducting current phase.

Feedback: The sequencer should allow external inputs to alter the state of a pulse program while it is running. This is needed for adaptive algorithms, such as quantum error correction. The feedback latency should be on the order of 10 nanoseconds and as close to the minimum delay as possible.

Programmability: All pulse sequence characteristics should be configurable in software using pulse programs. Pulse programs should support a minimum instruction set for producing pulse sequences that can change based on feedback. A detailed discussion of pulse encoding and machine instructions can wait until pulse programs are introduced in Chapter 5.

Ease of use: The sequencer should work with commonly-available consumer networks, desktop computers, and operating systems. Specifically it should have a web interface usable in any web browser and a well-defined network protocol for use in popular scientific software packages such as LabVIEW or MATLAB. It should be simple to install and require little or no maintenance. Users should be able to write and run pulse programs with a minimal learning curve.

Flexibility: The sequencer should be easy to upgrade or modify with new features, especially new instructions for pulse programs, user interface changes, or new communication protocols with other devices. New analog channels should be easy to add and remove in a scalable way to achieve on the order of tens of channels and hundreds of bits.

1.3 Approach

The goals in the previous section can be divided among three subsystems in increasing order of abstraction: hardware, firmware, and software. With well-defined interfaces, the layers can be designed and modified independently of each other and errors can be more easily isolated. The relationship between these layers can be seen in Figure 1-5.[2] The hardware acts as a container for the firmware, which in turn is a container for software. This enables a lower layer to accomplish some subset of the system's goals and defer the remainder to a higher layer.

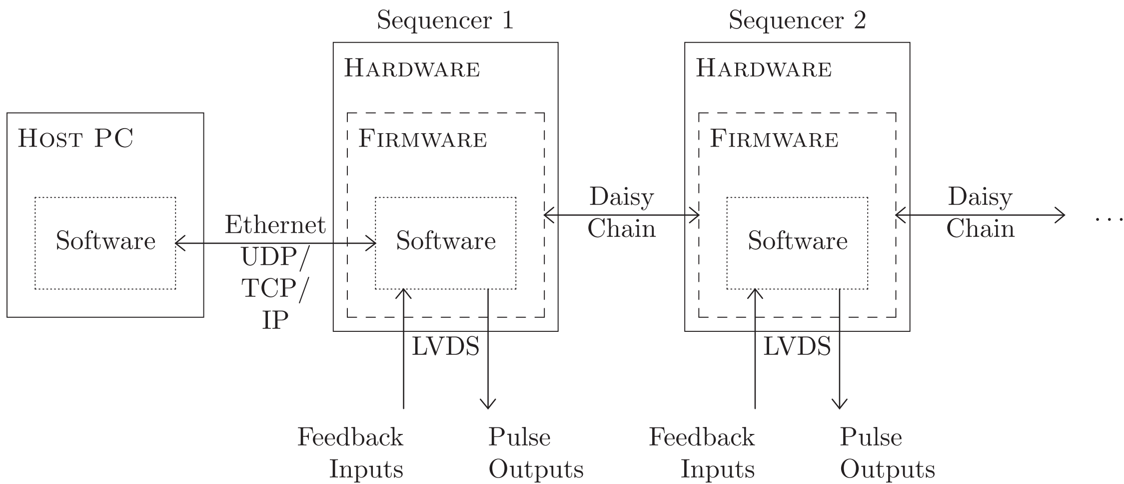

Figure 1-5: The three layers of the pulse sequencer.

The *hardware layer* consists of a high-speed digital printed circuit board (PCB) that multiplexes on-board and external clock sources. It communicates with an external waveform synthesizer through a low-voltage differential signaling (LVDS) bus and high-speed edgemount connector. It also contains connectors for off-board clock sources using Sub-Miniature A (SMA) connectors and general-purpose input/output using an inter-integrated circuit (I^2C) bus. A static random access memory (SRAM) chip provides volatile storage for general-purpose software as well as specialized pulse programs. In addition, multiple boards can be daisy-chained together and triggered from the same source; additional channels can be added in a modular fashion.

[2]Hardware, firmware, and software components in all subsequent figures will be indicated by solid, dashed, and dotted lines, respectively.

Using a field-programmable gate array (FPGA), the PCB can provide reconfigurable "glue" logic, known as the *firmware layer*, between the various inputs and outputs of the system. New features can be added simply by reprogramming the FPGA without redesigning the board. Firmware modules include a network stack for communicating with the user over TCP/IP, a daisy-chain stack for passing messages between sequencer devices, a general-purpose AVR processor[3] for running user interface software, and a specialized Pulse Control Processor (PCP) core for running pulse programs.

Highly configurable tasks such as the user interface and the pulse programs themselves are implemented in the *software layer*. Two user interfaces are provided: a graphical interface for new users and a programmatic interface for advanced users. The graphical interface uses the HyperText Transfer Protocol (HTTP) running over the Transmission Control Protocol (TCP) and is accessible with any web browser. The programmatic interface runs over the User Datagram Protocol (UDP) and allows sequencer actions to be scripted efficiently. Pulse programs are written as text source files in a low-level assembly language and compiled using a port of the GNU assembler.

The sequencer works with commonly-available consumer technology: it can connect to an Ethernet network, obtain a dynamic Internet Protocol (IP) address, and can be controlled from any personal computer (PC) on the Internet. In addition, host-side software runs on Linux, most UNIX systems, and Microsoft operating systems. To increase its usefulness to the research community, all design documents are released under the BSD license, including all circuit schematics, PCB layouts, source code, and documentation. The latest release of the sequencer's three subsystems and all related materials can be found at the project website:

`http://qubit.media.mit.edu/sequencer`

[3]Advanced Virtual RISC (reduced instruction set computer), a family of processors from Atmel Norway made popular in embedded systems because of their low cost, C language optimizations, and free development tools.

1.4 Organization

The remainder of this thesis is organized as follows.

Chapter 2 gives typical pulse requirements for several physical qubit representations and describes how a general-purpose pulse device can satisfy all of them. It also contains a survey of existing pulse programmers, providing context for the current work and motivating several key features.

The next three chapters describe the layers of the sequencer according to the abstraction hierarchy shown in Table 1.1. Each layer from Figure 1-5 is expanded into a stack of three sublayers, also discussed in order of increasing abstraction. Each chapter begins with an overview of how the given layer helps achieve the overall system goals, followed by sections detailing each sublayer, important design decisions, tradeoffs, and alternatives. An implementation section presents post-design issues, the tools used to construct a given layer, and lessons learned. After these, acknowledgments are given of third-party contributions as well as references for further reading.

Layer	**Chapter**	**Sublayer**	**Section**
Hardware	3	Printed circuit board	3.1
		I/O interfaces	3.2
		Storage and logic	3.3
Firmware	4	Bus primitives	4.1
		Communication stacks	4.2
		Processor cores	4.3
Software	5	Pulse programs	5.1
		Development tools	5.2
		Common gateway interface	5.3

Table 1.1: Hierarchy of abstraction layers.

Chapter 6 measures the performance and resource consumption of the device as an integrated whole. Chapter 7 concludes the thesis with an evaluation of the current work's success in meeting its design goals and its suitability for future experiments. A comparison is also provided against contemporary pulse programmers. Suggestions are given for further incremental improvements to this project as well as major extensions.

The appendices provide a technical reference for users and developers of the sequencer which can be read independently from the preceding chapters and in any order. Appendix A contains information useful for modifying the hardware layer, including circuit schematics, layouts, and design calculations for the printed circuit board. Appendix B gives details of the firmware layer, in particular the daisy-chain application protocol. Appendix C is a programmer's reference for PCP assembly language including architectural details, a complete list of opcodes, and their binary formats. Appendix D describes the maps used for linking and loading target software and also specifies the CGI variables used in the web interface.

Chapter 2

Background

The role of a pulse sequencer in the larger framework of quantum computing is best illustrated with examples of actively-researched qubit representations. Thus, Section 2.1 describes pulsed modulation requirements for nuclear magnetic resonance, trapped ion states, and superconducting current phases. The physics of each qubit is briefly described along with the addressing scheme for qubits the carrier being modulated. These examples also show how a pulse sequencer meeting generic requirements but with a wide operating range can be used in many different experiments. Because pulsed programming has a substantial body of prior art, the chapter concludes by surveying the notable features of previous and contemporary devices in Section 2.2. Along the way, key requirements of pulse programming are presented along with some common techniques for meeting them.

2.1 Physical Representation of Qubits

A classical bit can be implemented as any deterministic two-state system. By coupling them into a universal set of gates, one can compute arbitrary Boolean functions. Bits can achieve robustness to noise, correct initialization, measurable output, and determinism itself by virtue of their non-linearity. In contrast, the requirements for a quantum bit are more complicated. Quantum states must be coupled with outside systems for preparation and measurement, but they must also be isolated from

unwanted couplings during gate transformations and fault-tolerant storage [NC00]. Undesirable couplings can be sources of quantum noise or error, and quantum error correction is challenging due to the linearity of quantum systems.

The three physical qubits considered in this section are nuclear spins within a molecule, energy states within a trapped atomic ion, and the current phase through a superconducting junction. Each choice has benefits and drawbacks compared to the others; however, the method for performing unitary transformations using programmed pulses is identical for these three and many other types of qubits. Typical examples of these pulse requirements are given in Table 2.1.

Qubit	**Carrier Frequency**	**Pulse Duration Range**	**Typical Sequence Length**
Liquid-state NMR	200-600 MHz	1-10 μs	300 pulses for 7 qubits
Trapped ions	200 MHz	1-200 μs	40 pulses for 4 qubits
Supercond. phases	DC or 10 GHz	10 ns - 1 μs	10 pulses for 2 qubits

Table 2.1: Pulse requirements for different qubits (adapted from [Chu04]).

2.1.1 Nuclear Magnetic Resonance

Time-domain pulsed programming techniques were first pioneered for nuclear magnetic resonance (NMR) spectroscopy in order to perform Fourier analysis on the free induction decay of nuclear spins [FR81]. DiVincenzo first suggested NMR as a setting for quantum computing experiments [DiV95] and noted that some common NMR pulse sequences were equivalent to single-qubit gates. As a natural two-state system with strong spin-spin couplings, nuclear spins in a molecule are nearly ideal qubits but are difficult to measure due to their small magnetizations, requiring the use of large numbers of molecules (on the order of 10^{23}). Microsecond pulses modulate a transverse magnetic field at the desired Larmor frequencies, which is on the order of 100 MHz, applying unitary transformations to spins within a bulk sample of molecules. NMR quantum computers have been used to implement Grover's quantum search algorithm by Chuang, Gershenfeld, and Kubinec [CGK98] and Shor's quantum

factorization algorithm by Vandersypen, Steffen, Breyta, et al. [VSB+01]. Unfortunately, a major limitation of NMR is the exponential decrease of signal strength with a linear increase in nuclear spins due to pure state preparation [GC97].

2.1.2 Ion Traps

A more scalable approach also uses spin states but in isolated atomic ions coupled by vibrational modes. Cirac and Zoller first suggested the use of linear ion traps to perform quantum computations [CZ95], where quantum gates could be performed on arbitrary subsets of ions, not necessarily adjacent ones. Trapped ions are first prepared using Doppler and sideband cooling, after which individual energy transitions can be excited by shining a tuned laser on individual atoms. In practice, two lasers are often used such that the difference in frequencies is tuned to a desired transition, on the order of 200 MHz. Ions can also be moved in a trap using electrodes which are driven in the kilohertz to megahertz range. Both of these operations require modulated pulses which are tens to hundreds of microseconds wide. Ion traps have already been used to achieve deterministic qubit teleportation at the University of Innsbruck [RHR+04] and NIST [BCS+04].

2.1.3 Superconducting Current Phase

Another approach uses one of three available qubits of a superconducting junction which combines scalability with long coherence times. Superconductors enjoy very low dissipation and offer several possible qubits including the current phase and magnetic flux across the junction, which suffer less noise than using an electric charge representation [MNAL03]; they can also be mass-produced and coupled using integrated-circuit techniques. Superconducting qubits use an artificially limited subspace of energy states with transition energies dependent on a non-linear inductance. This inductance is controlled by high-impedance bias currents which can be modulated by pulses to perform unitary transformations [SMC03]. In theory, superconducting flux qubits can perform gates very quickly (in less than a nanosecond) but require

pulse durations shorter than the capabilities of current technology. Martinis has implemented a superconducting phase qubit [MO99], which has a more feasible pulse width requirement of 10 nanoseconds, using the sequencer presented in this thesis.

2.2 Related Work

Time-domain pulse programming was first pioneered in NMR due to their usefulness in determining molecular structure, imaging biological systems, and measuring chemical properties [FR81]. Pulse programmers themselves have become an important vehicle for transferring nuclear spin techniques to other physical systems in quantum computing. Likewise, the increasing demand for flexible pulse generation has made these devices less integrated with other instrumentation and more modular.

The first pulse programmers were hard-wired RC delay circuits that output a specific, linear pulse sequence when triggered. Unfortunately, this made it difficult to tune experimental parameters or repeat the sequences many times for signal averaging. An early digital/analog hybrid circuit for NMR was build by Conway and Cotts [CC77] which allowed operators to adjust pulse durations with manual controls and contained hardware loop counters to automate experiment repetitions. Additional features, such as automatically incrementing the delays between pulse groups, still required modifications to the circuit [AHT77]. It introduced the idea of modular circuits for the easy addition of new channels, and it promoted the use of the durable and inexpensive TTL logic family over potentially faster and more expensive ECL components.

Today, most pulse programmers use a purely digital approach, which has gone through two main stages as new integrated circuit technologies have become available. The first stage used discrete microprocessors to execute pulse programs and exploited the flexibility of software. The second stage, which includes the pulse sequencer, uses programmable logic devices to both configure hardware and execute pulse programs, retaining the advantages of software and extending them to the performance of firmware. These stages are described below.

2.2.1 Microprocessor Approaches

Adduci and Gerstein [AG79] developed a microprocessor-based system, also for NMR, with delays that were controllable from a host computer through a dedicated parallel port interface. However, pulse durations were still separately controlled through RC delay networks. Pulse programs were written in a machine language developed for NMR experiments and stored in on-board digital memory. The system contained a rudimentary operating system for accepting ASCII commands from a terminal.

The advantage of a dedicated microprocessor is twofold. First, general-purpose computers and operating systems cannot guarantee the predictable scheduling of pulses with nanosecond precision, which is still true for modern systems. Second, tightly coupling a pulse programmer to a host computer increases the cost of the entire system and does not isolate failures between the two. Furthermore, writing pulse programs in a machine language enabled the flexibility of loops and branches; being digitally stored also facilitated modification and reuse. The success of this design is evident in all modern pulse programmers, which are essentially specialized computers.

Thomann, Dalton, and Pancake [TDP84] developed a similar pulse programmer for electron spin resonance spectroscopy. However, it could produce pulses two orders of magnitude faster than NMR devices. Although the decay times of electron spin states are much shorter than for nuclear spins, the pulse techniques used are basically the same. The major advance of this programmer was the output of multiple bits in parallel, associated with a duration, for each step of a pulse program. This unified pulse outputs with timer loading under digital control, which decreased the circuit complexity and allowed much smaller minimum values.

The state-machine pulse programmer by Wachter, Sidky, and Farrar [WSF88] used a novel composite counter to increase the range of possible delays while maintaining a fast resolution due to improvements in TTL components. The fan-out of the high-speed clock is limited to a single fast counter, which in turn clocks slower counters. Also, the terminal value of loop counters and delay counters are used to decode the next instruction. The inclusion of two loop counters allowed a single level of nesting.

Wu, Patterson, Butler, and Miller's spectrometer [WPBM93] included a pulse programmer which controlled the timing of both amplitudes and phases using multiple-bit channels. Pulse programs were written as modules in LabVIEW, a graphical programming language for desktop computers. These were then compiled into the specialized machine code for the pulse programmer hardware and written to a parallel interface card.

Varian, the company which first produced commercial NMR instrumentation, uses a customized bytecode language in its ^{UNITY}INOVA spectrometers [Var00a]. These devices are usually sold as an integrated equipment cabinet with waveform generators and Programmable Test Source (PTS) synthesizers for frequency generation. Digital sequencing is performed by an embedded computer called the digital acquisition controller housed on a PCB that is approximately 9 by 8.5 inches. It contains a 200 MHz PowerPC-based microprocessor running the VxWorks real-time operating system and a custom high-speed interface to analog ports and RF controllers. Table 2.2 applies the sequencer's figures of merits to the acquisition controller. The ^{UNITY}INOVA is controlled over Ethernet using a dedicated Sun Microsystems workstation, which comes pre-configured with Varian's VNMR pulse programming software [Var00b]. This guarantees that the entire instrument works well as a monolithic unit but at greater material cost and less flexible upgrading.

Cycle time	25 ns
Minimum duration	100 ns
Number of digital outputs	34
Number of analog outputs	4
Maximum carrier frequency	50 MHz
Maximum program size (words)	4K
Flow control features	Nested looping, subroutines

Table 2.2: Figures of merit for the Varian ^{UNITY}INOVA spectrometer.

2.2.2 Programmable Logic Approaches

While the microprocessor approach offers significant flexibility over hard-wired pulse programmers, many signal assignments and I/O connections are still held fixed in hardware. More recently, Yun, Yu, Hongyan, and Gengying use a complex programmable logic device (PLD) to interface an on-board DAC to an industry standard architecture (ISA) bus [YYHG02]. The pulse programmer and the ISA interface controller is entirely implemented in the gate logic of the PLD, which increases the configurability of their system without decreasing its speed.

The commercial pulse programmers from SpinCore[1] are some of the most sophisticated devices to date and offer both waveform synthesis and sequencing, These devices use an Altera Stratix FPGA and proprietary firmware cores; they are implemented as ISA or peripheral component interconnect (PCI) cards installed inside the host PC. The PulseBlaster™, like the pulse sequencer, only produces digital outputs to control the timing of other devices; the pulse processing core is called a pattern generator by SpinCore [Spi04c]. The PulseBlaster DDS™ is a "direct digital synthesis" pulse programmer which combines sequencing and waveform synthesis to produce both digital outputs and analog channels [Spi04a]. The PulseBlaster ESR™ is a high-performance version for ESR spectroscopy [Spi04b]. The capabilities of the most powerful device in each product line is shown in Table 2.3. Parenthesized figures are for alternate models which make different feature trade-offs.

Each device is controlled over the PC's expansion bus by using device drivers running on the host. The user does not write pulse programs directly in the device's machine language. Rather, the C programming language and SpinCore's application programming interface (API) are used to create host binaries which, when executed, generates instructions and downloads them into the pulse programmer. This approach has the advantage of using existing compilers and languages to generically implement a new language that is local to the PCI bus and internally hidden from the user.

The integrated peripheral card approach of the previous two works have the advantage of a small form factor (approximately 4.5 by 7 inches) but several disadvantages

[1] http://www.spincore.com

Parameter	**PulseBlaster**	**PB DDS**	**PB ESR**
Cycle time (ns)	8 (10)	10	3.0 (3.3)
Minimum duration (ns)	40 (80)	50 (90)	3.0 (20)
Number of digital outputs	12 (24)	10	21 (24)
Number of analog outputs	n/a	3	4
Maximum carrier frequency (MHz)	50	100	4
Maximum program size (words)	512 (32K)	512 (32K)	4K (512)
Flow control features	Nested looping, subroutines		
Triggering	From host software or hardware inputs		
Trigger Latency	80 ns	80 ns	26.6 ns

Table 2.3: Figures of merit for SpinCore's pulse programmers.

due to tight coupling. This scheme is inflexible in that the PC must be powered off before the pulse programmer can be installed or removed. It lacks fault-tolerance in that the pulse programmer is not electrically isolated from the PC. The device depends on a shared power supply and is located closed to other unshielded devices with generate electromagnetic interference.

The direct predecessors of the current device were developed by Huang, Martinis, Waltman, and Chuang [Hua03]. The first generation device used a Rabbit Semiconductor board to provide both an Ethernet interface and a web server; it was easy-to-use, but had a slow cycle time and a small number of digital outputs. The second generation device used a Digilent board with a Xilinx FPGA to achieve faster cycle times and more digital outputs; however it suffered from an inflexible and less user-friendly serial interface.

The goals for the current sequencer listed in Section 1.2 are chosen to retain the advantages while overcoming the limitations of all prior devices, especially the novel feature of feedback support. The contemporary devices in this section will be revisited in the conclusion where they will be compared with the pulse sequencer. While some competing devices combine both sequencing and waveform synthesis, only sequencing capabilities will be compared.

Chapter 3

Hardware Design

This chapter describes the hardware layer of the pulse sequencer, which consists of I/O interfaces and electrical components connected by a PCB. In terms of the overall goals, the hardware ultimately determines system performance by setting the maximum clock speed, the number of available pulse outputs, and the maximum size of a pulse program. The hardware also maintains flexibility by avoiding hard-wired assumptions in its interface to the software and firmware layers. Implementing common I/O bus standards and connectors makes the hardware easy to use with other equipment. Programmability is implemented indirectly by including reconfigurable hardware components which can be made to execute software. Finally, feedback is supported at the hardware level by including appropriate connector inputs for external sources.

More concretely, the sequencer PCB must perform the following tasks which higher layers will depend on. It must run at fast clock speeds with low noise, requiring the high-speed digital design techniques that begin this chapter in Section 3.1. The PCB must also interface with other devices and especially the user; the choice of electronic components, bus standards, and connectors for interfacing with the outside world is justified in Section 3.2. Finally, the PCB must be able to connect its I/O interfaces in a configurable way. The components used to achieve storage for pulse programs, logic for their execution, and switching between multiple clock sources are described in Section 3.3. The resulting PCB is shown in Figure 3-1 after fabrication and assembly.

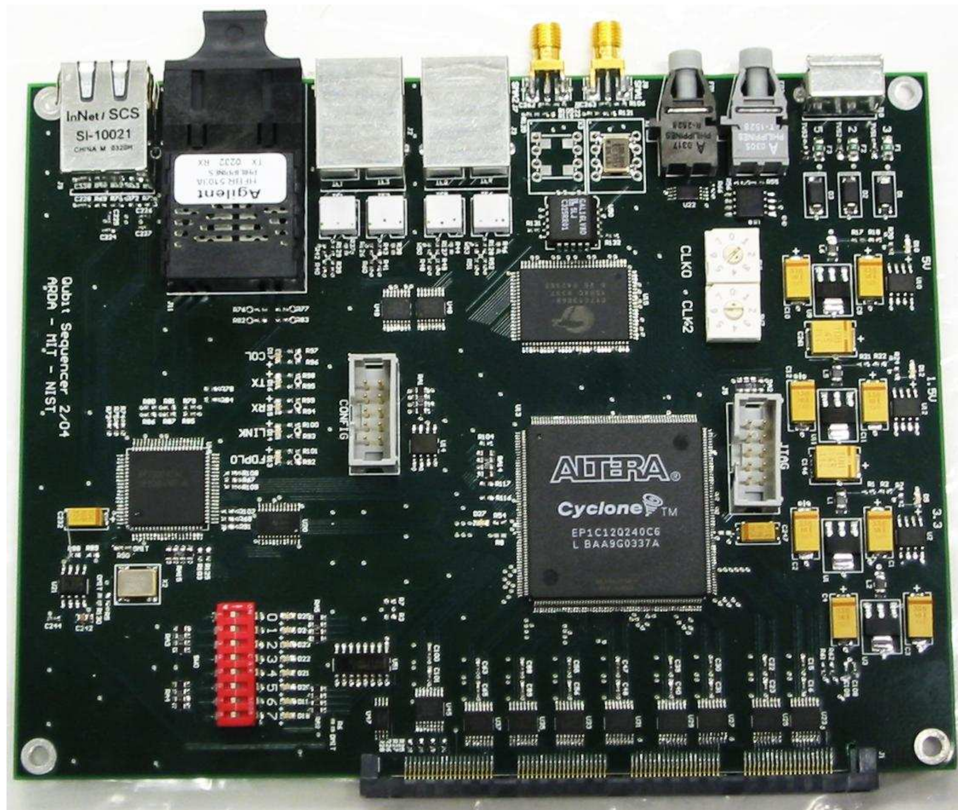

Figure 3-1: The assembled printed circuit board for the pulse sequencer.

3.1 PCB Design

The reliable driving and sampling of digital data at high speeds in the presence of noise is a feature known as *signal integrity*. When the round-trip propagation time of a signal approaches its transition time, the conventional circuit model using Kirchoff's laws cannot maintain signal integrity. On PCBs with dimensions of a few inches, this limit is usually reached at 100 MHz. At these speeds a PCB trace is more usefully considered as a transmission line, which predicts two chief sources of noise. Internal noise from an isolated signal's self interaction is called *reflection*; external noise from the coupling of multiple signals is called *crosstalk*. A third type of noise can be introduced by power supply fluctuations. Mitigating these sources of noise places certain constraints on trace dimensions, routing, and layer thicknesses, which must be balanced against available manufacturing tolerances.

3.1.1 Transmission Line Parameters

A transmission line consists of two conductors separated by a dielectric material, where the separation distance is much smaller than the signal wavelength and the cross-section is homogeneous down the line length. It is a useful model for describing signal propagation down coaxial cables, parallel plates, and especially PCB traces. In particular, traces on surface layers can be modeled as microstrips as shown in Figure 3-2. These have a rectangular cross-section and are separated from a ground plane on one side by a dielectric layer and are exposed to air on the other side.

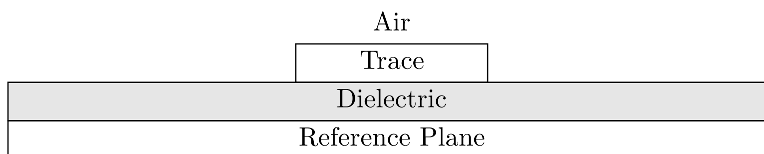

Figure 3-2: A PCB trace modeled as a microstrip transmission line.

To analyze an equivalent circuit of a transmission line, the lumped circuit approximation in Figure 3-3 is used; it includes a series resistance R, a series inductance L, a parallel capacitance C, and a shunt admittance G. Taking the limit of these cascaded lumped elements and applying Maxwell's Equations, the telegrapher's equations can be derived[1] to find the ratio between voltage and current changes in the transmission line, shown in Equation 3.1.

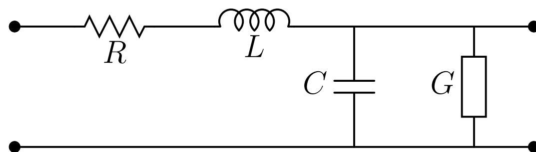

Figure 3-3: Lumped element approximation of transmission lines

This characteristic impedance, denoted by Z_0, is in general frequency-dependent and complex-valued. However, its expression can be simplified depending on the desired range of operating frequencies.[2] The pulse sequencer operates at frequencies much greater than $\omega_{LC} = R/L$, where the inductive term dominates the resistive term

[1]As done in [Kon00] and [Som99]

[2]A detailed treatment of performance regions can be found in Chapter 3 of [JG03].

in the numerator and the capacitive term dominates the admittance in the denominator. The lumped model then reduces to an LC circuit where Z_0 is relatively constant; using this approximation, specific transmission line parameters are calculated for the sequencer's microstrip PCB trace in Appendix A.

$$Z_0(\omega) = \sqrt{\frac{j\omega L + R(\omega)}{j\omega C + G(\omega)}} \tag{3.1}$$

This model possesses an under-damped step response due to its lower resonant frequency, which is more easily excited and produces unwanted signal transients. This form of noise, known as "ringing" or reflection, is the primary obstacle to faster clock speed. The characteristic impedance is used to calculate the additional resistance needed to reduce a signal's internal reflection, a process which is described next.

3.1.2 Minimizing Reflection

The amount of a wave that reflects from each end of a transmission line is determined by the reflection coefficients. The formulae below (for the source and the load, respectively) show these as real values between -1 and +1, indicating the relative magnitude and polarity of the reflected wave to the incident wave.

$$\Gamma_S = \frac{R_S - Z_0}{R_S + Z_0} \tag{3.2}$$

$$\Gamma_L = \frac{R_L - Z_0}{R_L + Z_0} \tag{3.3}$$

These coefficients should be made as close to zero as possible to minimize reflections; the sequencer accomplishes this using *termination*, or the addition of a resistance in series with the source (R_S) or parallel to the load (R_L) matched to Z_0. The three main termination schemes are depicted in Figure 3-4. The disadvantage to end termination is its use of two resistors and its larger constant current draw. Likewise, both-ends termination uses three resistors and halves the signal amplitude in exchange for greater reflection damping. Source termination was chosen as a compromise between minimizing reflections, power consumption, and component count.

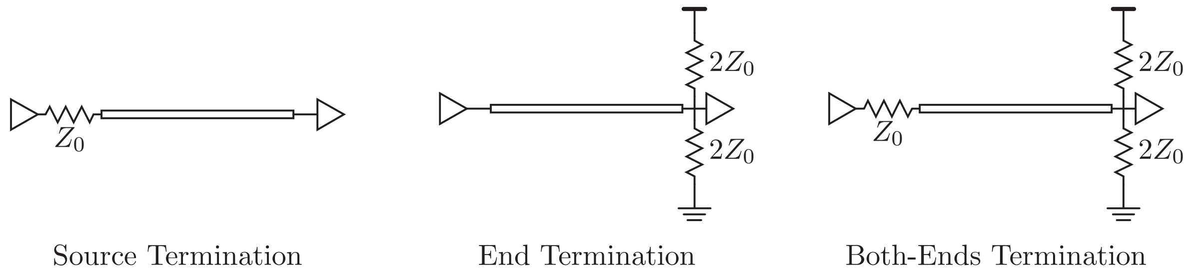

Figure 3-4: Three different termination schemes.

Terminating clock signals is especially important, since these transition twice as fast as any data signal in the system. Reflection on a clock signal will cause false triggering on transient logic levels in other components, which might seem mysterious if their data signals do not switch fast enough to require termination by themselves. Bulk propagation delays introduced by the termination resistor do not effect the overall throughput of the clock, since a real impedance will not distort the waveform.

While impedance matching and termination compensates for a signal's delayed echoes, it cannot distinguish between a signal's incident wave and coincident interference from an adjacent signal. Dealing with unwanted couplings from external sources is the subject of the following section.

3.1.3 Minimizing Crosstalk

Crosstalk is the interference of two signals primarily through mutual inductance. In an ideal, lossless transmission line, the conductor carrying the outgoing signal is closely coupled to the conductor carrying return current, which serves as a *reference*. This minimizes the current's total path in the circuit (the *current loop*), and thus also minimizes inductance. In differential signaling such as LVDS, there are two explicit conductors, and return paths between signals are isolated [Nat04]. This is a key advantage over single-ended signaling, where a single ground trace may be shared among many signals. On a PCB, current loops are sometimes close enough to act as transformer windings, allowing transients on one signal to show up with opposite polarity in neighboring signals.[3]

[3]This problem is sometimes called *ground bounce*.

The pulse sequencer uses two conventional techniques to minimize crosstalk: inner reference planes and multiple ground pins in connectors. Solid inner ground planes, rather than power planes, are chosen to distribute the return current evenly. In theory, any DC inner plane can act as an AC signal reference. However, on densely populated PCBs such as the sequencer, high-speed signal traces must be routed across layers using vias, and their signal references must also be "stitched" together with neighboring vias in order to be continuous. Overlapping power planes often carry different voltages and cannot be shorted, precluding their use as signal references.

In connectors, ground pins are evenly distributed between signal pins to eliminate crosstalk in off-board communications. This provides a short, well-defined current loop at regular intervals. Crosstalk is a more serious problem at these external interfaces because connector pins are usually packed closer together than any PCB traces, making it more likely that noise will propagate from one device to another. Specific examples of how connectors minimize crosstalk is deferred until Section 3.2.

Both reflection and crosstalk cause noise from signal couplings on transmission lines, but noise can also be introduced at the sources driving these lines. Filtering is required to attenuate and isolate fluctuations in the power supply.

3.1.4 Power Filtering

The pulse sequencer requires three voltage levels to interface a wide variety of components: +1.5 volts, +3.3 volts, and +5 volts.[4] Each one requires a separate regulator and a portion of a power plane for distribution. Furthermore, the +3.3 volt supply is further divided into two regulators to isolate critical LVDS subsystems from single-ended signals. Ferrite beads and varistors suppress current and voltage transients from the unregulated supplies. After regulation, local current transients are handled by the crosstalk-handling techniques above.

Bypass capacitance is the standard solution to local voltage transients. However, these are only effective at attenuating AC voltages in a certain frequency range, according to the relation $Z_C(\omega) = 1/j\omega C$. Thus, in addition to discrete bypass ca-

[4]The nominal unregulated input voltages are +1.8 volts, +3.6 volts, and +5.4 volts, respectively.

pacitors, the board's inner planes can also serve as extremely high-frequency parallel-plate capacitance using the formula $C_{pp} = \epsilon A/d$. Because a given voltage supply is not needed everywhere on the board, a single power plane can be split among two or more voltages, as shown in Figure 3-5.

Dielectric		
Ground Plane 1		
Dielectric		
Power Plane 1		Ground
Dielectric		
Ground	Power Plane 2	
Dielectric		
Ground Plane 2		
Dielectric		

Figure 3-5: Split power planes provide high-quality bypass capacitance.

The previous three subsections have discussed techniques for dealing with noise both inherent to the PCB as well as from external sources. The final stackup resulting from these design constraints is detailed in Section A.2. Its careful design allows it to maintain signal integrity at speeds greater than 100 MHz and support off-board I/O connections at the same speed. The components to achieve this fast communication have their own electrical and mechanical constraints separate from the PCB design. They represent the next hardware sublayer and are described in the following section.

3.2 I/O Interfaces

In order to generate pulses, the sequencer must communicate through input/output interfaces with the following entities: the user (both inputs and outputs), the waveform synthesizer (outputs only), other instances of the sequencer (both inputs and outputs), feedback sources (inputs only), clock sources (inputs only), and instruments (both inputs and outputs). These interfaces have logical and physical parts. Logical interfaces are internal to the PCB, and their signals are interpreted by the FPGA firmware. Physically, these interfaces must conform to established mechanical form factors and an electrical bus standard, including allowable voltage levels and signal-

ing speeds. PCB traces form the physical "standard" between all connectors, while common voltage levels and signaling speeds are determined by the FPGA and its clocks. Many standard I/O components, such as the Ethernet controller, I^{2}C driver, and fiberoptic connectors, still support the 5V TTL standard and use a FET bus switch to shift to the FPGA's LVCMOS levels.[5]

This section organizes logical I/O interfaces by the bus standards used to implement them; it then describes the physical connectors that implement each bus standard. As shown in Table 3.1, multiple logical interfaces can share the same physical interface, and a single logical interface can also use multiple physical connectors. Two interfaces have bus standards which are not discussed in this section: the SMA clock is only limited by LVCMOS input tolerances and has no real bus standard, while discussion of the daisy-chain interface is deferred to the firmware layer.

Bus Standard	**Speed**	**Logical Interfaces**	**Physical Connectors**	**Media**
Ethernet	100 MHz	User (TCP/IP)	RJ-45 SC	Unshielded twisted pair Fiberoptic
LVDS	200 MHz	Pulses Feedback Clock	Edgemount	Direct connection
		Daisy-chain	RJ-45	Unshielded twisted pair
I^{2}C	400 KHz	General-purpose	Edgemount	Direct connection
VersaLink	20 MHz	General-purpose	VersaLink	Fiberoptic
LVCMOS	18 GHz	Clock	SMA	Coaxial

Table 3.1: I/O bus standards, logical interfaces, and physical connectors.

3.2.1 Ethernet

Ethernet is the most popular technology for connecting local computer networks, especially to the Internet. The physical layer for the Fast Ethernet specification provides full duplex 100 Mbps data transmission over a variety of media. It can be incorporated into a TCP/IP stack for creating more reliable network connections and

[5] LVCMOS is the low-voltage 3.3V equivalent of the 5V CMOS standard.

running higher-level applications. These characteristics make it ideal for implementing an embedded user interface for downloading pulse programs.

The physical layer (PHY) is provided by the DP83843 chip from National Semiconductor [Nat99], chosen for its low cost and simple interface circuitry. It communicates with the FPGA through a standard media-independent interface (MII) described in [Eth02] and supplies two 25 MHz clocks for transmitting and receiving. These occupy two of the FPGA's four clock inputs, and the receive clock is used to synchronize incoming Ethernet frames with the firmware described in Chapter 4. Because no collisions occur during full-duplex operation and carrier detection is not commonly needed, these two pins are not connected.

The Ethernet interface can also make use of two different physical connectors: a 4-pair registered jack (RJ) 45 modular connector, of which only 2 pairs are used, and a 2-conductor fiberoptic subscriber connector (SC). Only one can be used at a time; switching between the two involves soldering a single jumper. Unshielded twisted pair cable is preferred for short runs and convenient connection to consumer PCs. Fiberoptic cable is preferred for longer runs due to its greater immunity to electromagnetic interference and ground isolation from other equipment.

3.2.2 LVDS

The low-voltage differential signaling (LVDS) standard was developed by National Semiconductor to provide fast, low-power interconnects between PCBs with low noise generation and high noise rejection [Nat04]. These match the requirements for several of the sequencer's logical interfaces, including the most important, the fast pulse outputs to the waveform synthesizer. In addition, the speed of LVDS is suitable for receiving moderately fast external clocks (up to 200 MHz). Also, its conversion to and from CMOS/TTL levels makes it ideal for receiving feedback inputs, and providing the physical layer for the daisy-chain interface.

National Semiconductor also provides two convenient components for interfacing LVDS buses with LVCMOS logic levels, a matching transmitter and receiver. The DS90LV047 [Nat03] takes 4 LVCMOS inputs and produces 4 pairs of LVDS outputs.

The DS90LV048 [Nat01] takes 4 pairs of LVDS inputs and produces 4 LVCMOS outputs. In a disconnected state, the receiver outputs high LVCMOS levels; this feature is used to detect the end of the daisy-chain in firmware.

The first three logical interfaces that use LVDS (pulse outputs, feedback inputs, and clock inputs) share the same physical connector, a male edgemount connector from Samtec with 2 rows of 80 pins (part number QSE-EM-80-01-F-D-EM2). A cross-section is shown in Figure 3-6. Right-angle connectors have the disadvantage of skewing differential signals due to different pin lengths, impedance discontinuities due to the right-angle bend, and reduced pin count because of through-hole soldering. Edgemount connectors introduce less impedance, have higher pin counts, and provide better grounding through a plated edge to inner PCB planes. Waveform synthesizers and other devices should use the matching female connector from Samtec (part number QTE-EM-80-01-F-D-EM2) using the same pinout given in Appendix A.

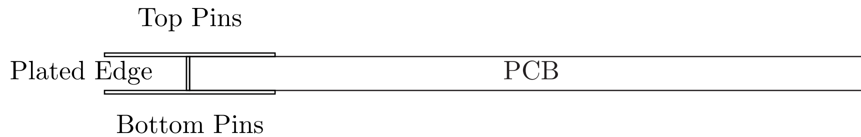

Figure 3-6: The Samtec edgemount connector.

LVDS also provides high-speed, point-to-point channel coding for the daisy-chain interface; its use as a bus standard for other on-board connectors made it a convenient choice. The physical daisy-chain connectors are implemented using two RJ-45 connectors and normal unshielded twisted-pair cable to connect each device to a master and a slave. All four pairs are used in each interface, providing two inputs and two outputs for both the master and slave interfaces. This physical layer forms the beginning of the daisy-chain communications stack which is primarily implemented in firmware. A detailed discussion of the higher layers can be found in 4.2.1.

The edgemount connector is also shared by the general purpose I^{2}C bus, while the remaining two buses (VersaLink and SMA) each have their own connectors.

3.2.3 I^{2}C, VersaLink, and SMA

The pulse sequencer provides two buses for general-purpose peripherals. The I^{2}C bus allows the FPGA to control low-speed peripherals such as the on-board LED controller; since I/O pins are at a premium on the FPGA, it is advantageous to address less critical peripherals using a shared 2-wire bus. The sequencer provides the pull-up resistors and positive power supply for this open-drain standard. These two pins are also exported to the edgemount connector for addressing off-board peripherals.

A faster pair of general-purpose fiberoptic connectors allow the board to interface with a wide variety of Versatile Link^{TM} products from Agilent. These can range from extending an RS-232 or RS-485 bus to acting as a second, higher-bandwidth set of daisy-chain connectors.

The final physical connector, the Sub-Miniature Version A (SMA) does not technically have its own bus standard but can be driven at CMOS logic levels to provide an extremely fast external clock. The connector itself can receive signals many times faster (18-24 GHz) than the PCB is capable of using. It also uses an edgemount connector and multiple ground pins to minimize reflections.

3.3 Logic and Storage

A high-speed PCB is a physical substrate for housing I/O connectors and components; the previous sections have described these two sublayers that are designed for performance. The remaining sublayer must not only provide fast clock speeds, it must supply the seed for flexibility in more abstract layers. The ideal solution to these problems is a complex, reconfigurable logic device. An FPGA provides abundant programmability but with a limited array of peripherals. The small amount of built-in storage and clock inputs requires external SRAM and clock switching components. Once these are secured, the sequencer can use FPGA technology to multiplex I/O connections, encode and decode higher-layer communication protocols, and provide a platform for executable code such as user interfaces and pulse programs. The remainder of this section describes the components chosen for logic and storage.

3.3.1 SRAM

To achieve fast storage, SRAMs provide the fastest access times and lowest power consumption. In addition, to avoid combinational delays and increase throughput, the fastest SRAMs are synchronous and have registered ports. The CY7C1386b from Cypress Semiconductor provides a suitable pipelined memory that is available in a wide data package (36 bits), fast clock speed (150 MHz), and large memory depth (512k words) [Cyp01]. Two unconnected address pins allow future upgrades to double or quadruple the memory depth while remaining footprint-compatible.

This SRAM's large depth provides ample storage space for pulse programs and is well-suited for pipelined burst reads, which will occur frequently in later firmware modules. Moreover, it can serve as memory for many different modules through a single physical interface if it is connected to configurable logic. Its fast speed means that logic components could use its stored state directly without caching. However, its large width makes it less suitable for general-purpose random memory access. Additional logic and cycles are needed to "resize" the physical width of 36 bits to a more standard 16 bits or 8 bits.[6] As a result, single reads have one-third the throughput of burst reads due to pipeline delays.

Because it is volatile, it cannot be used to bootstrap the system. This limitation cannot be overcome without incurring the increased cost, slower access times, more complex write logic, and greater power consumption of flash chips. In these cases, it makes sense to rely on the built-in flash memory of separate logic devices, which are described next.

3.3.2 FPGA

An FPGA typically consists of a two-dimensional grid of logic elements, each of which contains flip-flops, a lookup-table, and combinational logic to implement arbitrary Boolean functions. FPGAs usually contain other devices such as phase-locked loops (PLLs), built-in RAM blocks, and I/O pin drivers that support different logic families.

[6] Sizing and resulting memory segments are described in more detail in Sections 4.1.1 and B.2.

The bulk of an FPGA's semiconductor consists of a vast interconnection network to distribute signals between these elements.

In recent years, the need for configurable logic and easy prototyping for integrated circuits has driven down the cost of programmable logic devices. FPGAs, as the most advanced examples, are a natural choice to provide the "glue" logic for different I/O standards and clock speeds. The Cyclone EP1C12Q240C6 from Altera provides a low-cost component with a fast clock speed (up to 450 MHz), high logic element count (12,060), built-in memory (288 Mbits), 2 PLLs, and a variety of I/O standards (LVCMOS is the only one currently used) [Alt03]. Its lookup tables are contained in volatile memory, and it is reprogrammed by a serial flash device (the Altera EPCS4) on each power-up.

Historically, pulse programming devices have included only a hardware layer and a software layer. Unfortunately, many tasks such as memory arbitration are too complicated to implement in hardware; furthermore, such tasks cannot execute in software fast enough to produce accurate pulse timings. The introduction of a firmware layer, made possible by an FPGA, allows logic to approach the programmability of software and the speed of hardware. Including one device relegates almost all of our logic functions and I/O control to the firmware layer, as long as all peripherals are physically connected to the FPGA.

I/O connections also represent the major limitation of the FPGA. For a plastic quad flat package (PQFP), the maximum feasible number of pins is 240, with 60 on each side; 169 of these available for I/O, and these must be divided among storage lines (control, address, and data) and peripheral lines (control and data). Because I/O pins are limited, some features of the Altera Cyclone cannot be used. For example, the FPGA can drive differential outputs directly using several popular standards, but this requires two I/O pins for every signal. The final design compromised by having the FPGA drive single-ended signals, effectively doubling the pin count while requiring discrete components to convert them into differential pairs.

Moreover, the FPGA limits the available bandwidth of external storage. Ideally the memory data width would equal the number of desired programmable outputs,

allowing each memory bit to drive an output bit directly. In practice, however, daisy-chaining memory chips to achieve this width would introduce routing and noise problems with a shared address bus. Therefore, the task of multiplexing 36 bits of memory among 64 digital outputs was deferred to the firmware and software layers.

The FPGA is not a bottleneck in system clock speed; its high-speed internal bus can transition at over 450 MHz. Using its PLLs, it can multiply a 100 MHz clock from the PCB by any half-integer multiple up to 4.5. This extremely fast clock would be local to the FPGA's internal logic and the pulse output circuitry. In practice, however, the setup and hold times of the FPGA's programmable logic limit the maximum clock speed. In fact, due to several reasons not due to speed, it is desirable for the sequencer to manage clocks without using the FPGA at all.

3.3.3 Clock Multiplexing

Multiplexing clock signals is difficult compared to data signals because they must transition twice as fast, propagate farther distances, and drive much larger fan-outs. Furthermore, the sequencer has six available clock sources for maximum flexibility, so any clock switching scheme must support many inputs. A manual switch would suffice for user convenience, but this provides no voltage protection against external clock sources. A logic device could be used to multiplex clocks, but it must possess a high-speed interconnect so that clock signals are not attenuated at high frequencies. The FPGA would be ideal except that it only possesses 2 available input clocks.

The sequencer combines the approach of manual switches with a an separate programmable logic device (PLD), which switches the clocks directly. The GAL16LV8D from Lattice Semiconductor provides a 5 nanosecond transition delay, limiting the fastest clock to about 200 MHz. As a 3.3V CMOS component, it automatically converts any input clock voltages to acceptable levels for the FPGA, protecting it from potentially unsafe external levels. Unlike the FPGA, it must be programmed before stuffing and cannot be reprogrammed afterward. In exchange for its convenient clock switching, the PLD introduces a major clock speed limitation and complicates the assembly procedure for new sequencer boards.

3.4 Implementation

While the preceding sections have discussed major design decisions in order of abstraction, the actual interdependencies were not a linear sequence. The logic and storage components were chosen first due to the proven success of configurable logic and on-board buffering in previous approaches [Hua03]. Commonly used bus standards in other scientific apparatus determined the initial list of I/O connectors: SMA, VersaLink, and I^2C. LVDS is less common in existing equipment; it represents a degree of freedom in designing the sequencer and a fixed compatibility standard for later waveform synthesizers and daughterboards. The Ethernet subsystem was added relatively late to the sequencer, requiring the MAC controller to be implemented in firmware. While it occupies two of the FPGA's four clock inputs, this no longer a concern due to the PLD clock switch; moreover, it obviates the need for a separate Ethernet controller board and its associated cables and headers.

In a previous design, the problem of low I/O pin counts was solved using a ball grid array (BGA) package for the FPGA, which had 400 pins and a smaller PCB layout area. However, the higher pin density increased the cost and difficulty of fabrication due to the use of blind and buried vias, and it was eventually abandoned.

The circuit schematics and PCB layout were both implemented using the Protel 99 SE package[7] (with Service Pack 6) from Altium. However, the use of Protel limits development to Microsoft operating systems, and third parties cannot view and modify the designs without a Protel license. Moreover, collaboration proved difficult without version control for the design files; entire databases were copied wholesale between different sites. In keeping with the open nature of the project, future versions should use free software for electronic design automation (EDA). Possibilities include the freeware version of Eagle from Cadsoft[8] or open source tools like the gEDA suite.[9] The latter uses open text-based formats that are suitable for versioning with, e.g., Concurrent Versions System (CVS).[10]

[7] `http://www.protel.com`
[8] `http://www.cadsoftusa.com`
[9] `http://www.geda.seul.org`
[10] `http://www.cvshome.org`

3.5 Contributions

The printed circuit board for the pulse programmer was developed in close collaboration with John Martinis, then at NIST Boulder, and Steve Waltman for their work with Josephson phase qubits. In particular, Martinis suggested the use of a PLD as a clock switch and the integration of the Ethernet interface directly onto the sequencer board. He also assisted with the PCB layout and debugged the hardware errors in the initial run of PCBs. Waltman designed the Ethernet interface circuit and suggested the use of inner planes as high-frequency bypass capacitance. The immediate predecessors of this project are the two pulse programmers by Wei-han Huang [Hua03]. The schematics and layout of the current work are based on an initial design by Steve Huang for his Master's thesis [Hua03]. An early proposal for the current design can be found in the conclusion of that document.

3.6 References

High-Speed Signal Propagation by Johnson and Graham [JG03] is an indispensable and very readable guidebook to advanced topics in high-speed digital design; it contains useful approximations and practical examples at every stage of the engineering process from the circuit schematic to PCB manufacturing. *Electromagnetic Wave Theory* by Kong [Kon00] is a mathematically rigorous and concise treatment of classical electrodynamics and transmission line theory. It can be supplemented with Someda's *Electromagnetic Waves* [Som99] for a more in-depth introduction to waves in media. *Signals and Systems* by Oppenheim, Willsky, and Nawab [OWN97] remains the classic introductory text on signal processing and is useful in analyzing circuits as linear, time-invariant systems and transmission lines as system functions.

Chapter 4

Firmware Design

The sequencer's reconfigurable hardware enables an intermediate abstraction layer, the FPGA firmware, which is presented in this chapter. The sequencer's goal of flexibility can be achieved by using firmware to route data between I/O interfaces and to defer programmability to the software layer; new functionality can be added later without hardware modifications. Firmware also provides access to hardware performance with minimal overhead compared to software drivers. Unlike the other two layers, the firmware is effectively hidden from the user; therefore, its ease-of-use requirements are directed towards a clean, maintainable design for developers.

To achieve the objectives above, the firmware should be divided into modules[1] connected by well-defined interfaces, called buses. Section 4.1 outlines the sequencer's bus structure augmented with three important primitive modules. Section 4.2 describes the two controllers used by the sequencer for off-board communication over the daisy-chain and a network connection. Section 4.3 builds on these transport channels to describe the endpoints that generate and interpret data: the processor cores. The two firmware processors, the AVR and PCP, execute user interface software and pulse programs, respectively. All firmware modules can grouped into six top-level blocks as shown in Figure 4-1. The I^2C and AVR controllers were available as free third-party cores, while the network, daisy-chain, SRAM controllers and the PCP were designed for the pulse sequencer.

[1] Sometimes called intellectual property (IP) cores.

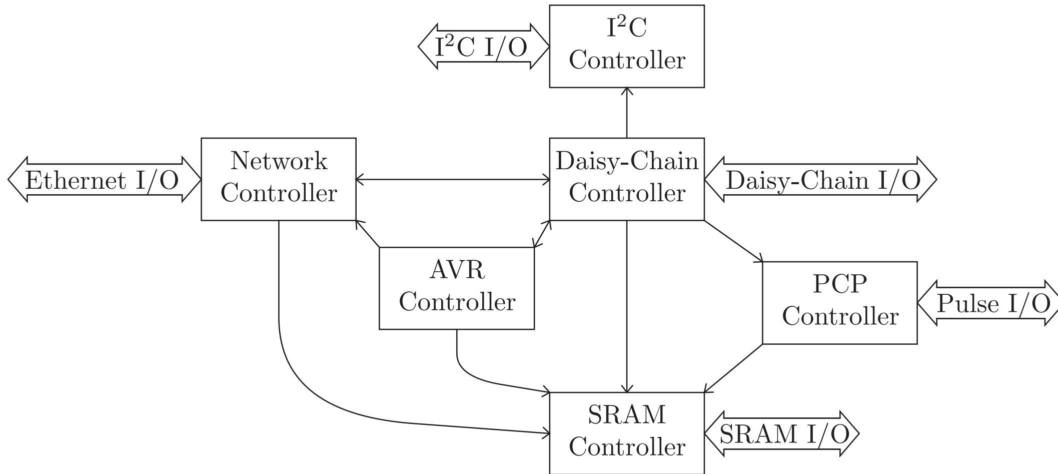

Figure 4-1: Top-level functional blocks of the firmware.

4.1 Bus Primitives

One of the recurring problems of System-on-Chip (SoC) designs is the logical interface between firmware modules for transferring control signals and data. A *bus interconnection architecture* is a standard protocol that defines how these transfers are clocked, what the signals are named, and when data is presented and latched. The sequencer firmware uses the Wishbone bus architecture [Her02] throughout its design to facilitate the replacement and independent testing of modules. Compared to other bus standards, it has the advantage of being simple, independent of hardware technology, and free for use without royalties.

The firmware layer can be viewed as chains of modules connected by buses The module which requests a data transfer (either reading or writing) is called a *master* and the module which services the request is called a *slave*. A module in a chain usually has two such interfaces since it is both a master and slave to other modules further down and up the chain, respectively. Two exceptions are shown in Figure 4-2. Sources, where data is generated or enters the chain, have a master on the hardware or software layer. Likewise, sinks, where data is consumed or leaves the chain, have slaves on layers other than firmware.

Recall that the main purpose of the firmware layer is to connect the multiple, heterogeneous hardware interfaces described in Section 3.2. In contrast, the Wishbone

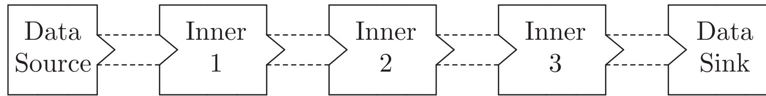

Figure 4-2: A chain of firmware modules acting as masters and slaves.

bus connects one master to exactly one compatible slave. In addition to using a uniform bus architecture, the sequencer must augment it with three additional bus primitives. Buses have variable widths and need a *sizer* to convert between them. In addition, a bus interface forces a module to be either a master or a slave, but modules can reverse their direction if necessary using a *proxy*. Finally, because the bus uses handshaking to throttle data transfer, chains that split or merge require an *arbiter* to serialize a data transfer. The use and implementation of these primitives are discussed in the remainder of this section.

4.1.1 Sizers

Consider that the Ethernet receive interface, which is a nibble (4 bits) wide, is too narrow to directly transmit data to the 64-bit pulse output interface, ignoring for the moment communication protocols. Conversely, the 8-bit feedback inputs are too wide to directly transmit to the 4-bit wide Ethernet transmit interface. Somewhere in these chains, a sizer module[2] must be interposed to translate data words from one width to another. In the sequencer firmware, sizers are only used to convert internal memory widths (usually 8 bits or 16 bits wide) into the 32-bit external SRAM interface by shifting virtual words into a larger physical word. These sizers must perform binary address conversions as well as memory sizing, as described in Section B.2.

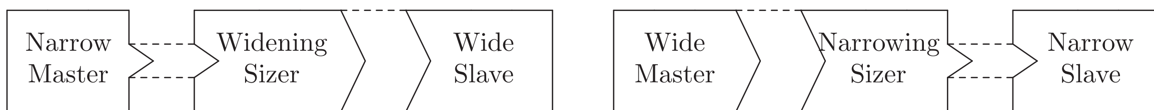

Figure 4-3: A sizer converts from one data width to another.

The two kinds of sizers convert from a narrow data stream to a wide one or vice versa, as shown in Figure 4-3. The widening type is sufficient to perform all sizing

[2]The descriptive but not widely-used term "sizer" is borrowed from the OpenCores project.

for the following reasons. In most cases, slave widths are wider than master widths; in the remainder of cases, the narrowing sizer can be implemented by reversing the direction of the first type and using the second busy primitive, a proxy.

4.1.2 Proxies

Certain modules are constrained to have one direction (master or slave) but must connect to other, non-complementary modules. For example, data received over the network is asynchronous with respect to processors, but must be read synchronously between these two masters. The reverse case is asymmetrically: data is transmitted synchronously, but it is desirable to make this process asynchronous by adding an intermediate slave. Therefore, the two kinds of *proxies* in Figure 4-4 are needed: one to convert a master into a slave and one to convert a slave into a master.

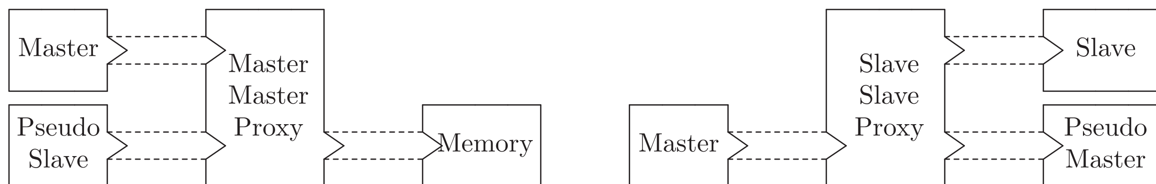

Figure 4-4: A proxy can convert a master into a slave and a slave into a master.

Using a slave-slave proxy, the network receive master can write data into a memory buffer, which is a slave, and signal the processor master that a packet is ready. The processor master must continually poll the proxy to discover and process this incoming data from its pseudo-slave. In the opposite direction, a master-master proxy and a memory buffer are needed to act as a pseudo-master between the processor (a master) and the network transmit interface (a slave) due to efficiency concerns in Section 4.3.1.[3] Although sizers and proxies can change data widths and data directions, they cannot enable separate chains to interact. Thus, a third bus primitive is needed to split and merge chains.

[3]Note that conventional processors and networking hardware also use a proxy called a Direct Memory Access (DMA) controller to achieve the same functionality using either hardware interrupts or polling.

4.1.3 Arbiters

One of the requirements for the firmware is that some physical interfaces must be shared among several logical interfaces, which requires multiplexing in two directions. As an example, more than one module requires exclusive access to the network interface for transmitting packets at different times. Conversely, an incoming network packet must be decoded and passed on to the correct receiver. If a bus standard only defines a point-to-point interconnection between one master and one slave (such as Wishbone), bus arbitration is necessary to ensure that at most one master is given access to at most one slave at any time. At the same time, all master and slave combinations must be possible.[4]

Both kinds of arbiters can suffer from starvation. In many-to-one arbiters, an unfair scheduling algorithm can starve one or more requesting masters in favor of others for an unbounded period of time. The sequencer firmware uses priority scheduling, in which masters are granted service in order of a fixed ranking, because it is simple to implement. In theory, it is not completely fair, but in practice no master requests service often or long enough to starve a lower-priority request. One-to-many arbiters can stall if the single master tries to address a non-existent slave. If the address space of slaves is not complete, the arbiter must act as a pseudo-slave for the unused addresses. In the current implementation, data addressed to unknown slaves is acknowledged and discarded. Both kinds of arbiters are represented pictorially in Figure 4-5.

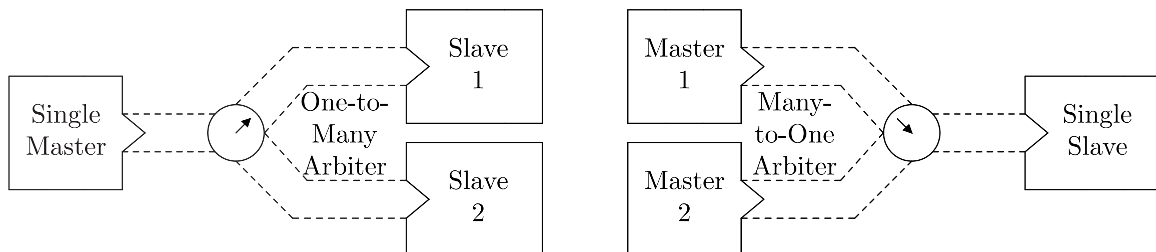

Figure 4-5: An arbiter can serialize transfers from many masters (one master) to one slave (many slaves).

[4]The appendix of [Her02] contains a good discussion of various shared bus schemes. More complicated many-to-many scenarios do not occur in the sequencer firmware, so simple multiplexing suffices.

4.2 Communication Stacks

Equipped with the bus primitives in the previous section, the sequencer firmware adds communication *stacks*, collections of modules to generate and parse a related "ladder" of protocols. Stacks have two anti-parallel chains for transmitting (carrying data from an internal source to a physical interface sink) and receiving (carrying data from a physical interface source to an internal sink). Data moving to and from a physical interface can be treated as purely encoded serial data, whereas data from an internal source or sink is purely decoded parallel data.

The two stacks in the firmware are the daisy-chain controller and the network controller. The daisy-chain controller uses the Pulse Transfer Protocol (PTP) to pass messages among devices; it can be transported between separate devices using a custom wire protocol or over an Ethernet/IP/UDP connection from the user. The network controller makes Internet communication with the user possible with an Ethernet Medium Access Controller (MAC) and an IP/UDP stack. Each controller's protocol stack roughly corresponds to the Open Systems Initiative (OSI) reference model as shown in Table 4.1. Each communications controller includes an application layer for bootstrapping in addition to software running on the AVR.

OSI Layer	**Functions**	**Network Layer**	**Daisy-Chain Layer**
Physical	Connector interface, channel modulation.	Ethernet PHY, RJ45	LVDS and RJ45
Datalink	Logical link, medium access	Ethernet MAC	Daisy-chain link UDP/AVR proxies
Network	Global addressing, routing	IP	PTP routing
Transport	Multiplexing, reliability	UDP, TCP	None within daisy-chain
Application	API for high-level functions	DHCP, AVR	PTP server

Table 4.1: Layers in the network and daisy-chain protocol stacks.

4.2.1 Daisy-Chain Controller

Multiple sequencer devices can be connected in a serial daisy-chain to allow the user to add and remove channels in a modular fashion; a three-node chain is shown in Figure 4-6. Devices in a chain can be synchronized to run the same pulse programs and are controlled by the user through a single network interface.

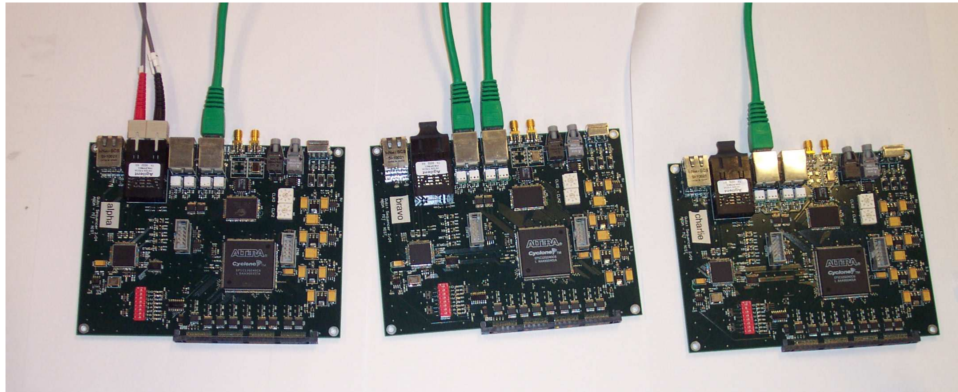

Figure 4-6: Three sequencer devices connected in a daisy-chain.

Each device has two physical interfaces for a single master and a slave, and each interface has two wire pairs (for input and output, respectively). A device which detects incoming network packets assumes it has no master and becomes the *chain initiator*; it serves as the entry point for the entire chain, and its AVR core runs the web interface which communicates with the user. A device which detects no slave becomes the *chain terminator*. This can be the same as the initiator in a one-device chain.

The daisy-chain controller passes messages between nodes using a store-and-forward approach which compares favorably with sharing direct wires. Benefits include transformer isolation for safety, matched impedance transfers, coexistence of pseudo-nodes like UDP and AVR, faster throughput due to registered logic, and dynamic address assignment based on the serial connection topology. The drawbacks include greater logic complexity, additional memory resources, and longer transmission latency.

The daisy-chain stack consists of a datalink layer, a routing layer, and an appli-

cation layer. The datalink layer provides medium access control to the physical layer for communicating frames between devices within the daisy-chain. Because frames can originate from AVR software or over the network, the daisy-chain UDP and AVR memory proxies as alternate datalink layers on the chain initiator. The routing layer provides addressing and routing between the PTP servers of all devices in the chain, the AVR of the chain initiator, and the user's host PC. The application layer performs higher-level actions, such as starting and stopping other firmware and software modules; it is implemented by the PTP server.

Datalink Layer

The datalink layer defines the semantics of the wires in each interface and provides medium access control. In each pair, one wire acts as a clock/strobe while the other acts as a data/cycle line.[5] The serial wire protocol uses these two anti-parallel pairs to transmits octets least-significant-bit first as shown in Figure 4-7. Each signal change involves a 4-way handshake to ensure that both master and slave proceed in lockstep. The master can abort a transfer at any time by signaling a stop.

This scheme is half-duplex because of master-slave synchronization. Clock recovery simply becomes handshaking, but at the cost of reduced bandwidth and the difficulty of recovering from corrupt transfers. Moreover, because this layer has finite storage to buffer messages, it provides medium access control by serializing both interfaces. At any time, the device is only receiving from a master, transmitting to a slave, transmitting to a master, receiving from a slave, or idle, in that order of precedence.

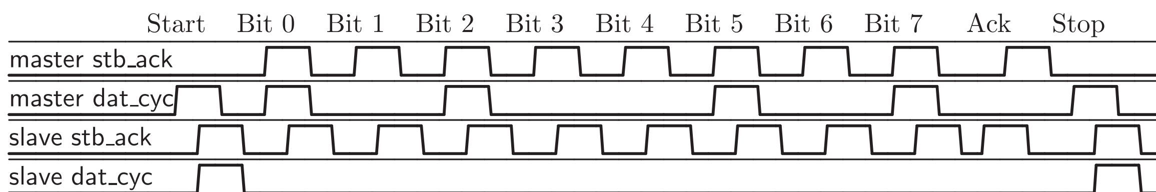

Figure 4-7: Daisy-chain wire protocol transferring one byte.

[5]These signals are called stb_ack and dat_cyc, respectively, after analogous Wishbone functions.

Routing Layer

The payload transferred by the datalink link layer is a frame for the Pulse Transfer Protocol (PTP), which is encoded and decoded by the routing layer and application layer. PTP currently refers to both the routing and application protocols, as there is little reason to use one without the other. The routing layer only performs addressing and not fragmentation; the minimum frame size is the header length (10 octets) and the maximum size is 984 octets.[6]

The fields of the frame are shown in Figure 4-8 and are transmitted from left to right in big-endian order. The major and minor versions of the frame indicate what version of the firmware is being run by the sender. The opcode indicates the type of frame and the interpretation of the payload. The zero field is reserved for future use and should be set to 0x00. The length is a two-octet field, most significant first, indicating the total length of the frame including both the header and the payload. The remainder of the frame is the opcode-dependent payload of variable length.

Field	Source ID	Dest ID	Major Ver.	Minor Ver.	Op-code	Zero	Total Length		Unused		Payload
Octets	1	1	1	1	1	1	2		2		variable
Address	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA ...

Figure 4-8: Fields of a Pulse Transfer Protocol frame

The source and destination IDs for a PTP frame are taken from the address space described in Table 4.2; by treating the user's host machine, the AVR of the chain initiator, and the broadcast address as pseudo-devices, PTP frames can be sent and received uniformly among these entities. Routing is performed by comparing the destination ID of an incoming frame with the self ID of the current device. If it is less, the routing layer forwards the frame to the master interface. If it is equal, the frame is passed up to the application layer. If it is greater, the frame is forwarded to the slave interface.[7]

[6] The current firmware implementation is limited to 1 KB of memory buffers for the network controller, and UDP and IP both have 20 octet headers.

[7] This is a simplification which ignores the bootstrapping case. A more accurate discussion can be found in Section B.3.

Entity	Address
User's host PC	0x00
AVR of chain initiator	0x01
PTP server of chain initiator	0x02
PTP server of next device	...
Broadcast pseudo-device	0xFF

Table 4.2: ID address space for the Pulse Transfer Protocol.

Three special routing cases must be considered for the chain initiator, the AVR core of the chain initiator, and the terminator. The initiator has two masters, the user's host PC and the AVR core, and it performs explicit comparisons for destination IDs of 0x00 and 0x01. The terminator discards any frames to its slave; otherwise it would stall while waiting for handshaking to complete.

The PTP routing layer, which performs this address routing, is shown in Figure 4-9. The terminal modules on the left connect to the datalink layer while those on the right connect to the application layer. The UDP and AVR interfaces are shown as terminal master and slave blocks, indicating two additional datalink channels for PTP other than the daisy-chain wire protocol. The slave link receiver is repeated in the figure to simplify the topology.

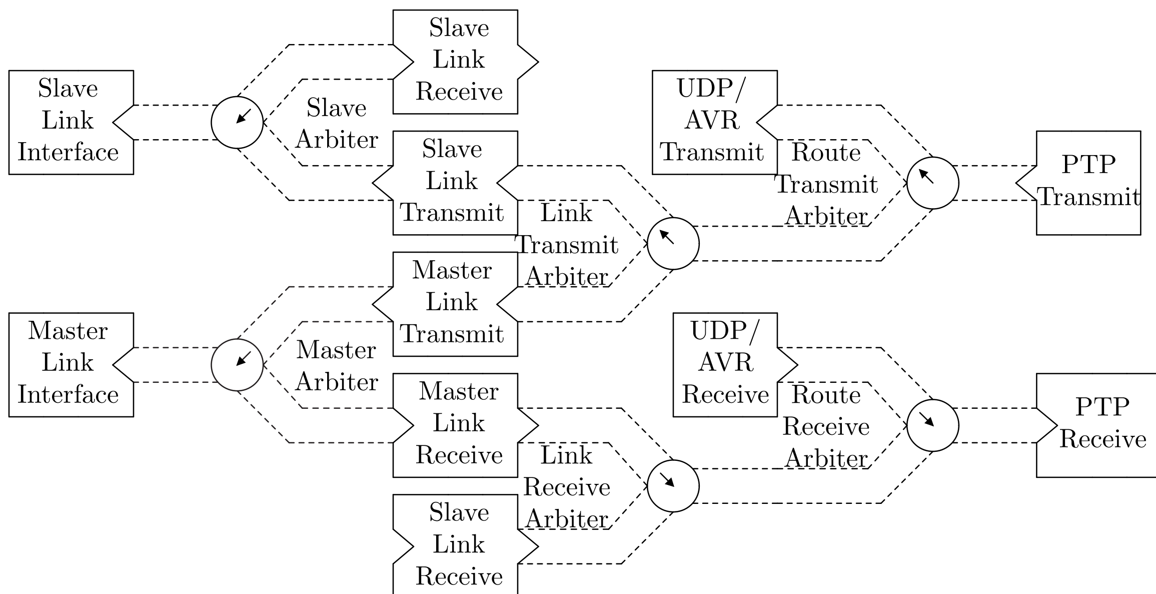

Figure 4-9: PTP routing layer transmit and receive chains.

Application Layer

The top layer of the daisy-chain controller is the application layer, which is implemented in firmware as the PTP server. This protocol has a request/response architecture suitable for clients and servers in other transport methods; for example, the UDP interface to PTP listens on port 8738. Each request opcode has a corresponding response opcode and is idempotent,[8] making it simple to recover from errors by placing the burden for retransmitting on the client. A detailed description of each opcode can be found in Section B.3.

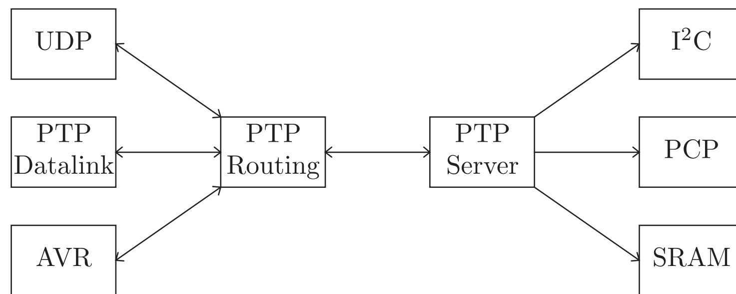

Figure 4-10: Interconnections within the daisy-chain controller stack.

The strength of the daisy-chain controller design is the decoupling between the application and routing layers, as depicted in Figure 4-10. The routing layer handles addressing among three different PTP clients and presents a uniform interface to the application layer, while the application layer provides its clients with uniform access to the resources of SRAM storage, I^2C communication, and the PCP. Not only does PTP make daisy-chain operation possible, it is a central firmware connective.

Because the entire daisy-chain stack operates internally, it is not bound by existing standards and represents a heavily-used degree of freedom in the design of the pulse sequencer. However, eventually the device must communicate with a fixed standard, the TCP/IP stack, in order to exploit the advanced user interfaces of desktop PCs and to be accessible over the Internet. This inter-operation is enabled by the network controller, the second communication stack in the firmware.

[8] Calling it multiple times in succession is equivalent to calling it once.

4.2.2 Network Controller

A shortcoming of early pulse programmers was tight coupling with a host PC using a proprietary interface, increasing the system's cost while decreasing its flexibility. In contrast, the pulse sequencer is accessible "out-of-the-box" from any PC using commodity networking hardware and software. A network controller in firmware contains protocol engines for standard Ethernet, IP, and UDP connections and can request a dynamic IP address using DHCP. It also enables the more complicated TCP and HTTP layers to be implemented in software for a user-friendly web interface.

The network controller contains independent transmit and receive chains which do not interact except at several endpoints. Figure 4-11 displays the receive chain using the bus primitives introduced earlier; the transmit chain is almost identical but flows in the opposite direction. The ARP module within the IP layer responds to address resolutions requests by transmitting replies and storing its own replies in a lookup table. The DHCP module extracts configuration data from received packets in order to transmit corresponding request packets. The PTP interface from UDP gives direct firmware access to the PTP server.

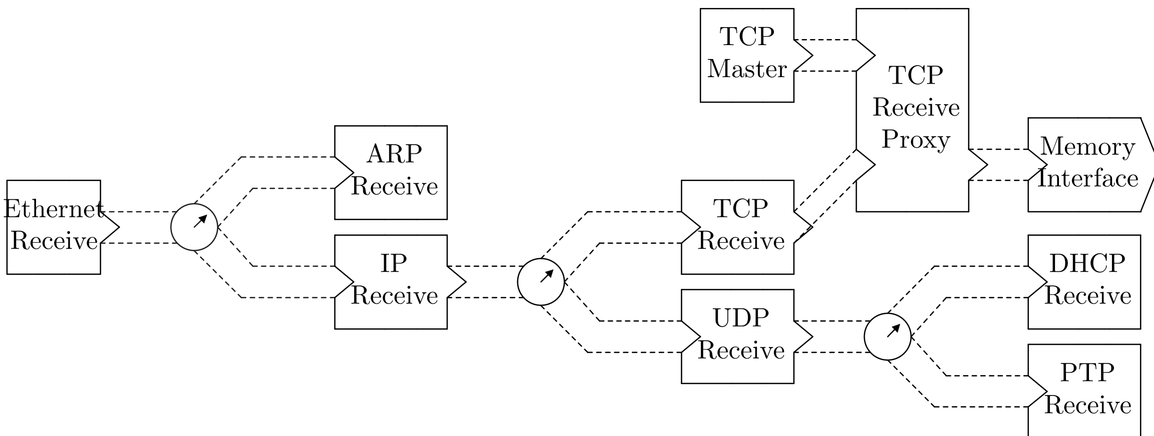

Figure 4-11: The receive chain of the network controller.

The network controller draws an unusual boundary between firmware and software at the TCP layer. Because the TCP specification requires handling many more error conditions than UDP, it is handled by software running on the AVR processor core. The AVR treats the network controller as just another peripheral. It sends and

receivers Internet packets using a memory proxy and a buffer at a fixed address, allowing firmware and software to communicate asynchronously without blocking.

Both the daisy-chain and network controllers in this section contain layers of protocol engines that make communication possible, but they do not consume or generate data by themselves. On one side are the bus primitives and beyond them, the hardware layer, other devices, and the user. On the other side are the embedded processor cores, which are the execution engines within the firmware layer.

4.3 Processor Cores

The processor cores represent the most abstract firmware modules because they enable a higher software layer. These are virtual processors have the advantage of dynamic modifiability and replaceability over discrete microcontrollers, since it would be prohibitively expensive to manufacture custom processors in silicon. On the other hand, they have a disadvantage of higher power consumption and slower speed.

The sequencer contains two of these processor cores. The AVR runs the general-purpose user interface and is described in 4.3.1, while the PCP runs timing-critical pulse programs and is described in 4.3.2. The AVR is an adapted third-party core while the PCP was written from the ground up to support special instruction timing modes. Both cores are described in their own subsections below.

4.3.1 AVR Controller

The sequencer's general-purpose processor emulates Atmel's ATmega103 8-bit AVR microcontroller. The modified controller is shown in Figure 4-12. All six I/O ports are retained for interfacing with the firmware network and daisy-chain controllers, and the AVR core implements the full register file and instruction set of the `avr3` family. All other built-in timers, I/O peripherals, and interrupt handlers were removed. The modified AVR controller is shown in Figure 4-12.[9]

[9]The use of "core" in the figure refers more specifically to the part of a processor that contains the instruction decoder and the register file, not just a firmware IP core as referred to everywhere else.

Because the AVR architecture has a separate program and data address space, a firmware emulation needs to support two ports: a read-only 16-bit address space of 16-bit words and a read-write 16-bit address space of 8-bit words. However, most FPGA designs only have a single port to external SRAM, including the pulse sequencer.

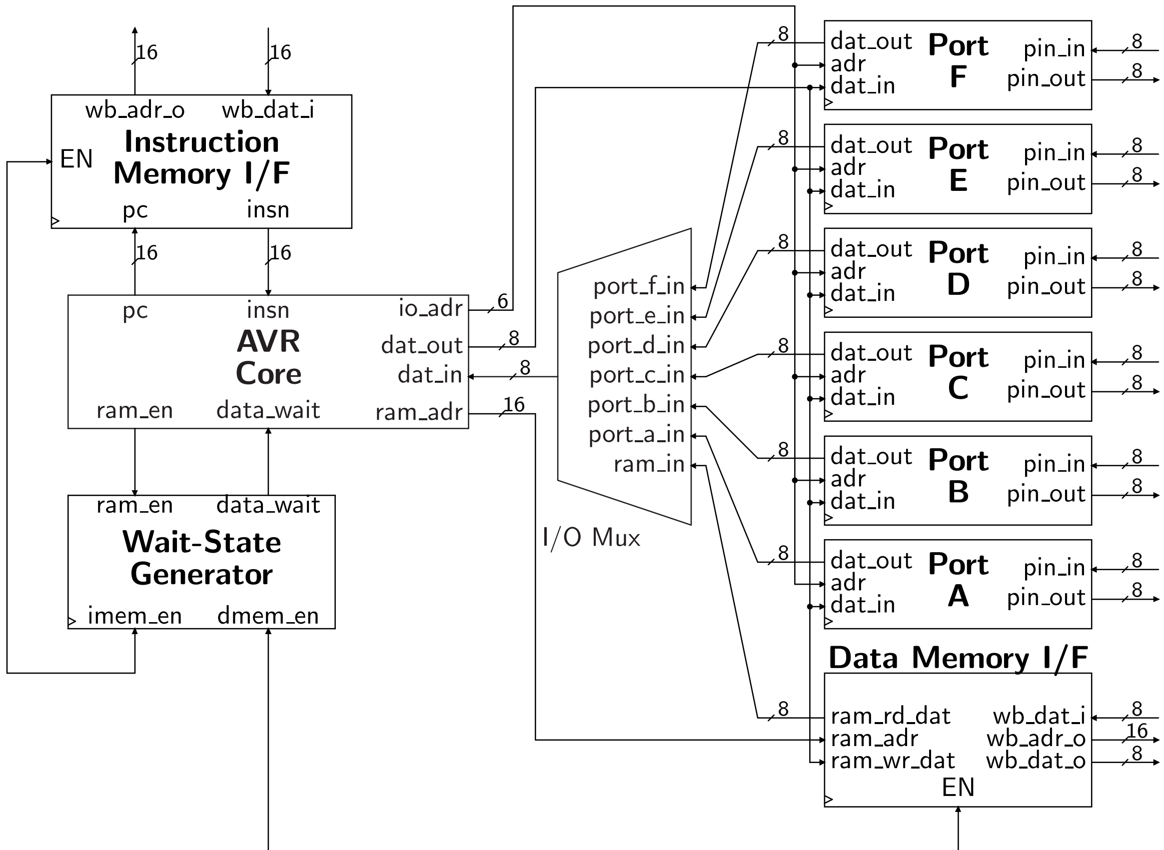

Figure 4-12: Machine model of the AVR controller adapted for the sequencer

The core's original author used a statically-compiled binary image as program memory and used the single SRAM port for data memory. This approach is undesirable for two reasons: it requires recompiling the entire firmware to update the software, and it consumes gates that are needed to implement other cores. One alternative would be the use of built-in FPGA RAM for one or both of the needed memories; however, the AVR does not need to run at fast clock speeds and has a small data width, neither of which is true of the other processor core, the PCP.

Given these resource constraints, the sequencer's design chooses to multiplex the external SRAM between the AVR's instruction port, its data port, and other mod-

ules which need non-critical memory access. This involved adding two Wishbone-compliant memory interfaces which connect to the SRAM controller. A wait-state generator was implemented to control these interfaces and stall the AVR core until either instructions or program data were available on its internal memory ports.

4.3.2 Pulse Control Processor

The Pulse Control Processor (PCP) is a specialized architecture supporting a simple instruction set and optimized for timing pulses. It provides a a general framework of instruction formats and memory layouts, which can be customized for particular PCP families. It has a RISC architecture and exploits a configurable number of registers and memory address widths. There is one dedicated register for the pulse output value. For simplicity of loading plain binaries, it has a unified instruction and data memory.[10] These parameters are quantified in Table 4.3 for the 64-bit version of the architecture, called PCP64.

Note that the PCP64 can switch all 64 hardware outputs simultaneously from a register, as well as associate a delay with each pulse instruction. In addition, pulse values can be encoded directly in the instruction due to the large number of available bits.[11] This achieves smaller minimum delay and faster minimum pulse durations compared to other approaches which load timers and pulse outputs as separate instructions. The choice of a 64-bit width was determined by the FPGA hardware and its available I/O count. The instruction width characterizes an architecture by determining the binary format it accepts, the widths of its registers, and ultimately its performance: smaller width processors generally have a lower latency but a corresponding lower throughput.

The PCP64 family constitutes an assembler target for writing software. Specific machines within the family will support a subset of its parameters and instruction set, which the assembler can check statically. Currently there is only one machine, `pcp0`, which has a 5-bit register address width, 32 64-bit registers, an 11-bit memory

[10] Also known as the von Neumann architecture

[11] This is sometimes called a very long instruction word (VLIW) approach.

Parameter	**Allowed Range**
Register Address Width	$0 - 5$ bits
Register Data Width	64 bits
Register Count	$1 - 32$ registers
Instruction Width	64 bits (8 bytes)
Memory Address Width	$0 - 32$ bits
Maximum Program Size	2^{32} instruction words
Instruction Word Endianness	big-endian
Instruction Opcode Width	6 bits (plus 2 flags)
Instruction Opcode Count	64, with 4 variants each

Table 4.3: Parameters of the PCP64 architecture.

address width, and a maximum program size of $2^{11} = 2048$ words.

The `pcp0` implements three classes of instructions: data transfer (loading values from main memory into registers), flow control (jumping to program addresses either unconditionally or conditionally on feedback), and pulse output. These instructions and their assembly language mnemonics are introduced in the next chapter, but Appendix C should be consulted for a complete programming reference.

A programming model for these instructions is provided in Figure 4-13 showing the major components of this processor. The feedback inputs are similar to interrupt requests; the 64 pulse outputs, which can be loaded in 32-bit groups, have no analog in general-purpose processors. The three largest components in `pcp0` are described below: the program memory including the register file, the timer for delaying pulse values, and the instruction decoder.

Memory

The PCP program memory is implemented in built-in FPGA RAM to support full-speed (100 MHz) 64-bit access to data. Although programs could be run directly from the external SRAM by a two-stage fetch of 32 bits each, this would not allow 64-bit pulse values to switch simultaneously. If only 32-bit simultaneous switching is required, a PCP32 family could be defined which would be able to make use of a much larger address space (512k words minus the 64k memory used by the AVR).

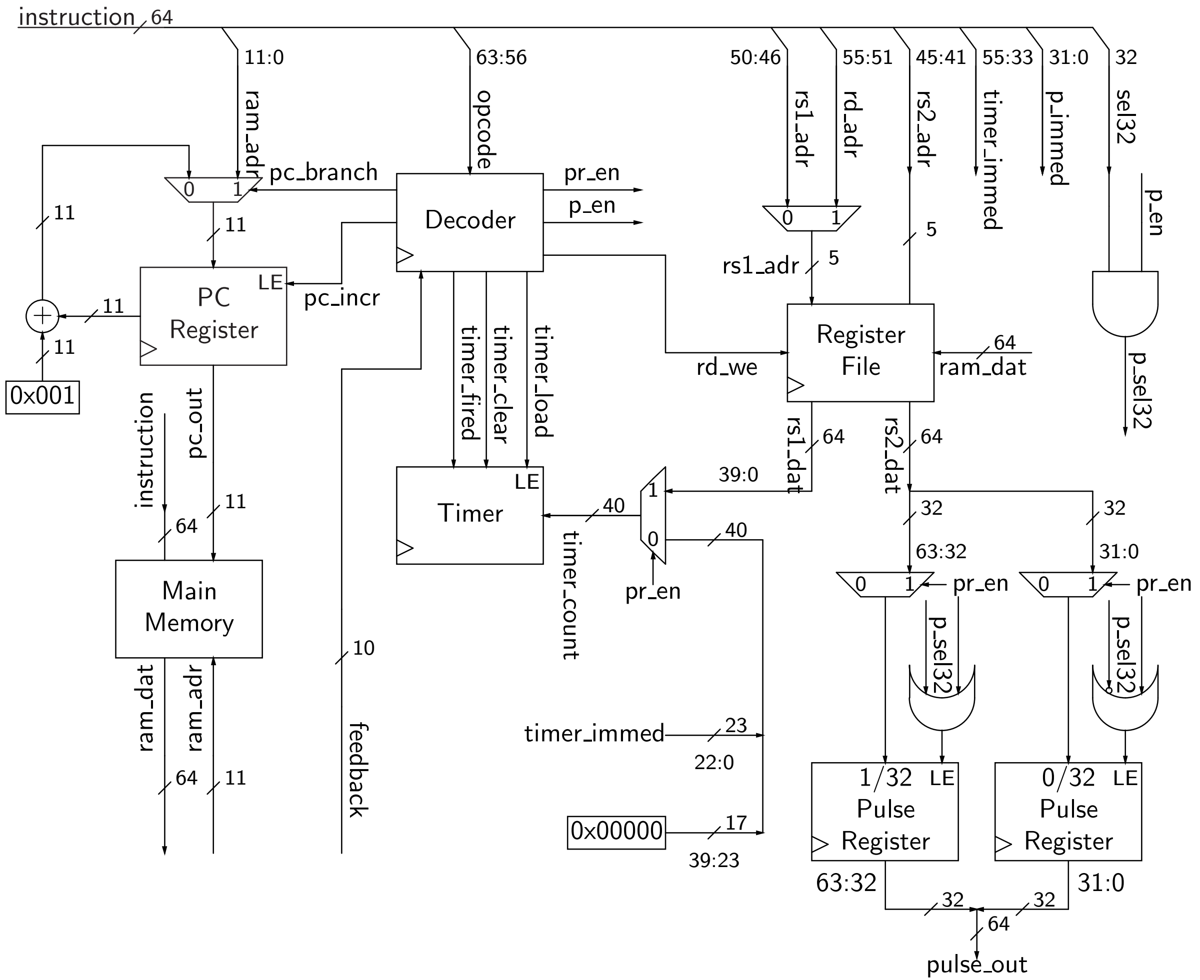

Figure 4-13: Machine model of the pcp0.

There are two separate address spaces that can be used by pulse programs. In the first, a 5-bit address selects a 64-bit register as the destination for a load or the source for a pulse. In the second, an 11-bit address select a 64-bit word in memory as the source for a load (a data word) or the destination of a branch (an instruction word). Since the program memory must be loaded from external SRAM, the PCP controller contains an asynchronous FIFO to perform inter-clock data transfers. This allows the PCP to be clocked much faster than and independently from other firmware modules.

Timer

The firmware uses a composite timer to control pulse delays and the execution of instructions in the pulse controller. An atomic subtimer is limited to 31-bits by

programming language limitations. Each one can be loaded with a value and clocked until the delay expires and it *fires*, producing a one-cycle output pulse. Due to the one-cycle delays of both the initial load and the firing event, the timer's master (the instruction decoder) should subtract two from its desired count. The composite timer consists of multiple subtimers cascaded together so that the firing of a faster subtimer enables the clock of a slower subtimer.

Timer Width	**Maximum Delay**
16-bits	655.36 microseconds
23-bits	83.89 milliseconds
31-bits	21.47 seconds
34-bits	171.80 seconds
36-bits	11.45 minutes
38-bits	45.80 minutes
40-bits	3.05 hours
62-bits	1,1462.35 years

Figure 4-14: Maximum delays for various timer widths at 10 nanosecond resolution.

Because the composite timer is implemented in programmable logic, it has a configurable width, which determines the longest delay it can conveniently and accurately produce in one firing interval. A comparison of maximum delay values and timer widths is given in Figure 4-14. The longest desired pulse delays (for NMR) are typically no more than 3 minutes, which would give an ideal timer width of 34 bits. The sequencer implements a slightly larger 40-bit timer since in practice it is more efficient to implement five 8-bit subtimers.

Decoder

As with any other processor core, the bulk of the PCP's complexity is found in its instruction decoder. It controls instruction fetching and decoding, performs reads and writes on the register file, loads the timer and receives its firing event, and outputs pulses. The `pcp0` has a two-stage instruction pipeline (fetching and decoding) and one branch delay slot.

Like conventional RISC decoders, the `pcp0` decoder runs general-purpose instruc-

tions in constant time. However, it must also run specialized pulse instructions with variable delays, and it must be able to overlap pulse and non-pulse instructions.[12] This overlap allows the `pcp0` to produce a train of pulses with zero delay in most cases.[13] The decoder cannot aggressively pipeline successive fetching and decoding stages because it must stall a pulse instruction at the fetching stage if the preceding pulse is not complete. Therefore, all instructions have a fixed two cycle execution time, while pulse instructions have a variable delay beyond this two cycles.

4.4 Implementation

The sequencer firmware is implemented in VHDL fixed up with macros in the GNU m4 language. VHDL was chosen for its strong typing, generic parameters, and support for design partitioning. Because it is text-based, it is also easier to control source code versions than a visual description, although it is initially harder to visualize and understand. Unfortunately, VHDL was developed in the 1980s before the object-oriented programming paradigm was widespread. It supports limited hierarchical design via composition, but in many cases it is more convenient for one design unit to inherit and specialize behavior without exposing additional ports.

In these cases, the GNU m4 macro language is used to parameterize commonly used design idioms, such as a pipelined memory master. The firmware is actually implemented as m4 source files which generate intermediate VHDL files; these are then compiled into a programmable design. The m4 macros were interpreted on the command-line using the Minimal GNU System to provide native POSIX development tools in Windows. The generated VHDL files were synthesized using the Quartus II design software from Altera,[14] version 4.1 with service pack 1, running on Windows 2000. Ideally, an open source synthesis tool would allow development and contributions from any platform. However, all current FPGAs and their synthesis tools use

[12]Pulse and non-pulse instructions are defined in more detail in the description of the event programming model on page 77.

[13]A minimum delay of one cycle occurs in the case of the shortest pulse duration, also one cycle.

[14]`http://www.altera.com`

proprietary, closed formats.

An initial design for the firmware layer sought to execute both general-purpose software and pulse programs on the same processor core, but this proved impractical. Due to its large instruction set, capable of multiple addressing modes and arithmetic operations, the AVR core can only be clocked at 25 MHz, giving a 40 nanosecond timing resolution. Simplifying the AVR was not an option, since it presented a well-defined development target for third-party software and an existing C compiler. Therefore, the PCP was designed as a custom core specialized for pulse timings.

4.5 Contributions

The following third-party controllers were used from the OpenCores project,[15] a collection of open-source FPGA modules and a community of contributing developers. The I^2C controller uses a Wishbone-compliant module with no modification. It was originally written by Frédéric Renet and is currently maintained by Richard Herveille. The module acts as a Wishbone master, uses 7-bit slave addresses, and operates at a maximum of 400 KHz. The AVR core and the original ATmega103 controller was implemented by Ruslan Lepetenok. Members of the OpenCores general mailing list[16] offered advice about FPGA programming, digital logic, and implementing Wishbone.

Some ideas about implementing a network stack in VHDL are taken from a similar project[17] by James Brennan, Ashley Partis, and Jorgen Peddersen at the University of Queensland. In particular, the current work also separates the stack into transmit and receive chains, and data is passed between layers as byte streams for simplicity.

The libnet packet construction library[18] by Mike D. Schiffman, version 1.1.1, was used to generate standards-compliant Ethernet frames, IP datagrams, and UDP packets. The libpcap packet capture library[19] enabled link-level access to the sequencer

[15] http://www.opencores.org
[16] cores@opencores.org
[17] http://www.itee.uq.edu.au/~peters/xsvboard/stack/stack.htm
[18] http://www.packetfactory.net/Projects/Libnet/
[19] The original UNIX library is maintained with the tcpdump utility at http://www.tcpdump.org. The port to Microsoft operating systems, wpcap, is located at http://winpcap.polito.it/

devices. The Ethereal network analyzer,[20] which depends on libpcap on both Windows and Linux, was used heavily with libnet to test the network protocol engines in firmware.

The idea of using an embedded processor core to output pulse instructions and run a web interface was suggested by Isaac Chuang. Early versions of the Ethernet MAC layer, I^2C controller, and SRAM controller were implemented in firmware by John Martinis.

4.6 References

Ashenden provides a comprehensive commentary on VHDL in [Ash04]; it is the recommended tutorial and reference for newcomers. This should be supplemented by the IEEE specification for VHDL [VHD02], which contains an exhaustive grammar of the language.

The Wishbone specification [Her02] is maintained by the OpenCores Organization; it contains examples and recommendations for Wishbone in common applications, including a multiplexed shared bus implementation. The I^2C protocol is a simple 2-wire, serial bus designed by Philips that is documented in [Phi00]. The detailed operation of Ethernet, including its frame structure and Media-Independent Interface (MII) can be found in the IEEE 802.3 standard [Eth02].

In understanding and implementing Internet protocols, the definitive sources are the Request For Comments (RFCs) from the Internet Engineering Task Force (IETF).[21] In particular, the following draft proposals and standards were used: Address Resolution Protocol (ARP) [Plu82], Internet Protocol (IP) version 4 [Pos81b], Internet Control Message Protocol (ICMP) [Pos81a], UDP [Pos80], Dynamic Host Configuration Protocol (DHCP) [Dro93], and DHCP-specific bootstrapping protocol (BOOTP) options [AD93].

[20] `http://www.ethereal.com`
[21] `http://www.ietf.org`

Chapter 5

Software Design

Software forms the highest abstraction layer on the sequencer device and is the topic of this chapter. It is responsible for satisfying most of the sequencer's goals for programmability and ease-of-use through pulse programs and the user interface, since these are the features most likely to change. Pulse programs complete the goal of feedback by conditionally starting or branching on feedback inputs made available by the hardware and firmware. The software does not participate in the performance and flexibility goals of the lower layers; it only introduces initial latencies to pulse timing, without affecting the relative accuracy or throughput of pulse durations.

Three different platforms for running software are associated with the sequencer device, as shown in Figure 5-1; two are virtual and one is physical. While the PCP's firmware architecture has been described in the previous chapter, this chapter begins with a discussion of its programming model and assembly language syntax in Section 5.1. Because of the PCP's optimized nature, it is not self-hosting; the user's host PC provides a more flexible development platform using the command-line assembler and tools in Section 5.2. The host also possesses a more intuitive graphical interface: a web browser. A third platform, the AVR, runs the corresponding web server and Common Gateway Interface (CGI) to communicate with TCP and PTP devices; this graphical interface is presented in Section 5.3. Although an overview of software usage is included in the following sections, the end user should consult the sequencer's technical manual[Pha04] for more detailed operating instructions.

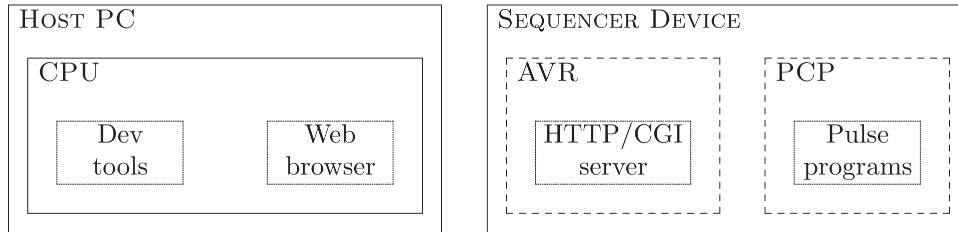

Figure 5-1: Execution platforms for the software layer.

5.1 Pulse Programs

Unlike conventional software, which usually places no constraints on execution time, pulse programs are explicitly designed to control events at specific times. Similar to real-time embedded systems, pulse programs require both predictability (a completely deterministic schedule) and performance (the ability to schedule many events accurately and precisely). Pulse programs at the level of assembly language must take into account the implementation details of a particular pulse processor, including its pipeline delays and available pulsing modes.

It is apparent that the software layer was the traditional boundary in system design even before the advent of firmware. Although lower layers have concentrated on the goals of performance, flexibility, and feedback, programmability and ease-of-use have largely been deferred until now. Moreover, previous chapters have described the design and implementation of subsystems which are meant to be used without modification; they are provided by the author. In contrast, pulse programs are a subsystem for which the *user* must provide the implementation; this section merely describes the available facilities and limitations.

First, requirements for the input capabilities of the sequencer are discussed as promised in Section 1.2. Next, the programming event model of the PCP is introduced, which motivates the PCP assembly language whose syntax is presented last in this section to make concrete the previously nebulous structure of pulse programs. An overview is provided for the available instruction set, which enables pulse programs to load data from memory, branch program flow control, and, of course, output pulses.

5.1.1 Encoding Pulses

To generate digital pulses, the user must first encode the desired sequence into a pulse program. The sequencer is agnostic to how the encoding is interpreted by a connected daughterboard, such as a waveform synthesizer, but for illustration, assume that the pulses will control the amplitude of a carrier wave. Further, assume that 8 discrete levels and a resolution of 20 nanoseconds are required, resulting in three digital outputs and a 50 MHz clock. These parameters are illustrated in Figure 5-2.

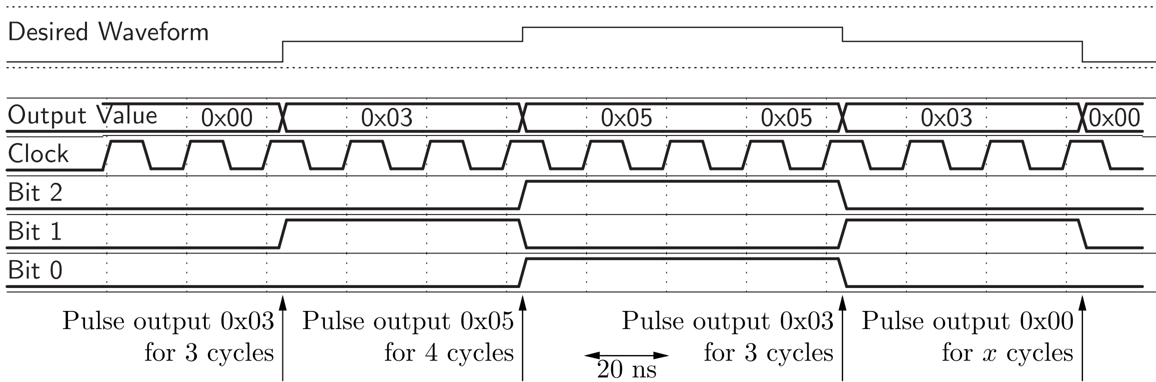

Figure 5-2: Encoding pulse outputs

It is important to note that a pulse program is not completely self-contained; its correct operation depends on two main external factors. First, the pulse outputs are specified with the assumption that they will drive a particular daughterboard of the sequencer which interprets the digital outputs a certain way. Second, there is no absolute time scale; the pulse durations are specified in cycles of the processor clock with the assumption that the provided clock has a certain frequency. It is a good idea to document these two factors as source file comments for each pulse program. This practice will increase the portability and maintainability of pulse programs.

5.1.2 Pulse Events

In the previous section, pulse sequences were encoded as a series of *events* consisting of changes in output value and separated by durations. In the PCP programming model, a pulse sequence always begins with an output value of all zeros for indefinite duration until the first pulse event. Each event specifies the new output value and the

minimum time duration until the next event; an event is caused by a special pulse instruction. There are three important, related consequences of this event model, which are depicted in Figure 5-3: zero output values, instruction overlapping, and minimum durations. Pulse instructions are denoted by the letters A, B, and C; non-pulse instructions are numbered 1, 2, 3, and 4.

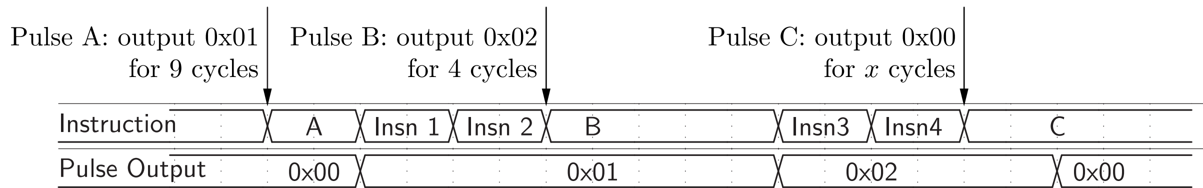

Figure 5-3: Pulse events and overlapping instructions.

Starting with the initial state of a pulse sequence, all output value changes are an event, including an output of all zeros. The first consequence is that there is no special value “in between” output values. To zero the output, e.g. at the end of a program for safety, the user must explicitly specify a zero pulse event as pulse instruction C does in the figure. In most applications, a value of all zeros represents a unique OFF state and any other value represents various kinds of ON states; moreover, the accurate durations of the OFF state are often just as important as those of ON states. The event model gives you the flexibility to ensure that both timing requirements are met.

The second consequence arises from having two kinds of instructions: pulse and non-pulse instructions. Non-pulse instructions are the kind found in normal processors, which perform loading data and branching; they provide a framework in which pulse instructions can be controlled. Non-pulse instructions do not directly affect the pulse output value, and their duration is fixed. However, they can *indirectly* affect pulse events because they take a finite time to execute.

The event model solves this problem by allowing non-pulse instructions to *overlap* with pulse instructions. Pulse instructions have two parts: an initial overhead of fixed duration to load a timer and output a value, and a delay of variable duration until the next event. The first part is like a non-pulse instruction and cannot overlap, but the second part can execute non-pulse instructions while waiting for a timer to fire.

If another pulse instruction is encountered before the previous duration is over, such as pulse instruction B in the figure, its execution is stalled.

The last consequence follows as a result of overlapping instructions. Pulse events specify a minimum duration and not an exact duration because overlapping instructions may prevent the next pulse event from being executed in time. In the figure, pulse event B specifies a minimum duration of 4 cycles but actually executes for 6 cycles because of two intervening non-pulse instructions. Even if there were no intervening instructions, the durations may still not be exact due to the fixed overhead of each pulse instruction. In the figure, this overhead is shown as two cycles, since pulse outputs appear two cycles after their corresponding instruction has been fetched. Any duration shorter than the fixed overhead will always be lengthened.[1]

The three consequences of the event model above can actually be expressed as a single consequence: generating accurate pulse sequences requires knowledge of instruction timing details. These details will provided at the end of the next section after the syntax and instruction set of the PCP have been explained.

5.1.3 PCP Assembly Syntax

Although the actual pulse programs consist of binary numbers, most users will write pulse programs using English-like mnemonics for the actual instructions and operands. These mnemonics form a low-level assembly language similar to those for general-purpose CPUs. An example pulse program demonstrating PCP assembly syntax is shown in Figure 5-4. Line numbers are added for further explication below but are not part of the pulse program. A gentler introduction to the same material can be found in the sequencer manual [Pha04].

PCP assembly language inherits all GNU assembler syntax due to its implementation, as described in 5.2.2. Lines 1-5 contain the `.equ` preprocessor directive to define symbols; they do not generate instructions. Comments extend from the semicolon character until a newline is reached. Operands to all directives and instructions

[1]The only exception is the minimum pulse duration of 1 cycle, which is always followed by at 1 cycle of an all-zero output value as explained on page 70.

```
 1             .equ ZeroReg, r0        ; r0 will hold zero value
 2             .equ TRIGGER_7, 0x80     ; 7th feedback bit
 3             .equ PATTERN, 0x12345678 ; output pattern while in loop
 4             .equ FourReg, r2         ; r2 will hold value four
 5             .equ EndReg, r1          ; r1 will hold the ending pattern
 6             ld64i ZeroReg, Data_Zero
 7             ld64i FourReg, Data_Four
 8             ld64i EndReg, Data_End
 9  Start:     btr TRIGGER_7, Break     ; break out of loop if trigger high
10             pr EndReg, FourReg       ; output end pattern, branch delay slot
11             p PATTERN, 0x04, 0       ; output loop pattern in lower half
12             p PATTERN, 0x04, 1       ; output loop pattern in upper half
14             j Start                  ; loop to beginning
15             nop                      ; branch delay slot
16  Break:     halt
17             pr ZeroReg, FourReg      ; branch delay slot; zero outputs
18  Data_Four: .quad 0x0000000000000004
19  Data_Zero: .quad 0x0000000000000000
20  Data_End:  .quad 0xABCDEF1234567890
```

Figure 5-4: An example pulse program in PCP assembly language.

are comma-delimited. Lines 6-8 contain the only data transfer instruction, `ld64i`, which loads the data addressed by the second operand and stores it into the register addressed by the first operand. Labels, terminated by colons, can be placed in the program for later use as data addresses for loading or destination addresses for branches. For example, `Start` is a branch address on line 9 used as a destination later on and `Data_Four` is a data address on line 18 used in a preceding `ld64i` instruction.

The `btr` instruction on line 10 implements conditional branching; in this case, it will direct program execution to the label `Break` on line 16 if the seventh trigger bit is high.[2] The instruction following `btr` is a branch delay slot. Due to the instruction pipeline, it is always executed regardless of whether the branch is taken or not.

Lines 11-13 produce the actual pulse outputs and demonstrate the two kinds of pulse instructions. The register pulse instruction, `pr`, can change all 64 bits simultaneously from a register addressed by the first operand, previously loaded with `ld64i`. The duration is also loaded from a register and is addressed by the second operand. The immediate pulse instruction, `p`, can only change the upper or lower 32 bits at one

[2]The corresponding value of the symbol `TRIGGER_7`, 0x80, is equal to 0x01 shifted seven bits to the left.

time. The `pr` instruction produces the output value of `Data_End` which was loaded into `EndReg`, namely `0xABCDEF1234567890`, which is 64-bits wide. The first operand is the immediate output value, the second operand is the immediate duration value, and the third operand indicates which 32-bit half of the pulse output to change. Thus, the first `p` instruction produces the value of `LOOP_PATTERN` in the lower 32 bits for a duration of 4 cycles, and the second `p` instruction duplicates it in the upper 32 bits for the same amount of time.

On Line 14, the `j` instruction unconditionally jumps to the `Start` address and continues the loop indefinitely. It also has a branch delay slot, which is filled with a null instruction (`nop`) that is guaranteed to do nothing except occupy a word in the program address space and two cycles of execution time. To make more efficient use of all program addresses, branch delay slots can be filled by rearranging the flow control instructions; `nop`s are often added, though, for simplicity.

The end result is that this pulse program will output three different values in sequence forever or until the seventh trigger bit is raised. In that case, control will branch to the `Break` location, where the `halt` instruction stops the PCP. Because it alters program execution, the `halt` also has a branch delay slot, which is customarily filled with a register pulse instruction turning all pulse outputs off (to zero) for safety. The duration is irrelevant as this last pulse persists until the PCP is reset.

The timing details of the PCP promised in the previous section are are summarized in Table 5.1. The duration for all instructions is the time it takes from one instruction fetch to a consecutive fetch. The pulse loading overhead is the time between a pulse instruction fetch and the appearance of its pulse outputs, coinciding with the fixed duration. Although these timing details may interfere with pulse timing, they are completely deterministic and allow the user to compensate for them at compile-time.

Fixed duration for all instructions	2 cycles
Immediate pulse loading overhead	2 cycles
Register pulse loading overhead	3 cycles
Pulse-instruction variable delay	$0 - (2^{40} - 3)$ cycles

Table 5.1: Timing parameters of the PCP.

5.2 Development Tools

After a pulse program is written in PCP assembly as described in the previous section, it must be compiled into a PCP binary and transferred to the sequencer device. These operations are performed using development tools which run on the host to take advantage of existing libraries and a convenient user interface. A typical software development session with the pulse sequencer is depicted in Figure 5-5.

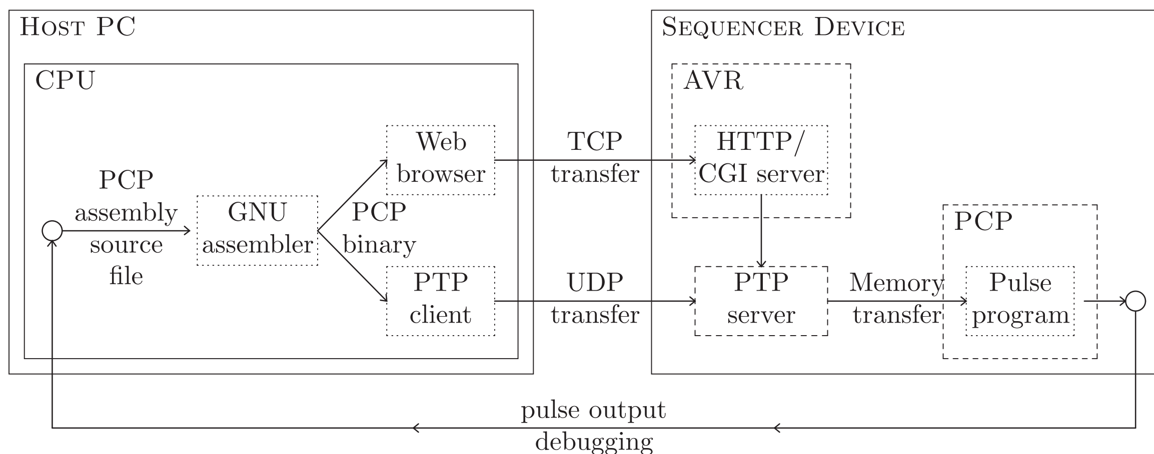

Figure 5-5: The software development process for pulse programs.

The user's first goal is to detect the IP address of any sequencer devices on the network and load the web server software onto them. Running the bootstrapping program, `ptpboot`, on any connected host computer accomplishes this.[3] Afterward, the development cycle begins by developing PCP assembly programs such as the example above.

To compile the necessary opcodes from higher-level mnemonics, a command-line assembler is provided based on the widely-used GNU binutils collection. After PCP binaries are generated on the host, the user's next task is to transfer them to the sequencer and device and run them. One way to do this uses a simple command-line PTP client, which is similar to the bootstrapping program, which transfers the binary to the PTP server directly using a UDP connection. Both of these command-line tools are described in the rest of this section along with examples of their usage.

[3]The bootstrapping process is omitted from the figure for simplicity, but must occur some time before the UDP or TCP transfer.

The second and easier way to transfer binaries and control the sequencer is using a web browser and the HTTP/CGI interface over a TCP connection. A description of this more user-friendly method will follow in the next section. As with any development activity, developing pulse programs is an iterative process using the pulse outputs as debugging feedback to correct or improve the next PCP assembly source file. Even with simulation, users should always verify pulse outputs with oscilloscope measurements.

5.2.1 PTP Client

The PTP command-line client was originally a debugging tool for the daisy-chain controller in firmware and the web interface software. However, it can also be used to efficiently script non-interactive actions from the command-line.

An example of the command-line syntax is shown in Figure 5-6. The program requires a command name plus optional command-specific parameters. The `discover` command detects the IP address of the sequencer device (in this case, 192.168.1.222); all other commands require an IP address and a device ID (recall that 02 denotes the PTP server on the first device). The `status` command requests the status of the given device. The `write` command loads the given program binary to the address 0x1a0000 in byte-addressable memory. The `trigger` command loads the newly written program into the PCP cache and waits for trigger source 9 (which is simply a START command). The `start` command is the awaited trigger which begins execution of the pulse program. The available commands roughly correspond to PTP opcodes described in Section B.3; complete online help is available by running the program with the `--help` switch.

The PTP client provides a command-line alternative to the web interface for transferring and controlling pulse programs. This particular sequence of commands also outlines the procedure for manually loading and running a pulse program, although the status command is not necessary; on the web interface, the process of triggering and starting are combined into the same form. However, command-line tools are currently the only option for developing and compiling pulse programs. These tools

```
> ptpclient discover
> ptpclient status 192.168.1.222 02
> ptpclient write 192.168.1.222 02 program.bin 1a 0000
> ptpclient trigger 192.168.1.222 02 9 1a 0000
> ptpclient start 192.168.1.222 02 pcp start
```

Figure 5-6: An example of command-line syntax for the `ptpclient` tool.

are presented in the next section.

5.2.2 GNU Assembler

The GNU binutils collection provides tools for generating and manipulating binary object files. It includes a linker, an assembler, a disassembler, and a binary file descriptor library for inspecting and converting object files. For simplicity, the sequencer only runs plain binary files on the PCP, so any relocations or loading information added by the linker must be stripped before it is executable. Since many of these tools share common code, only the assembler port is discussed.

The GNU assembler contains generic routines for parsing tokens, preprocessing macros, and emitting instructions that are common across all processors. It also contains *targets*, or architecture-specific opcodes and parameters for different combinations of processor families and output formats.[4] Within each family, specific machines may only implement a subset of available opcodes or have other peculiarities which are handled by the target using command-line parameters and static program checking. Therefore, an assembler port to the sequencer requires specifying the opcodes for a processor family (PCP64) and an output format (ELF64); consequently, the sequencer target is called `pcp-elf`, which is the prefix for the resulting command-line tools. Currently there is only one machine, the `pcp0`, but it could be extended to future machines with more instructions and capabilities; the `pcp-elf` target would be able to produce valid binaries for all of them.

[4]For example, a common target for PCs running Linux is `i386-elf`, meaning the binaries produced are Executable and Linking Format (ELF) files which run on i386 processors.

An example of the command-line syntax is shown in Figure 5-7. In the first command, the assembler for the `pcp-elf` target is called on the source file `program.s` to produce the ELF64 output `program.elf` for the machine `pcp0` (the linker `pcp-elf-ld` is called implicitly). The second command converts the ELF file into a plain binary file which is suitable for transferring to the sequencer device and running on the PCP. The programs themselves contain complete online help which can be accessed using the `--help` switch.

```
> pcp-elf-as -m pcp0 -o program.elf program.s
> pcp-elf-objcopy -O binary program.elf program.bin
```

Figure 5-7: An example of command-line syntax for the `pcp-elf` tools.

5.3 Common Gateway Interface

The previous sections have described the capabilities of pulse programs and their development environment using PCP assembly language. The missing step is to transfer pulse programs after being developed on the host PC to the sequencer device for execution. The embedded HTTP (web) interface allows the user to accomplish this and other tasks in a graphical, interactive way. It can call native functions using variables encoded in the HTTP post method and return the value of those functions as HyperText Markup Language (HTML) pages using the Common Gateway Interface (CGI) standard.

To achieve this, the software running on the AVR has three components: a driver for TCP that interfaces with the firmware network controller, a driver for PTP that interfaces with the firmware daisy-chain controller, and a web server augmented with CGI functions that call these two drivers. The AVR core is able to run the same executables as physical AVRs, simplifying its development; however, it requires large memory buffers to run its TCP and PTP drivers and to store a read-only filesystem. Normal AVRs have a 4K data memory limit and place global variables beneath the

downward-growing stack which will cause the stack to get stomped by the embedded web server. The alternate memory map used to avoid this is given in Section D.1.

PTP is a stateless protocol, as evidenced by the command-line interface in 5.2.1; all requests must fully specify all inputs, which can be repetitive across many requests. To automate this repetition, the CGI interface operates in two modes, as shown in Figure 5-8. The user can address a particular device in *selection mode* and then perform operations on that device for all subsequent requests in *operation mode*. These two modes and screenshots from the actual CGI interface are shown in the following two subsections. It is also possible to control the CGI interface directly without a web browser using the variables in Section D.2. Using this approach, however, does not detect all available devices and bypasses the error-checking provided by the web forms.

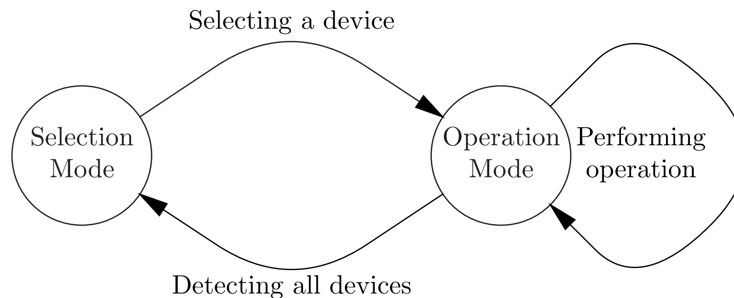

Figure 5-8: Modes of operation in the web interface

5.3.1 Selection Mode

Initially, the CGI interface sends a broadcast status request down the daisy-chain to discover all available sequencer devices, which is displayed to the user in selection mode. For a daisy-chain with only one device, there will be only one choice, as shown in Figure 5-9. Each device is shown with its unique ID from the space depicted in Table 4.2 and its current status, including whether it is a chain terminator or initiator and which processor cores (the AVR and/or the PCP) are running; the AVR core will always be running on the chain initiator in order to execute the CGI interface in the first place. After a device is selected, the interface will switch to operation mode and

address all future requests to that device. The selected device can be changed by re-entering selection mode at any time using the Devices link in the top frame.

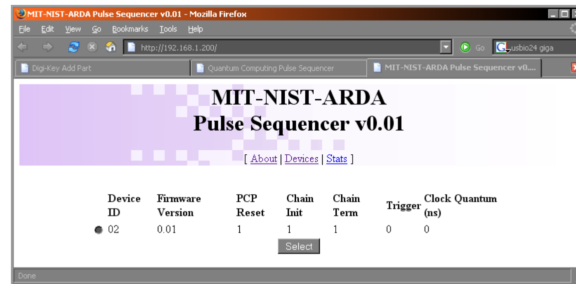

Figure 5-9: The device selection page of the web interface.

5.3.2 Operation Mode

Once a device is selected, the CGI interface persists in operation mode, allowing the user to perform the tasks shown in Figure 5-10. Separate forms are provided for refreshing the status of the device, reading memory contents, writing pulse programs to memory, starting and stopping a pulse program on the PCP, and addressing slaves over the I^2C bus. After each operation, the interface returns to an updated operation page showing the results. If a web browser is used, the selected device ID is guaranteed to be valid since operation mode is only reachable from selection mode.

5.4 Implementation

The primary platform for software development was Windows 2000, given the prevalence of Microsoft operating systems in scientific computing. However, the Minimalist GNU System for Windows (MinGW) was used to provide a POSIX development environment that is compatible with Linux and most UNIX variants. Thus, all source code can be configured and compiled identically on almost all existing operating systems. MinGW was used instead of the popular Cygwin POSIX emulation layer in order to generate native Win32 binaries; this would free users from having to download and install the Cygwin dynamically-linked library (DLL).

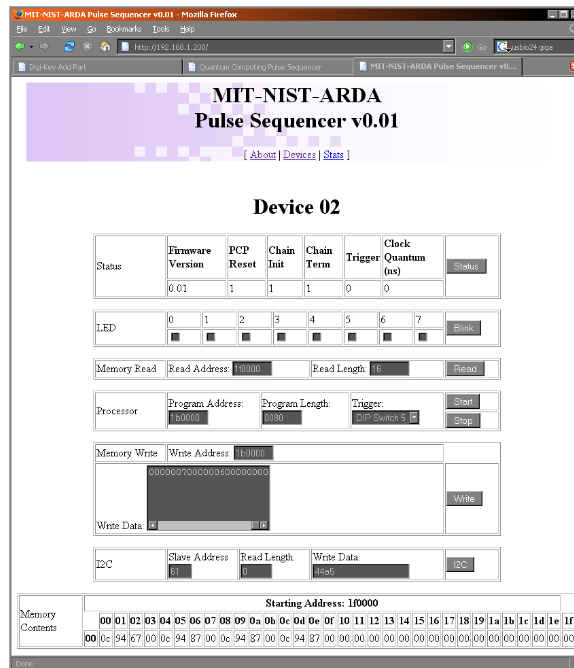

Figure 5-10: The device operation page of the web interface.

The development of the web interface used the AVR target of the GNU Compiler Collection (avr-gcc), also ported to Win32; this project is developed by the WinAVR team on SourceForge.[5] ANSI C library routines were provided by the AVR Standard C Runtime Library (avr-libc), maintained in GNU's Savannah Repository.[6] The availability of these tools made the AVR core a more viable choice than other freely-available microprocessor cores.

The web interface was tested with the Mozilla Firefox browser,[7] which is available on Microsoft, Linux, and UNIX operating systems. However, the debugging process was complicated by the fact that the target platform was not the same as the development platform; therefore, advanced tools like the GNU Debugger (gdb) could not be used to inspect the runtime state of the embedded web server and the CGI interface. In retrospect, it would have been wiser to construct a simulation framework on the x86 host and write unit tests for the AVR software.

[5] http://winavr.sf.net
[6] http://www.nongnu.org/avr-libc
[7] http://www.mozilla.org/firefox

5.5 Contributions

The uIP embedded TCP/IP stack and embedded HTTP server is written by Adam Dunkels at the Swedish Institute of Computer Science.[8] Only the TCP layer from the network stack is used, since the network controller implements an UDP/IP stack in firmware. Louis Beaudoin ported the uIP's machine-dependent portions to the AVR.[9] The GNU binutils collection is maintained by volunteers of the Free Software Foundation and is hosted by RedHat, Inc.[10] The PCP64 target is based on the (big-endian) PowerPC64 ELF target by Ian Lance Taylor of Cygnus Support.

5.6 References

The Atmel ATmega103 datasheet [Atm01] and the AVR instruction set documentation [Atm02] were useful in adapting target software to run on the embedded AVR processor core and interfacing it with other firmware modules.

The following Internet Engineering Task Force (IETF) standards were referenced in adding CGI support to the embedded web server: TCP [Pos81c] and HTTP 1.1 [FGM+99].

UNIX Networking Programming by W. Richard Stevens [SFR03] is the classic text for using the BSD sockets API on POSIX systems; portions of the command-line client use code from this book verbatim. The client was ported to Microsoft operating systems with the help of the Winsock Programmers' FAQ.[11]

Although no comprehensive document exists for porting the GNU binutils collection to a new target, the equivalent information is distributed across the GNU assembler user's manual [EF94], the Binary File Descriptor Library [Cha91], and the binutils source code itself.

---

[8] `http://www.sics.se/~adam/uip/`
[9] `http://www.laskater.com/projects/uipAVR.htm`
[10] `http://sources.redhat.com/binutils`
[11] `http://tangentsoft.net/wskfaq/`

Chapter 6

Measurements and Results

The previous three chapters have described the implementation of the three subsystems (hardware, firmware, and software) and how each one is optimized to perform a subset of the system goals. Considering each subsystem in isolation encouraged design modularity and focused the discussion of the sequencer thus far. However, the system must now be regarded as an integrated whole to quantify its behavior in terms of the previously-defined figures of merit. Of the five goals presented in the first chapter, the timing parameters for performance and feedback lend themselves most naturally to measurement; consequently their results are presented in this chapter.

First, Section 6.1 defines terminology, methods, and equipment which are common to all the remaining sections. Next, the skew between digital outputs is presented in Section 6.2 to determine the synchronization of the pulse output bits. Then the system's primary limiting factor, the resolution, is measured in Section 6.3. Both the minimum duration and the minimum delay depend on the resolution and are measured next; Section 6.4 discusses duration and Section 6.5 discusses delay. Next, triggering and feedback latencies are measured in Section 6.6. Related to the timing parameters is signal integrity and noise tolerance; hence, Section 6.7 measures the noise inherent to the sequencer's construction arising from both reflection and crosstalk. Finally, as a matter of practical concern, the sequencer's power consumption is measured in Section 6.8. With these measurements in hand, it is possible to evaluate the success of the sequencer's approach and implementation in the next chapter.

6.1 Prerequisites

All timing measurements in this chapter characterize how far a given parameter deviates from its nominal value. This difference has a deterministic, constant part called a *timing error* and a nondeterministic part that is modeled as a random Gaussian process whose standard deviation is called *jitter*.[1] Timing measurements are performed using an Agilent 5432B Infiniium Oscilloscope with a sampling rate of 4 GSa/s. All measurements use the 100 MHz on-board oscillator unless otherwise noted.

Most measurements take advantage of the sequencer's programmability by running a pulse program to exercise some quantifiable feature. Their source code is provided along with a concise explanation of their operation. Thus, these examples also serve a pedagogical purpose in helping users write their own pulse programs to perform similar tasks. While no knowledge of pulse program syntax is necessary to use the measured results, the interpretation of the results often rely on a familiarity with the design and implementation discussion in Chapters 3, 4, and 5.

These test programs have the following features in common. Most flip bit 0 of the output or other bits which are located on the top layer of the sequencer PCB. Loops are used to create a periodic signal for triggering the oscilloscope and averaging across many measurements. Macros or comments indicate configurable parameters which are changed between compilations to test a wider range of some timing parameter. The last instruction is always a zero pulse of an arbitrary duration that fills the branch delay slot of the `halt` instruction; this is a defensive programming measure to return the pulse outputs to a known safe state upon halting.

To simulate actual usage patterns, the test environment uses an LED test board photographed in Figure 6-1. This test board was designed and assembled by Steve Waltman and John Martinis at NIST Boulder; it acts as a daughterboard to the sequencer, connecting to it via the edgemount connector. In actual experiments, it would be replaced with a waveform synthesizer or some other interface board, but the use and arrangement of LVDS receivers will remain unchanged.

[1]This differs from the usual definition of jitter as the total range of timing errors, but it is a useful shorthand for our purposes.

Several common causes of timing error and jitter should be mentioned here to avoid repetition later on. LVDS chips and their differential signaling paths cannot be routed and placed uniformly on the PCB, and manufacturing tolerances can differ widely. Both can cause skew from one bit to another and are sources of deterministic error. Sources of random jitter can include electromagnetic interference, thermal noise, and system clock jitter. The most important source of jitter is introduced by the LVDS components, since all measurements from the sequencer are actually filtered through at least one round of LVDS conversion to reach a connected daughterboard. Some of these error sources have specific effects on certain measurements and are discussed where appropriate in the following sections.

6.2 Output Skew Measurements

Most pulse programs will depend on the sequencer outputting multiple synchronized bits operating in parallel, but in reality there is always some skew between these bits. These are not subtracted from later measurements, since the deterministic error introduced by PCB layout and LVDS conversion does not affect relative timings on the same bit. However, the maximum output skew determines the useful upper bound of the maximum clock speed, measured in the next section; increasing the resolution of pulse outputs beyond this will result in loss of synchronization.

A test program which flips all bits on and off as fast as possible with an equal duty cycle is shown in Figure 6-2; this code will also be useful in measurements of clock frequency dependence, such as the maximum clock speed in Section 6.3 and power consumption in Section 6.8. Because all output bits need to switch simultaneously, a register pulse instruction is used, which requires loading registers with output values and the shortest register duration, 3 cycles.

The average skew of each bits relative to bit 0 is shown in Figure 6-3. From visually inspecting the top PCB layer (Figure A-10), one can see the expected variation of PCB trace lengths shortening to a minimum and then lengthening out again. This correlates with the growing negative skew (indicating a shorter propagation delay)

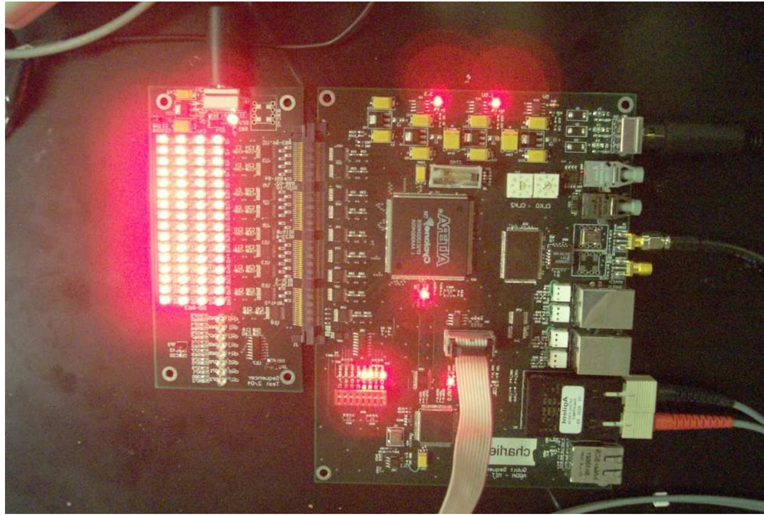

Figure 6-1: The LED test board constructed at NIST Boulder.

```
        ld64i r0, Zero
        ld64i r1, Out1
        ld64i r2, Time1
        nop
Start:  pr r1, r2
        pr r0, r2
        j Start
        nop                     ; branch delay slot
        halt
        nop                     ; branch delay slot
Out1:   .quad 0xffffffffffffffff
Zero:   .quad 0x0000000000000000
Time1:  .quad 0x0000000000000003
```

Figure 6-2: Test pulse program for measuring output skew and clock frequency-dependence.

until a minimum at bit 36, after which the skew becomes less negative but never returns to zero.

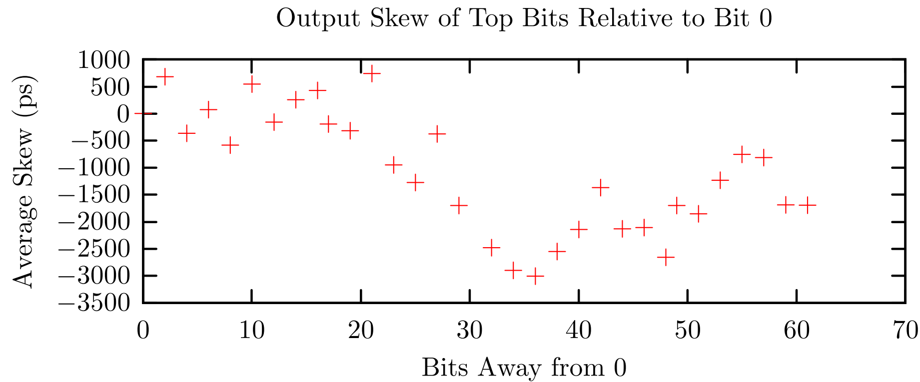

Figure 6-3: Plot of output skew.

6.3 Clock Speed Measurements

In digital systems, all timing parameters are based on clock speed, which is the reciprocal of resolution for the sequencer. Therefore equivalent clock speed measurements can be used to determine the finest resolution possible as a basis for other timing parameters. The hardware and firmware chapters have described a process of designing a PCB, selecting components, and programming modules to optimize this primary limiting factor of system performance, and the effectiveness of this approach is delineated in Table 6.1. The limits on clock speed can be viewed as a series of bottlenecks, where the slowest bottleneck determines the maximum speed for the whole system. To vary the clock speed, an external clock source was used in place of the on-board oscillator; an Agilent 8648B 2 GHz signal generator was used to drive a sine-wave clock through an SMA connector. The clock signal was amplified with a +24V Mini-Circuits ZHL-32A module to a fixed peak-to-peak amplitude of 5.0 volts, centered about zero. Such an input over-drives the clock switch PLD, which is a 3.3V component, but overcomes attenuation at higher frequencies.

Clock Frequency	**Cycle Time**	**Bottleneck**
330 MHz	3.03 ns	Clock switch
200 MHz	5.0 ns	LVDS (from datasheet)
164 MHz	6.10 ns	FPGA firmware (measured)
150 MHz	6.67 ns	SRAM (from datasheet)
105 MHz	9.52 ns	FPGA firmware (simulated)

Table 6.1: Maximum clock speed constraints.

On a bare PCB, only internal trace resonances constrain the maximum clock speed, but other constraints are added with each successive component in the clock path. The first drop in clock speed is reached by the addition of the clock switch PLD, which essentially acts as a low-pass filter for clock signals. Its maximum clock speed is reached when it no longer passes valid LVCMOS levels to the FPGA; this is depicted graphically in the power consumption measurements in 6.8.3. The next slowest components are the LVDS drivers and receivers which run at 200 MHz; the conversion from LVCMOS to LVDS and back limits the speed of any clocks transported this way. The SRAM is the next slowest hardware component, but in actual implementation it is only used for low-speed firmware modules such as the network and daisy-chain controllers.

The largest actual bottleneck is due to the FPGA firmware, which must route an external clock to many thousands of logic gates internally. In this case, the maximum clock speed is reached not when reflection overcomes LVCMOS thresholds but when the clock period drops beneath the setup and hold time constraints of the firmware. Although the FPGA can physically switch clock inputs as fast as 450 MHz, the PCP decoder cannot run that fast. Hence, clock speed represents a trade-off between adding new features to the PCP and minimizing the firmware timing constraints.

The maximum clock speed can be measured by plotting the output jitter versus clock frequency as done in Figure 6-4 and noting when the PCP decoder stalls; this occurs when the jitter falls to zero at 164 MHz and remains there for all higher frequencies. In timing simulations performed with the Quartus II design software, a much more conservative estimate of 105 MHz was given. Even the measured maxi-

mum clock speed, however, is not fast enough that the output skew in the previous section becomes a problem. The test program for this measurement is the same one used for output skew in Figure 6-2.

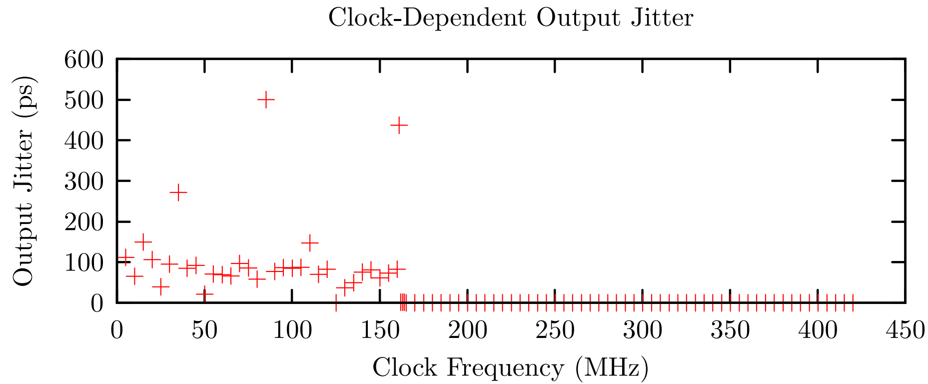

Figure 6-4: Plot of clock frequency-dependent output jitter.

Because jitter represents additive phase noise, it is useful to determine the base timing error and jitter of the system clock before measuring the noise contribution of other components. These are presented in Table 6.2.

Expected Period	10.0000 ns
Expected Frequency	100 MHz
Actual Period	9.9978 ns
Actual Frequency	100.0230 MHz
Error	2.2 ps
Jitter	31.26 ps

Table 6.2: Timing error and jitter for the on-board 100 MHz oscillator.

6.4 Duration Measurements

After the resolution, the most interesting figure of merit is the minimum duration, since it represents the real limit of the device's performance. There is a trade-off between making the resolution finer with aggressive pipelining and decreasing the minimum duration by shortening the decoding pipeline overhead. Since the maximum

clock speed and finest resolution was fixed in the previous section, the minimum duration can now be determined in this section. More generally, the minimum duration is the lower bound of the discrete spectrum of durations measured in this section.

Figure 6-5 contains the test code for making duration measurements. To achieve minimum delays of 1 cycle, the immediate pulse instruction `p` must be used. The first operand in each instruction is a hexadecimal number specifying a single unique bit; all other bits are turned off. The second operand uses a macro to easily change which 16 consecutive duration values are being tested in any particular compilation. The duration is specified in cycles which correspond to 10 nanosecond time steps. The third operand indicates that only the lower 32 pulse outputs are ever used.

```
        .equ BASE, 0x00            ; starting duration
Start:  p 0x00000001, BASE+0x01, 0
        p 0x00000004, BASE+0x02, 0
        ...
        p 0x08000000, BASE+0x0f, 0
        p 0x20000000, BASE+0x10, 0
        j Start
        nop                        ; branch delay slot
        halt
        p 0x00000000, 0x03, 0      ; branch delay slot
```

Figure 6-5: Test pulse program for measuring duration error and jitter.

The `j` instruction creates a loop so that the 16 bits which are in the lower half and located on the top side of the PCB toggle different durations, which can be measured for both timing error and jitter, as done in 6.4.1 and 6.4.2. Three different compilations were used with `BASE` values of 0x00, 0x10, and 0x20, respectively, for a total of 48 durations. Linear fit parameters are given in Table 6.3 for later plots of error and jitter.

	Duration Error	**Duration Jitter**
Slope (ps / ns duration)	0.216	0.00212
y-intersect (ps at 0 ns duration)	9.244	73.006

Table 6.3: Linear fit parameters for duration error and jitter.

6.4.1 Error

Figure 6-6 plots the timing error versus duration for 1 cycle up to 48 cycles. As expected from the base clock jitter in Table 6.2, initial duration errors are negative. However, the error increases roughly linearly with duration as the additive error accumulates in each successive duration. The y-intersect is validated by giving a relatively small timing error for a theoretical zero nanosecond duration; the small slope of picosecond errors per nanosecond duration ensures that the percent error will approach zero for arbitrarily long durations.

Several special cases are worth mentioning. Pulse durations for one and two cycles are handled as special cases in the PCP decoder and have markedly more negative errors than durations of three cycles or more, which are handled more generally. A pulse duration of zero is not shown but results in a pulse duration of 10 nanoseconds (1 cycle), indicating that 10 nanoseconds is indeed the minimum duration for the sequencer. Because each instruction takes two cycles to fetch and decode, odd durations utilize a different path in firmware than even durations; however, the expected bipartite distribution is only seen in isolated stretches around 300 to 360 nanoseconds or 400 to 450 nanoseconds. This trend may become more apparent for longer durations, but at the low end its effects are obscured by the propagation delays of the PCB layout.

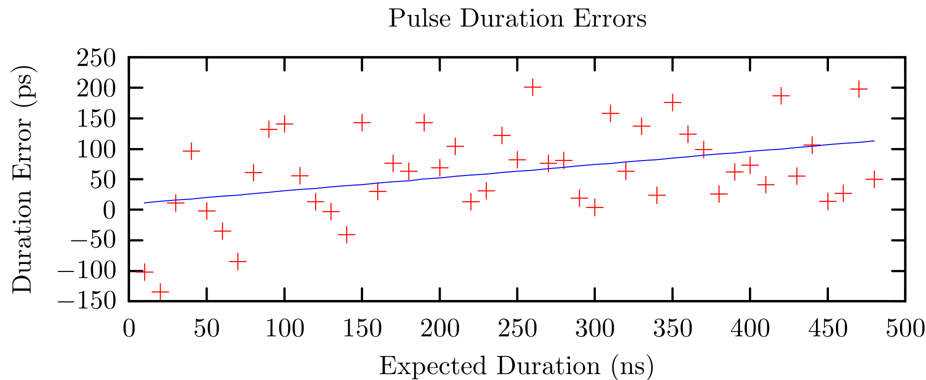

Figure 6-6: Plot of duration error.

6.4.2 Jitter

Figure 6-7 plots the jitter versus duration for 1 cycle up to 48 cycles. Outliers are due to PCB layout skew and initially, from the special-case handling of short durations. Jitter increase for successively longer durations is negligible. The predicted jitter at zero nanoseconds is greater than the system clock jitter, which can be attributed to the timer firmware.

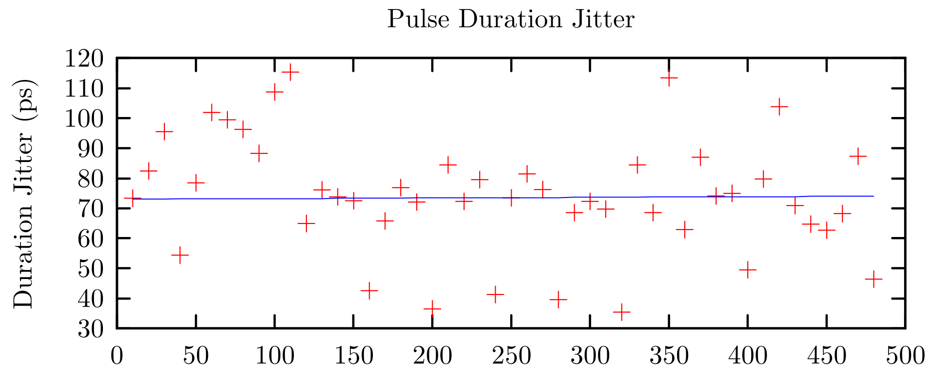

Figure 6-7: Plot of duration jitter.

6.5 Delay Measurements

While a duration is an explicitly-specified time interval, delays are the implicit overhead between durations. This overhead is incurred when the instruction decoder loads the timer, whereas duration measurements in the previous section characterize the PCP timer after it is loaded. The test code for measuring delay is shown in Figure 6-8. Bit 0 is toggled on for a duration of one cycle. Note that the immediate pulse instruction for 1 cycle is a special case which zeros all pulse outputs afterward; this is an effective 10 nanosecond delay after all pulses of 1 cycle and represents the minimum delay of the sequencer. For durations of 2 cycles or greater, the pipeline of overlapping instructions described in Section 5.1 ensures that there is theoretically zero delay between pulses, which will be useful in explaining the linear fit later on.

Delays can be greater than the minimum value due to non-pulse instructions inserted between pulse instructions. This is the rationale behind the test code, which inserts nops to stretch the delay between consecutive pulses in 2 cycle (20 nanosecond) intervals. Error and jitter are measured against this implicitly-specified nominal delay in the rest of this section. Although it may seem unnecessary to measure error for an overhead, writing deterministic pulse programs requires both accurate durations and delays. Linear fit parameters for both error and jitter are provided in Table 6.4 for plots in the remainders of this section.

```
Start:  p 0x00000001, 0x01, 0
        nop
        ... ; insert nops here to stretch the loop
        nop
        j Start
        nop
        halt
        p 0x00000000, 0x03, 0
```

Figure 6-8: Test pulse program for measuring delay error and jitter.

	Delay Error	**Delay Jitter**
Slope (ps / ns delay)	0.0120	0.035
y-intersect (ps for 0 ns delay)	0.5697	52.373

Table 6.4: Linear fit parameters for delay error and jitter.

6.5.1 Error

The results of delay error are shown in Figure 6-9. As with duration, systematic timing errors accumulate in successively longer intervals, but the rate of increasing error diminishes rapidly against increasing delay. The extremely small timing error for a zero nanosecond delay corresponds to duration jitter, since there is no delay between pulses of duration greater than 1 cycle.

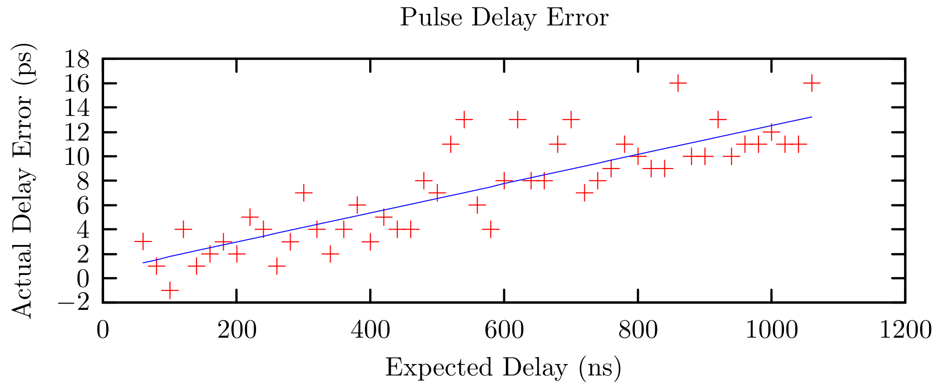

Figure 6-9: Plot of delay error.

6.5.2 Jitter

The corresponding results for delay jitter are plotted in Figure 6-10. The average jitter increases slightly with increasing delay as expected; however, there is a bipartite distribution between cycles with an even or odd number of `nop`s. This is due to jitter inherent to decoding each type of instruction. A pair of identical consecutive instructions will cancel out part of this jitter, while an unpaired instruction will exhibit all of it. Corroboration is provided by the lower jitter for even numbers of `nop`s relative to odd numbers as seen in the graph.

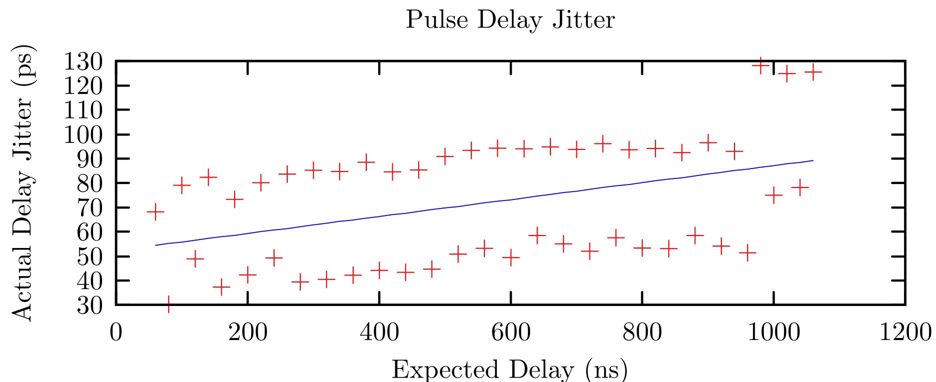

Figure 6-10: Plot of delay jitter.

6.6 Latency Measurements

The measurements in the previous three sections characterize the throughput, or the steady-state overhead, of the pulse output pipeline. However, the pipeline also exhibits overhead when starting or stopping called *latency*. Latency in the pulse sequencer can occur during triggering, feedback, and LVDS transport.

Even in traditional pulse programmers, throughput alone does not describe the system's behavior even for simple, linear pulse sequences. Most devices, including the sequencer, have the capability of waiting for an external trigger before starting a pulse program. However, one of the sequencer's chief advantages is that it also support feedback, allowing programs to take multiple actions based on external events. There is an inherent overhead associated with detecting and reacting to these events which is measured in the rest of this section.

Another distinctive feature of the sequencer is its use of LVDS to transport off-board signals with low power loss at the expense of additional timing error (propagation delay) and jitter. Moreover, each of these parameters is different for rising edges and falling edges due to device asymmetries. Measurements were conducted by running the program in Figure 6-11, which toggles bit 0 on and off in a loop. The propagation delay was taken to be the interval between the sequencer-side signal reaching 1.0 volts and the corresponding daughterboard-side signal reaching 1.0 volts. Even when treating the LVDS conversion as a resistive channel, the sequencer and the LED test board have different resonances. Therefore, reflection distorts the transmitted and received signals in different ways, and 1.0V was chosen rather than a logical CMOS high or low to measure the propagation delay and not reflection distortion.

```
Start:  p 0x00000001, 0x01, 0
        j Start
        nop
        halt
        p 0x00000000, 0x03, 0
```

Figure 6-11: Test pulse program for measuring LVDS latency.

The resulting histograms for LVDS latency are shown below. The rising edge is described by Figure 6-12 and the falling edge is described by Figure 6-13. Gaussian fit parameters are given in Table 6.5. The mean of the Gaussians represents the average LVDS latency while the standard deviations are the jitter. Since all triggering and feedback tests propagate through the LVDS channel twice, their responses must be convolved with two Gaussians. This directly leads to the latencies measured in the following two subsections.

	Rising Edge Latency	**Falling Edge Latency**
Mean (ns)	2.9396	3.2441
Standard Deviation (ns)	0.34876	0.14547

Table 6.5: Gaussian fit parameters for LVDS edge latency.

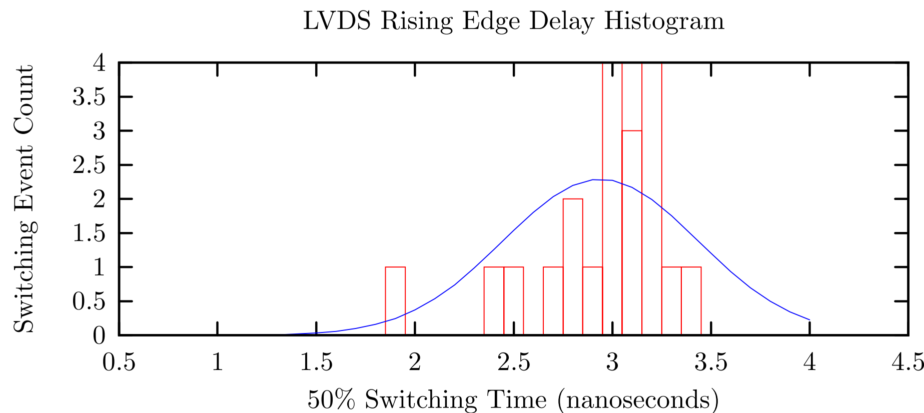

Figure 6-12: Histogram of LVDS rising edge latency.

6.6.1 Trigger Latency

Trigger latencies are measured with a pulse program whose first instruction raises an output bit, as shown in Figure 6-14. No looping is necessary since triggering is a single, asynchronous event. The time interval between the trigger signal reaching a logical LVCMOS high level (3.0 V) and the output bit reaching the same level is the latency. It includes one cycle to detect the trigger and lower a synchronous reset

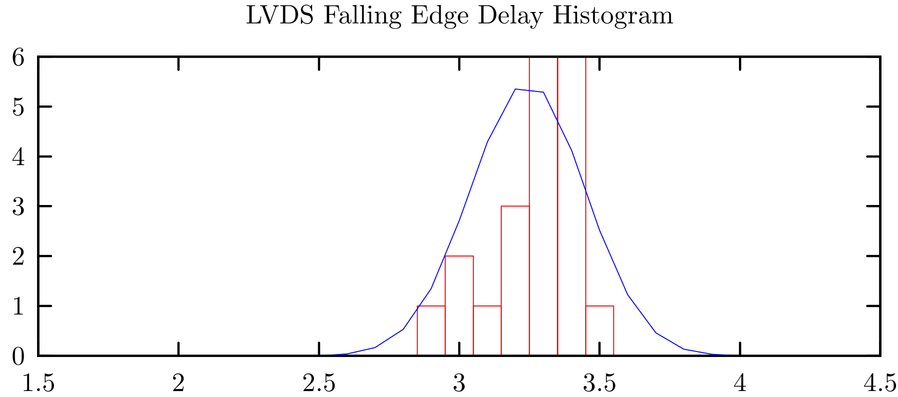

Figure 6-13: Histogram of LVDS falling edge latency.

signal, one cycle to detect the reset signal and start the PCP, and another cycle to start the instruction pipeline. However, this starting circuitry is not part of the PCP itself and is clocked at a much slower 25 MHz, accounting for 120 nanoseconds of latency. Any additional latency must be due to LVDS conversion and propagation through firmware bus primitives.

```
        p 0x00000001, 0x01, 0
        nop
        halt
        p 0x00000000, 0x03, 0
```

Figure 6-14: Test pulse program for measuring trigger latency.

The test program was triggered from a feedback channel on the LED test board which was controlled over I^2C and ultimately by user input. Triggering is handled completely in firmware before software execution has begun; therefore the trigger can be treated as a uniformly random process with respect to the starting circuitry. However, in practice, the user's original trigger input must pass through LVDS conversion before reaching the PCP firmware, and the resulting first output pulse must also be converted to reach the daughterboard. This results in the distribution measured in Figure 6-15 and the Gaussian fit parameters in Table 6.6. Notice that no measured

latency is shorter than 120 nanoseconds or is much longer than 160 nanoseconds, corresponding to the 40 nanosecond synchronization window for an asynchronous signal and a 25 MHz clock.

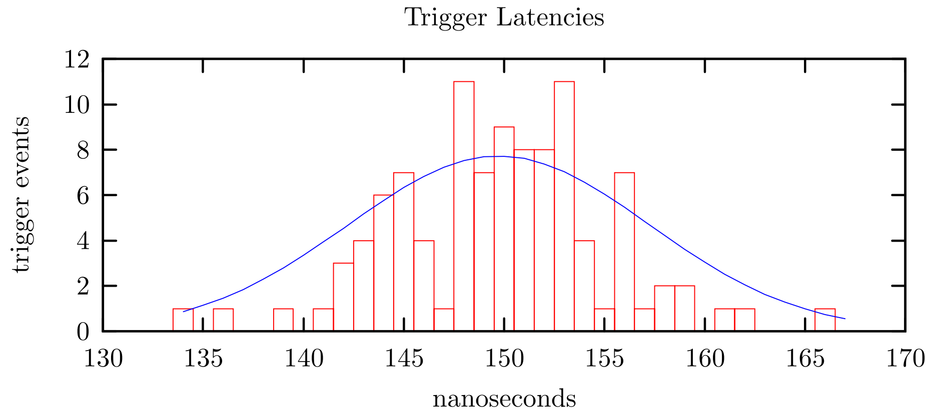

Figure 6-15: Histogram of trigger latency.

Mean (ns)	149.73
Standard Deviation (ns)	5.3199

Table 6.6: Gaussian fit parameters for trigger latency.

6.6.2 Feedback Latency

A pulse program to test feedback latency must loop indefinitely while waiting for a feedback event. To facilitate multiple measurements, the program should be reloadable in the following sense: a rising edge on the feedback signal will trigger a measurable change in output and a falling edge will reset the output. Such a program is listed in Figure 6-16. The `btr` instructions use the seventh feedback input, represented with the number 0x80. The program consists of two loops with entry points at `Start` and `Jump`. Because the `btr` instruction can only branch when a positive condition is met (in this case, when the seventh feedback input is high), the branch destination labels must be the same for both loops in order to handle both positive and negative cases.

```
        .equ TRIGGER, 0x080
Start:  btr TRIGGER, Jump
        p 0x0000000000000000, 0x01, 0 ; branch delay slot
        j Start
        nop
Jump:   p 0x0000000000000001, 0x01, 0
        btr TRIGGER, Jump
        nop
        j Start
        nop
        halt
        p 0x00000000, 0x03, 0   ; branch delay slot
```

Figure 6-16: Test pulse program for measuring feedback latency.

The resulting measured latencies are plotted in a histogram shown in Figure 6-17. Unlike triggering latency, which is entirely due to firmware running at a 25 MHz clock, feedback branching involves a software component which runs at a faster 100 MHz clock. In a loop where the branch is taken, a minimum of 3 instructions and a maximum of 7 instructions must be executed between a feedback input high and a pulse output high. The minimum theoretical latency occurs when the feedback rises right before `Start` and the maximum occurs when the feedback rises right afterward. Recall that each instructions take 2 cycles and 20 nanoseconds to decode; correspondingly, no latency measurement is less than 60 nanoseconds or much greater than 140 nanoseconds. The large standard deviation reflects the 80 nanosecond synchronization window along with jitter from LVDS conversion.

Mean (ns)	112.98
Standard Deviation (ns)	22.251

Table 6.7: Gaussian fit parameters for feedback latency.

6.7 Noise Measurements

In the previous sections, throughput and latency were measured as a kind of "digital phase noise," or unwanted timing artifacts. These depended on the notions of timing

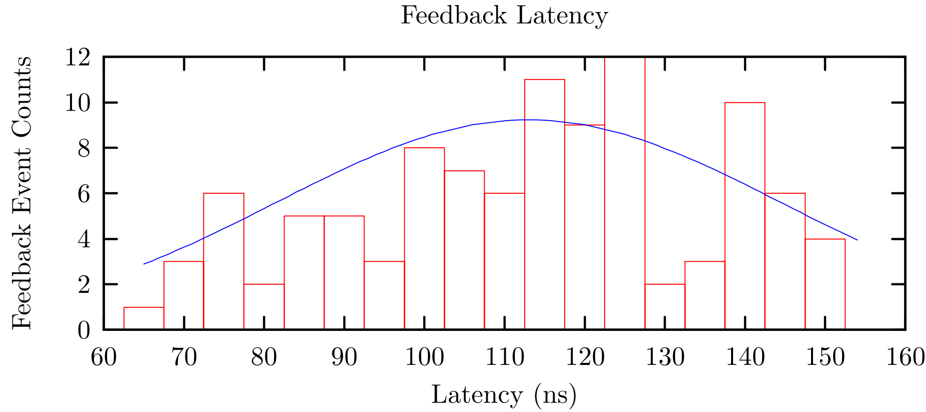

Figure 6-17: Histogram of feedback latency.

error and jitter for analog phase noise. However, unwanted amplitude noise can also affect the signal integrity of the sequencer. In Chapter 3, several techniques were discussed for mitigating the two main sources of amplitude noise, reflection and crosstalk. In this section, the success of these techniques is measured using power ratio comparisons.

6.7.1 Reflection Noise

In timing measurements, a periodic signal with an asymmetric duty cycle was sufficient to measure throughput and latency. In contrast, noise measurements must clearly distinguish signal frequencies from noise frequencies, requiring a symmetric duty cycle. The test program in Figure 6-18 produces a 50% square wave with strong components at multiples of a single fundamental frequency. Since individual pulse resolution has been fixed at 10 nanoseconds and a single cycle consists of two pulses (On and Off), the square wave period can be incremented in intervals of 20 nanoseconds. Because of the overhead in looping, including a branch delay slot, the shortest period is 6 cycles (60 nanoseconds).

Typically, the ratio of signal power to random noise power measures the amplitude error introduced by switching components. However, the reflection noise from parasitic resonances is greater than any background noise. Thus, taking the signal-

```
        .equ HALF_PERIOD, 0x03 ; in cycles, >= 3
Start:  p 0x00000001, HALF_PERIOD, 0 ; pulse high
        j Start
        p 0x00000000, HALF_PERIOD, 0 ; branch delay slot; pulse low
        halt
        p 0x00000000, 0x03, 0        ; branch delay slot; turn off for safety
```

Figure 6-18: Test pulse program for measuring reflection noise.

to-reflection noise ratio is a conservative estimate of the sequencer's noise rejection properties and measures how well termination impedances were matched. It is customary to plot this ratio versus frequency, which is inversely proportional to period. Since periods are increased linearly, the horizontal frequency axis can be linearized by taking its logarithm and fitting the ratios to a linear curve. As expected, the ratio drops as frequency increases as shown by the parameters in Table 6.8; this is most likely due to greater excitation of parasitic resonances and a rising noise floor from switching noise.

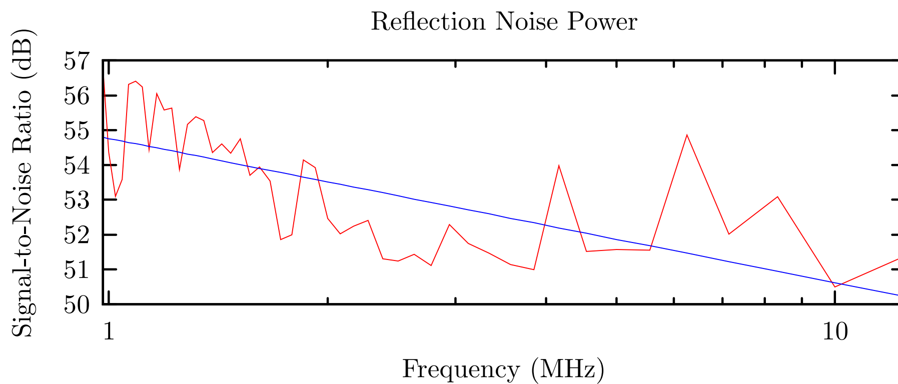

Figure 6-19: Plot of signal-to-reflection noise ratio.

Slope (dB / decade)	-1.7978
y-intersect (dB at 0 MHz)	54.7625

Table 6.8: Linear fit parameters for signal-to-reflection-noise ratio.

6.7.2 Crosstalk Noise

The amplitude noise from crosstalk constitutes the switching noise mentioned above. The ratio of signal power to noise power indicates how well the digital outputs are isolated from one another and how well-defined the current return paths are. These measurements use the same code as for reflection in Figure 6-18; however, the noise power is taken from the strongest frequency component of the signal in an adjacent, non-switching output. The results are shown in Figure 6-20. The linear fit parameters in Table 6.9 show the expected increase in crosstalk noise with frequency due to increased switching currents and therefore increased mutual induction.

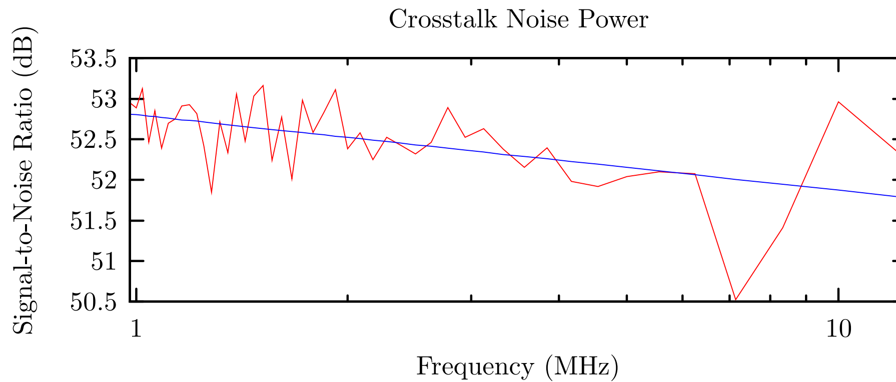

Figure 6-20: Plot of signal-to-crosstalk noise ratio.

Slope (dB/ decade)	-0.4039
y-intersect (dB at 0)	52.8027

Table 6.9: Linear fit parameters for signal-to-crosstalk noise ratio.

6.8 Power Consumption

Although power consumption is not an explicit requirement, an ideal device would meet the stated goals while consuming as little power as possible. Three different

power measurements are presented in this section which depend on three independent factors: firmware programming, normal user tasks, and clock frequency. While electrical requirements are primarily limited to the hardware layer, measuring the power consumption before and after programming the FPGA gives another measure of the static overhead introduced by firmware and the relative efficiency of module implementation. Dynamic firmware overhead can be measured during the performance of user tasks. Also, power consumption is closely related to clock frequency, and it is useful to know this relationship independently of the maximum clock speeds measured in Section 6.3. Pragmatically, knowledge of the device's power consumption is also useful in in selecting an appropriate supply. The measurements below were taken with an Agilent E3631A bench supply, but in practice the sequencer can be powered from a commodity AC adapter. A single DC voltage between +5.4 and +9 volts which can source 2 amperes suffices for one device.

6.8.1 Firmware Power Consumption

The measurements in Table 6.10 show the idle power consumption of the sequencer device, before and after programming, when powered from three separate voltages. The increase in current draw after programming reflects the power efficiency of the firmware alone, since no other hardware parameters were changed and the processor cores are held in reset. Along with logic resource utilization, this firmware power consumption is a useful benchmark for quantifying the improvement of future versions.

	Idle, Unprogrammed		Idle, Programmed		
Supply Voltage	**Current Draw**	**Power**	**Current Draw**	**Power**	**Sinks**
+1.8V	63 mA	113 mW	121 mA	218 mW	FPGA core
+3.6V	292 mA	1051 mW	362 mA	1,303 mW	FPGA I/O, oscillator, SRAM, LVDS, clock switch, LEDs
+5.4V	481 mA	2,597 mW	631 mA	3407 mW	Ethernet, fiberoptic connector

Table 6.10: Idle and peak power consumption of sequencer device.

6.8.2 Task-Dependent Power Consumption

The power consumptions of the sequencer in different states is shown in Table 6.11 when powered by a single +5.4V supply and running from the off-board 100 MHz oscillator.

Current Draw (A)	**Power Consumption (W)**	**Sink**
1.292	6.977	During FPGA programming.
1.502	8.111	Before DHCP.
1.496	8.078	After DHCP (idle) and during reset
1.499	8.095	PCP started but halted.
1.503	8.116	PCP running program in Figure 6-11.
1.536	8.294	AVR started.
1.518	8.197	Receiving network packet.
1.527	8.246	Using the web interface.

Table 6.11: Power consumption of various tasks at +5.4V

6.8.3 Clock-Dependent Power Consumption

Frequency-dependent power consumption was performed using the same test program as for output skew in Figure 6-2. which simply toggled all 64 outputs on and off. This made the power relationship to clock frequency more measurable. It also provided a worst-case estimate for power consumption during actual use at high speeds.

The results in Figure 6-21 show a linear increase of power consumption with clock frequency initially, with the PCP consuming more power while running than while idle, as expected. The rapid drop in power usage at 164 MHz indicates that this is the maximum clock speed for the PCP decoder before it fails; after this point, the stalling of the PCP also stalls other firmware modules, so that the PCP's running power usage is actually lower than its idle usage. After 330 MHz, the power consumption becomes more erratic as the clock switch no longer passes valid LVCMOS levels to the FPGA. Linear fit parameters for power consumption are shown in Table 6.12, with the high values at 0 MHz indicating the power consumed by non-PCP firmware.

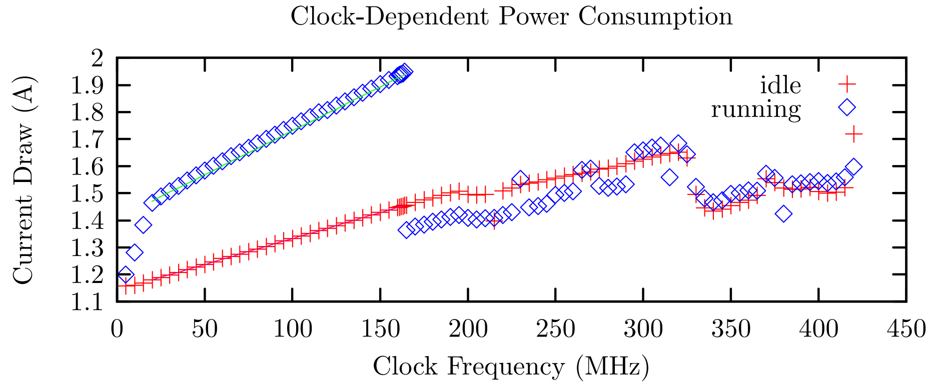

Figure 6-21: Plot of clock frequency-dependent power consumption.

	PCP running	PCP idle
Slope	3.243 mA / MHz	1.911 mA / MHz
y-intersect	1.4062 A at 0 MHz	1.1416 A at 0 MHz

Table 6.12: Linear fit parameters for clock-dependent power consumption.

Chapter 7

Conclusion

In the first chapter, the sequencer was presented as an abstract specification of five goals. This goals were motivated by the requirements of different qubit representations and the capabilities of contemporary devices in Chapter 2. Chapters 3, 4, and 5 made the sequencer concrete with descriptions of a design and implementation intended to meet the given specification. The success of this approach was measured in in Chapter 6 for the performance and feedback goals. These results are interpreted and the remaining subjective goals are evaluated for the sequencer in Section 7.1 of this chapter. However, the goal of flexibility also extends to future versions of the sequencer. Thus, improvements are proposed in Section 7.2 to further increase the current work's usefulness and efficiency and provide opportunities for future work.

7.1 Evaluation

In evaluating the sequencer, its goals must be understood both objectively and subjectively. Several goals mention desired values for figures of merit, which were measured in the preceding chapter. However, evaluation is not a simple decision problem since it is often useful to know *by how much* a figure of merit was satisfied and how it compares to other contemporary devices which have similar goals and which have made different trade-offs. This quantity also represents a margin for improvement in future versions of the sequencer.

7.1.1 Performance

To meet the performance goal, the sequencer achieved a cycle time of 10 nanoseconds (running at a 100 MHz clock) and is currently capable of decreasing this to about 6.25 nanoseconds (running at a 160 MHz clock). Due to its pipelining of pulses and overlapping instructions, it is able to achieve a delay of 0 nanoseconds for most pulse sequences; a special case is the minimum duration of 1 cycle, which must always be followed by a 1 cycle pulse of all-zero outputs. At 100 MHz, this satisfies the goals for a minimum duration of 10 nanoseconds and a minimum delay of 10 nanoseconds.

The combination of PCP instruction decoder's timing constraints and the current FPGA hardware cannot run faster than 164 MHz. Moreover, due to the skew between output bits of the LVDS subsystem, pulses much shorter than 5 nanoseconds will become unsynchronized, corresponding to LVDS's theoretical maximum speed of 200 MHz. At slower speeds, however, the device is able to switch 64 digital outputs simultaneously, with output skew no greater than 3 nanoseconds.

7.1.2 Feedback

The feedback goal consists of a hard requirement to enable the sequencer to respond to external signals and a soft requirement to minimize this response latency. By including LVDS receivers as well as drivers, the hard requirement is fulfilled. The soft requirement applies to the sequencer's two possible responses to an external event: triggering the start of a pulse program and enabling conditional branching in a running program. Measurements have shown that a pulse program can produce a measurable response at most 170 nanoseconds after a trigger (with an average of 150) and at most 160 nanoseconds after a conditional branch (with an average of 113).

While these latencies are greater than the desired value of tens of nanoseconds, there are clear solutions to decreasing them further. To decrease the trigger latency, the trigger mechanism should be incorporated into the PCP and clocked at its faster speed, rather than at the slower 25 MHz of all other firmware modules. To decrease the feedback latency, an additional branch-wait-on-trigger instruction can be added.

7.1.3 Programmability

Satisfying the programmability goal requires the ability to control outputs with pulse program software using a minimal instruction set for conditional feedback branching. The sequencer accomplishes this with the specialized firmware processor core, the PCP. It is able to switch all 64 of its digital outputs independently in software using a 64-bit register file and a unified, 64-bit wide program memory. Pulses can be output indirectly using registers previously loaded from memory or immediate values directly encoded into instructions. Program execution can branch to arbitrary locations unconditionally (for infinite looping) or conditionally (for feedback control); the PCP can also safely halt at the end of a pulse program. These represent the minimum instruction set for generating conditional pulse sequences. More sophisticated features, such as finite looping and subroutines, are possible by modifying the decoder in exchange for greater logic complexity and slower clock speed.

7.1.4 Ease-of-Use

In fulfilling the ease-of-use goal, the sequencer uses standard Ethernet connections and a TCP/UDP/IP protocol engine to communicate with commodity hardware and software. Network interface cards (NICs) and web browsers are available for any consumer PC and operating system conforming to open standards such as PCI and HTTP. The use of an assembly language and command-line compiling is still difficult for non-programmers, but the learning curve is lessened by the instruction set's small size. In contrast with pulse program development, the control of the sequencer via the web interface is much more intuitive because no external program is needed except a web browser, and no command syntax must be remembered. On the hardware side, the device's power requirements can easily be satisfied with an off-the-shelf AC adapter, making it easy to deploy in any setting.

7.1.5 Flexibility

In order to meet its goals for flexibility, the sequencer includes a daisy-chain interface allowing multiple devices to be synchronized to run the same pulse programs, detect the same inputs, and use the same clocks. Channels can be added and removed easily in multiples of 64 bits without redesigning the system. Most importantly, the daisy-chain controller decouples the control of pulse programs from various clients, including host access over UDP, the web interface, and other devices in the daisy-chain.

The use of well-chosen interfaces and a modular design allows many components of the sequencer to be refactored easily and tested independently. The use of a standard Wishbone interconnection bus between firmware modules makes them interchangeable and reusable. Running a web server in software on a general-purpose core, the AVR, isolates critical pulse program execution from non-critical user interface changes. It also allows the web server to be upgraded remotely by downloading new software, rather than reprogramming the firmware. The existence of well-defined machine opcodes and a transfer protocol allows third-party development tools to target the sequencer and to create more user-friendly graphical programming interfaces. Section 7.2 describes these extensions in greater detail, but the examples here attest to the sequencer's flexibility.

7.1.6 Comparison

Based on the figures of merit, goals, and secondary features such as cost and physical size, the sequencer is compared with three similar devices in Table 7.1. The first column refers to the current work, which has the benefits of high performance, wide outputs, a flexible Ethernet interface, and uniquely among all the devices, feedback support. It also possesses an open design, making it suitable for future development beyond the original author. The second column details the most powerful SpinCore device available, the PulseBlaster ESR-333-PRO; it achieves the fastest clock speed and includes waveform generation at a greater cost and a less flexible PCI interface.

The third column refers to the sequencer's predecessor by Huang et al., from which

the sequencer inherited several features. This device was implemented with an off-the-shelf Xilinx FPGA development board from Digilent; it has a relatively low cost, fast speed, and the widest outputs. However, its serial interface is difficult to use and it is no longer actively developed. The fourth column represents the digital acquisition controller of Varian's ^{UNITY}INOVA spectrometer. This is the oldest device in the comparison and hence has the slowest performance and largest PCB size; however, it is integrated with sophisticated waveform generators and control software, and it is also backed by the greatest level of technical support. The price for the Varian's data acquisition controller is not provided because this part is never sold separately.

Parameter	**Sequencer**	**SpinCore**	**Huang**	**Varian**
Year released	2004	2004	2003	2000
Minimum cycle time (ns)	10	3	8	25
Minimum duration (ns)	10	3	8	100
Minimum delay (ns)	$0-10$	3	8	25
Number of digital outputs	64	10	72	34
Number of channels	1-64	24	1-72	4
Maximum program size (words)	2K	32K	512K	4K
Maximum trigger latency (ns)	170 ns	80 ns	n/a	n/a
Feedback support	yes	no	no	no
Configurable clock sources	yes	no	yes	no
Waveform synthesis	no	yes	no	yes
Host interface	Ethernet	PCI	RS-232	Ethernet
PCB size (in)	5.5 x 6	4.5 x 7	5 x 5	9 x 8.5
Unit cost	$700	$3,200	$200	n/a

Table 7.1: Feature comparison between pulse sequencer and competing devices.

7.2 Future Work

As with any complex project, the sequencer device has many avenues for future extensions; as part of its goal of flexibility, it was designed to make these extensions easy. Some of these extensions are actively-developed features which did not make it into the first release while others represent longer-term objectives. These improvements are grouped by layer below.

7.2.1 Hardware Extensions

Most improvements to the hardware revolve around further increasing the maximum clock speed and increasing the number of digital outputs by upgrading, adding, or removing components. The PCB layout is indirectly a performance bottleneck due to reflection. The current design can attenuate reflection noise even further by terminating traces between the FPGA and the LVDS subsystem.

The most challenging hardware extension is the design of a daughterboard to perform digital-to-analog conversion. Because the sequencer only produces digital outputs over LVDS, it generally requires this additional board to interface with other apparatuses. Typically, quantum computing experiments will require amplitude shaping, frequency generation, and phase shifting, which can be encoded in separate channels. One board with such capabilities is a programmable RF source currently being developed by John Martinis at the University of California, Santa Barbara, and Steve Waltman at NIST Boulder. Other daughterboards may simply convert LVDS to TTL levels on coaxial connectors to drive circuits directly using square pulses. Such pulses are also suitable for driving a PTS synthesizer for frequency generation.

7.2.2 Firmware Extensions

The simplest improvements to the firmware involve optimizing firmware modules to use fewer gates and to execute in fewer cycles; both steps would decrease power consumption and signal delays while increasing maximum clock speed. Decreasing the resources used by the existing modules would free up space to implement new modules, and extensive regression tests have been provided to ensure that new optimizations preserve existing functionality.

Larger performance gains, especially in increasing clock speed, will involve the use of multiple clocks (*interclocking*) and asynchronous FIFOs.[1] Two areas where this is most applicable are the datalink layer for the PTP and the PCP instruction decoder. These may be considered separately, depending on whether the user needs

[1]Sometimes called a double-clocked queue.

higher daisy-chain throughput or additional pulse instructions.

In the first case, the link layer must operate between two devices running different clocks. It samples a signal several times before making bit decisions to avoid metastability, and it must transmit a byte serially bit-by-bit. During this time, the PTP routing layer must stall and remain idle. It would be possible to clock the link layer faster than the rest of the firmware in order to account for this sampling and serialization overhead at the increased cost of an asynchronous FIFO with its two interfaces for writing and reading.

In the second case, another asynchronous FIFO could be placed after the instruction decoder, which is a system bottleneck for maximum clock speed, and before the timer and pulse output register, which can be clocked much faster. The decoder would indirectly output pulses by writing new timer counts and output values into a queue. The timer would read new timer counts from the queue, stall for the specified duration, and output the specified value without intervention from the decoder. Because the queue is asynchronous, the decoder and the timer can be clocked independently.

The primary obstruction to adding new instructions and capabilities to the decoder is the corresponding increase in cycle time and decrease in maximum clock speed. By using the FPGA's built-in PLLs as clock multipliers, the timer can run up to 4.5 times faster than the decoder and maintain a relatively fine resolution. Asynchronous queues have their own pipeline, which would increase the feedback and branching latency from the decoder by at least three cycles; however, this would not cancel out the improvement in minimum duration and minimum delay. However, due to output skew, it may not be feasible to clock the timer at its maximum speed.

Three useful additions to the instruction set include arithmetic operations, comparisons, and subroutine calls. One possible implementation would use a smaller register file equal in width to the program address. This is generally much smaller than the full 64-bits of program data; it would be small enough to allow fast arithmetic and comparison operations but large enough to store return locations. However, even the fastest operations and smallest register file will introduce greater timing constraints on the instruction decoder, decreasing its maximum clock speed.

7.2.3 Software Extensions

Software extensions can be divided between the development tools on a host PC and the CGI interface on the sequencer device.

Although the command-line assembler is a great improvement over assembling opcodes by hand, it still exposes the user to unnecessary implementation details such as branch delay slots and special cases within each pulse instruction. An obvious extension to the development tools is a port of a compiler[2] for C or some other high-level language that will shield the user from instruction scheduling and pulse encoding. An alternate approach is the SpinCore development model in which pulse programs are written indirectly through a host binary. This can be accomplished most directly by creating and distributing a shared library and header files, similar to the existing PTP network library used by the command-line client.

The web interface is useful because it exploits the ubiquity of web browsers on all major computing platforms; however, it depends on a separate bootstrapping program to discover and program the sequencer device. An alternative user interface would be an integrated graphical application which would perform both bootstrapping and user operations over the network in place of both the web browser and the current command-line client. To provide uniformity across all host operating systems, this application should use a cross-platform toolkit such as GTK+ or wxPython.

Finally, users may wish to control the sequencer or generate pulse programs from popular scientific software such as LabVIEW, MATLAB, or Octave. These packages provide interfaces for making network calls, such as constructing and parsing packets according to a custom protocol, as well as sending and receiving them. PTP provides a convenient and well-defined network protocol over UDP for non-interactive control of the sequencer. Likewise, the CGI interface can be used non-interactively over HTTP and TCP using the post and get methods and the fully-specified modes and variables in Section D.2. A useful extension of the host development tools would be interface libraries for the above software packages.

[2]The GNU Compiler Collection (gcc) is an ideal choice since it already uses the GNU binutils collection to assemble binaries for many other targets.

7.2.4 Methodology

An important extension to the sequencer does not involve modifying the current system as a whole. Rather, different parts of its design and implementation can be applied to new experiments unknown at the time of writing. Pulse programming in general represents a technology transfer from NMR spectroscopy to other areas of physics and quantum computing in particular. Aside from being a tangible device with immediate applications, the pulse sequencer described in this thesis can serve as a template for future instrumentation and experimental techniques combining programmable logic, machine languages, and network interfaces.

Appendix A

Hardware Design Documents

A.1 Transmission Line Model of a Microstrip

The following approximations for a microstrip transmission line are taken from [IPC95] and are only valid for

$$0.1 < \frac{W}{H} < 3.0 \qquad\qquad 1 < \epsilon_r < 15$$

where the variables are defined as follows:

W is the trace width in mils.

T is the trace thickness in mils.

H is the trace height above the reference plane in mils.

ϵ_r is the relative permittivity of dielectric.

σ is the conductivity of the trace material.

The characteristic impedance of the microstrip in ohms is:

$$Z_0 = \frac{87}{\sqrt{\epsilon_r + 1.41}} \ln\left(\frac{5.98H}{0.8W + T}\right) \tag{A.1}$$

The shunt capacitance in picofarads per meter is:

$$C_0 = \frac{26.38(\epsilon_r + 1.41)}{\ln\left(\frac{5.98H}{0.8W+T}\right)} \tag{A.2}$$

The external series inductance in nanohenries per meter is:

$$L_0 = C_0 Z_0^2 \tag{A.3}$$

The propagation delay in nanoseconds per meter is:

$$t_{pd} = 3.33\sqrt{0.475\epsilon_r + 0.67} \tag{A.4}$$

The series DC resistance of the trace and its return path in ohms per meter is:

$$R_0 = \frac{1}{\sigma w t} \tag{A.5}$$

Supplying values from [JG03] specific to surface mount PCB traces

$W =$ 8 mils

$T =$ 1.4 mils (1 ounce per square foot electro-deposited copper)

$H =$ 6 mils

$\epsilon_r =$ 4.5 (for FR-4 laminate at room temperature up to 1 GHz).

$\sigma =$ 5.8×10^{-8} for annealed copper at room temperature.

The following approximations result:

Z_0 $= 54.61\Omega$

C_0 $= 102.17 \times 10^{-12}$ F / meter

L_0 $= 5.58 \times 10^{-9}$ H / meter

R_0 $= 239 \times 10^{-6}$ Ω / meter

t_{pd} $= 0.432$ ns / meter

Using a 49.99 Ω series source termination and a typical trace length of 5 centimeters, the source reflection coefficient is:

$$\Gamma = \frac{R_s - Z_0}{R_s + Z_0} = -0.044$$

For a driving voltage of 3.3V, this corresponds to a negative source reflection of -145 mV arriving at the load two delay times (21.6 picoseconds) after the initial incident wave, which is within the noise margin of LVCMOS.

A.2 Layer Stackup

The noise-tolerance mechanisms described in Section 3.1 place certain constraints on the dielectric thicknesses of a PCB Minimizing reflection requires the calculation of the characteristic impedance for a microstrip trace; this, in turn, depends on the thickness separating signal traces and their references. Minimizing crosstalk and power filtering requires placing signal and reference layers in the correct order and controlling the thicknesses between power and ground planes. Moreover, the large number of desired programmable outputs required at least two signal layers for routing traces, where cost considerations limited the layer count to 8 or less. The resulting layer stackup is shown in Table A.1.

Layer Name	**Thickness (mils)**	**Material**
Top Routing	1.4	1 oz. copper traces
Prepreg	6.0	FR-4
Ground Plane 1	0.7	1/2 oz. copper
Core	5.0	FR-4
Power Plane 1 (+3.3V)	0.7	1/2 oz. copper
Core	35.0	FR-4
Power Plane 2 (+1.5V and +5V)	0.7	1/2 oz. copper
Core	5.0	FR-4
Ground Plane 2	0.7	1/2 oz. copper
Prepreg	6.0	FR-4
Bottom Routing	1.4	1 oz. copper traces

Table A.1: The six layer, controlled dielectric stackup for the PCB.

The first constraint on the board stackup was matching of trace impedances on the top and bottom routing layers to 50 Ω, the lowest value common in PCBs; lower values tend to stress components which can overdrive the traces. Trace thickness and height above the reference plane were both constrained by standard manufacturing tolerances for electro-deposited copper and FR-4 prepreg, respectively. The calculation of trace impedance can be found in Appendix A and yields an approximate value of 54.61 Ω using the values above. This agrees within 1.4% of the answer calculated by the 2D-field solver in Protel, 55.6 Ω.

The placement and thicknesses of the inner planes represented the next constraints. The ground planes must be exterior to the power planes in order to serve as signal references to the routing layers. The smallest standard thickness of core layer (5 mils) was chosen to separate each power and ground pair to improve its bypass impedance at high frequencies. The symmetry of the layer stackup is necessary to match impedances on the top and bottom layer. The final degree of freedom, the core between the two power planes, was chosen to bring the total board thickness to 61.2 mils, a standard PCB thickness that is used by most edgemount connectors.[1]

[1] See Section 3.2 for the advantages of edgemount connectors.

A.3 Circuit Schematics, Layouts, and Drawings

The following schematics are for Revision A of the hardware. All net identifiers are global across all sheets; intersecting nets are only connected if marked with a dot.

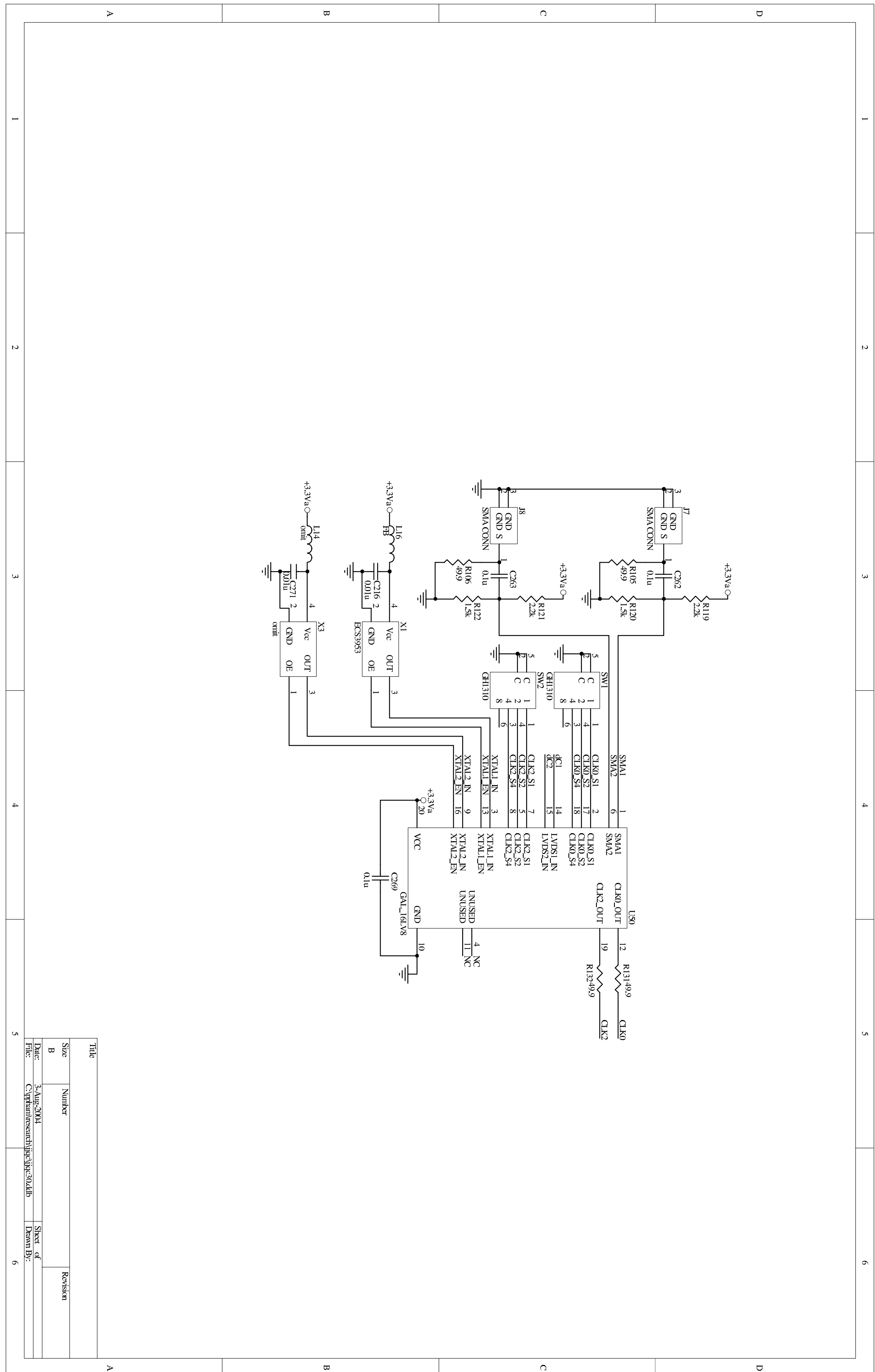

Figure A-1: Schematic for clock switch and clock sources.

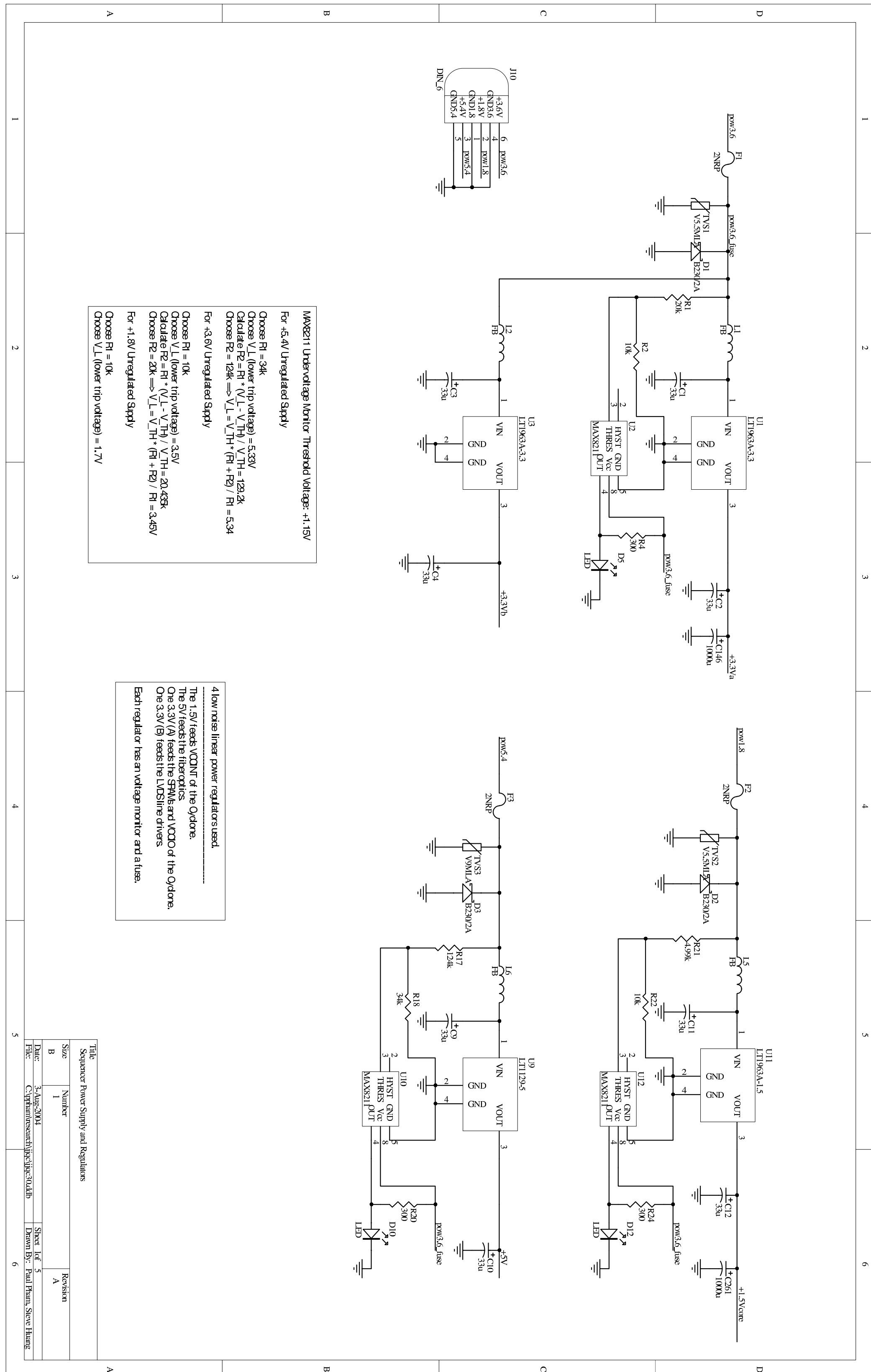

Figure A-2: Schematic for power supplies and safety devices.

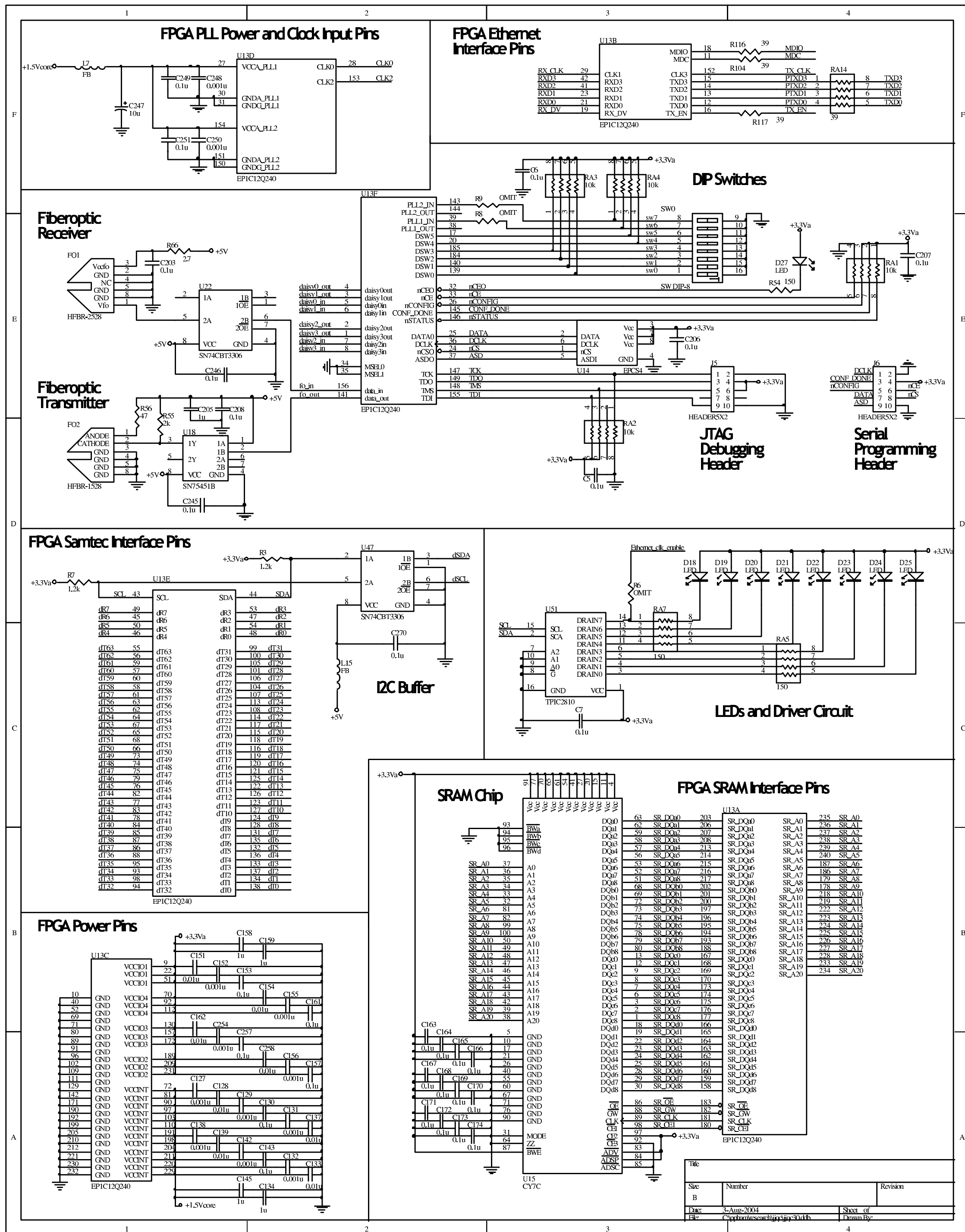

Figure A-3: Schematic for FPGA, SRAM, LED driver, and fiberoptic connectors.

Figure A-4: Schematic for Ethernet controller and connectors.

Figure A-5: Schematic for LVDS drivers and receivers.

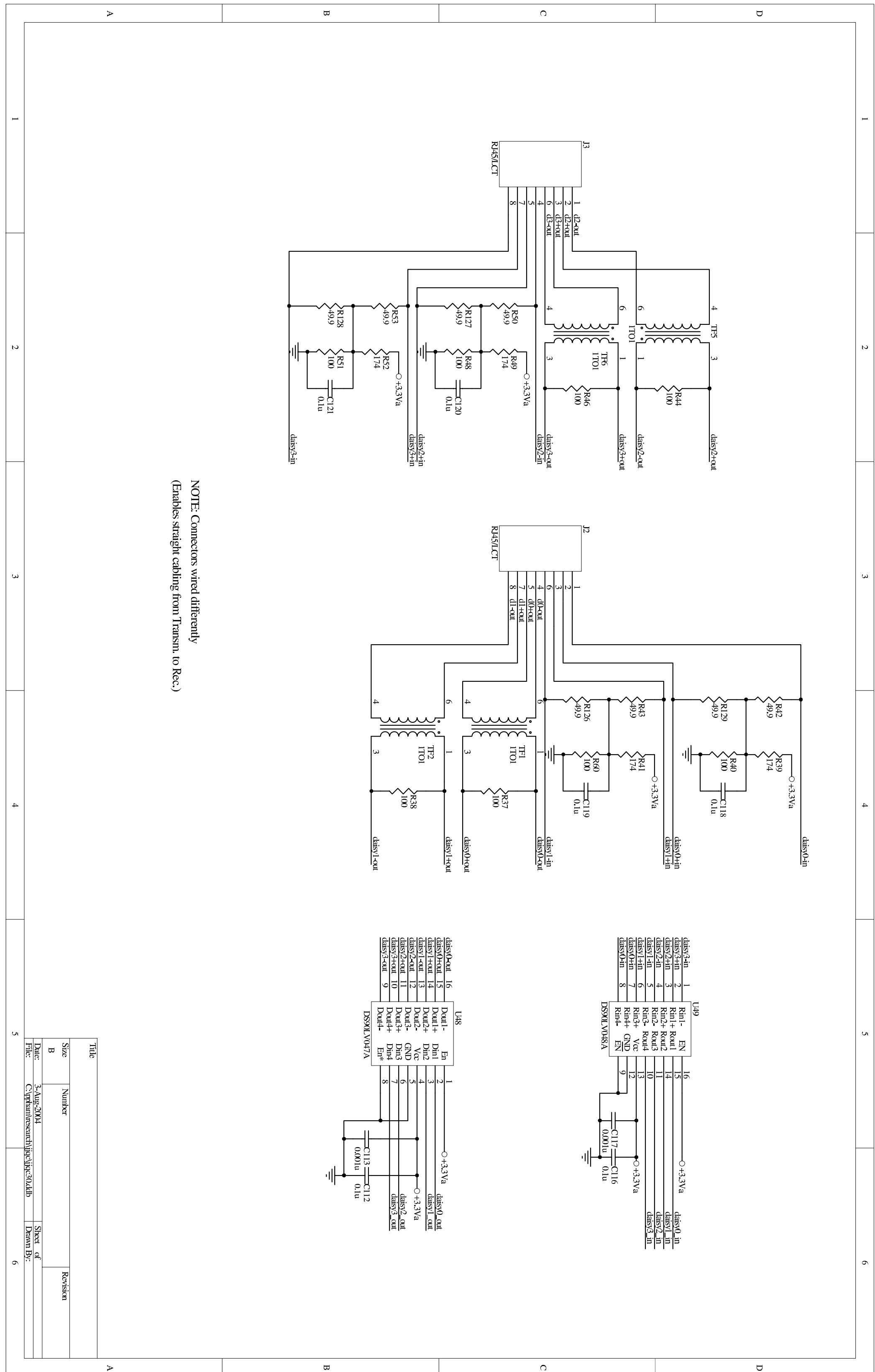

Figure A-6: Schematic for daisy-chain connectors and isolation transformers.

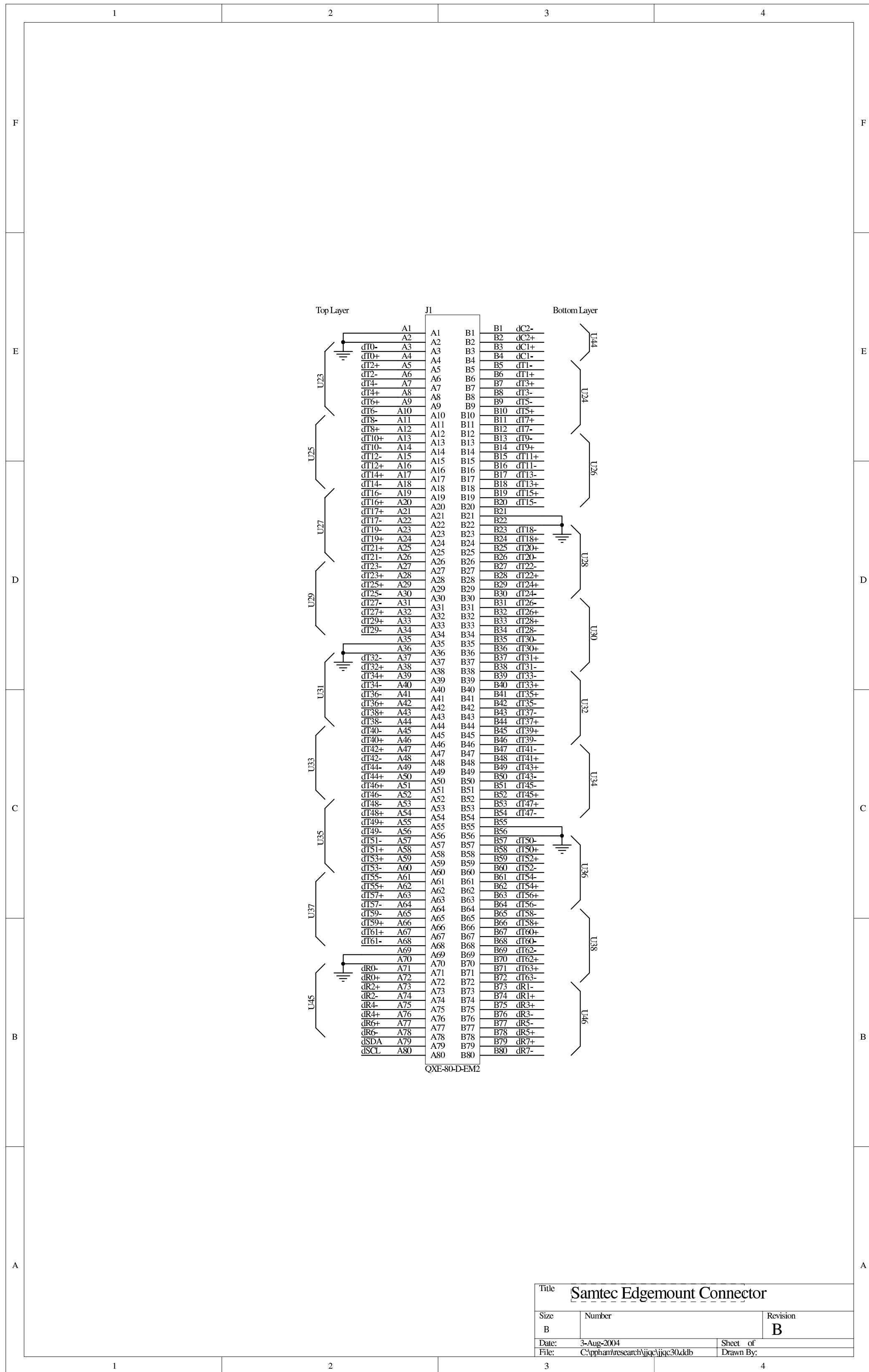

Figure A-7: Schematic for Samtec edgemount connector.

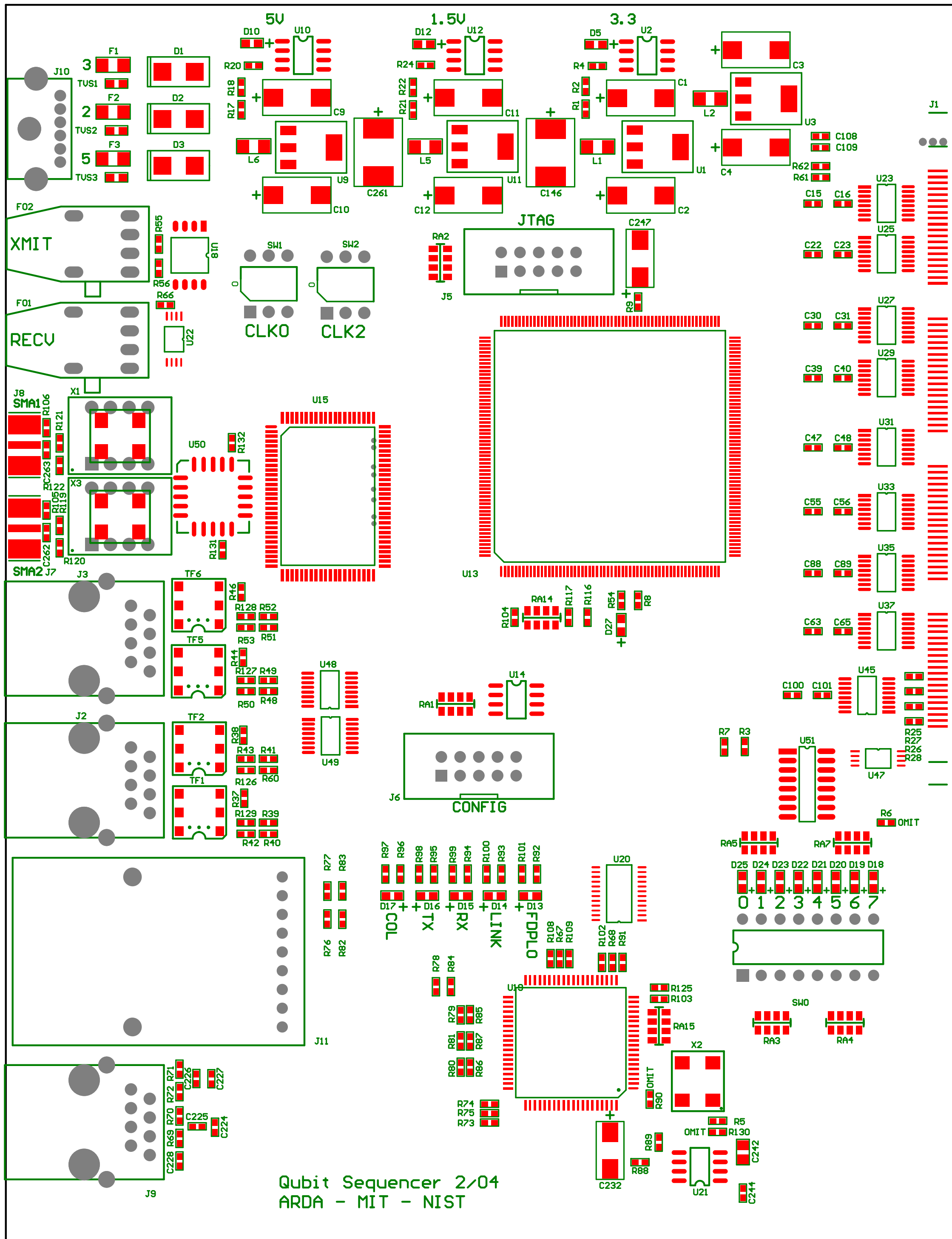

Figure A-8: Assembly drawing for top side (pads and silkscreen).

Figure A-9: Assembly drawing for bottom side (pads and silkscreen).

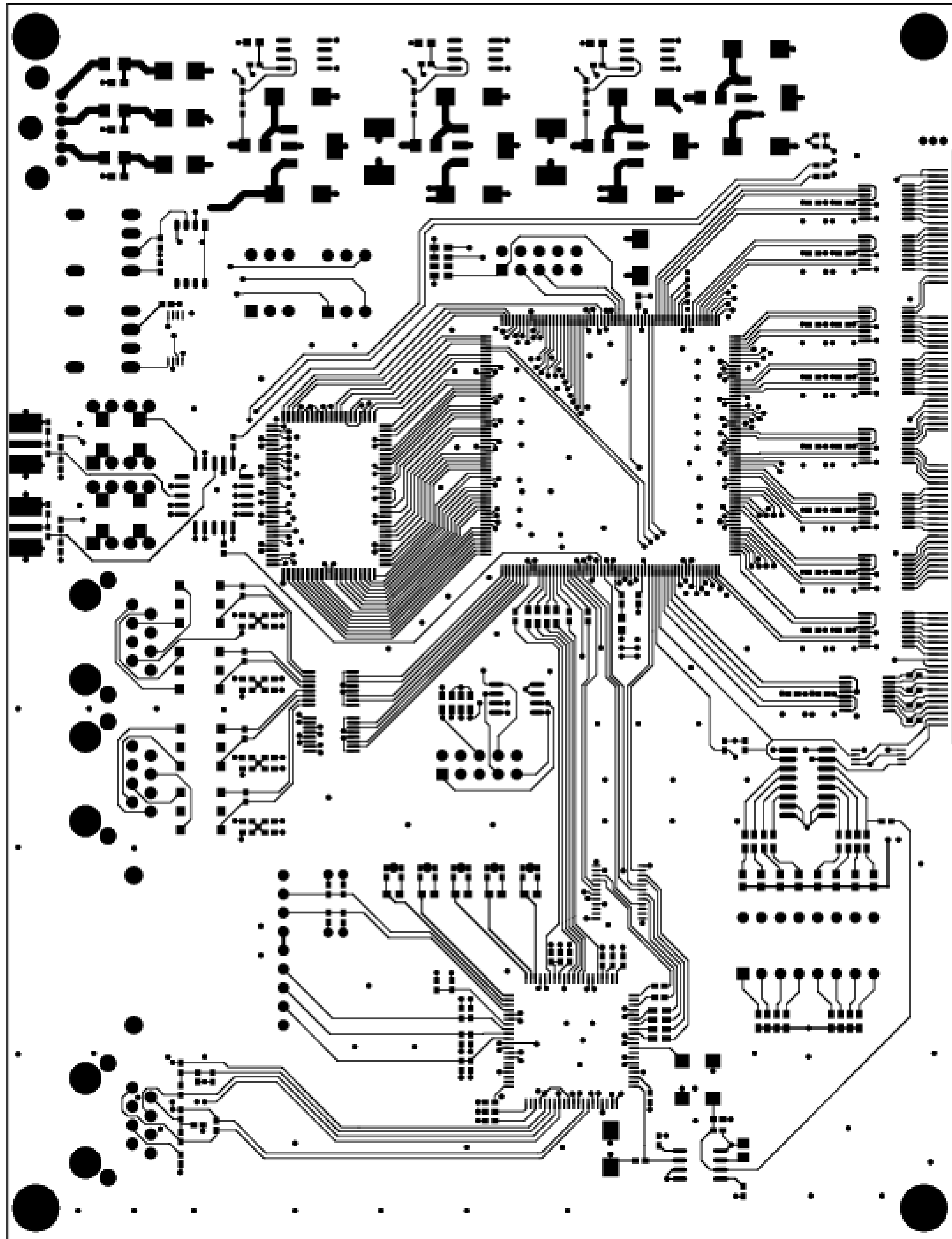

Figure A-10: PCB layout for top routing layer (traces and pads).

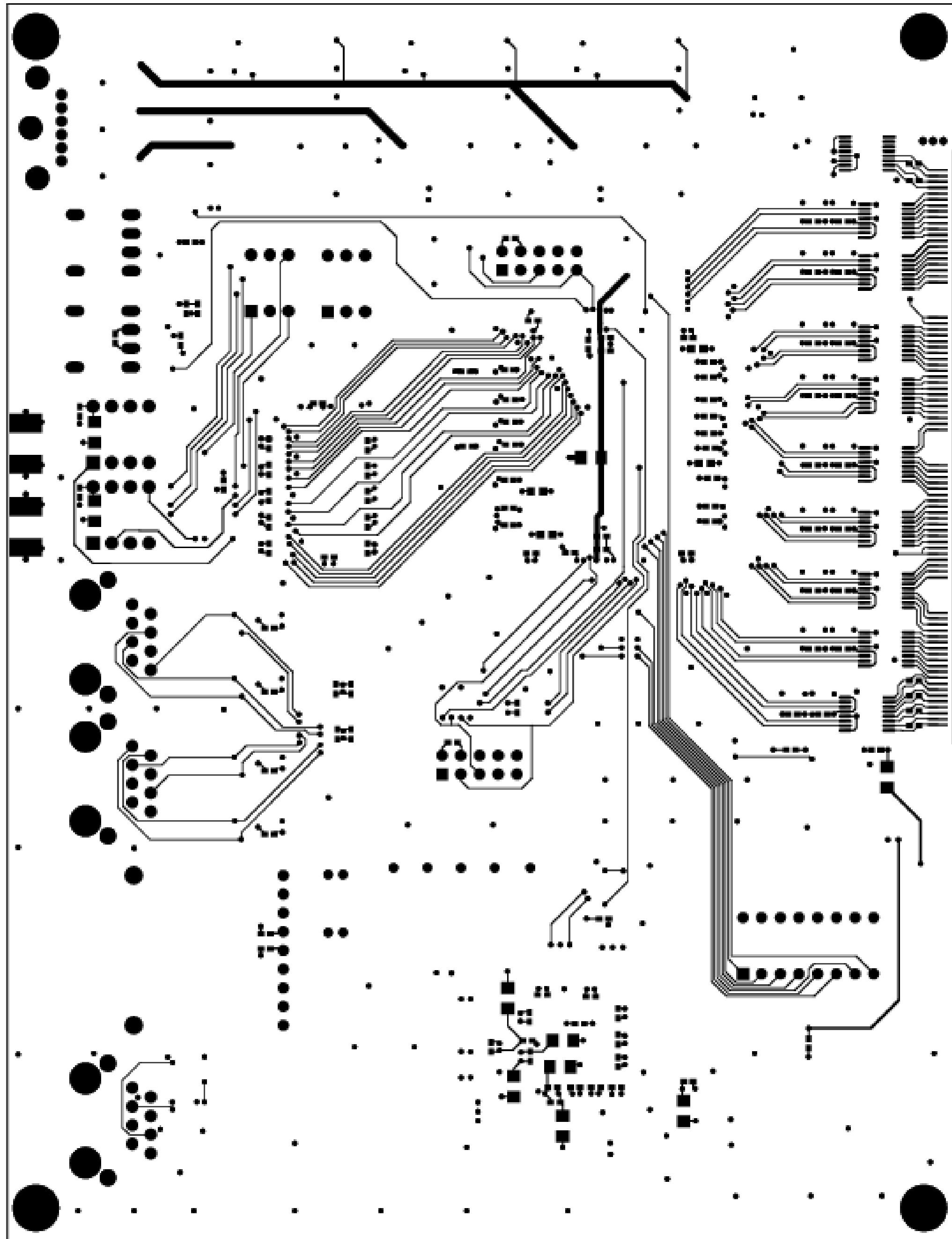

Figure A-11: PCB layout for bottom routing layer (traces and pads).

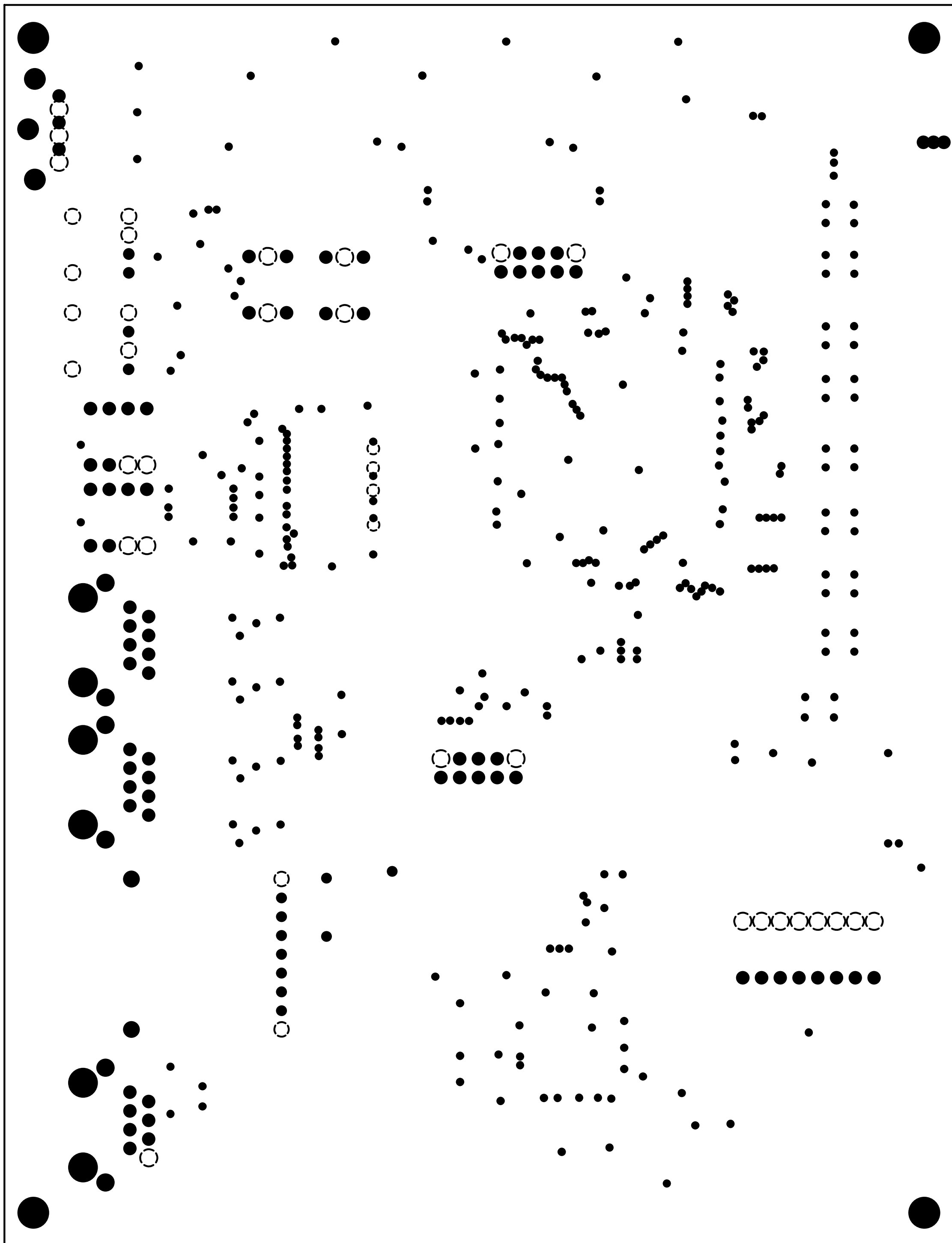

Figure A-12: PCB layout for inner ground planes (keepout regions).

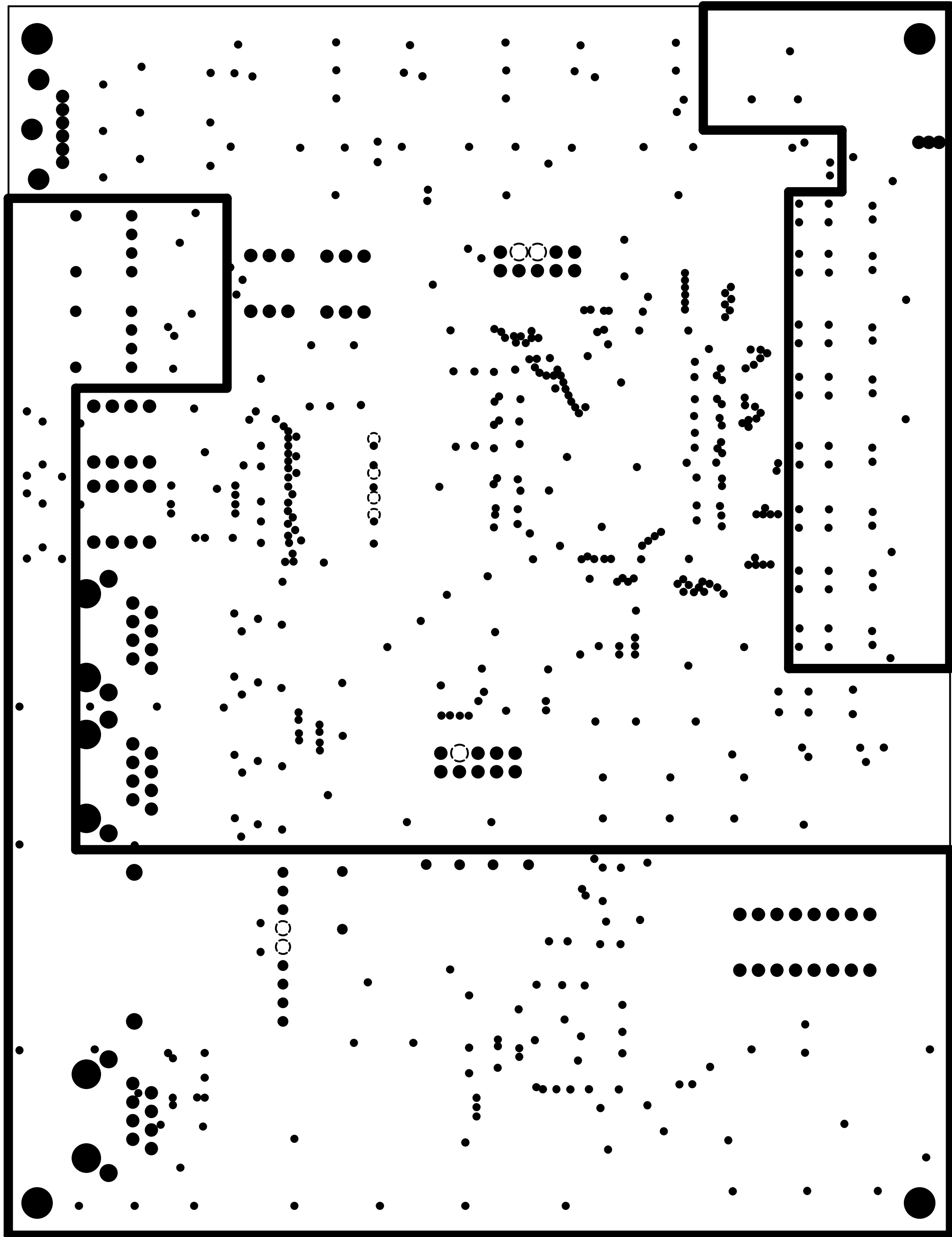

Figure A-13: PCB layout for inner power planes (keepout regions).

Appendix B

Firmware Design Documents

B.1 Power-On Reset Circuit

The underlying analog layer of an FPGA is usually reset with an RC delay circuit connected to the positive power supply. However, a similar feature is usually not available for the digital logic programmed into the FPGA. Many modules use state machines and variables which must begin in a well-defined state in order to function properly. The catch is that any power-on reset circuit must not itself depend on any initialization; all it can use are power, ground, a clock, and combinational gates. The circuit used in the pulse sequencer's firmware is shown in Figure B-1.

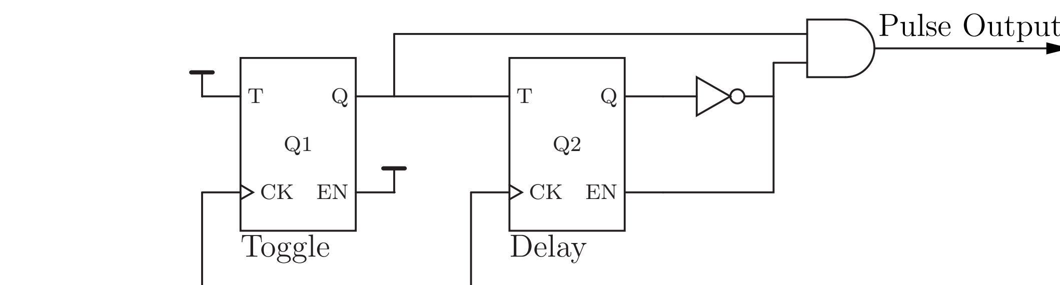

Figure B-1: Power-on reset circuit for bootstrapping digital logic.

It is constructed from T flip-flops and produces a pulse one clock cycle wide, on the first clock cycle after it powers up. Note that this is a structural, or RTL (register-transfer logic), description using the industry standard Library of Parameterized Modules (LPM); in general, RTL descriptions are not otherwise portable.

This circuit depends on the flip-flops powering up in an off state. Many FPGAs, the Altera Cyclone included, allow one to specify the power-on states of uninitialized registers; thus, the above circuit could be implemented in an equivalent VHDL behavioral description.

B.2 Memory Segments

The sequencer's external SRAM chip has an external data width of 36 bits, which is not directly usable by most firmware modules. In particular, the communication controllers operate on 8-bit wide data, since Internet protocols are octet-oriented; the AVR core has a 16-bit wide instruction bus and an 8-bit wide data bus; and the PCP core has a 64-bit wide instruction bus. Moreover, these usable data widths are powers of two; to keep the sizer implementation simple, the SRAM data width is rounded down to 32 bits and the upper 4 bits are not currently used.

Consequently, the SRAM controller must be multiplexed among three different sizers, and the 8-bit sizer must be further multiplexed among several different clients, making SRAM storage the source of greatest resource contention in the system. These relationships are depicted in Table B.1.

Virtual Data Width	**Virtual Address Width**	**Data Width Multiple**	**Physical Address Width**	**Segment Prefix**	**Masters**
8	16	2^2	14	0x1D (5 bits)	AVR data memory TCP/AVR proxy PTP/AVR proxy
					PTP (read/write) PCP (read-only)
16	16	2^1	15 (4 bits)	0x1E	AVR instruction memory

Table B.1: Data, address, and segment prefix widths for SRAM masters.

Note that the PCP memory master actually uses a proxy and another sizer in reverse, going from an 8-bit read-only of external SRAM to a wider, write-only master, the 64-bit PCP program memory.

B.3 Pulse Transfer Protocol Application Layer

The Pulse Transfer Protocol (PTP) describes both a routing (network) layer and an application layer. The routing layer provides addressing and datagram-oriented network transfer, analogous to the Internet Protocol, for passing messages up and down a daisy-chain of sequencer devices. The application layer is a request/response architecture suitable for client/server operation. Each client request generates a corresponding reply from the server; the client is responsible for retransmitting requests until a reply is received. The server side is implemented in the sequencer firmware as a PTP server module. On the AVR, the client side is implemented with software drivers and a firmware proxy. On the host PC, the client side is implemented with the command-line bootstrapping program or any third-party library conforming to the protocol; in these cases, PTP uses Ethernet/IP/UDP as an initial datalink layer in traveling from the host PC to the sequencer device.

The opcodes and their payloads are described below in request/reply pairs. They conform to the format of a PTP frame in Figure 4-8. Although the frame length field accommodates 16 bits, the current implementation only supports packets shorter than or equal to 984 octets in length (1024 octets minus headers for UDP and IP). PTP follows the big-endian order convention of IP.[1] Payloads are described in a table where fields are listed in the order they appear in the payload and along with their length in octets. The payload lengths specified are the minimum. In most request opcodes, octets beyond the specified payload are ignored except for variable-length fields. Neither the client nor server should depend on any particular value in PTP packets beyond the stated payload lengths.

B.3.1 Null Opcode

This opcode is guaranteed to have no effect on the PTP server and does not generate a reply. It is currently not used in any implementation.

[1]That is, the most significant bit in an octet has index 0 and the least significant bit has index 7, hence the term “octet” instead of byte. Correspondingly, the most significant octet in a multi-octet word comes first (has the lowest address).

Request

Hex Value: 0x00

Payload Length: 0 octets

Payload Description: Payload is ignored.

B.3.2 Status Opcodes

Request

This opcode requests the status of the sequencer to be returned by the corresponding reply.

Hex Value: 0x01

Payload Length: 0 octets

Payload Description: Payload is ignored.

Reply

This opcode returns the status of the device addressed in the corresponding request.

Hex Value: 0x11

Payload Length: 2 octets

Length in Octets	**Bits**	**Function**
1	0-3	4-bit trigger source, uses the values in Table C.4.
	4	1 if AVR is in reset, 0 if it is running.
	5	1 if PCP is in reset, 0 if it is running.
	6	1 if the device is the chain initiator, 0 otherwise.
	7	1 if the device is the chain terminator, 0 otherwise.
1	0	1 if PCP has halted, 0 otherwise.
	1-7	Lower seven bits of the clock scale quantum (unused).

Table B.2: Payload description for the Status Reply opcode.

B.3.3 Memory Opcodes

PTP can manipulate the entire address space of the external SRAM as byte-addressable memory. It cannot write across segment boundaries. Note that it is unsafe to perform

a memory write operation into the address space of the AVR while it is running, since AVR instructions are retrieved directly from SRAM without any cache.[2]

Request

This opcode requests a memory read or write operation to be performed on the addressed device. Write data is included in the request payload, and the write length (denoted by *length* below) is calculated from the payload.

Hex Value: 0x02

Payload Length: 4 octets + (2 octets for reading, *length* octets for writing.)

Length in Octets	**Function**
1	Subopcode indicating which memory function to perform. 0x01 for writing, 0x02 for reading
1	Address prefix for 16-bit address, 8-bit wide segment (lower 5 bits)
2	Starting 16-bit offset in 8-bit wide segment
2 or *length*	Length for reading Data for writing

Table B.3: Payload description for the Memory Request opcode.

Reply

This opcode reports the success of a requested memory operation after it has completed. For reads, it returns the requested read data.

Hex Value: 0x12

Payload Length: 1 + (*length* for reads)

Octet	**Function**
1	Subopcode of request, same as Table B.3.
length	Data from reading.

Table B.4: Payload description for the Memory Reply opcode.

[2]The bootstrapping program stops the AVR automatically.

B.3.4 Start Opcodes

The PTP server controls both processor cores in the firmware using start opcodes, subject to waiting on triggers. Initially, both cores are held in reset. The Start Request does not perform any program loading for the PCP; pulse programs should be loaded with a Trigger Request first, although the Start Request can be specified as the trigger.

Request

This opcode lowers the reset line for either the AVR or the PCP, causing that core to begin executing instruction starting at address 0x0.

Hex Value: 0x04

Payload Length: 1 octet

Length in Octets	**Function**
1	Subopcode indicating which start operation to perform. 0x01 for resuming the PCP.
	0x02 for suspending the PCP.
	0x03 for resuming the AVR.
	0x04 for suspending the AVR.

Table B.5: Payload description for the Start Request opcode.

Reply

This opcode reports the success of a requested start operation, using the same subopcode as the original request.

Hex Value: 0x14

Payload Length: 1 octet

Octet	**Function**
1	Subopcode of request, same as in Table B.5.

Table B.6: Payload description for the Start Reply opcode.

B.3.5 Trigger Opcodes

The PCP can be made to wait on a specified trigger by the PTP server. Triggers can include any of the feedback inputs, a manual DIP switch position, a PTP Start Request (which does not wait on any trigger), and a null trigger, which stalls the PCP indefinitely until a new trigger is specified. Trigger opcodes are also used to load pulse programs for running; this is the only way to write into PCP program memory, which the PCP treats as read-only. Note that if `clk0` is not running, then the PTP server will stall.

Request

This opcode sets the current trigger for the PCP to the specified source and loads a pulse program beginning at the specified prefix and address in the slow external SRAM into the fast PCP program cache. Changing the trigger without loading a pulse program can be accomplished by setting the read length (*length*) to 0.

Hex Value: 0x05

Payload Length: 6 octets

Length in Octets	**Function**
1	Trigger source using the values in Table C.4.
1	Address prefix for segment containing the pulse program (lower 5 bits).
2	Starting address offset for the pulse program.
2	Length of pulse program.

Table B.7: Payload description for the Trigger Request opcode.

Reply

This opcode reports the success of a trigger operation, echoing the trigger source in the original request.

Hex Value: 0x15

Payload Length: 1 octet

Octet	Function
1	Trigger source of original request in Table B.7.

Table B.8: Payload description for the Trigger Reply opcode.

B.3.6 I^{2}C Opcodes

These opcodes define requests for I^{2}C operations and their corresponding replies. The PTP server can address slaves with 7-bit addresses on the I^{2}C bus to read and write data. Either read or write lengths or both can be zero. This command should never stall because the firmware I^{2}C controller, as a bus master, will simply read back zeros if a slave never responds to a given address.

Request

Because most I^{2}C slaves require an initial write (e.g. of an internal memory address), the I^{2}C Request always performs a write first using all bytes in the payload beyond the read length (*length*) field.

Hex Value: 0x06

Payload Length: 3 + *write length* octets

Length in Octets	Function
1	Slave address (lower 7 bits)
2	Number of octets to expect while reading (*read length*).
write length	Data to write to the slave.

Table B.9: Payload description for the I^{2}C Request opcode.

Reply

This opcode reports the success of an I^{2}C operation and returns any requested read data.

Hex Value: 0x16

Payload Length: (1 + *read length*) octets

Octet	Function
1	Slave address from original request (lower 7 bits).
read length	Octets read from the slave.

Table B.10: Payload description for the I^2C Reply opcode.

B.3.7 Debug Opcodes

These opcodes define requests for the PTP server to perform a debugging operation and the corresponding replies upon completion.

Request

Currently the only debugging function is to blink the general-purpose LEDs in the 1 byte pattern that is the suboperand.

Hex Value: 0x08

Payload Length: (1 or more) octets

Length in Octets	Function
1	Subopcode indicating which debugging function to perform. 0x01 for displaying debugging information on LEDs (1 suboperand).

Table B.11: Payload description for the Debug Request opcode.

Length in Octets	Function
1	Suboperand indicating the 8-bit pattern to display on LEDs.

Table B.12: Payload description for the Debug LED Request subopcode.

Reply

This opcode reports that the corresponding requested debugging action has completed.

Hex Value: 0x18

Payload Length: (1 + *read length*) octets

Octet	Function
1	Subopcode, same value as original request from Table B.11.

Table B.13: Payload description for the Debug Reply opcode.

B.3.8 Discover Opcodes

These opcodes allow the user to both discover the dynamic IP address of the daisy-chain initiator as well as dynamically assign IDs to all other devices in the chain starting with 02. The chain initiator can only be discovered once until it is reset.

Request

Hex Value: 0x09

Payload Length: 1 octet

Length in Octets	Function
1	Current slave address, starting with 0x02.

Table B.14: Payload description for the Discover Request opcode.

Reply

Hex Value: 0x19

Payload Length: 1 octet

Octet	Function
1	Slave address from original request in Table B.14.

Table B.15: Payload description for the Discover Reply opcode.

Appendix C

Programming Reference for the PCP

This appendix serves as a reference for the PCP64 architecture and the only machine currently in it, the `pcp0`.

C.1 Architectural Parameters

The PCP64 family has configurable parameters which are determined by each machine. These parameter values for the `pcp0` are shown in Table C.1.

Parameter	**Allowed Range**
Register Address Width	5 bits
Register Data Width	64 bits
Register Count	32 registers
Instruction Width	64 bits (8 bytes)
Memory Address Width	11 bits
Maximum Program Size	2048 instruction words

Table C.1: Machine parameters for the `pcp0` in the PCP64 architecture.

Every `pcp0` instruction takes two cycles to execute, with the exception of the pulse instructions which take a variable number of cycles depending on their delay value. Every control flow instruction has one branch delay slot due to the pipeline.

C.2 Instruction Set for the pcp0

This section describes the instruction set supported by the pcp0, which may be extended by future machines. Each instruction is described in its own section with a summary of its binary format, and a textual description of its operation.

For each instruction, the binary format summary follows the example of Table C.2. The mnemonic name is given along with a one-line description. The instructions opcode and modifiers are given as binary and hexadecimal values. Operands, their names, bit widths, locations, and interpretations are defined. The result of each instruction is given as one or more modifications of processor states, including the program counter, the output of the pulse register, the pulse delay, the value of general-purpose registers, and the value of locations in memory. The last field describes the symbolic usage of the instruction in an assembly-language program. The notation and abbreviations used in the remainder of this section are explained in Table C.3.

<table>
<tr><td>mnemonic</td><td colspan="8">Description of instruction.</td></tr>
<tr><td>63 58</td><td>57 56</td><td>55 51</td><td>50 46</td><td>45 41</td><td>40</td><td>39 32</td><td colspan="2">31 27 23 19 15 11 7 3 0</td></tr>
<tr><td>6 bits</td><td>2 bits</td><td>5 bits</td><td>5 bits</td><td>5 bits</td><td>1 bit</td><td>8 bits</td><td>21 bits</td><td>11 bits</td></tr>
<tr><td>opcode</td><td>i u</td><td colspan="7">operands appear here</td></tr>
<tr><td colspan="2" rowspan="2">bbbbbb bb
0xhh</td><td>Result:</td><td colspan="5">result_1</td><td>result_2</td></tr>
<tr><td>Usage:</td><td colspan="6">mnemonic operand_1, operand_2</td></tr>
</table>

Table C.2: Example of a PCP64 instruction format table.

Symbol	Description
opcode	6-bit value determining which operation to perform.
i	1-bit opcode modifier denoting an immediate operand.
u	1-bit opcode modifier denoting an unsigned operand.
pc	Program counter (same width as machine address space).
r_p	Pulse output register (64 bits wide).
t_p	Pulse delay time (40 bits wide). For 23-bit immediate timer values, upper 17 bits are zeroed. For 64-bit register timer values, only lower 24 bits are used
sel_x	Selects where an x-bit constant is loaded in a 64-bit register ($\log_2 64/\log_2 x$ bit wide).
$r_n(sel_x)$	A selected subset of the specified 64-bit register.
ui	Unsigned immediate address (same width as machine address space)
[ui]	Dereferenced value or location of unsigned immediate address (64 bits)
uc	Unsigned constant (32 bits)
ti	Immediate timer constant (23 bits)
halt	TRUE when processor has halted. FALSE initially.
$b \dots b$	a binary value.
0x$h \dots h$	a hexadecimal value.

Table C.3: Notation for the PCP64 instruction set.

C.2.1 Load 64-bit Immediate Instruction

ld64i	Load 64-bit unsigned data at an immediate address.							
63 58	57 56	55 51	50 46	45 41	40	39 32	31 27	23 19 15 11 7 3 0
6 bits	2 bits	5 bits	5 bits	5 bits	1 bit	8 bits	21 bits	11 bits
opcode	i u	r_d	unused					ui
000100 10		Result:	$r_d \leftarrow +[ui]$					$pc \leftarrow pc + 8$
0x12		Usage:	ld64i r_d, ui					

This instruction is used to load a 64-bit data word into a general-purpose register. The source of the load is immediately addressed by ui within the program address space; any bits beyond the machine data address width are ignored. The destination of the load is immediately addressed by r_d; any bits beyond the machine register address width are ignored.

C.2.2 Jump Instruction

<table>
<tr><td>j</td><td colspan="8">Jump to an immediate address</td></tr>
<tr><td>63 58</td><td>57 56</td><td>55 51</td><td>50 46</td><td>45 41</td><td>40</td><td>39 32</td><td colspan="2">31 27 23 19 15 11 7 3 0</td></tr>
<tr><td>6 bits</td><td>2 bits</td><td>5 bits</td><td>5 bits</td><td>5 bits</td><td>1 bit</td><td>8 bits</td><td>21 bits</td><td>11 bits</td></tr>
<tr><td>opcode</td><td colspan="7">unused</td><td>ui</td></tr>
<tr><td colspan="2">010111 00</td><td>Result:</td><td colspan="5"></td><td>pc ← ui</td></tr>
<tr><td colspan="2">0x5c</td><td>Usage:</td><td colspan="6">j ui</td></tr>
</table>

This instruction can unconditionally set the program counter to an arbitrary location ui within the program address space. Only the lower bits of ui that are within the machine's address width are used. There is one branch delay slot following a jump which is always executed before the first instruction at the destination.

C.2.3 Branch on Trigger Instruction

<table>
<tr><td>btr</td><td colspan="8">Branch to an immediate address if triggered</td></tr>
<tr><td>63 58</td><td>57 56</td><td>55 51</td><td>50 46</td><td>45 41</td><td>40</td><td>39 32</td><td colspan="2">31 27 23 19 15 11 7 3 0</td></tr>
<tr><td>6 bits</td><td>2 bits</td><td>5 bits</td><td>5 bits</td><td>5 bits</td><td>1 bit</td><td>8 bits</td><td>21 bits</td><td>11 bits</td></tr>
<tr><td>opcode</td><td colspan="4">unused</td><td colspan="2">tr</td><td>unused</td><td>ui</td></tr>
<tr><td rowspan="2">010100 00
0x50</td><td>Result:</td><td colspan="5">if tr, pc ← ui</td><td colspan="2">else pc ← pc + 1</td></tr>
<tr><td>Usage:</td><td colspan="7">btr tr ui</td></tr>
</table>

This instruction allows programs to conditionally branch program execution based on the value of digital feedback inputs in hardware. The operand tr specifies a 9-bit mask which causes a branch when any of the specified feedback bits are high; thus, it provides a logical OR operation on all 9 bits. Pulse programs can then conditionally branched based on one of several different feedback inputs at any point. Table C.4 describes the interpretation of each mask bit.

tr Bit	**Hexadecimal Representation**	**Interpretation**
0	0x001	Feedback input 0
1	0x002	Feedback input 1
2	0x004	Feedback input 2
3	0x008	Feedback input 3
4	0x010	Feedback input 4
5	0x020	Feedback input 5
6	0x040	Feedback input 6
7	0x080	Feedback input 7
8	0x100	DIP switch position 5
9	n/a	PTP start command
n/a	0x000	Null trigger

Table C.4: Interpretation of the tr operand in the btr instruction.

C.2.4 Halt Instruction

halt	Halt program execution							
63 58	57 56	55 51	50 46	45 41	40	39 32	31 27	23 19 15 11 7 3 0
6 bits	2 bits	5 bits	5 bits	5 bits	1 bit	8 bits	21 bits	11 bits
opcode	unused							
011001 00		Result:	halt ← TRUE				pc ← pc	
0x64		Usage:	halt					

After this instruction is executed, the processor enters a halted state, the program counter is no longer incremented, and any pulse outputs in progress remain constant until the processor is reset.

Like branch control instructions, halt has one branch delay slot. Therefore, the last instruction ever executed by a program is not the halt itself, but the instruction immediately after the halt.

C.2.5 Pulse Immediate Instruction

p	Pulse a 32-bit immediate output for up to a 23-bit immediate delay							
63 58	57 56	55 51	50 46	45 41	40	39 32	31 27 23 19 15 11 7 3 0	
6 bits	2 bits	5 bits	5 bits	5 bits	1 bit	8 bits	21 bits	11 bits
opcode		ti	sel_{32}				uc	
011100	00	Result:	$r_p(sel_{32}) \leftarrow uc, t_p \leftarrow ti$				$pc \leftarrow pc + 1$	
0x70		Usage:	p uc, ti, sel_{32}					

This instruction produces a 32-bit wide pulse output for a 23-bit delay constant. Both the output (uc) and the delay (tc) are immediate constant values encoded directly in the instruction. Since the timer normally takes 40-bit values, the upper 17 bits of the timer count for this instruction are zeroed.

Because an instruction is 64 bits wide, an immediate pulsing instruction will never be able to encode all 64-bits; the opcode and timer values occupy some number of bits as overhead. Therefore, 32 bits was chosen as the largest convenient subset of 64. Therefore, the p instruction takes a third parameter, sel_{32}, which is a single bit that determines in which half of the output register the pulse is loaded. The value is interpreted as in Table C.5. This allows the two 32-bit halves of the pulse output register to be modified independently.

This instruction has an output loading overhead of 2 cycles, meaning its output value will appear on the pulse output register 2 cycles after it is fetched.

sel_{32}	**Interpretation**
1	uc is loaded in upper half of r_p (bits 63-32).
0	uc is loaded in lower half of r_p (bits 31-0).

Table C.5: Interpretation of the sel_{32} operand in the p instruction.

C.2.6 Pulse Register Instruction

This instruction loads a 64-bit pulse output and a 40-bit timer value from two registers, r_o and r_t, which can be the same. The timer value is taken from the lower 40 bits of a register.

The instruction has an output loading overhead of 3 cycles. This means its output value will appear in the pulse output register 3 cycles after it is fetched. Consequently, the minimum pulse duration for the instruction is also 3 cycles. It can maintain state for all 64 output bits together in the register file, but it cannot maintain independent state for any of the individual bits.

p	Pulse a 64-bit register output constant for up to a 40-bit register delay constant							
63 58	57 56	55 51	50 46	45 41	40	39 32	31 27 23 19 15 11 7 3 0	
6 bits	2 bits	5 bits	5 bits	5 bits	1 bit	8 bits	21 bits	11 bits
opcode		unused	r_t	r_o	unused			
011101	00	Result:	$r_p \leftarrow r_o, t_p \leftarrow r_t$				$pc \leftarrow pc + 1$	
0x74		Usage:	pr r_o, r_t					

Appendix D

Software Design Files

D.1 AVR Maps

The sizes of the `.text` and `.data` sections will vary from version to version; the numbers here correspond to release 0.01. When linking with the host development tools, a binary has the layout shown in Table D.1 in a unified file. After loading into separate instruction and data memory on the sequencer, the image has the layout shown in Table D.2 in data memory; instruction memory remains identical to the linking map.

Location	**Description**
0x0000	Beginning of program instructions (`.text` section). Interrupt vectors.
0x00CE	ANSI C runtime initialization (`.init` section). Data and BSS copying.
0x0112	Beginning of functions.
0x3E60	Beginning of initialized data (`.data` section). Virtual, static web server filesystem.
0x6BDE	End of file.

Table D.1: The AVR linking map for the web server binary on the host.

Location	**Description**
0x0FFF	Beginning of stack (grows downward).
0x1000	Beginning of initialized data (`.data` section).
	End of initialized data and beginning of heap.
0x9fff	End of heap (`__heap_end` symbol).
0xa000	TCP driver transmit buffer.
0xb000	TCP driver receive buffer.
0xc000	PTP driver transmit buffer.
0xd000	PTP driver receive buffer.

Table D.2: The AVR memory map for the web server binary on the sequencer.

D.2 CGI Variables

Variable Name	Mode	Length (bytes)	Description
dev_id	Sel	2	Hexadecimal number identifying a device to select for subsequent operations.
submit	Ops	5	Determines which form was submitted. One of: Blink controls debugging LEDs. Status updates the status of the selected device. Read reads data from memory. Write writes data into memory. Start starts running a pulse program. Stop stops a pulse program. I2C performs a slave read and/or write over I^2C.
led$x$, $0 \leq x \leq 7$	Ops	0	If this variable is present at all with submit=Blink, turns the given LED bits on and turns all other LED bits off.
read_addr	Ops	6	Six-digit hex address with leading zeros specifying the starting address for submit=Read, submit=Write, and submit=Start.
read_len	Ops	4	Decimal number specifying number of bytes to read with submit=Read or submit=Start
write_file	Ops	0-max	ASCII-encoded binary string representing data to write with submit=Write, 2 hexadecimal characters per byte.
trigger	Ops	1-2	Decimal number representing the trigger source with submit=Start.
i2c_addr	Ops	2	Hexadecimal number representing a 7-bit I^2C slave address with submit=I2C.
i2c_len	Ops	4	Decimal number specifying number of bytes to read with submit=I2C and i2c_addr.
i2c_data	Ops	0-max	ASCII-encoded binary string representing data to write with submit=I2C and i2c_addr.

Table D.3: CGI variables and allowed values.

Bibliography

[AD93] Steve Alexander and Ralph Droms. DHCP options and BOOTP vendor extensions. RFC 1533, Internet Engineering Task Force, October 1993.

[AG79] D.J. Adduci and B.C. Gerstein. Versatile pulse programmer for nuclear magnetic resonance. *Review of Scientific Instruments*, 50(11):1403–1415, November 1979.

[AHT77] D.J. Adduci, P.A. Hornung, and D.R. Torgeson. Auto-increment circuit for a digital pulse programmer. *Review of Scientific Instruments*, 48(6):661–663, June 1977.

[Alt03] Cyclone device handbook. Designer guide version 1.4, Altera Corporation, October 2003.

[Ash04] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, San Francisco, California, 2nd edition, 2004.

[Atm01] ATmega103(L) 8-bit AVR microcontroller with 128k in-system programmable flash. Datasheet rev. 0945G, Atmel Corporation, September 2001.

[Atm02] 8-bit AVR instruction set. Developer manual rev. 0856D, Atmel Corporation, August 2002.

[BCS+04] M.D. Barrett, J. Chiaverini, T. Schaetz, J. Britton, W.M. Itano, J.D. Jost, E. Knill, C. Langer, D. Leibfried, R. Ozeri, and D.J. Wineland.

Deterministic quantum teleportation of atomic qubits. *Nature*, 429:737–739, June 2004.

[CC77] John L. Conway and R.M. Cotts. Circuit for a digital pulse programmer. *Review of Scientific Instruments*, 48(6):656–660, June 1977.

[CGK98] I.L. Chuang, N. Gershenfeld, and M. Kubinec. Experimental implementation of fast searching. *Physical Review Letters*, 18(15):3408–3411, 1998.

[Cha91] Steve Chamberlain. LIB BFD, the binary file descriptor library. User manual for bfd-2.9.1, Cygnus Support, April 1991.

[Chu04] Isaac L. Chuang. Agile pulse control systems for quantum processors. August 2004.

[Cyp01] CY7C1386 512k x 36 pipelines DCD SRAM. Datasheet, Cypress Semiconductor Corporation, December 2001.

[CZ95] J.I. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Physical Review Letters*, 74(20):4091–4094, 1995.

[DiV95] D.P. DiVincenzo. Two-bit gates are universal for quantum computation. *Physical Review A*, 51(2):1015–1022, 1995.

[Dro93] Ralph Droms. Dynamic host configuration protocol. RFC 1531, Internet Engineering Task Force, October 1993.

[EF94] Dean Elsner and Jay Fenlason. Using as, the GNU assembler. User manual for gas-2.9.1, Free Software Foundation, January 1994.

[Eth02] Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. Technical Report IEEE Std 802.3-2002, IEEE LAN/MAN Standards Committee, IEEE Computer Society, March 2002.

[FGM+99] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henry F. Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol (HTTP/1.1). RFC 2616, Internet Engineering Task Force, June 1999.

[FR81] Eiichi Fukushima and Stephen B.W. Roeder. *Experimental Pulse NMR: A Nuts and Bolts Approach*. Perseus Books, Reading, Massachusetts, 1981.

[GC97] Neil A. Gershenfeld and Isaac L. Chuang. Bulk spin-resonance quantum computation. *Science*, 275(5298):350–356, January 1997.

[Her02] Richard Herveille. Specification for the WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores. Developer manual revision B.3, OpenCores Organization and Silicore Corporation, September 2002.

[Hua03] Wei-Han Huang. Instrumentation for quantum computers. Master's thesis, Massachusetts Institute of Technology, December 2003.

[IPC95] Design guidelines for electronic packaging utilizing high-speed techniques. Technical note IPC-D-317-A, Section 5, Institute for Printed Circuits, January 1995.

[JG03] Howard Johnson and Martin Graham. *High-Speed Signal Propagation: Advanced Black Magic*. Pearson Education, Upper Saddle River, New Jersey, 2003.

[Kon00] Jin Au Kong. *Electromagnetic Wave Theory*. EMW Publishing, Cambridge, Massachusetts, 2nd edition, 2000.

[MNAL03] John M. Martinis, S. Nam, J. Aumentado, and K.M. Lang. Decoherence of a superconducting qubit due to bias noise. *Physical Review B*, 67(094510), 2003.

[MO99] John Martinis and Kevin Osborne. Superconducting qubits and the physics of josephson junctions. 1999. Submitted to Les Houches conference proceedings; cond-mat/0402415.

[Nat99] DP83843BVJE PHYTER Ethernet PHY controller and MII. Datasheet, National Semiconductor Corporation, July 1999.

[Nat01] DS90LV048 LVDS quad CMOS differential line receiver. Datasheet, National Semiconductor Corporation, May 2001.

[Nat03] DS90LV047 LVDS quad CMOS differential line driver. Datasheet, National Semiconductor Corporation, January 2003.

[Nat04] LVDS owner's manual. Designer guide 3rd edition, National Semiconductor Corporation, 2004.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, U.K., 2000.

[OWN97] Alan V. Oppenheim, Alan S. Willsky, and Hamid Nawab. *Signals and Systems*. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 1997.

[Pha04] Paul Pham. MIT-NIST-ARDA pulse sequencer. Technical manual release 0.01, MIT Center for Bits and Atoms, December 2004.

[Phi00] The I^{2}C-bus specification. Designer guide version 2.1, Philips Semiconductors, January 2000.

[Plu82] David C. Plummer. An Ethernet address resolution protocol. RFC 826, Internet Engineering Task Force, November 1982.

[Pos80] Jon Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 1980.

[Pos81a] Jon Postel. Internet control message protocol. RFC 792, Internet Engineering Task Force, September 1981.

[Pos81b] Jon Postel. Internet protocol, version 4. RFC 791, Internet Engineering Task Force, September 1981.

[Pos81c] Jon Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[RHR+04] M. Riebe, H. Häffner, C.F. Roos, W. Hänsel, J. Benhelm, G.P.T. Lancaster, T.W. Körber, C. Becher, F. Schmidt-Kaler, D.F.V. James, and R. Blatt. Deterministic quantum teleportation with atoms. *Nature*, 429:734–737, June 2004.

[SFR03] W. Richard Stevens, Bill Fenner, and Andrew Rudoff. *The Sockets Networking API*, volume 1 of *UNIX Network Programming*. Addison-Wesley, Boston, Massachusetts, 3rd edition, 2003.

[SMC03] Matthias Steffen, John M. Martinis, and Isaac L. Chuang. Accurate control of josephson phase qubits. *Physical Review B*, 68(224518), 2003.

[Som99] Carlo G. Someda. *Electromagnetic Waves*. Chapman & Hall, London, 1999.

[Spi04a] Pulseblasterdds™ PCI board SP3. User manual, 2004.

[Spi04b] PulseBlasterESR™ PCI board SP4. User manual, 2004.

[Spi04c] PulseBlaster™ PCI board SP2. Owner's manual, 2004.

[TDP84] Hans Thomann, Larry R. Dalton, and Charles Pancake. Digital pulse programmer for an electron-spin-resonance computer-controlled spectrometer. *Review of Scientific Instruments*, 55(3):389–398, March 1984.

[Var00a] ^{UNITY}INOVA technical reference. Technical Report Pub. No. 01-999047-00, Rev. C0200, 2000.

[Var00b] VNMR user programming. User manual Rev. A0800, 2000.

[VHD02] IEEE standard VHDL language reference manual. Technical Report IEEE Std 1076-2002, IEEE Design Automations Standard Committee, IEEE Computer Society, May 2002.

[VSB+01] Lieven M.K. Vandersypen, Matthias Steffen, Gregory Breyta, Constantino S. Yannoni, Mark H. Sherwood, and Isaac L. Chuang. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883–887, December 2001.

[WPBM93] Xiao Wu, Donald A. Patterson, Leslie G. Butler, and Joel B. Miller. A broadband nuclear magnetic resonance spectrometer: Digital phase shifting and flexible pulse programmer. *Review of Scientific Instruments*, 64(5):1235–1238, May 1993.

[WSF88] E.A. Wachter, E.Y. Sidky, and T.C. Farrar. Enhanced state-machine pulse programmer for very-high-precision pulse programming. *Review of Scientific Instruments*, 59(10):2285–2289, October 1988.

[YYHG02] Jiang Yun, Jiang Yu, Tao Hongyan, and Li Gengying. A complete digital radio-frequency source for nuclear magnetic resonance spectroscopy. *Review of Scientific Instruments*, 73(9):3329–3331, March 2002.