OpenPiton FPGA Prototype Manual

Wentzlaff Parallel Research Group

Princeton University

openpiton@princeton.edu

Version 3.0

Revision History

Revision	Date	Author(s)	Description
1.0	06/30/15	AL	Initial version
2.0	2/29/16	AL	Added support VC707, Genesys2 and NexysVideo development boards
2.1	4/2/16	MM,AL	SD boot image generation. Porting to another boards. Debugging and Simulation.
2.5	10/19/16	AL	Described UART DMW. Removed BRAM OBP. New location of FPGA specific files.
3.0	3/22/17	KL, AL	Added details on network support and updated figures according to new chipset design. Updated boards' frequencies. Decribed oled, port options for tools.
3.1	6/9/17	JB	Simplified ramdisk generation to remove bespoke script. Added recommendation to use our images rather than creating a new one.

Contents

List of Figures

List of Tables

Example	Description
$DV_ROOT/tools/src/proto	*Italic text* is used to indicate paths to scripts and folders
`source add_files.tcl`	`Courier` font is used for commands, scripts' names, IP core names and file names
`NEXYSVIDEO_BOARD`	Text in all capital `COURIER` font is used for defines
Tcl Console	Capitalized text is used for menu options
Note:	Any **text in bold** is used to highlight a special topic or particular option

Table 1: Notation conventions

1 Preface

1.1 Notation Conventions

Text conventions used in the manual are specified in Table 1.

1.2 Supported Tools and Environment Set Up

1.2.1 SW Requirements

- In order to be able to run scripts, *$DV_ROOT* and *$MODEL_DIR* environment variables should be defined. They can be set by moving to the topmost directory of OpenPiton framework folder, and executing the following commands:

  ```
  export PITON_ROOT=$PWD
  source piton/piton_settings.bash
  ```

- To be able to run FPGA implementation of OpenPiton FPGA prototype, Xilinx Vivado tool should be installed on a development machine. For this release Vivado 2015.4 version has been tested and is supported. You can check Vivado version by running
  ```
  vivado -version
  ```

1

- For loading tests from a PC through UART on FPGA the next requirements should be met:

  1. Python used on PC should have `pyserial` library installed.

  2. Depending on the OS used, it could be required to recompile `configsrch` script (sources located in *$DV_ROOT/tools/src/proto/configsrch*). Run
     ```
     make; make install
     ```
     to compile and install `configsrch` for your system. In addition to the above step, you could have to install required Perl packages used by the tools. In Ubuntu you can run `perlpkg.sh` script located in *$DV_ROOT/tools/src/proto* to do this. For other systems, list of additional packages can be taken from the script and installed according to the system.

  3. Support for 32-bit programs should be enabled and libraries should be installed. On Ubuntu, run the next commands:
     ```
     dpkg -add-architecture i386
     apt-get update
     apt-get install libc6:i386 libncurses5:i386
     libstdc++6:i386
     apt-get clean && apt-get update && apt-get
     upgrade
     apt-get install -f
     ```

  4. Install the 32-bit version of gmp. On Ubuntu run:
     ```
     wget http://mirrors.kernel.org/ubuntu/pool/universe/
     g/gmp4/libgmp3c2_4.3.2+dfsg-2ubuntu1_i386.deb;
     dpkg -i libgmp3c2_4.3.2+dfsg-2ubuntu1_i386.deb
     ```

  All the above steps can be run on Ubuntu using *$DV_ROOT/tools/src/proto/syscfg.sh* script. This script can also be used as a reference for other OSes.

- Current implementation of processor memory on BRAMs implies that some tests should be run in SW simulator first. The Synopsys VCS program has to be installed (see Simulation Manual Section 2.5.2 Verilog Simulator for more details).

1.2.2   HW Requirements

You need one of a supported development boards, listed in Table 2. If configuration implies usage of an SD card, you will need one micro SD. Because of SD controller limitations, only micro SDSC cards are supported (**not SDHC**).

1.2.3   Job Queue Managers

SLURM (Simple Linux Utility for Resource Management) is optional `protosyn` script supports using it via `--slurm` command option. Currently, SLURM version 15.08.8 has been tested with.

Development Board, FPGA name, Part Number	Core Clock (1 core)	Max # Cores	DDR type, Size, Data width
Xilinx VC707 Virtex-7 XC7VX485T-2FFG1761C	60 MHz	3	DDR3 1GB 64 bits
Digilent Genesys2 Kintex-7 XC7K325T-2FFG900C	66.667 MHz	2	DDR3 1GB 32 bits
Digilent NexysVideo Artix-7 XC7A200T-1SBG484C	30 MHz	1	DDR3 512MB 16 bits

Table 2: Supported development boards and their parameters

2 Introduction

High speed and relatively low price of FPGAs make them the most common choice for hardware/software interface verification of a processor design. However, design bring-up time, limited capacity of logic cells and width of external interfaces can introduce additional challenges during prototyping step of a project flow [1].

In order to speed up prototyping of a processor and enable a flexible framework for hardware/software interface verification, the FPGA prototype for OpenPiton processor [2] is publicly released. It targets several development boards with Xilinx FPGAs. The release includes all scripts required to implement OpenPiton from Verilog sources down to bit files to program an FPGA. In addition, all the changes applied to a design to enable FPGA synthesis are described, so it can be easily ported to other development boards and FPGAs.

Because of limited FPGA capacity at most three processor tiles fit on one VC707 development board [3]. For all currently supported development boards, maximum number of cores fitting on them, and core clock frequencies see Table 2.
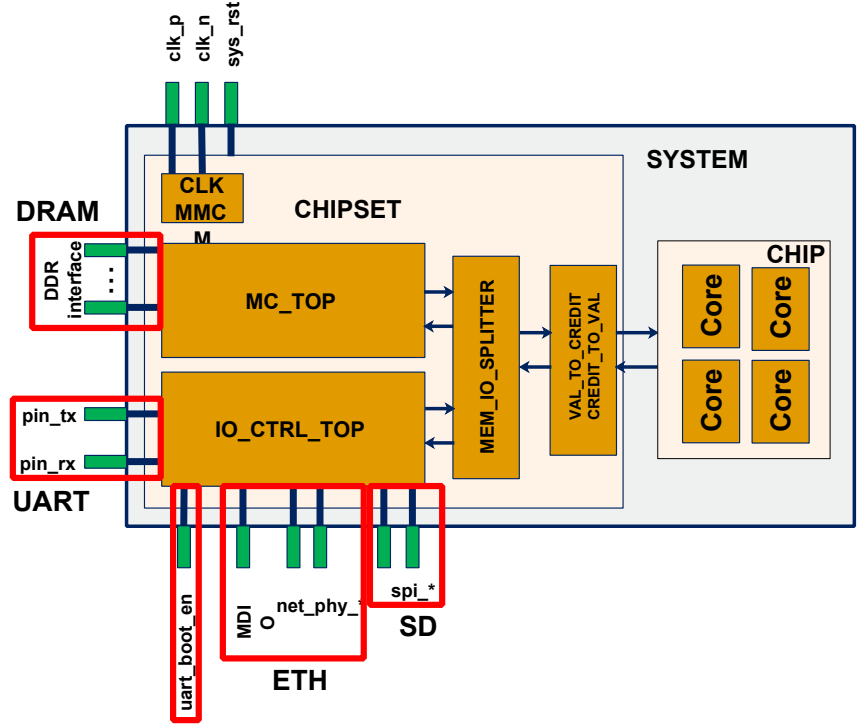
Figure 1: Top level prototype architecture with a default configuration. FPGA pins are shown as green rectangles and are connected to logical signals at `system` module.

3 Prototype Architecture

3.1 Top Level Architecture

Top module of the FPGA prototype design is named `system` (located in `$DC_ROOT/design/rtl`). As shown in Figure 1, it can be divided into the `chip` and `chipset` logical blocks with respective submodule names. In turn, `chipset` is composed out of the next submodules:

- `val_to_credit/credit_to_val` modules - converts off-chip val/rdy NOC interface to a credit based interface. Since the original chip design has a credit based interface, another converter from a credit based to val/rdy interface is added for compatibility.

- `mem_io_splitter` - module for splitting memory requests

from a chip to DDR memory and IO controller[1].

- `io_ctrl_top` - top module for external IO controllers (UART and SD card* by this moment). It has an internal splitter to direct memory requests to a correct IO device and arbitrate between responses[2].

- `mc_top` - a top level for a memory controller. Because of the application side of the MIG 7 IP core working at a higher frequency than the chip and having an interface different from NOC's val/rdy, asynchronous FIFOs and packet transceivers are required. Width of DDR data bus depends on a board type (see Table 2), therefore memory controller was designed to have a configurable application data interface to MIG 7 IP core. As a result, the number of cycles required to read/write one cache line varies from board to board.

- `clk_mmcm` - Xilinx's IP core for clock generation.

Pins shown in Figure 1 as green rectangles are self-explanatory. Input pin `uart_boot_en` is connected to the last SW7 on each FPGA board, which is used to control whether if the system is waiting for a test to be loaded through UART or starts booting Linux from SD card. Depending on the board type, `clk_n`/`clk_p` pin pair can be a single ended clock or not.

3.2   IO_CTRL_TOP Architecture

Architecture of `io_ctrl_top` module is shown in Figure 2. It uses NOC1, NOC2 and NOC3 interfaces for communication with the chip, SPI interface to communicate with SD card (not shown on the Figure), `uart_tx`/`uart_rx` interface to communicate with a host PC through UART, and additional control signals to check test completion conditions (not shown). Its subblocks are described below:

- `iob_splitter` - module which splits requests from cores between I/O devices and `ciop_iob`

---

[1] This is not the same as splitting the IO space and main memory. See precise description of its operation for different project configurations in Section 5.4

[2] The presence of an SD card controller depends on project configuration. See Section 5.1

- `ciop_iob` - module which sends interrupts to cores through NOC2. Among other interrupts, the module send a wake-up packet to a core (see reset sequence in Section 6.1). This module is also responsible for sending network interrupt packets to the core when they are enabled

- `iodev_storage_splitter` - splits memory requests between I/O devices and chipset storage memory spaces.

- `uart_eth_splitter` - splits memory requests between UART and Eth memory spaces. UART memory space should be the same as configured in the Hypervisor or assembly test.

- `noc_axilite_bridge, uart_top` - bridge and top level UART module. `uart_top` contains logic processing bit stream from PC when loading a test into DDR and sends control sequences after each test completion.

- `noc_axilite_bridge, mac_eth_axi_lite` - bridge and Ethernet Lite MAC IP core. There is also an external converter from RGMII interface of IP core to MII interface of PHY chip on Genesys2 and NexysVideo boards.

- `fake_boot_ctrl` - module with a BRAM containing an assembly test in case of `protosyn` was run with `--bram-test` option.

- `noc_sd_bridge, axi_sd_bridge, spi_master` - modules composing SD controller and its interface.
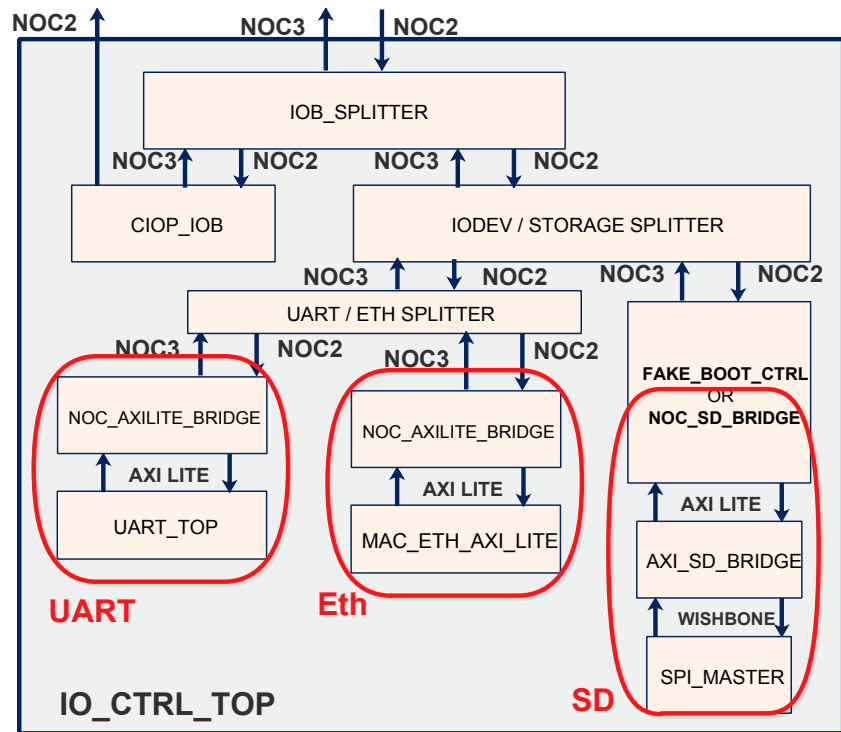
Figure 2: Architecture of the io_ctrl_top module. Labels for arrows represent interface and NOC types.

4   Mapping of a Processor to FPGA

In order to port processor design on FPGA, its design should be transformed to an identical synthesizable form for a chosen FPGA. On top of that, infrastructure used for simulation should be converted to a synthesizable format as well to allow running of assembly tests and booting OS on a prototype. These changes require removing and/or replacing all of the chip specific modules (like RAMs, phy levels, etc.) with their equivalents supported by FPGA tools.

The following IP blocks were created specifically for prototyping OpenPiton:

- Block memories to emulate on-chip SRAMs

- Main Memory controller (MIG7 Xilinx IP core)

- Asynchronous FIFOs for inter-clock domain transfers

- Generator of system clocks

- UART16550 - serial interface for communication with host PC

- MAC for communication with Ethernet PHY chip on Genesys2 and NexysVideo boards

In addition to IP blocks, we designed infrastructure modules which replace and/or extend capabilities of software simulation infrastructure:

- Main Memory/IO and UART/BOOT splitters

- Converter between NOC interface and Memory controller interface with configurable data width

- Converter between NOC interface and SD controller

- A wrapper around UART 16550 IP core for enabling Direct Memory Writes from a host computer

- Multiplexers for choosing a master module for driving Memory Controller

- Emulation of assembly test memory with BRAMs and corresponding controllers

- A wrapper around SD card controller

For a complete schematic of FPGA design see Figure 1. For defines used to configure a project and provided scripts options see Section 5.1.

4.1 Mapping of Processor's SRAMs

OpenPiton has several on-chip SRAMs which are implemented as register arrays for simulation. Some of these memories have bit-enable write masks. Because of IP cores for Xilinx FPGA BRAMs can have only byte-enable write mask, we designed wrappers which implement bit-enable write mask functionality on top of available BRAMs.

SRAMs which are implemented as block memories are shown on Figure 3. All of them except a memory for L2 states (`2p_256_176` on Figure 3) are simple dual port memories with one read and one write port. For a write request we first read the content of memory block, perform a bitwise operation based on a mask and write data back to memory in the next cycle. Since original SRAMs have only one port (with selection of operation type), it guarantees that there is no read conflict when two memory blocks are read at the same cycle. The only case which requires a special attention is writing and reading of the same block in two adjacent cycles. In this case for a read operation we have to return a result after bitwise operation, not to read it from memory (because of writes to a BRAM occur one cycle after a request).

Original SRAM for L2 states is a simple dual-ported memory with one port for read and one port for write requests. A wrapper used for it in FPGA design instantiates a true dual-ported memory because in contrast to above case this memory can have read conflicts. To enable bit-enable write mask the wrapper pads incoming data and mask bits which always have the same mask with zeros.

Memory wrappers for BRAMs are located in the *$DV_ROOT/design/proto* directory.

4.2 Mapping of Main Memory

Simulation infrastructure for OpenPiton implements memory using Verilog PLI. Obviously, this solution can not be used for a prototype because of if is not synthesizable.

There are several options for memory implementation, available for Open-Piton FPGA protoype. Depending on design configuration, processor memory can be emulated with on-board DDR memory, on-FPGA BRAMS and SD card. For a list of available configurations see Section 5.2.
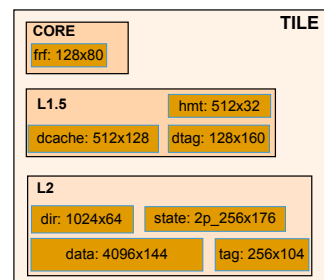
Figure 3: Memory blocks and their sizes inside an OpenPiton tile

4.2.1 Using on-Board DDR3 Memory

All currently supported development boards have DDR3 memory on it. This allows us easily implement processor main memory using Xilinx MIG 7 IP core [4].

4.2.2 Memory Emulation With on-FPGA BRAMs

Though DDR memory has large capacity, it requires a complex controller, which uses on-FPGA logic elements. At the same time most of assembly tests (as well as OpenBoot) are relatively short (less than several MB), which makes them well suited for storing them in BRAMs. This is a synthesizable option which allows to create a self-contained FPGA design (without any additional external I/O controller and storage).

In order to map a test to map a BRAM, it should be run on a software simulator (using `sims` command) first to get a set of addresses accessed by a test. The flow of mapping of an assembly to a BRAM is described below:

- `test.s` is passed to `sims` through `--bram-test` option of `protosyn`, which compiles it and creates `mem.image` file in *$MODEL_DIR* directory

- after test finishes, `sims.log` is analyzed by `make_mem_map.py` script. It extracts addresses accessed by a test and corresponding data read from them[3]

---

[3] Even if results of memory reads are printed in a cache line granularity (64 bytes), not the whole cache line can have valid data in a log file. Because of that each address met in `sims.log` is checked in `mem.image` file to make sure it is initialized. If there is a read of an uninitialized address, `make_mem_map.py` script maps it to zero

- accessed memory addresses are aggregated into sections and based on that a `.coe` (used to load data to a BRAM) and `storage_addr_trans.v`. The latter one is used to map physical addresses to BRAM addresses.

Refer to Section 5.1 for `protosyn` man page.

4.2.3 Memory Emulation With SD Card

SD card is a natural way to replace an CD drive for an FPGA. If design is built with an SD controller, all memory requests to I/O space except those with UART address (`addr[39:13] == 28'hfff0c2c`) are forwarded to an SD controller (see Figure 2). Currently it supports only micro SDSC card (not SDHC card) and works at 20MHz frequency.

5 Design Configurations and Tools

5.1 Prototype Project Defines

FPGA version of OpenPiton can be configured using next defines:

- `NO_SCAN*` - define is used to remove logic responsible for memory BIST and instantiating flip-flops without scan. The presence of this define doesn't allow to use a scan chain.

- `FPGA_SYN*` - enables FPGA specific optimization for SPARC core

- `FPGA_SYN_1THREAD*` - allows to synthesize a design with one thread per core

- `NO_USE_IBM_SRAMS*` - forces to use memories suited for FPGA synthesis

- `PITON_PROTO*` - enables OpenPiton specific optimization for FPGA

- `PITON_FULL_SYSTEM*` - enables synthesis of both chip and chipset on one FPGA

- `PITONSYS_UART_BOOT` - used to indicate that design should be synthesized with support of UART controller responsible for communication with PC for loading test into DDR memory

- `PITON_NO_CHIP_BRIDGE*` - removes chip bridge from the chip, exposing chip interface as three credit based NoCs

- `PITON_UART16550*` - instantiates Xilinx's UART16550 IP core

- `PITONSYS_NO_MC` - indicates that design does not require memory controller. This can be the case when the test is stored in BRAM

- `PITON_FPGA_SYNTH*` - depricated from `protosyn1,0`

- `PITON_FPGA_NO_DMBR` - removes MITTS (former DMBR) from a project [5]

- `VC707_BOARD`, `GENESYS2_BOARD`, `NEXYSVIDEO_BOARD` - used to specify board type. Depending on this define single

ended or differential input clock for an FPGA and active level of reset pin is selected, corresponding memory interface is instantiated and respective instances names for BRAMs are used. See
$DV_ ROOT/verif/env/manycore/manycore_ top.v.xlx.v,
$DV_ ROOT/design/fpga/mc/rtl/mc_ top.v,
$DV_ ROOT/design/fpga/include/mc_ define.h,
$DV_ ROOT/design/proto/fpga_ top.v
and memory wrappers (Section 4.1) source files for how exactly these defines affect the design

- `PITON_FPGA_MC_DDR3` - enables instantiation of memory controller and interfaces for DDR3.

- `PITON_FPGA_SD_BOOT` - enables instantiation of an SD card controller and necessary converters for NOC interface.

- `PITON_FPGA_BRAM_TEST` - enables instantiation of a BRAM with a synthesized assembly test in it (see Section 4.2.2) and memory mapping module for it. The define also affects Memory/IO splitter (see Section 5.4).

- `PITON_FPGA_BRAM_BOOT` - enables instantiation of a BRAM with Open Boot synthesized in it and memory mapping module for it. Exclusive with `PITON_FPGA_BRAM_TEST` define. Should be used along with `PITON_FPGA_MC_DDR3` define.

- `PITON_FPGA_NO_FPU` - removes FPU from a project. **Note:** this module is required if you want to boot Linux.

- `PITON_FPGA_ETHERNETLITE` - enables networking on supported boards (see Section 9)

- `OLED_STRING` - defines a string to be displayed on OLED display (Genesys2 and NexysVideo boards)

*Note: default defines used for a project

For complete set of defines which are set for each particular configuration by `protosyn`, refer to the source *$DV_ ROOT/tools/src/proto/protosyn,2.5.*

5.2   Prototype Configurations

As described in Section 3, chipset includes memory controller, SD card controller, BRAM controllers, and UART16550 wrap-

	OS + SD + Eth (default)	BRAM_TEST	UART_DMW to DDR
BRAM with test + `bram_map` module		X	
UART/PC interface			X
DDR Memory controller	X		X
SD card controller	X		
UART 16550 wrapper	X	X	X
Ethernet Lite MAC	X		

Table 3: Suggested prototype configurations (OS_SD, BRAM_TEST and UART_DMW DDR) with infrastructure blocks included into them

per with control logic (see Figure 1). Using defines described above, different configuration of OpenPiton prototype can be created. Three most useful configurations are shown in Table 3 with modules included to each of them. The first and the last column can be combined together, which allows to run assembly tests or boot OS on using the same design. This can be selected using SW7 on boards.

The main script for running all necessary preparation steps and FPGA implementation for a targeted development board is named `protosyn` and its source is *$DV_ROOT/tools/src/proto/protosyn,2.5*. Its operation is explained in the next subsection.

5.3 Running Implementation for Supported Development Boards

The main script used for preparations and FPGA implementation of OpenPiton prototype is `protosyn` script. Its manpage can be found in Appendix A. Major operations performed by `protosyn` and their presence in the flow for different prototype configurations are presented below.

`protosyn` consists of the next steps, listed in order of their execution:

- **test run** - compiles a simulation model of a chip and runs

	sims	test run	test map	project creation	impl
no extra options (default) / --uart-dmw	X			X	X
--bram-test <test>	X	X	X	X	X
--make-mem-map (with --bram-test)	X	X	X		

Table 4: Steps of `protosyn` run for some of its options

simulation of a specified test using `sims` script. For tests which are using UART, `protosyn` passes and argument specifying UART baud rate (board frequences are hard-wired in the script)

- **test map** - creates a `.coe` file for BRAM synthesis from `mem.image` and `sims.log` files (located in *$MODEL_DIR*) and `bram_map` module for mapping from physical to BRAM addresses (see Section 4.2.2)

- **project creation** - preprocessing of `.v.pyv/h.pyv` files and generatin of `.tmp.v/.tmp.h` using `pyhp` script (see Simulation Manual). Creation Vivado project for a specified board*
  NOTE: **Overrides a previous project**

- **implementation** - runs implementation for a targeted development board down to bitstream generation

In the Table 4 shown which steps are run depending on extra `protosyn` options.

5.4 Memory Address Spaces for Different Prototype Configurations

During software simulation of a chip, where memory is implemented using Verilog PLI, all requests are directed at the same place. However, for a prototype, final destination of a request is defined by configuration of splitters: `mem_io_splitter`, `iob_splitter`, `iodev_storage_splitter`, `uart_eth_splitter` (see Figure 1, Figure 2).

In case of test stored in BRAM (BRAM_TEST configuration), all memory requests are directed to `fake_boot_ctrl` except those send to UART.

When streaming testing of assembly test through UART is on (UART_DMW configuration), all memory requests are directed to main memory except those send to UART.

For all other configurations, destination of memory requests is defined by respective memory address spaces (for Ethernet, SD, UART, etc.)

6   Prototype Operation

6.1   Reset Sequence

After reset button is pushed, the signal is converted to an internal system reset with zero active level.

If there is no DDR controller, system reset is passed to a `io_ctrl_top` module (see Figure 1), `mem_io_splitter` and clk control logic. In the case there is a memory controller, system reset is used for `MIG7 Xilinx IP core` first. Reset control logic is waiting until memory controller finishes calibration of DDR interface and after that reset is sent to `io_ctrl_top`.

If design was configured with support of UART DMW and it was enabled by SW7 on a board, control logic waits for UART stream to be finished (see Appendix D), and after it is finished, reset is deasserted for chip wake-up logic. In case of there is no UART DMW support in the design or it is not enabled by a SW7, system reset is passed to core wake-up logic directly.

After a predefined delay for wake-up logic, an interrupt packet is send to Core0 through NOC1. This packet wakes up a core and it starts fetching data from `0xffff_f000_0020` address.

6.2   Loading Assembly Tests from PC to DDR

Option `-uart-dmw ddr` makes `protosyn` to use define `PITONSYS_UART_BOOT` during synthesis. It enables interaction between FPGA and host PC connected to it through UART. The format of the file with assembly test being transmitted to FPGA is described in Appendix D.

In order to load an assembly test from a PC, `pitonstream` script can be used (see Appendix B). Board type and filename should be provided to the script. After serial port is configured, the script is waiting for configuration completion message from FPGA (which is a sequence of ASCII codes for a string "DONE"). After receiving a keyword, script copmiles an assembly test (until `--ustr` option is provided) from the specified file, converts it to .ustr format and loads trough serial interface to an FPGA.

When test is loaded, the script is waiting for one of the three keywords: "PASSED", "FAILED", "TIMEOUT". These keywords are send by UART control logic from FPGA, and therefore can not present in the test output. There are two addresses used in

$DV_ROOT/verif/diag/assemlby/include/good_bad_trap_handler.s to determine if test failed or passed. Memory access at address `0x8100000000` indicates a passed test, while an access at address `0x8200000000` indicates a failed test. These addresses are from reserved memory spaces of SPARCv9 specification and therefore do not interfere with test memory accesses. Checks for mentioned addresses are done in *$DV_ROOT/design/chipset/mem_io_splitter/rtl/mem_io_splitter.v.* After receiving any of the above keywords, `pitonstream` starts compiling and loading the next text.

When running an assembly test on FPGA, the whole memory space except UART address range is mapped to a DDR. The mapping is defined by *$DV_ROOT/design/chipset/rtl/storage_addr_trans_unified.v* module. This module was generated by `pitonunimap` script, provided in a release. Its current version corresponds to a set of tests used for OpenPiton development. However it can turn out that a new test will have some memory sections not mapped in the module. `pitonstream` makes this check every time before loading a new test to FPGA, so in the case of `ERROR: Address * is not mapped...` regenerate mapping module using `pitonunimap` script and copy its output file over *storage_addr_trans_unified.v.*

This setup requires changing only board type for `pitonstream` script and therefore allows to use the same interface and the same logic across multiple boards, which decreases development efforts and time.

6.3   Assembly Test Execution

Assembly tests can be compiled into BRAM (see Section 4.2.2) using `--bram_test <test name>` option for `protosyn`. The result of an assembly test can be either checked in the terminal (for those having output to a console) or PC can be checked using build-in FPGA Logic Analyzer. See Section 7.2 on how to set up debug cores for Xilin FPGAs.

6.4   Booting OS from an SD Card

Micro SD card allows to store both OpenBoot and OS image on it. OpenBoot requires processor clock frequency in order to be able to set up divisor latch for UART16550. For current core

frequencies on different board refer to Table 2.

Precompiled OS image files are located *os_ images* folder of a release. They can be written on SD card using `dd` Linux command or `Win32 Disk Imager` on Windows. For a step by step instructions on how to boot Linux and play tetris see Appendix F.

7 Simulation and Debugging

It is possible to run software simulation of OpenPiton prototype from Vivado. This feature is helpful for debugging reset sequence for your project and checking initial initialization sequence of a processor. This framework can be easily extend to incorporate custom tests targeting prototype specific modules, but we leave this discussion out of scope of this documentation.

For debugging a processor on FPGA build-in hardware logic analyzer are used. It allows to check states of internal signals. Instructions on how to run software simulation from Vivado and how to add debug cores are below.

7.1 Software Simulation from Vivado

You can run simulation of a prototype from Vivado to debug initial processor initialization and warm up. Simulation from Vivado allows you to you IP cores used for synthesis and ensure that you logic is interpreted in an expected way. Top level module for simulation if `fpga_top`. It generates clock are reset control signals for prototype.

- click on Tools -> Compile Simulation Libraries from Vivado GUI

- select VCS as a targeted simulator and check Overwrite the current pre-compiled libraries

- click Compile and wait until it finishes

- in Flow Navigator on the left chose Simulation -> Run Post-Synthesis Functional Simulation

- processing of netlist and compilation can take some time. After it is finished, DVE waveform viewer will be opened

7.2 Inserting Debug Cores for Logic Analyzer

Build-in logic analyzer allow you to debug FPGA design while it's running. Next steps briefly describe how to add debug cores. For more information see [6].

- find signals in the design which you want to debug. To make sure that Vivado doesn't optimize the logic corresponding and you will be able to access a signal with debug

cores, add `(* MARK_DEBUG = "TRUE" *)` before it. This directive works with flip-flops and ports, but can not work well with wires. If you need, add additional logic to flip-flop signals.

- RUN SYNTHESIS of a design

- after synthesis finished, expand OPEN SYNTHESIZED DESIGN tab of FLOW NAVIGATOR and click on SET UP DEBUG

- follow the steps in the prompt to add signals for monitoring and to assign clock domain to them

- save the design and finish FPGA flow down to bitstream generation

- when programming FPGA from Vivado, in addition to `.bit` files specify `.ltx` files with debug signals names

8   Description and Structure of Prototype Specific Files and Scripts

8.1   Source Files and Scripts

BRAM memory wrappers have extension *.xlx.v* and located in *sram_wrappers* sub-folders of the paths: *$DV_ROOT/design/chip/tile/sparc/srams/rtl*, *$DV_ROOT/design/chip/l15/rtl*, *$DV_ROOT/design/chip/tile/l2/rtl/*. For complete list of chip memory which was mapped to BRAM see Section 4.1.

Source code for prototype scripts is located in *$DV_ROOT/tools/src/proto* directory. See Appendix B, Appendix A, and Appendix C for more information on how to use provided scripts.

8.1.1   Synthesis and Implementation Files

Board specific files are located in *xilinx* directories in design source tree. `protosyn` script creates Vivado project for each board using `gen_project.tcl` script in *$DV_ROOT/tools/src/proto/vivado* directory. This script in turn calls `setup.tcl` script in the same directory and adds all necessary files (Verilog files, IP descriptions, .coe files, etc.) to the project. All source files are preprocessed by `pyhp` script using `$DV_ROOT/tools/src/proto/common/pyhp_preprocess.tcl`. All new files required for synthesis and implementation should be included in *$DV_ROOT/tools/src/proto/common/rtl_setup.tcl* script.

Constraint files for each board are in *$DV_ROOT/design/xilinx/<BOARD name>*. Description of IP cores consists of *.xci* files located in *xilinx/<BOARD name>* directories across design source tree. During synthesis, all IP related files are generated from *.xci* files in the same directories.

8.2 Generated Files

All files generated by a software simulator are put into *$MODEL_ DIR* folder. `protosyn` creates a separate sub-folder for each board. After that, depending on design type being synthesized there is a respective sub-folder (e.g *system* or *chipset*). This folder has `additional_defines.tcl` script used during implementation to set additional defines during implementation defines when running `protosyn`.

`protosyn_logs` is an output folder for `protosyn` script and contains logs for each of its step. *make_ project.log* and *implementation.log* present for every run of `protosyn`. In the case an assembly tests is synthesized into BRAM, there will be *bram_ map.log* file.

<board name>_ <design type> is a working directory for Vivado. It has *<board name>_ <design type>.xpr* file which is the description of Vivado project.

Example
Running `protosyn -b genesys2` will create the next folders and files in *$MODEL_ DIR* an shown in Figure 4.

```
genesys2
  system
      genesys2_system
        genesys2_system.runs
            impl_1
              runme.log
            synth_1
              runme.log
      protosyn_logs
        implementation.log
        make_project.log
      additional_defines.tcl
```
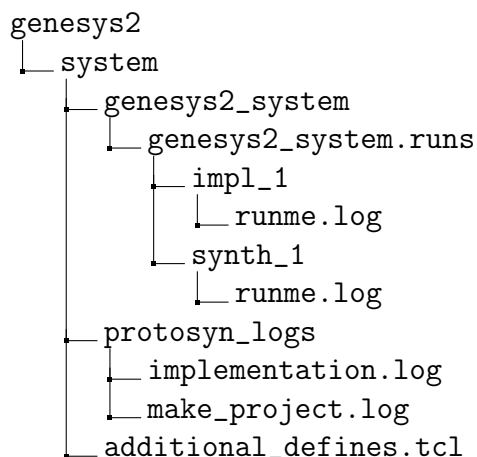
Figure 4: $MODEL_DIR directory structure

runme.log, *implementation.log*, and *make_ project.log* can be used for debugging synthesis/implementation errors during project development.

9   Networking

Ethernet is supported on the Genesys2 and the NexysVideo. It is not supported on the VC707. On supported boards, networking is enabled using the define `PITON_FPGA_ETHERNETLITE`, which is set by default for Genesys2 and NexysVideo when building with `protosyn`.

9.1   Interrupts

Interrupts are generated by the Ethernet Lite IP core upon a successful transmit or receive. The signal is synchronized to the clock domain of `ciop_iob`, which is responsible for sending the actual interrupt packet to the core.

9.2   Bringing up the network interface

Using one of the provided disk images, boot and login as described in Appendix F. Once booted, run the following commands in the terminal to set the MAC address and bring up the network interface:

```
ifconfig eth0 hw ether <MAC ADDRESS>
dhclient -v eth0
```

Replace `<MAC ADDRESS>` with the MAC address to be used for the board.

Note: If you are planning to boot and bring up the interface multiple times, it is highly recommended to put your board behind a NAT, because `dhclient` will request a new DHCP lease every time.

A protosyn manpage

```
protosyn -b <board type> [options]

Required Options:
    -b, --board             Name of a supported
                            development board.
                            Available options are:
                                vc707
                                genesys2
                                nexysVideo

Additional options:

    --bram-test <test name> Name of an assembly test
                                to be mapped into a BRAM

    --no-ddr                Implemenet design without
                            DDR memory controller (MIG7)

    --uart-dmw <storage type> Implement design with
                                Direct Memory Write (DMW)
                                from UART module turned on.
                                Default type:
                                    "ddr"

    --eth                   Add Ethernet controller
                            Default:
                                enabled

    --define <comma separated list of defines>
                                Comma separated list
                                of custom Verilog macro
                                defines

    --make-mem-map          Create a mapping of a test
                            specified by --bram-test
                            option

    --from <flow step>      Start prototype implementation
                            flow from a particular step
                            (e.g. when for generating a
                            new
                            bitfile w/o creating a
                            project).
                            Available options are:
```

```
                        project
                        impl


--to <flow step>       Run  prototype implementation
                       flow to a specified step
                       including it (e.g. for creation
                       a project without running
                       an FPGA implementation).
                       Available options are:
                            project
                            impl


--oled <string>        String to be displayed on OLED
                       display.
                       (Genesys2 and nexysVideo
                        boards only)


--slurm                Run steps of flow using
                       SLURM job scheduler


-h, --help             Print this usage message
                       and exit
```

B pitonstream manpage

```
pitonstream -b <board type> -s <storage type> -f
<filename> [options]

Required Options:
    -b, --board <board type>     Name of a supported
                                 Xilinx's development
                                 board. Available options:
                                     vc707
                                     genesys2
                                     nexysVideo

    -f, --file <filename>        File name with
                                 test names. If option
                                 --ustr is not specified,
                                 they should be assembly
                                 test names, otherwise
                                 list of generated
                                 .ustr files

    -p <portname>                Port name for serial
                                 device of FPGA board
                                 on a host PC.
                                 Default:
                                     ttyUSB0

    --ustr                       Specifies that test names
                                 in the file should be
                                 treated as .ustr file
                                 names

    -h, --help                   Display this help message
                                 and exit
```

Note: port name can be checked in /dev on Linux. Simply unplug and plug again a cable connecting a board, and check which device appears when you plug back.

C pitonunimap manpage

```
pitonunimap -b <board type> -f <filename>

Required Options:
    -b, --board             Name of a supported
                            development board.
                            Available options are:
                                vc707
                                genesys2
                                nexysVideo


    -f, --file <filename> Filename with assembly
                            test names

Additional options:
    -h, --help              Print this message
                            and exit
```

D `.ustr` file format

`.ustr` file starts with
`40'haaaaaaaaaa`
- initial synchronization sequence for FPGA FSM.

After that there are N groups, each consisting of the next fields:
40'h - memory start address for data
4'h - number of sequential data blocks in a group
128'h x specified number of blocks - data blocks, 512-bits each, stored from MSB on the left to LSB on the right. Address of the next block increases from previous one by 8'h40

Group with address `40'hffffffffff`, `8'h00` number of blocks and `8'h00` data block serves as a stop sequence for UART stream.

E   Porting OpenPiton Prototype to a Custom Development Board

FPGA project can be ported to development boards different from supported ones. Follow the steps below to create a new project and configure it for OpenPiton prototype. Instructions are presented based on Xilinx Vivado 2015.4 tool, but they also can be used as a guideline for other versions or tools. Project for Genesys2 development board with respective scripts in *$DV_ROOT/tools/src/proto/genesys2* can be taken as a reference.

1. Create a new folder in *$DV_ROOT/tools/src/proto* directory (e.g. `new_board`)

2. Create a new folder with the same name as above in *$MODEL_DIR* directory

3. Open your tool and choose "CREATE NEW PROJECT"

4. Set up new project name and its location (e.g. create a project with the name of your development board and choose *$MODEL_DIR/<new_board* location)

5. Choose RTL Project Type

6. Add source files to the project*

   *Vivado 2015.4: this step can be skipped and script `add_files.tcl` can be run instead from Vivado's Tcl Console (see below)

7. Add IP necessary cores*

   *Vivado 2015.4: folder *$DV_ROOT/tools/src/proto/genesys2/ip_cores* can be copied to *$MODEL_DIR/new_folder* and added as a directory to the project

8. Add a constraint file to your design*. File *$DV_ROOT/tools/src/proto/genesys2/constraints.xdc* can be taken as a reference. Note that pin names and clock constraints will require changes to match you FPGA pin-out.

   *Vivado 2015.4: copy *$PITON_ROOT/tools/src/proto/genesys2/constraints.xdc* to *$MODEL_DIR/new_folder* and add it to the project

9. Select a targeted FPGA or development board type

10. Finish project creation

11. **Vivado 2015.4:** copy all `.coe` files from *$DV_ROOT/tools/src/proto/common* to a recently created *$MODEL_DIR/new_folder*

12. **Vivado 2015.4**: copy *$PITON_ROOT/tools/src/proto/add_files.tcl* to *$MODEL_DIR/new_folder*, go to that directory in `Tcl Console` and execute `source add_files.tcl`. This command will add necessary source files, set the top level file, add a constraint file and set defines.

13. **Vivado 2015.4:** if you are receiving warnings about locked IPs, try to run `upgrade ip [get ips -all]` from its Tcl Console. If it doesn't remove all warnings, you will have to regenerate those IP cores manually.

    Note: `mig_7series_0` IP core will not be updated correctly in any case because of pin constraints. See the step below on a required procedure to generate it for Vivado 2015.4

14. **Vivado 2015.4:** Regenerate MIG with settings recommended for your FPGA. Due to design constraints it is recommended to choose 4:1 PHY to Controller Clock Ratio, System Clock - "No Buffer", Reference Clock - "Use System Clock", and enable XADC instantiation.

15. **Vivado 2015.4:** Update Write and Read width for `uart_mig_afifo` IP core, which depends on MIG settings and must be equal to 'MIG APP ADDR WIDTH+'MIG APP DATA WIDTH+'MIG APP MASK WIDTH

16. **Vivado 2015.4:** Set up necessary type of clock input for `clk_mmcm` IP (single ended VS differential). Set a targeted clock frequency and update `uart_16550`, `afifo_mem_splitter`, `afifo_splitter_mem`, and `uart_mig_afifo` IP cores to match the frequency of a generated clock.

17. Modify the next files to specify clock type, reset level, and DDR interface of your board.

    - *$DV_ROOT/design/proto/fpga_top.v*
        - Add additional preprocessor directive if you have

differential clock and/or high active reset level

- *$DV_ ROOT/verif/env/manycore/manycore_ top.v.xlx.v*

  - Add additional preprocessor directives if you have differential clock and/or high active reset level

  - if DDR is planned to use, create wires with respective interface width under a new define for module interface in instance of `mc_top` module

- *$DV_ ROOT/design/fpga/include/mc_ define.h*

  - set width of DDR data bus - DDR3 DQ WIDTH*

    *Vivado 2015.4: set parameters of MIG as they are in generated IP core

- *$DV_ ROOT/design/fpga/mc/rtl/mc_ top.v*

  - if DDR is planned to use, create wires with respective interface width under the same define as above

18. Set defines required for you configuration*

    *For Vivado 2015.4: If you ran *add_ files.tcl* script, it added the following defines to your project:
    ```
    NO_SCAN
    FPGA_SYN
    FPGA_SYN_1THREAD FPGA_SYN
    PITON_PROTO
    NO_USE_IBM_SRAMS
    PITON_FPGA_NO_DMBR
    PITON_FPGA_MC_DDR3
    PITON_FPGA_SD_BOOT
    ```
    Check define description in Section

19. Set a top level source file to *cmp_ top.v**

    *Vivado 2015.4: If you ran `source add_files.tcl`, it is already done for you

20. If you are planning to run OBP from BRAM, make sure that the frequency of a reference clock for UART in Hyper-Visor matches the system clock in the design. If necessary, regenerate *obp.coe* file and `bram_16384x512` IP core.

After these steps steps you should be able to run implementation
of OpenPiton for your FPGA.

F   Step-by-Step Instructions for Booting Debian Linux and Playing Tetris

Below are steps required to build an OpenPiton prototype targeting Digilent Genesys2 development board and run Tetris on a full stack Debian Linux:

- download `tar.gz` archive of OpenPiton release

- extract it and set up your environment and tools (see Section 1.2)

- run `protosyn -b genesys2` and wait until script exists

- because Vivado creates separate processes for its tasks, `protosyn` exits before Vivado generates a `.bit` file. You need to check `runme.log` in
  `$MODEL_DIR/genesys2/genesys2_piton/`
  `genesys2_piton.runs/impl_1`
  to make ensure that bit file was successfully generated.

- open HARDWARE MANAGER in Vivado or Vivado Lab Edition connected to Genesys2 board

- open a target and program the board with a generated `.bit` file

- put image file with OpenBoot and OS image onto SD card

- insert the SD card into the board and press RESET button

- wait for Open Boot to start OK boot prompt

- print `boot Linux` command in OK boot prompt

- wait for Linux to boot

- use `root` both as login and password

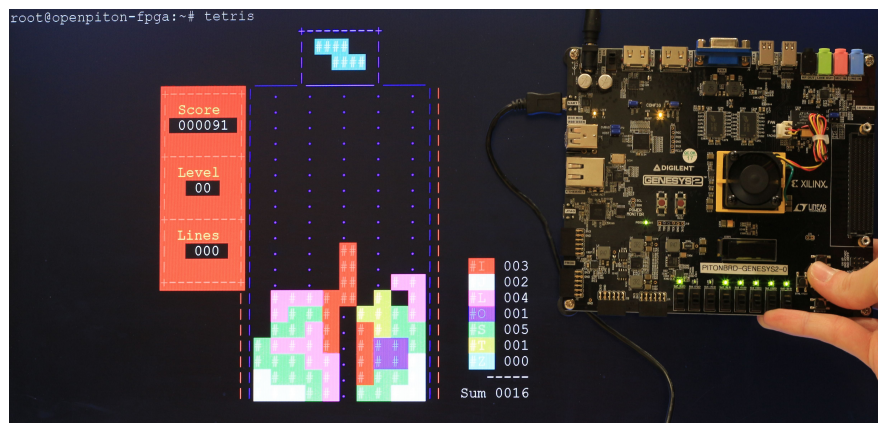- print `tetris` in Linux prompt and play the game!

Figure 5: Genesys2 board running Tetris on full stack Debian Linux

G   Generating an SD-Bootable Image

We consider this guide to be deprecated as we provide working disk images. Please check our download page for the most up-to-date images. If you wish to build an image of your own, you will want to adapt the instructions to fit a newer Debian release (one targeting SPARC64).

This appendix explains in detail how to generate an SD-bootable image of Debian for SPARC (not SPARC64) for OpenPiton on an FPGA. Each image consists of two items:

- An existing PROM file that contains a copy of the hypervisor and OpenBoot. This binary is tuned to your FPGA's primary clock frequency and RAM size.

- A ramdisk, which contains SILO, a bootable version of Linux, and a full filesystem.

The PROM files are pre-generated, but you can create and modify your own ramdisk to customize the software you want to run on OpenPiton. Ramdisks can be generated from scratch, but we recommend that you use our distribution as a starting point.

G.1   Building a Ramdisk from Scratch

In order to create a ramdisk from scratch, simply follow these steps on a machine running Linux.

G.1.1   Initialization

First, create an empty ramdisk. Simply execute the following instruction, replacing `{size}` with the desired size of your ramdisk in megabytes (be sure to add the extra 16 in the command below as noted - this space is for the PROM).

```
dd if=/dev/zero of=ramdisk bs=1M count={size + 16}
```

Next, create a Sun disk partition in this file. Run `parted` and enter the following commands in order to navigate its menus. Note that if /dev/loop0 is already in use, you can change the 0 to another number. `losetup -a` will show the status of all loop devices.

```
sudo losetup -o 16777216 /dev/loop0 ramdisk
sudo parted /dev/loop0
```

```
mklabel
sun
mkpart
ext3
0
{size}M
q
```

Save the VTOC from the front of the ramdisk.

```
dd if=/dev/loop0 of=vtoc count=1
```

Then, make the file system. Be careful as we have seen this command attempt to create a filesystem larger than the device. Check that the number of blocks fits within the device and if not, reduce the count by providing a block count at the end of the `mke2fs` command.

```
sudo mke2fs -j /dev/loop0
y
sudo losetup -d /dev/loop0
```

G.1.2 Mounting and Filling the Disk

Next, use a loopback device to mount the ramdisk.

```
mkdir mnt
sudo mount -o loop,offset=16777216 ramdisk mnt
```

Then, use `debootstrap` to initialize a basic Debian installation on your mounted disk (one command).

```
sudo debootstrap --arch=sparc --variant=minbase wheezy
mnt
```

This should fill the */mnt* directory with a mostly-complete minimal installation of Debian. However, there are some things you'll have to do yourself to get it to work on OpenPiton.

First, edit *mnt/etc/inittab* to specify the terminal and baud rate we're using for OpenPiton. Comment out the lines that start `getty`, leaving only two uncommented as shown below.

```
# Note that on most Debian systems tty7 ...
# so if you want to add more getty's go ...
#
1:2345:respawn:/sbin/getty -L 115200 tty1
```

```
#1:2345:respawn:/bin/login ttyS0 </dev/ttyS0 >/dev/ttyS0
2>&1
#2:23:respawn:/sbin/getty 38400 tty2
#3:23:respawn:/sbin/getty 38400 tty3
#4:23:respawn:/sbin/getty 38400 tty4
#5:23:respawn:/sbin/getty 38400 tty5
#6:23:respawn:/sbin/getty 38400 tty6

# Example how to put a getty on a serial line (for a
terminal)
#
#s0:2345:respawn:/sbin/agetty -L --noclear ttyS0 115200
vt100
T0:2345:respawn:/sbin/getty -L ttyS0 115200 vt100
#T1:23:respawn:/sbin/getty -L ttyS1 9600 vt100
```

Then, edit *mnt/etc/fstab* to enable use of the Sun ramdisk. Replace the contents of that file with those shown below.

```
# /etc/fstab: static file system information.

proc /proc proc defaults 0 0
/dev/sunhv_disk / ext3
defaults,errors=remount-ro,noatime 0 1
```

Finally, add a symlink to */proc/mounts* in the */etc* directory.

```
cd mnt/etc
sudo ln -nsf /proc/mounts mtab
```

Next, we're going to copy over a valid boot directory. Start with the boot directory from our provided ramdisk:

```
cd mnt
sudo cp path/to/existing/boot/* ./boot
```

Create symlinks in the root directory to point to *silo.conf* and *vmlinuz.*

```
sudo ln -nsf /boot/vmlinuz vmlinuz
sudo ln -nsf /boot/silo.conf silo.conf
sudo ln -nsf /boot/silo.conf /etc/silo.conf
```

If you wish to modify the version of Linux being used on the ramdisk, please note that you will need to add support for the sunhv disk and network drivers.

Finally, we're going to patch up the *silo* files. Change directories into the directory containing *mnt*. Then, run the following command.

```
sudo /sbin/silo -r ./mnt -f -p 0
sync
sudo umount mnt
dd if=vtoc of=ramdisk seek=16 conv=notrunc count=1
```

Your *ramdisk* file is now ready to be used.

G.2   Modifying an Existing Ramdisk

Once you have a ramdisk (either one you generated or our supplied one), you can add or remove files from it on any Linux machine. First, mount the ramdisk file to a loopback device:

```
mkdir mnt
sudo mount -o loop,offset=16777216 ramdisk mnt
```

Copy any files you want to use into the ramdisk's filesystem, which is now exposed in the *mnt/* directory.

If you want to run `apt-get` to install new packages, you first must `chroot` into the *mnt/* directory.

```
cd mnt
sudo mount -o bind /proc ./proc
sudo mount -o bind /dev ./dev
sudo mount -o bind /sys ./sys
sudo chroot .
```

Run `apt-get` to install any desired packages, making sure to run `apt-get clean` once you're finished to clear out the package manager's cache.

To exit, undo your `chroot`.

```
exit
sudo umount ./proc
sudo umount ./dev
sudo umount ./sys
```

To save the modified ramdisk back into the original file, simply unmount it.

```
cd ..
sync
sudo umount ./mnt
```

G.3   Creating an SD Image

With a PROM file and a ramdisk ready, simply use `dd` to write it to an SD card.

Note: As of the current release, only plain SD cards will work properly – any SDHC or SDXC cards will not work with Open-Piton's FPGA implementation.

References

[1] D. L. Weaver, "Opensparc internals," pp. 107–108, Sun Microsystems, Inc., October 2008.

[2] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "Openpiton: An open source manycore research framework," in *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), ACM, 2016.

[3] X. Inc., "Vc707 evaluation board for the virtex-7 fpga user guide," April 7, 2015.

[4] X. Inc., "7 series fpgas memory interface solutions," March 1, 2011.

[5] Y. Zhou and D. Wentzlaff, "Mitts: Memory inter-arrival time traffic shaping," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (New York, NY, USA), ACM, 2016.

[6] X. Inc., "Vivado design suite user guide," June 24, 2015.