

Quantum Computing Pulse Programmer

Technical Manual

by

Paul T. Pham

University of Washington, Department of Computer Science & Engineering
Max-Planck-Institut für Quantenoptik
Institut für Experimentalphysik, Universität Innsbruck
MIT Center for Bits and Atoms

Version 0.31

July 3, 2006

Contents

1	Overview	7
1.1	Other Documents	7
1.2	Version Numbers	8
1.3	Organization	8
1.4	Disclaimer	9
1.5	License	9
1.6	Acknowledgements	10
1.7	Online Resources	10
I	Introduction	7
2	Features	11
2.1	Feature Summary	11
2.2	Box Configuration	11
II	Planning	11
3	Step-by-Step Building Guide	12
4	User Setup	14
4.1	Setting up the Control PC	14
4.2	Setting up the Ethernet Network	14
4.2.1	Network Address Rendezvous	14
4.2.2	Private Subnet	15
4.2.3	Firewall	15
5	Ordering Parts	16
III	Building	16
6	PCB Fabrication and Assembly	17
6.1	Fabricating Printed Circuit Boards	17
6.1.1	Revision Letter	17
6.1.2	CAM Files	17
6.1.3	Fabrication	17
6.2	Assembling the Sequencer and Breakout Boards	18
7	Building the Box	20
7.1	Machining Connector Holes in the Faceplate	20
7.2	Connecting the the Faceplate	20
7.3	Drilling Mounting Holes into the Chassis	20
7.4	Making the Power Regulator Board	20
7.5	Crimping Ribbon Cables	20
7.6	Building the Chain Daughterboards	20
8	Mounting the Ground Level and Running First Power-On Tests	21
8.1	PCB Layout	21
8.1.1	Ethernet Interface	21
8.2	Booting Up	21
8.3	Troubleshooting	22

IV	Hardware Testing	21
9	FPGA Programming	23
9.1	Prerequisites	23
9.2	Installation Procedure	23
9.2.1	User Installation	23
9.2.2	Developer Installation	24
9.3	Programming Cable Installation	24
9.4	Using the Programming Cable	25
9.4.1	Programming New Firmware	25
9.4.2	Debugging Firmware	26
10	Testing Sequencer and Device Chains	28
10.1	Testing a Sequencer	28
10.2	Testing a Breakout Board	28
10.3	Testing a Chain Daughterboard	28
10.4	Testing a DAC Device and Chain Daughterboard	28
10.5	Testing a DDS Device and Chain Daughterboard	28
10.6	Testing a VGA Device and Chain Daughterboard	28
11	Firmware	29
11.1	Building the Firmware	29
V	Hardware Design	29
VI	Firmware Design	29
12	Writing Software Configuration Settings for a Box	30
VII	Software Design	30
13	Adding Software Devices	31
14	Running Software Tests on the Finished Box	32
15	Operating Information	33
15.1	DIP Switches	33
15.2	Configuring Clocks	33
15.3	Using the Daisy Chain	33
A	Pulse Transfer Protocol	34
A.1	Null Opcode	35
A.2	Status Opcodes	35
A.3	Memory Opcodes	36
A.4	Start Opcodes	37
A.5	Trigger Opcodes	37
A.6	I ² C Opcodes	38
A.7	Debug Opcodes	38
A.8	Discover Opcodes	38
VIII	Appendices	34

B	PCP32/16 Programming Reference	39
B.1	Architectural Parameters	40
B.2	Machine Model	41
B.3	Instruction Format	42
B.4	Instruction Set	43
B.4.1	Nop Instruction	43
B.4.2	Halt Instruction	43
B.4.3	Wait Instruction	44
B.4.4	Branch-On-Trigger Instruction	44
B.4.5	Jump Instruction	45
B.4.6	Move Instruction	45
B.4.7	Subroutine-Call Instruction	45
B.4.8	Subroutine-Return Instruction	45
B.4.9	Compare Instruction	45
B.4.10	Branch-On-Flags Instruction	46
B.4.11	Load-Counter Instruction	46
B.4.12	Store-Counter Instruction	46
B.4.13	Load-Register Instruction	46
B.4.14	Store-Register Instruction	47
B.4.15	Read-Segment Instruction	47
B.4.16	Write-Segment Instruction	47
B.4.17	Add Instruction	47
B.4.18	Multiply Instruction	47
B.4.19	Divide Instruction	48
B.4.20	Pulse-Phase Instruction	48
B.4.21	Load-Phase Instruction	48
B.4.22	Pulse-DAC Instruction	48
B.4.23	Pulse-DDS Instruction	48
B.4.24	Pulse-DDS-Profile Instruction	49
B.4.25	Pulse-TTL-Higher Instruction	49
B.4.26	Pulse-TTL-Lower Instruction	49
B.4.27	Pulse-Chain-Address Instruction	49
B.4.28	Increment Instruction	49
B.4.29	Load-Constant Instruction	49

IX	Pulse Control Processor Families	39
-----------	---	-----------

List of Figures

1-1	Where does a pulse sequencer fit into quantum computing?	7
3-2	?	13
6-3	Cost summary of initial fabrication run in February 2004.	18
6-4	Modifications for Revision A PCB assemblies.	19
8-5	The main components of the sequencer hardware.	21
9-6	A screenshot of the SignalTap II logic analyzer software.	27
B-7	This is not really the PCP32/16 machine model.	41

List of Tables

6-1	CAM filename extensions and descriptions for Protel 99 SE.	18
7-2	Canonical pinouts for an Innsbruck and MPQ box.	20
A-3	Fields of a Pulse Transfer Protocol frame	34
A-4	Summary of PTP opcodes	34
B-5	Fixed parameters for the PCP32/16 architecture.	40
B-6	Machine parameters for the pcp1 in the PCP32 architecture.	40
B-7	PCP32/16 example instruction format.	42
B-8	Notation for the PCP32/16 instruction set.	42
B-9	PCP32/16 instruction set summary.	43

1 Overview

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 using minimal resources and introducing as few errors as possible.

There are two parts to solving this problem: designing a language for specifying arbitrary pulse sequences using *pulse programs* and designing a device, called a *pulse sequencer*, for translating this language efficiently and accurately into digital outputs. An abstract model of this situation is depicted in Figure 1-1. As a concrete implementation of this model, this project contains a specialized Pulse Control Processor (PCP) for outputting digital pulses and a corresponding assembly language (PCP assembly); it also contains user interfaces and development tools to make using the pulse sequencer and writing pulse programs easier and more intuitive.

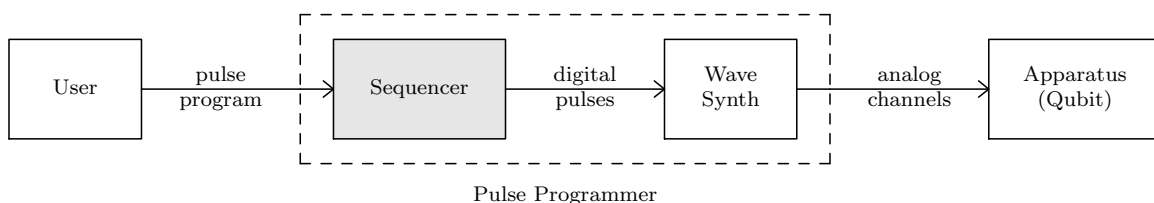


Figure 1-1: Where does a pulse sequencer fit into quantum computing?

By itself, the sequencer only controls the timing of digital outputs (bits) and is agnostic to how these outputs 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 specific to the chosen qubit; multiple bits from the sequencer can be assigned to a single channel, and one sequencer can control multiple channels at once. To make the pulse sequencer as general as possible, this project does not contain a waveform synthesizer but includes interface specifications and examples so that you can design your own.

1.1 Other Documents

There are two main documents associated with the pulse sequencer project. Each one serves a different purpose as described below but can be read as a self-contained unit. There is a small overlap in introductory material, just as there will probably be an overlap between the different audiences addressed. Rather than searching through an all-inclusive single document, you can choose to read only the material that is most relevant to your needs.

Manual This is the document you’re reading now. It is meant for users and engineers who already know what a pulse sequencer is in general and want to use and modify this device in particular. Read this if you want to write and run pulse programs, to interface them with other apparatuses (e.g. a D/A converter or a frequency generator), or to add new features. It is written as a collection of how-to guides.

Thesis dissertation This document gives some context for the use of a pulse sequencer in quantum computing and describes the design, implementation, and evaluation of the current device. Read this if you want to know what problems a pulse sequencer can solve in general or what rationale and alternatives exist for this device in particular. It is written as a logical narrative and evaluation [1]. You can find a copy on the project website in the package `sequencer-thesis`.

Printable (Postscript and PDF) and online (HTML) versions of each document, along with the L^AT_EX source, can be found on the project website.

1.2 Version Numbers

The project has many different interacting components, or *packages* which can be modified separately: hardware, firmware, software, and documentation. Clearly, a uniform scheme for managing changes is needed to ensure that these packages continue working with each other as expected. For the purposes of this manual, the terms *version* and *revision* are used interchangeably: they are a snapshot of a project component at a particular point in time and are assigned a number *x.xx* which never changes. This allows users and developers to refer to revisions unambiguously. A *release* is a file that is officially available from the project website; it always increments revision numbers. A *build* is any compiled binary file (either firmware, software, or documentation), which may or may not be publicly released; for example, a special build may be created for debugging purposes.

All packages except the hardware ones have non-overlapping version numbers in the same timeline. For example, the following packages were released in chronological order:

0.18 sequencer-docs

0.20 sequencer-firmware

0.21 sequencer-firmware

0.23 sequencer-python

No two versions have the same number, even if they belong to different components. This helps you remember that Python compiler v0.23 was released after the FPGA firmware v0.21 and is compatible with it. In general, backwards compatibility does not extend beyond one version number, so there may be some feature in firmware v0.20 that is no longer supported in Python compiler v0.23 and later.

There are two main reasons why the hardware packages are versioned separately. One is historical convention; software has version numbers, but hardware generally has revision letters. The other reason is functional: hardware is so hard and expensive to change that its interface to firmware and software must stay the same. Therefore, it should not matter when Revision C hardware was released relative to software version 0.25; any hardware revision should work with any other “soft” package version.

1.3 Organization

The remainder of this manual is organized as follows. All users and developers should read Chapter 2 first. This will tell you at a glance if this device can do what you want it to, and if not, how hard it is to modify.

The sections following this are divided into parts describing different stages in using and modifying the sequencer device. The first three parts are user guides and are relevant to people who wish to use the sequencer without necessarily understanding how it works. The last two parts are developer guides for modifying the sequencer.

Part ?? describes the basics of configuring, maintaining, and troubleshooting sequencer hardware, which should occur infrequently during normal use. Chapter 4 is a quick-start guide to setting up a sequencer device from out-of-the-box to loading the web interface; this is the only essential section in this part, and afterwards most users can skip to the next part. The remaining sections describe tasks that experienced users may need to perform in increasing order of complexity. Those who need to configure the sequencer’s clock sources should read Chapter ?. Others who need to operate multiple sequencer devices in a daisy-chain should read ?. Finally, Chapter ?? describes how to update the firmware of your sequencer device when a new update is released.

Part ?? describes pulse programs, the pieces of software that tell the sequencer how to generate pulses. Chapter ?? introduces the processor core that interprets pulse programs and its associated programming model. Chapter ?? covers the basic syntax of pulse programs, the compilation process, and the generation of simple pulses. Users can write most of the pulse programs they need after reading these two sections and will know enough to run their programs via the user interfaces in

the next part. For more advanced pulse programs, Chapter ?? describes how to generate more complicated pulse sequences that encounter the sequencer’s performance limits.

Part ?? describes the sequencer’s user interfaces. Most operations can be performed using the graphical web interface, which is documented in Chapter ?. A command-line version is convenient for automated, non-interactive tasks and is documented in Chapter ?.

Part ?? is for developers or engineers who need additional functionality not currently present in the sequencer. The advantage of an open design is that the preferred source forms are available, and the following sections will describe how to modify them. Developers should read Chapter ? to setup the development environment, but afterwards can jump directly to any of the following three sections to learn how to modify each of the device’s subsystems: Chapter ? for hardware, Chapter ? for firmware, and Chapter ? for software. Within each section, each subsection describes a particular module in the current design or how to perform a particular modification for a modified design. These listed in increasing order of abstraction, but most can be read independently from each other. Chapter ? gives guidelines for designing a matching daughterboard to the sequencer; these will interface the digital pulses to a particular experiment.

Part ?? is for a particular kind of developer, the one who will be maintaining the official versions of the sequencer, including the project website and this manual document. This part is provided in an end-user manual for completeness, since sometimes a developer or a user will need to perform a maintainer task. Also, in the open source spirit, if support from the original author(s) or maintainer(s) ever lapses, anyone can fork the project and keep on going. Chapter ? describes how to build the thesis document and this manual from source and how to update the project website. Chapter ? describes how to have a board fabricated and assembled from scratch and how to release a new version of the hardware design documents. Chapter ? and Chapter ? describe how to compile the firmware and software, respectively, from source; Chapter ? describes how to post a new version of the firmware and software (which are synchronized together).

Appendix ? contains the current version notes. Appendix ? is a complete instruction set reference. Appendix ? explains how to use the LED test board to evaluate the sequencer.

1.4 Disclaimer

The instructions in this manual have been tested with the versions of third-party software and hardware listed in the setup guides of Chapters 4 and ?, and the latest available versions of the firmware and software.

Other versions and version combinations may work, but the user is advised to upgrade or downgrade to the specified versions before submitting bug reports or asking for help. The author(s) or maintainer(s) will attempt to answer questions by e-mail, but do not guarantee any formal technical support.

1.5 License

The documentation for the sequencer project, including all circuit schematics, PCB layouts, source code (in C, VHDL, m4, and \LaTeX) and the website is released under the following 3-clause BSD license.

Copyright (c) 2006, Quantum Computing Pulse Sequencer Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the Quantum Computing Pulse Programmer Project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.6 Acknowledgements

The following contributors have helped debug and develop the Python compiler and finished hardware designs:

Philipp Schindler in the Quantum Optics and Spectroscopy Group at the University of Innsbruck.

Hector Schmitz and Lutz Petersen in the Quantum Analog Simulation Group at the Max Planck Institute for Quantum Optics in Garching.

Firmware cores were used from the following contributors as part of the OpenCores project:

Richard Herveille wrote the I²C controller core.
[Version 0.9, 3 July 2004]

The sequencer device and this manual benefitted from user testing and feedback from:

Mark Yeo in Chris Munroe's ion trap group at the University of Michigan.

1.7 Online Resources

The official website for this project is:

`http://pulse-sequencer.sf.net`

Check here first for the latest project news, releases, and solutions to known problems. Unless noted otherwise, all mentioned design documents and project materials are available online, including their preferred source forms for modification. Materials that are not online are usually available upon request.

All user feedback is welcome, including questions, comments, and bug reports. The author can be contacted at the e-mail address listed on the website above.

2 Features

Before you invest time and effort into learning how to use this sequencer, you may want to know its features at a glance, both what it can do currently and how hard it is to modify for new capabilities. These are listed below.

2.1 Feature Summary

10 nanosecond timing resolution corresponding to maximum clock speed of 100 Mhz.

64 digital pulse outputs that can switch simultaneously.

8 digital inputs can be used for either feedback branching or triggering.

10 nanosecond minimum duration for 32 simultaneously-switching outputs; 30 nanosecond minimum duration for all 64 simultaneously-switching outputs.

0 nanosecond minimum delay between pulses of 30 nanoseconds or longer; 10 nanosecond minimum delay after pulses of 10 and 20 nanoseconds.

60-150 nanosecond feedback latency between feedback input rising edge and corresponding pulse output.

135-165 nanosecond trigger latency between trigger input rising edge and corresponding pulse output.

6-opcode instruction set and assembly language for writing pulse programs with conditional branching on feedback.

Standard Ethernet interface for connecting to existing networks and commodity PCs.

Simple custom UDP protocol for loading and running pulse programs on any device in a daisy-chain. Can be used to implement interface libraries (not included) for third-party software like LabVIEW or MATLAB.

TCP/HTTP interface for an intuitive graphical interface using any standard web browser.

2.2 Box Configuration

3 Step-by-Step Building Guide

Building a pulse programmer box from scratch involves the following steps, with conservative time estimates given for a single person working full-time on this project.

1. Choose the hardware configuration of all boxes (Indeterminate time). [Section ??]
2. Set up software and hardware on a control PC (1 day). [Section ??]
3. Set up the network/Ethernet interface (Indeterminate time). [Section ??]
4. Order parts (1 week + indeterminate waiting time). [Section 5]
5. Have circuit boards fabricated (1 week). [Section ??]
6. Write additional devices (1 day). [Section 13]
7. Have sequencer and breakout board assembled (2 weeks). [Section ??]
8. Have connector holes machines out of box faceplate (1 week). [Section ??]
9. Drill mounting holes (2 days). [Section ??]
10. Make the power subsystem (1 day). [Section ??]
11. Make ribbon cables (1 day). [Section ??]
12. Build the chain daughterboards (0.5 days per board). [Section ??]
13. Mount the power subsystem, sequencer, and breakout boards and run first power-on tests (1 day). [Section 8]
14. Write the software configuration file for your box (1 day). [Section 12]
15. Program the FPGA (Less than an hour). [Section 9]
16. Test the sequencer and breakout board (1 day). [Section ??]
17. Assemble the faceplate connectors (1 day). [Section ??]
18. Connect and test the chain daughterboards (0.5 days per board). [Section ??]
19. Connect and test the first DAC board and chain (0.5 days per DAC/chain pair). [Section ??]
20. Connect and test the first DDS board and chain (1 day). [Section ??]
21. Connect and test the first VGA board (1 day). [Section ??]
22. Run software tests on the finished box. (3 days). [Section 14]

The time from initial planning to having a working box varies greatly depending on what features you want to include in your box, how many people you have working on it, and how long it takes to place orders. A simple one-channel box can be completed in two months in the best case; more complicated boxes or larger quantities can take three months or more.

The list gives a simple, high-level description for each task. It conveys the main difficulties in building a box and can be used as a handy check-off list during the building process itself. Each item references a later subsection in this manual where the task is described in excruciating detail with very specific steps. The task list is not linear, and some steps can be completed in parallel according to the dependency graph in Figure 3-2.

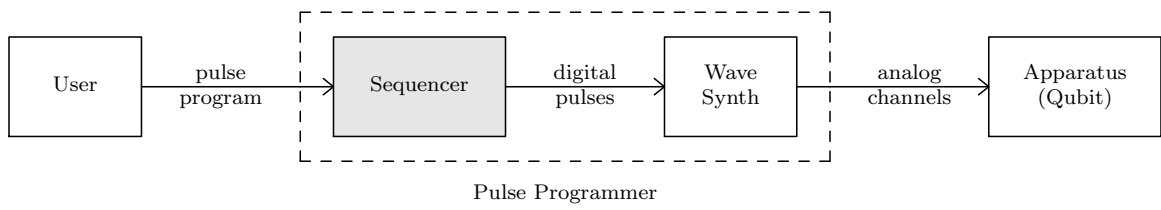


Figure 3-2: ?

4 User Setup

This section is a quick-start guide for a new user to prepare the lab environment for using the pulse programmer. It consists of two parts: setting up a control PC with the right software and hardware (Section 4.1) and setting up the local Ethernet network to connect the control PC and the pulse programmers (Section 4.2).

4.1 Setting up the Control PC

First, you need to choose a suitable PC to act as the control computer. For only running experiments, the following requirements are sufficient for the pulse programmer. You should also consider the overhead of running any other software needed by your experiment (e.g. LabVIEW, MATLAB, etc.)

- Any operating system that runs Python (includes Windows, Mac OS X, Linux, and most kinds of UNIX).
- The latest version of Python (tested with 2.4.1) to run the compiler and scripts for the pulse programmer.¹
- A web browser to configure your router/firewall (if applicable).
- A typical computer at the time of this writing had a 2 GHz processor, 512 MB of RAM, and 150 MB of free hard drive space for Python and the pulse programmer software. These are more than sufficient for the purpose.

If you want to *develop* firmware or hardware for the pulse programmer, there are further requirements on the control PC. For firmware, see Chapter 9. There is currently no section for hardware development, but the CAD files were done in Protel 99 SE, which only runs on Windows.

4.2 Setting up the Ethernet Network

The sequencer is controlled over an ordinary Ethernet network for flexible decoupling from the control PC. This allows you to control multiple pulse sequencers from the same PC simply by changing some program constants, and different sequencers can be added or removed independently of each other through the wonder that is Ethernet. Here we speak about the pulse sequencer specifically instead of the entire pulse programmer because the sequencer contains all the Ethernet functionality.

However, this would be too simple, and then we couldn't write a subsection about it. First, we describe the problem of *rendezvous*, or how software on the control PC can find the sequencer hardware on the network in Subsubsection 4.2.1. Then we'll describe the most common setup first in Subsubsection ???: a 100baseTX unshielded-twisted pair network, with PC and sequencers on the same subnet, and static IP addresses. Another, more exotic configurations is mentioned here for completeness. Subsubsection ?? describes how to operate sequencers which are behind a firewall.

4.2.1 Network Address Rendezvous

How do you know what IP address a particular sequencer has? How do you guarantee that different sequencers on the same network have different MAC addresses to avoid collisions? You must specify these in two places. First, in the hardware DIP switches on the sequencers themselves as described in Chapter 8. Second, in the software configuration of the Python compiler as described in Chapter ??. This is how the software and hardware will find each other, hence the name rendezvous.

While it is possible and sometimes desirable to use DHCP to dynamically allocate IP addresses, you must still specify static MAC addresses and the process is more complicated: (1) the control PC broadcasts a discover message, (2) all sequencers identify themselves, (3) the control PC parses all replies to find the one with the correct MAC address, and then (4) uses the corresponding IP address for all future communication. If your experiment reboots, or your PC reboots, or both, then

¹<http://www.python.org>

Switch Position	MAC Address	IP Address
0x00	00:01:ca:22:22:20	192.168.0.220
0x01	00:01:ca:22:22:21	192.168.0.221
0x02	00:01:ca:22:22:22	192.168.0.222
0x03	00:01:ca:22:22:23	192.168.0.223
0x04	00:01:ca:22:22:24	192.168.0.224
0x05	00:01:ca:22:22:25	192.168.0.225
0x06	00:01:ca:22:22:26	192.168.0.226
0x07	00:01:ca:22:22:27	192.168.0.227
0x08	00:01:ca:22:22:28	192.168.0.228
0x09	00:01:ca:22:22:29	192.168.0.229
0x0a	00:01:ca:22:22:2a	192.168.0.230
0x0b	00:01:ca:22:22:2b	192.168.0.231
0x0c	00:01:ca:22:22:2c	192.168.0.232
0x0d	00:01:ca:22:22:2d	192.168.0.233
0x0e	00:01:ca:22:22:2e	192.168.0.234
0x0f	00:01:ca:22:22:2f	192.168.0.235

you have to detect this and go through the process again. Plus, some implementations of Python don't give you access to the IP header to get the IP address, so you have to include it in the UDP payload, which involves changing the firmware and PTP.² With static addresses, you can go straight to step (4) above, and it doesn't matter if anything reboots.

The DIP switch positions on the sequencer correspond to the following hardcoded MAC addresses and IP addresses.

4.2.2 Private Subnet

4.2.3 Firewall

²And I am lazy.

5 Ordering Parts

6 PCB Fabrication and Assembly

6.1 Fabricating Printed Circuit Boards

Making a new hardware revision has a fabrication part, resulting in a bare PCB, and an assembly part, resulting in a PCB populated with components. There is no reason these must be separate processes, but originally they were handled by separate companies. Step-by-step instructions are not provided below, as these are likely to change rather frequently over time; however, examples of past orders are provided as a guidelines.

The hardware shouldn't change as rapidly as the firmware and software, so you won't be following the steps in this subsection for most versions. The hardware was designed in Protel 99 SE from Altium³, so you will need this package (or at least the free trial version) to view and edit the circuit schematics and PCB layout.

6.1.1 Revision Letter

The Protel database (`jjqcxx.ddb`, where *xx* is the current version number, should be updated with every new version. In addition, all changes in the current version should be noted and dated in the plaintext file `revisions.txt` within the Protel database and the version notes of this manual in Appendix ???. If a particular version results in a new PCB being fabricated, this counts as a hardware revision, and the associated letter should also be incremented.

The only reason intermediate numbers are used at all is to facilitate collaboration across organizations, such as MIT and NIST. The files are compressed in Protel and copied whole. Obviously an improvement would be to use a free EDA tool with an open file format that would support versioning using CVS or Subversion.

6.1.2 CAM Files

In order to fabrication and assemble the PCB, you must first generate computer-aided manufacturing (CAM) files in Protel, which will all share the same base filename with different extensions for different layers or purposes. Only a subset of the generated files is truly necessary. Table 6-1 describes the purpose of each filename extension and in which process they should be included. The files marked as optional are for informational purposes only and may be omitted without consequence.

6.1.3 Fabrication

The original PCBs were fabricated by Advanced Circuits⁴ in Aurora, Colorado, due to the trace widths of the PCB, which were originally 6 mils, the plated edge for the edgemount connector, and the controlled dielectric stackup. The trace widths have since been relaxed to 8 mils, and it is possible that the last two features above are overkill for signal integrity. We won't really know until the BNC breakout board is made and tested; these results will be documented on the website. Without these features, the board is much cheaper to produce and can even be done by quick-turn prototyping companies such as AP Circuits⁵. If no changes are made to the board from the last revision (Revision B as of this writing), Advanced Circuits can reuse their existing artwork and it will be cheaper to fabricate.

The basic steps when ordering a fabrication are:

1. Compress the fabrication files (Gerber files and NC drill file) into a zip file named `sequencer-pcb-x.zip` where *x* is the hardware revision letter. You will post this later to the project website as part of a release. You can either use GNU zip on the command-line in MSYS, Cygwin, or Linux/UNIX; PKZip in Windows 2000 and earlier; and the built-in right-click zipping in Windows XP and later).

³<http://www.altium.com>

⁴<http://www.4pcb.com>

⁵<http://www.apcircuits.com>

Filename Extension	Description	Process
.GTL	Top copper layer	Fabrication
.GB0	Bottom copper layer	Fabrication
.GTO	Top overlay/legend/silkscreen layer	Fabrication
.GB0	Bottom overlay/legend/silkscreen layer	Fabrication
.GP x	Power plane, x is a number (negative, shows keepout regions)	Fabrication
.GM x	Mechanical layer, x is a number	Fabrication
.GTS	Top soldermask (usually negative, shows keepout regions)	Fabrication
.GBS	Bottom soldermask (usually negative, shows keepout regions)	Fabrication
.DRL	NC Drill file (locations and sizes of drill holes)	Fabrication
.DRR	Drill tool list	Fabrication/Optional
.BOM	Top pad master	Assembly
.GPT	Top pad master	Assembly
.GPB	Bottom pad master	Assembly
.GPT	Top paste mask	Assembly/Optional
.GPB	Bottom paste mask	Assembly/Optional

Table 6-1: CAM filename extensions and descriptions for Protel 99 SE.

- (Optional) Advanced Circuits has a free online design checking tool called FreeDFM⁶ which is mostly helpful but sometimes misleading. You can upload your zip file and get an e-mail back with a URL telling you all the stuff you supposedly did wrong.
- Order the board using the appropriate web form, being sure to specify a plated edge for connector J1, which should cost marginally more per board. The summary of costs for the initial run of boards is shown in Table 6-3 for reference. You can try skipping the electrical test to save money, but it will make the fab engineers really nervous. Obviously the actual prices will vary.

Item	Quantity	Unit Cost	Total Cost
6-layer PCB with plated edge and controlled dielectric	30	\$53.70	\$1,611.00
NRE Sales	1	\$319.00	\$319.00
Shipping & Handling	1	\$15.86	\$15.86
Electrical Test	1	\$200.00	\$200.00

Figure 6-3: Cost summary of initial fabrication run in February 2004.

6.2 Assembling the Sequencer and Breakout Boards

The first three runs of the sequencer PCB were assembled by Axiom Operations⁷ in Beaverton, Oregon, part of Ambitech International. Mention should be made of these previous runs so that the staff can use existing stencils and refer to past modifications.

For Revision A boards, refer in the order to the MIT assembly job in February 2004 and request the modifications in Figure 6-4.

For Revision B boards, refer in the order to the NIST assembly job in April 2004 and the MIT assembly job in January 2005. No hardware modifications are necessary.

For all revisions, you will need to request that the assembly house program the GAL clock switch before stuffing. The necessary file is a standard JEDEC file called `clock_switch.jed` that is available on the project website with the other assembly files. Programming the part requires a

⁶<http://www.freedm.com>

⁷<http://www.axiomleadfree.com/>

1. Cut +5V line of R88. Connect this line to gnd (to C232). This enables the Ethernet chip to reset properly.
2. Cut trace to pin 87 on SRAM (U15) (7th pin from top right), and connect it to +3V at pin 84 (4th pin from top right). Short pin 87 to ground. This enables the global write line.
3. For transformers TF1, TF2, TF5, TF6, patch pads center top to right bottom. The wrong pinout was originally used for these parts.

Figure 6-4: Modifications for Revision A PCB assemblies.

special hardware programmer with an attachment for plastic leadless chip carrier (PLCC) packages, which was not available to the original author. Ambitech/Axiom is known to have this capability for a minimal additional charge.

7 Building the Box

7.1 Machining Connector Holes in the Faceplate

7.2 Connecting the the Faceplate

7.3 Drilling Mounting Holes into the Chassis

7.4 Making the Power Regulator Board

7.5 Crimping Ribbon Cables

Connector	Output Bits															
J11	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
	BA3	BA2	BA1	BA0	T20	T19	T18	T17	T16	BA3	BA2	BA1	BA0	PS0	PS1	FUD
	AD9744 DAC				TTL				AD9858 DDS							
J5	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
	T15	T14	T13	T12	T11	T10	T09	T08	T07	T06	T05	T04	T03	T02	T01	T00
	TTL															
J4	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	D7	D6	D5	D4	D3	D2	D1	D0	A5	A4	A3	A2	A1	A0	WRB	PSEN
	AD9858 DDS															
J3	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	D13	D12	D11	D10	D09	D08	D07	D06	D05	D04	D03	D02	D01	D00	CLK	T21
	AD9744 DDS															TTL

Table 7-2: Canonical pinouts for an Innsbruck and MPQ box.

7.6 Building the Chain Daughterboards

8 Mounting the Ground Level and Running First Power-On Tests

8.1 PCB Layout

Figure 8-5 shows the top-side of the sequencer PCB assembly with all the major hardware components marked. Refer back to this drawing when following future instructions to familiarize yourself with the board layout.

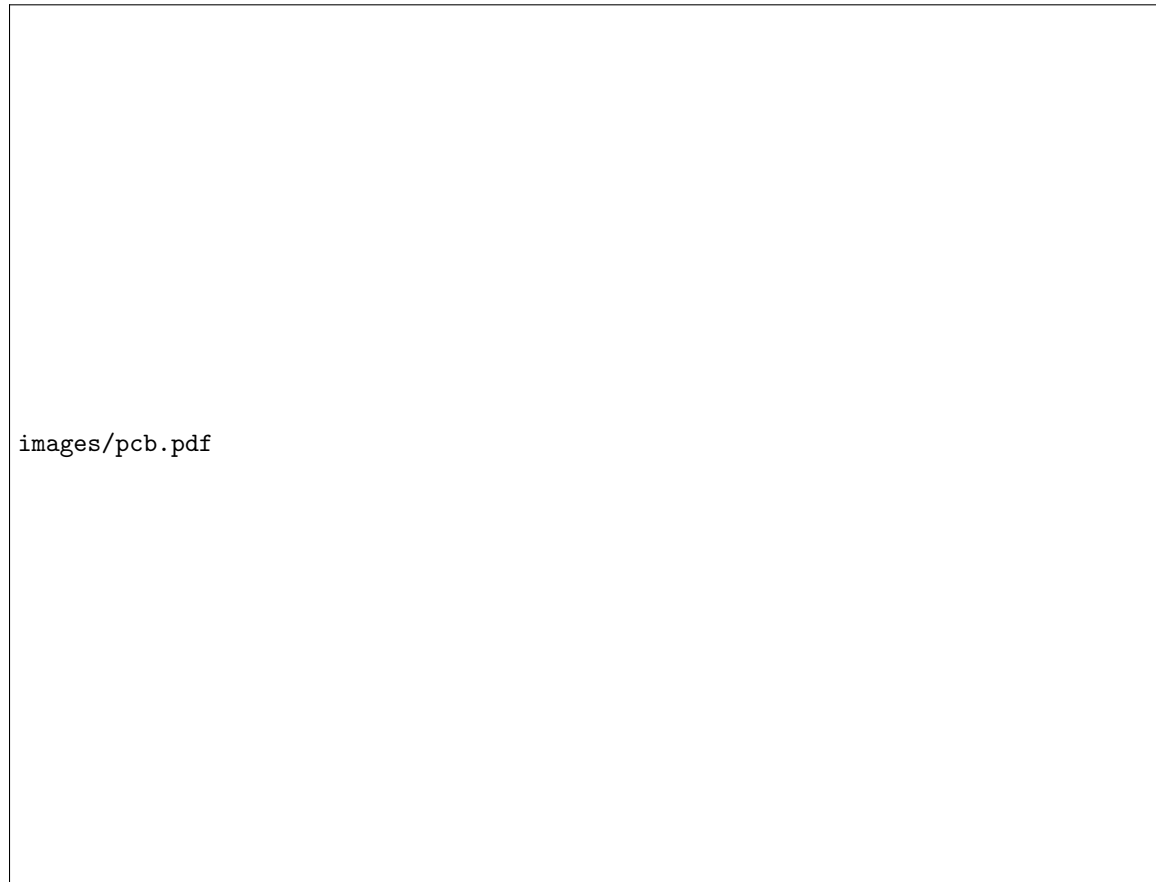


Figure 8-5: The main components of the sequencer hardware.

8.1.1 Ethernet Interface

The sequencer device communicates with a host PC through a standard Ethernet network. By default, it is configured to use a 100baseFX fiberoptic cable with SC connectors, so you will need a converter to the unshielded twisted pair (UTP) that most desktop PCs use. This provides optical isolation for safety and noise immunity, but you can convert the device to use the RJ45 connector and UTP cables directly by soldering in a surface-mount jumper at R90, near the Ethernet controller chip.

8.2 Booting Up

1. Connect the tinned leads of the sequencer power cable to your power supply. These can be separate for each voltage, or all three positive supplies (the red, blue, and yellow wires) can

be connected to the same supply.

2. Connect one end of a SC fiberoptic cable (or a Cat 5 UTP cable) to the appropriate connector on the sequencer (either J11 or J9) and connect the other end to a network with a DHCP server and a host PC containing the bootstrap program.
3. Connect the power cable to the power receptacle on the sequencer (J10). The general-purpose LEDs should light up one-by-one, starting from bit 0 and going to bit 7, while the Ethernet LEDs blink to indicate getting a dynamic IP address over DHCP. The general-purpose LEDs will continue to cycle until a dynamic address is obtained. It will timeout after three attempts, after which it will assume a static address of 169.254.34.34.
4. Run the program `ptpboot` either from the command-line or double-clicking the program icon. A terminal window should open showing if a sequencer device was discovered on the network and software loading progress. If it is successful, the last message will tell you the IP address of the discovered device; if it is unsuccessful, an error message will tell you what problem occurred.
5. Open any web browser and enter the discovered IP address. The sequencer's web interface should load.

8.3 Troubleshooting

Because the sequencer is still in development, you may encounter situations where it becomes unresponsive or “freezes.” You can reset the device by toggling DIP switch 0 on the PCB, after which you must wait for the LEDs to finish cycling and run the bootstrapping program again, as described above. Also, you can help improve the device by sending a bug report to the author, using the e-mail address on the project website. Please describe what you were trying to do when the problem occurred, the environment in which the sequencer was operating, and steps to reproduce the problem if possible.

9 FPGA Programming

The firmware consists of the configurable logic modules that program the FPGA; it is stored in a non-volatile flash programming device. Unlike the hardware, the firmware can be modified by the user in the field by reprogramming the flash device; unlike the software, it is available upon boot and does not have to be loaded separately.

Programming the firmware (burning it into flash memory) is separate from *developing* the firmware itself. Users can program the firmware without any knowledge of its underlying implementation, downloading firmware files from the project website when they are magically generated by the author. Eventually, users will only need to update their firmware infrequently to fix critical bugs. However, in these early stage where all users are essentially developers, all bugs are critical, and all features are requested, firmware updates will happen quite frequently.

Developers who wish to modify the firmware will need some knowledge of VHDL and m4, the languages used to implement the firmware, as well as a Windows development environment running a fully-licensed version of the Altera Quartus software. Maybe I will write a separate guide for firmware design and implementation in the future, but it is currently beyond the scope of this document. For now you will probably just have to e-mail me a lot.

This section is a guide for setting up the necessary hardware and software for both programming and developing the firmware, as well as for the programming process itself. Section 9.1 lists FPGA-specific prerequisites for both users and developers.

9.1 Prerequisites

Both users and developers need a development PC with the following resources:

- A Microsoft operating system (2000/XP recommended) where you have administrator permissions.
- An available parallel port.
- An ordinary parallel printer cable (make sure it is not null-modem or anything weird like that).
- A ByteBlaster II parallel programming cable from Altera or equivalent. I have built an inexpensive work-alike version whose design is also released with the pulse sequencer project on its website. If you contact me, I can send you one for free.

A user who only wants to program the FPGA further needs the following:

- 24.1 MB of free hard drive space for the installation program itself.
- An additional 94 MB of free hard drive space for the actual program.

A developer who wants to modify and build new version of the firmware needs the following:

- 174 MB of free hard drive space for the installation program itself.
- An additional 704 MB of free hard drive space for the actual program.
- A fast processor (1 GHz or faster) and lots of RAM (512 MB or more) is recommended to make the firmware compilation go faster.

9.2 Installation Procedure

9.2.1 User Installation

- Download the *Quartus II Programmer* Version 5.1, Service Pack 1, from Altera Corporation. Unfortunately, you will first need to register with Altera technical support before downloading.⁸

⁸<https://www.altera.com/support/software/download/programming/quartus2/dnl-quartus2-programmer.jsp>

- Run the installation program. Read and agree to the license and the default installation directions for both Quartus and the tutorial files.
- Install the drivers for the ByteBlaster II programming cable, as described in Section 9.3, then return here to continue.
- If you’ve reached this point, the program is installed. You can continue to Section 9.4 to learn how to use this software, along with your programming cable, to burn new firmware into your FPGA or debug existing firmware.

9.2.2 Developer Installation

1. Request a free license from Altera. This will involve signing up for a support account first here, if you do not have one already.

`http://www.altera.com/support/licensing/lic-index.html`

2. Wait for an e-mail from Altera containing your license file as an attachment. Save this to your hard drive.
3. Download and install the Quartus II Web Edition design software from Altera’s website. Be sure to download the version which includes all available service packs. The author has tested the VHDL code with Version 5.0, Service Pack 1, but later versions may also work.⁹
4. Run the installation program. Read and agree to the license and the default installation directions for both Quartus and the tutorial files.
5. Install the drivers for the ByteBlaster II programming cable, as described in Section 9.3, then return here to continue.
6. After installation completes, launch the Quartus II software. If you have a permanent network connection and a firewall running, you must either disable your network connection or allow Quartus II to access the Internet to check for updates the first time.
7. Select **Tools→License Setup** from the menu and choose the license file you downloaded earlier.
8. If you’ve reached this point, the program is installed and runs successfully. You can leave it open and continue following the instructions in Section 9.4, or you can close the program.

9.3 Programming Cable Installation

The following instructions are copied from an Altera webpage:

`http://www.altera.com/support/software/drivers/dri-bb-2k.html`

1. Click on the **Start** menu, choose **Settings**, and click on **Control Panel**.
2. Double-click the **Add/Remove Hardware** icon to start the **Add/Remove Hardware Wizard** and click **Next** to continue.
3. Select the **Add/Troubleshoot a device** radio button in the **Choose a Hardware Task** panel, and click **Next** to continue. Windows 2000 will search for new Plug and Play hardware (**New Hardware Detection** window).
4. Select **Add a new device** in the **Choose a Hardware Device** window, and click **Next** to continue.

⁹https://www.altera.com/support/software/download/altera.design/quartus_we/dnl-quartus_we.jsp

5. Select the “No, I want to select the hardware from a list” radio button in the Find New Hardware window, and click Next to continue. Select “Sound, video and game controllers” in the Hardware Type window, and click Next to continue.
6. Select Have Disk ... in the Select a Device Driver window.
7. Browse to the win2000.inf file in the \drivers\win2000 directory of your Quartus II software or MAX+PLUS II software installation and click OK.
8. Click Yes at the Digital Signature Not Found warning dialog box.
9. Select Altera ByteBlaster II in the Select a Device Driver window, and click Next to continue.
10. Click Next after the Start Hardware Installation window displays the hardware being installed.
11. Click Yes at the Digital Signature Not Found warning dialog box.
12. Click Finish in the Completing the Add/Remove Hardware Wizard window.
13. Reboot the computer.

9.4 Using the Programming Cable

Warning: Do not plug the programming cable into the sequencer board until you have started the programming software. Doing so may cause the programming cable to short power to ground and draw enough current to turn the configuration LED off.

There are two operations, and two corresponding pieces of software which users installed in Section 9.2, that you will want to perform with the programming cable.

1. Programming new firmware into the FPGA. You will do this with the *Quartus II Programmer*. The programming process is described in Subsubsection 9.4.1.
2. Debugging existing firmware on the FPGA. You will do this with the *Quartus SignalTap II Logic Analyzer* which is installed as part of the Quartus II Programmer. The debugging process is described in Subsubsection 9.4.2.

The full Quartus II design software, which developers installed, also includes both the Programmer and the SignalTap Logic Analyzer among its many other design features.

If you experience an error in programming the FPGA device, you may need to restart the software, or remove and add back the programming cable hardware within the software settings. Unfortunately, the process can be flaky even with an official \$150 cable.

9.4.1 Programming New Firmware

1. Download the most recent version of the firmware from the project website. It is under the package `sequencer-firmware`, and the file will have a name like `sequencer-firmware-x.xx.tar.bz2`.
2. Extract this file into a new directory and remember its location.
3. Open the *Quartus II Standalone Programmer* software.
4. Plug the 10-pin socket of the programming cable into the 10-pin header of the sequencer board marked **CONFIG**. If the configuration LED, right next to this header, goes out, immediately unplug the cable, restart the programmer software, and try again.
5. Next to the **Device Hardware** field, click **Setup...**
6. A dialog box should open. You should click **Add Hardware** to add the ByteBlaster II device on printer port LPT1 if it does not already exist.

7. Then select the Byteblaster II option by clicking on it. Then click on the button marked **Select Hardware**, followed by OK to close the dialog box.
8. Choose **Active Serial Programming** as the Mode.
9. Click **Add File...** and navigate to the file **sequencer_top.pof** in the directory where you extracted the firmware files in the first step. Select this file and click OK.
10. Check the box labelled **Program/Configure** next to the filename.
11. Click the button **Start** and wait for the procedure to finish. The configuration LED should go out while the flash memory is being reprogrammed, and a gauge bar will show you the progress in writing the new firmware into flash. Finally, the LED should come back on when programming is finished. You should perform a hardware reset to reinitialize the FPGA memory.

9.4.2 Debugging Firmware


Note: You can only debug a firmware revision that is specially compiled to include the SignalTap II Logic Analyzer, and you need the corresponding SignalTap file (with a filename ending in ***.stp**).

For example, the normal firmware programming file is named **sequencer_top.pof** and does not have SignalTap built-in. There may be some other programming files in the same package with names like **sequencer_top_blah.pof** which has a corresponding SignalTap file **sequencer_top_blah.stp**. If you burned the **.pof** file into your FPGA, then you can use the **.stp** file to debug it.

You may have previously used a hardware device called a logic analyzer to debug digital signals; SignalTap is an *embedded* logic analyzer compiled into the firmware and accessed through the same cable that you use for programming. When SignalTap was compiled into a programming file, the developer selected which internal signals to capture. These signals were then internally routed to a special memory which samples and records the signal changes over time according to some clock. When you run SignalTap, you can read back this memory in a graphical waveform display to inspect the internal operation of the firmware when it is running at full-speed.

You can also trigger SignalTap to wait on a particular signal before running, so you can examine signals only at the time of some rare event. You can also specify no trigger to have SignalTap run immediately; this is useful for determining why the firmware has frozen up. As you can imagine, SignalTap makes debugging most problems very easy, which would otherwise be quite difficult if not impossible to do with LEDs or other debugging methods.

- Download the firmware release from the project website which corresponds to the version programmed in your FPGA. It is under the package **sequencer-firmware**, and the file will have a name like **sequencer-firmware-x.xx.tar.bz2**.
- Extract this file into a new directory and remember its location.
- Run the *Quartus SignalTap II Logic Analyzer* software.
- Plug the 10-pin socket of the programming cable into the 10-pin header of the sequencer board marked **DEBUG**. If any LEDs go out, immediately unplug the cable, restart the programmer software, and try again.
- Choose **File→Open...** and navigate to the file **sequencer_top_blah.stp** in the directory where you extracted the firmware files in the first step, where **blah** is whatever special build you want to debug. Select this file and click OK. You should see a list of signals compiled into this SignalTap build, its trigger, memory size, and other parameters.
- If the **JTAG Chain Configuration** field says “No device is selected” then click the **Setup...** button next to the **Device Hardware** field. Otherwise skip to Step 9.4.2.
- A dialog box should open. You should click **Add Hardware** to add the ByteBlaster II device on printer port LPT1 if it does not already exist.



images/signaltap-data.pdf

Figure 9-6: A screenshot of the SignalTap II logic analyzer software.

- Select the Byteblaster II option by clicking on it. Then click on the button marked **Select Hardware**, followed by OK to close the dialog box.
- If the software does not immediately scan for a connected FPGA, click the **Scan Chain** button. The FPGA on the sequencer board, with model number EP1C12Q240C6, should show up in the **Device** field. If it does not, make sure your sequencer board is powered and that the programmer cable is connected to the **DEBUG** header and your PC's parallel port.
- Click the **Run Analysis** button (with a green right arrow) to start debugging.
- Wait for the trigger event to occur, or cause the trigger event if desired. If the SignalTap file does not have a trigger, the analysis will stop immediately with the data it captured. If you want to stop the analysis at any time, press the **Stop Analysis** button (with a black square).

Note that the **Data Log** windows keeps track of all different sets of signals, trigger settings, and data logs captured. Thus, you save the SignalTap file with the captured data and e-mail it to the developer or a colleague to help with debugging.

10 Testing Sequencer and Device Chains

10.1 Testing a Sequencer

10.2 Testing a Breakout Board

10.3 Testing a Chain Daughterboard

10.4 Testing a DAC Device and Chain Daughterboard

10.5 Testing a DDS Device and Chain Daughterboard

10.6 Testing a VGA Device and Chain Daughterboard

11 Firmware

I can't think of a good name for this section.

11.1 Building the Firmware

To build the firmware, you must have the following software installed on your development machine (running Windows).

- Altera Quartus (see Subsubsection 9.2.2)
- GNU m4
- GNU make
- GNU bash (or some other POSIX shell)

On Windows, I use Cygwin¹⁰ to get the GNU stuff above and to pretend that I'm using a real operating system. Once you meet the prerequisites above, follow these instructions:

1. Download the latest release of the **sequencer-firmware** package from SourceForge.
2. Extract the file **sequencer-firmware-x.xx.tar.bz2** to get a directory with the same base name. In your shell:

```
> tar xvf - sequencer-firmware-x.xx.tar.bz2 | bzip2 -dc
```

3. In your shell, run the following commands:

```
> cd sequencer-firmware-x.xx
> make sequencer_top.vhd
> make sequencer_top.map.eqn
```

4. Open the project **sequencer_top.qpf** in Quartus. Press **Ctrl+L** to completely build design.

At the end, you will have programming files called **sequencer_top.pof** and **sequencer_top.sof**, which you can then program as described in Subsubsection 9.4.1 instead of using downloaded firmware.

¹⁰<http://www.cygwin.com>

12 Writing Software Configuration Settings for a Box

13 Adding Software Devices

14 Running Software Tests on the Finished Box

15 Operating Information

15.1 DIP Switches

15.2 Configuring Clocks

The sequencer firmware can accept two configurable clock sources (`clk0` and `clk2`) which are switched by parts

15.3 Using the Daisy Chain

A Pulse Transfer Protocol

The Pulse Transfer Protocol (PTP) is a network protocol based on UDP for controlling the pulse sequencer board, including starting and stopping the PCP, reading and writing to various memory address spaces, controlling I²C lines, and any other function that requires user interaction. It uses a client-server model where the client may send one or more idempotent requests to the server until it receives a matching reply. The client runs on a control PC, preferably the Python pulse compiler, which abstracts away many low-level networking details about generating and parsing the frames. The server runs on the pulse sequencer board itself as a stack of firmware modules in the FPGA. More details about the implementation of the PTP stack can be found in [1].

The opcodes and their payloads are described below in request/reply pairs. They conform to the format of a PTP frame in Figure A-3. Although the frame length field accommodates 16 bits, all current firmware implementations 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.¹¹ 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.

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

Table A-3: Fields of a Pulse Transfer Protocol frame

Frame	Request Opcode	Reply Opcode	Description
Null	0x00		Does nothing.
Status	0x01	0x11	Returns status.
Memory	0x02	0x12	Performs memory operation.
Start	0x04	0x14	Starts/stops processors.
Trigger	0x05	0x15	Manages PCP instruction memory and firmware triggers.
I ² C	0x07	0x17	Performs operations on I ² C slaves.
Debug	0x08	0x18	Performs debugging operations.
Discover	0x09	0x19	Auto-assigns IDs to discovered device chains.

Table A-4: Summary of PTP opcodes

¹¹That 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).

A.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.

NULL	Request
OPCODE:	0x00
PAYLOAD LENGTH:	0 octets
PAYLOAD DESCRIPTION:	Payload is ignored.

A.2 Status Opcodes

These opcodes requests the status of the addressed sequencer device to be returned by the corresponding reply. It is useful for nonobtrusive pinging of devices on a network or on a chain.

STATUS	Request	Reply		
OPCODE:	0x01	0x11		
PAYLOAD LENGTH:	0 octets	2 octets		
PAYLOAD DESCRIPTION:	Payload is ignored	OCTET	BIT	DESCRIPTION
		1	0-3	4-bit trigger source, uses the values in Table ??.
			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.
		2	0	1 if PCP has halted, 0 otherwise.
				1-7

A.3 Memory Opcodes

PTP can manipulate all memory on the pulse sequencer board as a byte-addressable space. This includes the external SRAM as well as data memory built into the FPGA (for firmware versions that support it).

Firmware versions for the pcp0 (0.05 and prior) cannot write across segment boundaries, where a segment is a 16-bit address space with a 3-bit prefix (to address all 19-bits). Note that it is unsafe to perform a memory write operation into the instruction memory (external SRAM) of the PCP while it is running. The Python compiler stops the PCP before writing and restarts it after writing by default.

MEMORY	Request		Reply	
OPCODE:	0x02		0x12	
PAYLOAD	6 octets for reading		2 octets	
LENGTH:	4 + <i>length</i> octets for writing			
PAYLOAD DESCRIPTION:	OCTET	DESCRIPTION	OCTET	DESCRIPTION
	1	Subopcode for memory operation. 0x01 write external SRAM 0x02 read external SRAM 0x03 write PCP data memory* 0x04 read PCP data memory*	1	Subopcode of request
			2 → + <i>length</i>	Data read from memory.
	2,3,4	Starting address in MSB order		
	5,6	Length for reading in MSB order.		
	5 → + <i>length</i>	Data for writing in MSB order.		

The subopcodes marked with * are only available for pcp2 and later. In the starting address field above, the actual number of bits used depends on the address width of the desired memory. For example, the current external SRAM uses 18 bits, but the PCP data memory will be smaller than that. Therefore, the upper bits of the address will be neglected.

MSB order means that the most significant byte of the starting address has the lowest position in the payload. For example, to read 0x1234 bytes from external SRAM starting at address 0x34567, you would send the following payload:

Subopcode	Starting Address			Read Length	
0x02	0x03	0x45	0x67	0x12	0x34

The maximum length for reading is misleading, as it really can't exceed the length of a maximum Ethernet frame (1500 - 10 byte IP header - 10 byte UDP header - 10 byte PTP header = 1470). The current IP transmitter does not support fragmentation. I'm not really motivated to fix this.

A.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.

START	Request		Reply	
OPCODE:	0x04		0x14	
PAYLOAD LENGTH:	1		1	
PAYLOAD DESCRIPTION:	OCTET	DESCRIPTION	OCTET	DESCRIPTION
	1	Subopcode for start operation. 0x01 starting PCP 0x02 stopping PCP 0x03 starting AVR (deprecated) 0x04 stopping AVR (deprecated)	1	Subopcode of 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.

A.5 Trigger Opcodes

This opcode allows you to initially load the PCP's instruction memory, if it is separate from the external SRAM. It also allows you to start the PCP at arbitrary addresses within instruction memory. Triggers themselves are a deprecated feature used in the PCP64 architecture and pcp0 machine used to set the start condition for the PCP. Modern PCP architectures only start on a PTP start frame. and use branch-on-trigger instructions to achieve the same functionality. Note that PCP32 would respond to a trigger opcode, but it would have no effect on the starting address, which would always be 0x00.

In PCP64, 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.

TRIGGER	Request		Reply	
OPCODE:	0x05		0x15	
PAYLOAD LENGTH:	6		1	
PAYLOAD DESCRIPTION:	OCTET	DESCRIPTION	OCTET	DESCRIPTION
	1	Trigger subopcode (deprecated).	1	Trigger subopcode of request
	2,3,4	Starting address in SRAM		
	5,6	Program length for loading		

A.6 I²C Opcodes

These opcodes define requests for I²C operations and their corresponding replies. The PTP server can address slaves with 7-bit addresses on the I²C bus to read and write data. Either read or write lengths or both can be zero. This command should never stall because the firmware I²C controller, as a bus master, will simply read back zeros if a slave never responds to a given address.

I ² C	Request		Reply	
OPCODE:	0x07		0x17	
PAYLOAD	3 for reading		1 + <i>readlen</i> for reading	
LENGTH:	3 + <i>writelen</i> for writing		1 for writing	
PAYLOAD DESCRIPTION:	OCTET	DESCRIPTION	OCTET	DESCRIPTION
	1	Slave address (lower 7 bits).	1	Slave address of request
	2,3	<i>readlen</i> in MSB order	2 → + <i>readlen</i>	read data from slave
	4 → + <i>writelen</i>	Data for writing to slave.		

Because most I²C slaves require an initial write (e.g. of an internal memory address), the I²C Request always performs a write first using all bytes in the payload beyond the read length (*length*) field.

A.7 Debug Opcodes

These opcodes define requests for the PTP server to perform a debugging operation and the corresponding replies upon completion. They are currently deprecated; in development builds it was used to blink LEDs before the advent of SignalTap debugging.

DEBUG	Request		Reply	
OPCODE:	0x08		0x18	
PAYLOAD	2		2	
LENGTH:				
PAYLOAD DESCRIPTION:	OCTET	DESCRIPTION	OCTET	DESCRIPTION
	1	Subopcode. 0x01 blink LEDs (deprecated)	1	Subopcode of request.
	2	Suboperand	2	Returned debugging data

A.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 0x02. For historical reasons, the chain address 0x01 was reserved for the now defunct AVR processor on the chain initiator. The chain initiator can only be discovered once until it is reset.

DISCOVER	Request		Reply	
OPCODE:	0x09		0x19	
PAYLOAD	1		1	
LENGTH:				
PAYLOAD DESCRIPTION:	OCTET	DESCRIPTION	OCTET	DESCRIPTION
	1	First slave address in chain.	1	First slave address from request.

B PCP32/16 Programming Reference

PCP32/16 is the top secret codename of the third firmware architecture for the pulse programmer. It currently contains only one machine, the `pcp3`. The `pcp2` was a more modest extension of the PCP32 architecture with data memory that was implemented but did not meet several important experimental needs. PCP32/16 is so named because its instruction words are 32 bits long but its data constants and general-purpose (GP) register file are 16 bits long. It has a separate instruction memory and a true data memory (a Harvard architecture), both of which contain *segments* which are loaded from the larger external SRAM. This is similar to PCP64 and allows us to clock the processor at an optimal 125 MHz (divided down from the DDS 1 GHz sampling clock) and also write back instructions for runtime, self-modifying code generation.

The architecture also supports 16-bit integer arithmetic, storing and loading registers from data memory, reading and writing both data and instruction segments from external SRAM, branching on register equality, firmware triggers, dedicated pulse outputs for Analog Devices evaluation boards and TTL devices, and logical bit manipulation. Both the instruction and data memory are double-clocked to synchronize between the slower PTP clock and the faster PCP clock.

PCP32/16 makes extensive use of *preloading*, which is a term that I just made up. It means that to compensate for the one cycle delay in latching an address into the GP register file and reading out the value, each instruction contains dedicated operands for register addresses to be used by the *next* instruction. Whenever an instruction makes use of such a preloaded operand, it is listed in curly braces as “`{rs1}`”, “`{rs2}`”, or “`{ria}`”. Not all instructions are compatible with preloading, and this is specified in Table B-9.

PCP32/16 eliminates the overly general pulse outputs of earlier architectures in favor of more optimal outputs for current experimental setups at Innsbruck and MPQ. It also provides better isolation between output devices, improves update speeds of DACs and DDSs, and allows TTL devices to hold their values across subroutine calls and branch points. It retains the existing functionality of PCP32, such as phase-coherent frequency switching, nested subroutine calls, and nested finite looping. The motivation behind its design was to allow data capturing by incrementing a counter register based on pulses of feedback inputs and to dynamically generate new pulse instructions are run-time to quickly scan arbitrary amplitude shapes or frequency patterns.

B.1 Architectural Parameters

The PCP32/16 family has parameters which are either machine-dependent or fixed for all machines in the family. The fixed parameters are shown in Table B-5 and the machine-dependent parameters are shown in Table B-6 along with the allowed ranges the specific values for the only machine in this family, the `pcp3`.

Parameter	Value
Instruction Width	32 bits (4 bytes)
Data Width	16 bits (2 bytes)
Maximum Stack Depth	8 nested subroutine calls
Phase Register Data Width	32 bits (to match AD9858)
Phase Register Adjust Width	14 bits (to match AD9858)
Phase Register Address Width	4 bits
Phase Register Count	16 registers
GP Register Counter	15 registers
GP Register Address Width	4 bits
GP Register Data Width	16 bits
Timer Width	27 bits
Maximum Timer Value	1.07 seconds (for an 8 ns cycle)
Minimum Timer Value	4 cycles
Branch Delay Slots	2 cycles

Table B-5: Fixed parameters for the PCP32/16 architecture.

Parameter	Maximum Value	<code>pcp3</code> Values	Units
Data Memory Address Width	16	10	bits
Maximum Data Segment Size	65,536	1,024	data words
Instruction Memory Address Width	16	12	bits
Maximum Program Segment Size	65,535	4,096	instruction words
SRAM Address Width	19	18	bits
Maximum Total SRAM Size	524,288	262,144	instruction words

Table B-6: Machine parameters for the `pcp1` in the PCP32 architecture.

Most PCP32/16 instructions have a two stage fetch and decode cycle for pipelining simplicity. This is only relevant for instructions which modify registers, whose results may not be written back and read immediately after the decoding stage. Because PCP32/16 fetches instructions from a built-in instruction memory, it only has 2 branch delay slots.

The GP register file contains a constant zero register, `r0`, which can be used as a source for setting other registers to 0 and as a sink for writing values that should be discarded (e.g. overflow bits).

B.3 Instruction Format

For each instruction, the binary format summary follows the example of Table B-7. The mnemonic name and sample operand usage is given as short, convenient examples for developers, but there is no actual assembly language or assembler tool. Also given are: the hexadecimal value of the opcode; the bit widths, names, and locations of each operand; and the result of each instruction as a modification of processor state. The notation and abbreviations used in the remainder of this subsection are explained in Table B-8.

mnemonic	Description of instruction.																																								
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00									
5 bits					3 bits			4 bits				4 bits				16 bits																									
opcode					constant																																				
0×hh					Result:		result ₁																		result ₂																
					Usage:		mnemonic operand ₁ , operand ₂ ,...																																		

Table B-7: PCP32/16 example instruction format.

Symbol	Description
opcode	5-bit value determining which operation to perform.
pc	Program counter (same width as machine address space).
halt	TRUE when instruction fetching has halted. FALSE initially.
stack	Address stack used for nested subroutine calls.
pulse	64-bit pulse output register.
timer	28-bit timer which halts processor until it reaches zero.
fb	8-bit hardware feedback inputs.
ffb	3-bit firmware feedback inputs.
[r]	Takes the value stored in GP register <i>r</i> .
[r] _{C,s}	Takes the value stored in a trigger counter <i>r</i> . <i>s</i> is a single bit indicating a 16-bit half of the full 32-bit counter.
[r] _{P,s}	Takes the value stored in phase-coherent frequency counter <i>r</i> . <i>s</i> is a single bit indicating a 16-bit half of the full 32-bit counter.
[addr] _D	Takes the value stored at address <i>addr</i> in data memory
[addr] _I	Takes the value stored at address <i>addr</i> in insn memory.
[addr] _S	Takes the value stored at address <i>addr</i> in external SRAM.
([addr1] → [addr2]) ← ([addr3] → [addr4])	The contents of memory from <i>addr3</i> to <i>addr4</i> get stored from addresses <i>addr1</i> to <i>addr2</i> .
eq	Flag which is TRUE when <i>cmp</i> finds two GP registers equal.
gt	Flag which is TRUE when <i>cmp</i> finds one GP register greater than another.
of[ctr]	Flags which are TRUE when the corresponding counter <i>ctr</i> overflows its 32-bit value.
preg	The intermediate 32-bit register for loading phase-coherent frequency counters.
rc	The current phase-coherent frequency counter.

Table B-8: Notation for the PCP32/16 instruction set.

B.4 Instruction Set

Insn	Opcode	Operands	Description	Pre
nop	0x00		Does nothing.	Y
halt	0x01		Halts instruction fetching.	Y
wait	0x02	const	Waits for a 28-bit delay.	N
btr	0x03	ftr, tr, addr	Branch to an immediate address on matching input trigger.	N
j	0x04	addr	Jump to an immediate address.	Y
mv	0x05	rd, {rs2}	Copy contents of one register to another.	rs2
call	0x06	addr	Call a subroutine at an immediate address.	Y
ret	0x07		Return from a subroutine call.	Y
cmp	0x08	{rs1}, {rs2}	Compares if two registers and sets flags.	Y
bf	0x09	m, addr	Branches to an immediate address depending on flags.	Y
lcr	0x0A	s, rd1, rd2, ctr	Loads trigger counter into GP register.	N
scr	0x0B	s, rs1, ctr	Stores GP register in trigger counter.	Y
ldr	0x0C	rd, {ria}	Load a GP register from data memory.	rs2
str	0x0D	{rs1}, {ria}	Stores a GP register to data memory.	Y
rd	0x0E	{ria}, {ril}, addr	Reads segment from SRAM to insn memory.	Y
wr	0x0F	{ria}, {ril}, addr	Writes segment from data memory to SRAM.	Y
add	0x10	{rs1}, {rs2}, rd1, rd2	Adds two GP registers.	N
	0x11			
mul	0x12	{rs1}, {rs2}, rd1, rd2	Multiplies two GP registers.	N
div	0x13	{rs1}, {rs2}, rd2	Divides (integrally) two GP registers.	rs1
pp	0x14	s, u, w, const	Pulses out a phase-coherent frequency counter.	Y
lp	0x15	s, c, w, rp, const	Loads a phase-coherent frequency counter.	rs2
pdac	0x16	u, data	Pulses data and update to the DAC chain.	Y
	0x17			
pdds	0x18	u, w, addr, data	Pulses data, address, update to the DDS chain.	N
pddsp	0x19	e, p1, p0	Pulses the profile select and enable for the DDS chain.	Y
pttlh	0x1A	m, const	Pulses levels to upper 16 TTL outputs.	Y
pttll	0x1B	m, const	Pulses levels to lower 16 TTL outputs.	Y
pchain	0x1C	m, ca	Sets the current device for either DAC or DDS chains.	Y
inc	0x1D	rd	Increments the given GP register by one.	rs2
	0x1E			
ldc	0x1F	rd, const	Loads a constant into a GP register.	rs2

Table B-9: PCP32/16 instruction set summary.

B.4.1 Nop Instruction

nop					Null instruction.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00					
5 bits					27 bits																															
opcode																																				
0x00					Result:		Nothing															and more nothing														
					Usage:		nop																													

B.4.2 Halt Instruction

halt					Halts instruction fetching																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00					
5 bits					27 bits																															
opcode																																				
0x01					Result:		halt ← TRUE																													
					Usage:		halt																													

B.4.3 Wait Instruction

halt				Waits for a constant 27-bit delay.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits					27 bits																														
opcode					const																														
0x02				Result:		timer ← const															halt ← TRUE until (timer = 0)														
				Usage:		wait const																													

B.4.4 Branch-On-Trigger Instruction

btr				Branches to a target address on a trigger input.																																			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00								
5 bits				6 bits				8 bits								16 bits																							
opcode				ftr				tr								addr																							
0x03				Result:				pc ← addr																if ((fb ∧ tr) ∨ (ffb ∧ ftr)) ≠ 0															
				Usage:				btr ftr, tr, addr																															

PCP32/16 can detect triggers either in hardware inputs (**fb**) or firmware inputs (**ffb**), and the trigger masks for branching are specified in **tr** and **ftr** respectively. The branch will be taken if any of the high bits in the masks are detected in either hardware or firmware, performing a logical OR of the specified inputs.

B.4.5 Jump Instruction

j				Branches to a target address unconditionally.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits				11 bits												16 bits																			
opcode																addr																			
0x04				Result:		pc ← addr																													
				Usage:		j addr																													

B.4.6 Move Instruction

[illegible]

`mv` can be used to reset a GP register to 0 by using `r0` as the source.

B.4.7 Subroutine-Call Instruction

call		Calls a subroutine and pushes return address onto stack.																																			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00						
5 bits				11 bits												16 bits																					
opcode																addr																					
0x06				Result:		pc ← addr																stack ← stack.push(pc)															
				Usage:		call addr																															

B.4.8 Subroutine-Return Instruction

ret				Returns from a subroutine by popping return address from stack.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits				27 bits																															
opcode																																			
0x07				Result:		pc ← stack.top()														stack ← stack.pop()															
				Usage:		ret																													

B.4.9 Compare Instruction

cmp				Compares the values of two GP registers.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits				27 bits																															
opcode																																			
0x08				Result:		eq ← ([rs1] = [rs2])																gt ← ([rs1] > [rs2])													
				Usage:		cmp {rs1}, {rs2}																													

cmp sets the equal flag and greater-than flag in a single cycle for the specified source registers. All flags initially start out as FALSE when the processor is reset. **cmp** can also perform the less-than operation by switching the two source registers **rs1** and **rs2**. See **bf** to perform the operations for greater-than-or-equals and less-than-or-equals.

B.4.10 Branch-On-Flags Instruction

bf		Branch on processor flags.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00		
5 bits					8 bits											16 bits																	
opcode				m												addr																	
0x09				Result:		pc ← addr															if ((eq ∧ ¬m) ∨ (gt ∧ m))												
				Usage:		bf m, addr																											

bf branches the pc based on its mode bit m and the value of flags eq and gt previously set by cmp. m = 0 is the equal mode, and m = 1 is the greater-than mode. Normally bf is followed by two branch delay slots.

However, you can branch on the result of a greater-than-or-equals comparison by first branching on equals, then branching on greater-than, then inserting the branch delay slots. The first two branches can be interchanged.

B.4.11 Load-Counter Instruction

lcr				Loads a trigger counter into a GP register.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits					2 bits		4 bits				4 bits				13 bits													3 bits							
opcode					s	rd1				rd2																	ctr								
0x0A				Result:		[rd1] ← [ctr] _{C,s}																[rd2] ← of[ctr]													
				Usage:		lcr s, rd1, rd2, ctr																													

Each trigger counter ctr has an associated overflow flag of[ctr] which is set to TRUE when the counter reaches the maximum 32-bit value and wraps around to zero. This overflow flag is loaded in the GP register specified by rd2 and can be used to take conditional action with cmp and bf. If the overflow flag should be discarded, set rd2 = r0.

B.4.12 Store-Counter Instruction

scr		Stores a CP register into a trigger counter.																																						
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00									
5 bits						2 bits		4 bits				37 bits																									3 bits			
opcode						s		rs1																													ctr			
0x0B				Result:			[ctr] _{C,s} ← [rs1]																																	
				Usage:			scr s, rs1, ctr																																	

scr is primarily useful for resetting trigger counters to zero (rs1 = r0). Note that you must write to both 16-bit halves of the 32-bit counter (s = 0 and s = 1) to properly reset it to zero, and this should be done before most counter operations to get a calibrated value.

Each trigger counter ctr has an associated overflow flag of[ctr] which is set to FALSE when the processor is reset and whenever scr is decoded.

B.4.13 Load-Register Instruction

ldr		Loads a GP register from data memory.																																					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00								
5 bits				3 bits			4 bits				4 bits				16 bits																								
opcode							rd				{ria}				addr																								
0x0C				Result:		[rd] ← [[ria]] _D																																	
				Usage:		ldr rd, {ria}																																	

B.4.14 Store-Register Instruction

str		Stores a GP register value to data memory.																																		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00					
5 bits				3 bits			4 bits				4 bits				16 bits																					
opcode							{rs1}				{ria}				addr																					
0x0D			Result:		[[ria]] _D ← [rs1]																															
			Usage:		str {rs1}, {ria}																															

B.4.15 Read-Segment Instruction

rd				Reads a segment from external SRAM to instruction memory.																																																																																																																											
31				30				29				28				27				26				25				24				23				22				21				20				19				18				17				16				15				14				13				12				11				10				09				08				07				06				05				04				01				02				01				00			
5 bits								3 bits								4 bits								4 bits								16 bits																																																																																															
opcode								addr								{rs1}								{rs2}								addr																																																																																															
0x0E				Result:				([rs1] _I → [rs1] + [rs2] _I) ←																([addr] _S → [addr + [rs2] _S)																																																																																																							
				Usage:				rd {rs1}, {rs2}, addr																																																																																																																							

Note that rs2 in this case contains the read length in instruction memory. This is the same as the read length in external SRAM, because both memories contain 32-bit words.

B.4.16 Write-Segment Instruction

wr				Writes a segment from data memory to external SRAM.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits				3 bits			4 bits				4 bits				16 bits																				
opcode				addr			{rs1}				{rs2}				addr																				
0x0F				Result:		([addr] _S → [addr + [rs2/2]] _S) ← ([rs1] _D − [[rs1] + [rs2]] _D)																													
				Usage:		wr {rs1}, {rs2}, addr																													

Note that rs2 in this case contains the read length in data memory, which consists of 16-bit words. That is why the read length is divided by two in the destination, because the external SRAM contains 32-bit words.

B.4.17 Add Instruction

add		Adds the value of two GP registers.																																	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits				3 bits			4 bits				4 bits				16 bits																				
opcode				addr			{rs1}/rd1				{rs2}/rd2																								
0x10				Result:		[rd1], [rd2] ← [rs1] + [rs2]																													
				Usage:		add {rs1}, {rs2}, rd1, rd2																													

The sum of any two 16-bit values is at most a 17-bit value due to the high-order carry bit. After decoding this instruction, rd1 will contain the carry bit and rd2 will contain the lower 16 bits. Any combination of registers is valid except that rd1 and rd2 must be different registers. A common case is when the sum of registers is written back in-place ($rd1 = rs1$ and $rd2 = rs2$).

B.4.18 Multiply Instruction

mul		Multiplies the value of two GP registers.																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00
5 bits				3 bits			4 bits				4 bits				16 bits																
opcode				addr			{rs1}/rd1				{rs2}/rd2																				
0x12				Result:		[rd1], [rd2] ← [rs1] × [rs2]																									
				Usage:		mul {rs1}, {rs2}, rd1, rd2																									

B.4.19 Divide Instruction

The integral quotient of any two 16-bit values is at most a 16-bit value, with division by 0 being undefined. After decoding this instruction, `rd2` will contain the quotient of `rs1` and `rs2`. Any combination of registers is valid. A common case is when the quotient of registers is written back in-place (`rd2 = rs2`).

pp		Pulse out a phase-coherent frequency counter.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00		
5 bits								18 bits																		6 bits							
opcode				s	u	w																				const							
0x14				Result:				pulse ₃₁₋₂₄ ← [rc] _{P,S} , pulse ₄₈ ← u												pulse ₁₇ ← w, pulse ₂₃₋₁₈ ← const													
				Usage:				pp s, u, w, const																									

lp		Load a phase-coherent frequency counter.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00		
5 bits							4 bits								16 bits																		
opcode				s	w	c	rp								const																		
0x15		Result:		[rp] ← preg if w, preg _s ← const														rc ← rp if c															
		Usage:		lp s, c, w, rp, const																													

pdac		Pulses a gain level and update clock to DAC chain.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00		
5 bits																		14 bits															
opcode						u												data															
0x16			Result:		pulse ₁₅₋₀₂ ← data													pulse ₀₁ ← u															
			Usage:		pdac u, data																												

pdds				Pulses address, data, write, and update to DDS chain.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits								6 bits												8 bits															
opcode						u	w	addr												data															
0x18				Result:				pulse _{31–24} ← data, pulse ₄₈ ← u												pulse ₁₇ ← w, pulse _{23–18} ← addr															
				Usage:				pdds u, w, addr, data																											

B.4.24 Pulse-DDS-Profile Instruction

pddsp				Pulses profile select and enable bits to DDS chain.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits																																			
opcode					e	p1	p0																												
0x19				Result:				pulse ₃₂ ← e,												pulse ₄₉ ← p1, pulse ₅₀ ← p0															
				Usage:				pddsp e, p1, p0																											

B.4.25 Pulse-TTL-Higher Instruction

pttlh		Pulses levels to upper 16 TTL outputs.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00		
5 bits																16 bits																	
opcode						m										const																	
0x1A				Result:			pulse _{00,59–55} ← ∨const if (m = 1)											pulse _{00,59–55} ← ∧¬const if (m = 0)															
				Usage:			pttlh m, const																										

B.4.26 Pulse-TTL-Lower Instruction

pttl				Pulses levels to lower 16 TTL outputs.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits																16 bits																			
opcode						m										const																			
0x1B				Result:			pulse _{47–16} ← ∨const if (m = 1)											pulse _{47–16} ← ∧¬const if (m = 0)																	
				Usage:			pttl m, const																												

B.4.27 Pulse-Chain-Address Instruction

pchain	Pulses chain address to either DAC or DDS chains.																																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00		
5 bits																													4 bits				
opcode				s																									ca				
0x1C				Result:		pulse _{63–60} ← ca if (s = 1)														pulse _{54–51} ← ca if (s = 0)													
				Usage:		pchain s, ca																											

B.4.28 Increment Instruction

inc				Increment a GP register.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits												4 bits																							
opcode								rd/{rs1}																											
0x1D				Result:				[rd] ← [rs1] + 1																											
				Usage:				inc {rs1}, rd																											

B.4.29 Load-Constant Instruction

ldc				Load a constant into a register.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	01	02	01	00				
5 bits								4 bits																											
opcode								rd				const																							
0x1F				Result:				[rd] ← const																											
				Usage:				ldc rd, const																											

References

- [1] Paul T. Pham. A general-purpose pulse sequencer for quantum computing. Master's thesis, Massachusetts Institute of Technology, January 2005.