



OpenPiton FPGA Prototype Manual

Wentzlaff Parallel Research Group

Princeton University

openpiton@princeton.edu

Version 2.1

History of versions

Version	Date	Author(s)	Changes
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.

Contents

1	Preface	1
1.1	Notation Conventions	1
1.2	Required Tools and Environment Set up	1
1.2.1	SW Requirements	1
1.2.2	HW Requirements	1
2	Introduction	2
3	Prototype Architecture	3
4	Mapping of a processor to FPGA	6
4.1	Mapping of on-chip processor SRAMs	7
4.2	Mapping of main memory	7
4.2.1	Using on-board DDR3 memory	8
4.2.2	Memory emulation with on-FPGA BRAMs	8
4.2.3	Memory emulation with SD card	9
5	Design Configurations and Tools	9
5.1	Prototype Project Defines	9
5.2	Prototype configurations	10
5.3	Running implementation for supported develop- ment boards	11
5.4	Memory Address Spaces for Different Prototype Configurations	12
6	Prototype operation	13
6.1	Reset Sequence	13
6.2	Assembly test execution	13
6.3	Running an Open Boot from BRAM	13
6.4	Booting OS from an SD card	13

7	Simulation and Debugging	14
7.1	Software simulation from Vivado	14
7.2	Inserting Debug Cores for Logic Analyzer	14
8	Description and Structure of Prototype Specific Files	15
8.1	Source Files and Scripts	15
8.2	Generated Files	16
A	protosyn manpage	17
B	Porting OpenPiton Prototype to a Custom Development Board	18
C	Step by Step Instructions for Booting Debian Linux and Playing Tetris	22
D	Generating an SD-Bootable Image	24
D.1	Building a Ramdisk from Scratch	24
D.1.1	Initialization	24
D.1.2	Mounting and Filling the Disk	25
D.2	Modifying an Existing Ramdisk	26
D.3	Creating an SD Image	27
	References	29

List of Figures

1	FPGA prototype architecture with a default configuration. FPGA pins are shown as green rectangles and connected to logical signals at <code>CMP_TOP</code> module.	4
2	Memory blocks and their sizes inside an OpenPiton tile	8
3	Genesys2 board running Tetris on full stack Debian Linux	23

List of Tables

2	Notation conventions	2
3	Supported Development Boards and their parameters	3
4	Suggested prototype configurations (OS_SD, BRAM_TEST and BRAM_BOOT) with infrastructure blocks included into them	11
5	Steps of protosyn run for some of its options . .	12
6	Memory address spaces for different prototype configurations	12

1 Preface

1.1 Notation Conventions

In the manual next text conventions are used:

1.2 Required Tools and Environment Set up

1.2.1 SW Requirements

In order to be able to run `protosyn` script, `$DV_ROOT` and `$MODEL_DIR` environment variables should be defined. They can be set by going to the topmost directory of OpenPiton framework folder, and execute commands:

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

To be able to run FPGA implementation of OpenPiton prototype you need make sure that you have Xilinx Vivado installed on your machine it has 2015.4 version, which we were using during development. You can check its version by running

```
vivado -version
```

Current implementation of processor memory on BRAMs imply that some tests will be run in SW simulator first. This requires an installed Synopsys VCS program (see simulation manual for more details).

1.2.2 HW Requirements

You need one of a supported development boards, listed in Table 3. If configuration implies usage of an SD card, you will need one micro SD. Because of an SD controller limitations, only micro SD are supported (**not SDHC**).

Example	Description
<i>\$DV_ROOT/tools/src/proto</i>	<i>Italic text</i> is used to indicate paths to scripts and folders
<code>source add_files.tcl</code>	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 options

Table 2: Notation conventions

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 we have designed FPGA prototype for OpenPiton processor [2]. It targets several development boards with Xilinx's FPGAs. We provide all the scripts required to implement OpenPiton from Verilog sources down to bit files to program an FPGA. In addition, we describe all changes applied to a design to enable FPGA synthesis, so it can be easily ported to other development boards and FPGAs.

Because of limited FPGA capacity we implemented at most 4 processor tiles on one VC707 development board [3]. For all currently supported development boards, maximum number of cores fitted on them and core clock frequencies see Table 3.

Development Board, FPGA name, Part	Core Clock (1 core)	# Cores	DDR type, Size, Data width
Xilinx VC707 Virtex-7 XC7VX485T-2FFG1761C	67 MHz	4	DDR3 1GB 64 bits
Digilent Genesys2 Kintex-7 XC7K325T-2FFG900C	60 MHz	2	DDR3 1GB 32 bits
Digilent NexysVideo Artix-7 XC7A200T-1SBG484C	29 MHz	1	DDR3 512MB 16 bits

Table 3: Supported Development Boards and their parameters

3 Prototype Architecture

Top module of FPGA prototype design is named `cmp_top` (file `manycore_top.v.xlx.v`). As shown on Figure 1, it can be divided into next logical blocks (with respective submodule name except for `clk`, `rst ctrl` block), described below:

- `clk`, `rst ctrl` - several helper modules and control logic for generation of clock and reset signals. See reset sequence in Section 6.1.
- `chip` (described in `chip.v.xlx.v`) - a modified version of a chip with removed `chip_bridge` module (which converts from a NOC interface to an off-chip interface). Since FPGA off-chip infrastructure works at the same frequency, asynchronous FIFOs in `chip_bridge` are not required. Also, while having all interconnections on an FPGA removes pin limitation presented for a real chip. The last, but not least feature is that removing of `chip_bridge` saves some logic cells, which is critical to small FPGAs.
- `val_to_credit` and `credit_to_val` block - converts off-chip val/rdy NOC interface to a credit based. Since 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

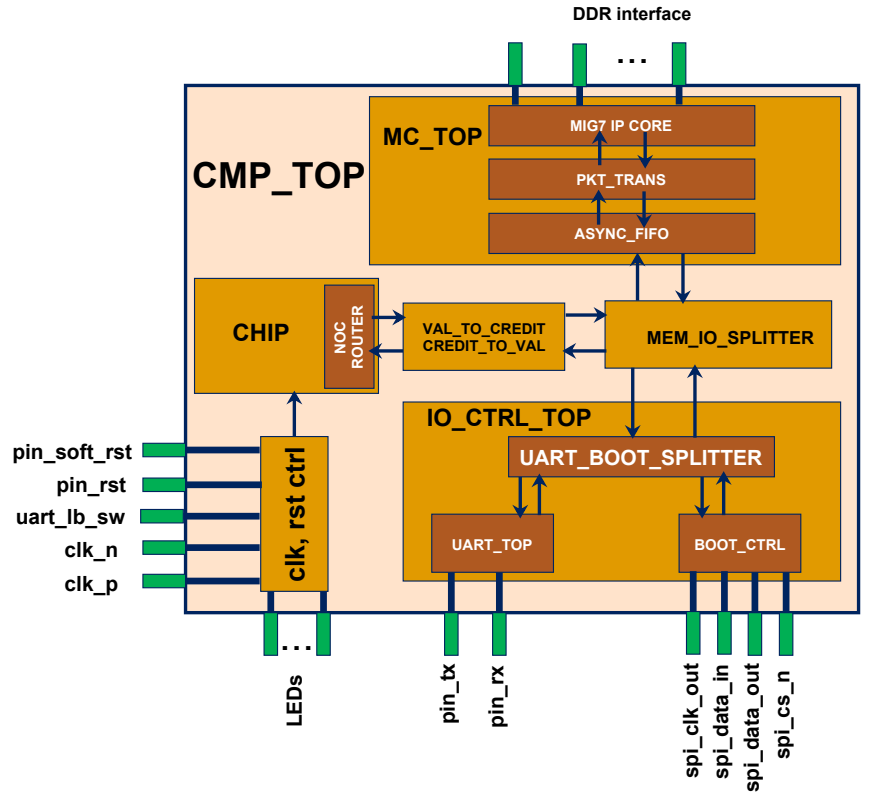


Figure 1: FPGA prototype architecture with a default configuration. FPGA pins are shown as green rectangles and connected to logical signals at **CMP_TOP** module.

from a chip to DDR memory and IO controller¹.

- `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 responses².
- `mc_top` - a top level for a memory controller. Because of application side of a MIG 7 IP core works at a higher frequency than the chip and has an interface different from NOC's val/rdy, asynchronous FIFOs and packet transceivers are required. Because of interface of a data bus depends on a board type (see Table 3), it 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 on the board type.

Pins are shown on Figure 1 as green rectangles and almost all are self-explained. The only one which requires explanation is `uart_lb_sw`, which stands for UART loop-back switch and controls if `pin_tx` is connect to UART16550 IP core or directly to a `pin_rx` for debugging purposes. Also, instead of `clk_n`, `clk_p` pin pair can be a single ended clock depending on FPGA type.

¹ this is not the same as splitting between 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

4 Mapping of a processor to FPGA

In order to enable FPGA prototyping of a processor 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 manufacture specific modules with their equivalents supported by FPGA tools.

The next IP blocks were created specifically for prototyping of OpenPiton:

- Block memories to emulate on-chip SRAMs
- Main Memory controller
- Asynchronous FIFOs for inter-clock domain transfers
- Generator of system clocks
- UART16550
- Controller for an SD card

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 BOOR and assembly test memory with BRAMs and corresponding controllers

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 on-chip processor 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 2. All of them except a memory for L2 states (2p_256_176 on Figure 2) 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 it is not synthesizable.

There are several options for memory implementation, available for OpenPiton FPGA prototype. 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.

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 IO 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:

- `text.s` is passed to `sims`, which compiles it and creates `mem.image` file in `$MODEL_DIR` directory
- after test finished `sims.log` is analyzed by `make_mem_map.py` script. It extracts addresses accessed by a test and corresponding data read from them³
- accessed memory addresses are aggregated into sections and based on that a `.coe` (which is used to synthesize a BRAM with data) and `bram_map.v` files are generated and placed in `$MODEL_DIR/<board name>` directory. The

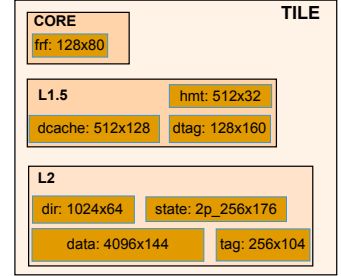


Figure 2: Memory blocks and their sizes inside an OpenPiton tile

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

latter one is used to map physical addresses to BRAM addresses.

As mentioned above, Open Boot can also be stored in a BRAM. However, it doesn't require a preliminary run by software simulator. An `obp.coe` file (located in `$MODEL_DIR/<board name>`) is used to synthesize a memory and `bram_map_obp.v` module maps physical addresses to BRAMs addresses.

To see which options are used to generate a BRAM with a test or OpenBoot stored in it refer to Section 5.1.

4.2.3 Memory emulation with SD card

SD card is a natural way to replace a CD drive for an FPGA. If design is built with an SD controller, all memory requests directed to IO space by `mem_io_splitter` (see Figure 3) except those with UART address (`0xffff0c2c`) are forwarded to SD controller. Currently it supports only micro SD 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*` - forces to use logic required only for 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_FPGA_SYNTH*` - deprecated 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.

***Note:** default defines used for a project

5.2 Prototype configurations

Off-chip prototype infrastructure consists of Memory controller, SD controller, BRAM controllers, UART 16550 wrapper and logic for controlling clk, rst signals (see Figure 1). Using defines described above, different configuration of OpenPiton prototype can be created. Three most useful configurations are shown below in Table 4 with modules included to each of them:

	OS_SD (default)	BRAM_TEST	BRAM_BOOT
BRAM with test + bram_map module		X	
BRAM with OBP + bram_map_obp module			X
DDR Memory controller	X		X
SD card controller	X		
UART 16550 wrapper	X	X	X

Table 4: Suggested prototype configurations (OS_SD, BRAM_TEST and BRAM_BOOT) with infrastructure blocks included into them

The main script for running all necessary preparation steps and FPGA implementation for a targeted development board is named **protosyn** and located in *\$DV_ROOT/tools/src/proto* directory. 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. Here described the main operations performed by **protosyn** and their presence in the flow for different prototype configurations.

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

- **sims compilation** - preprocessing step, which main goal is to generate `.tmp.v/.tmp.h` files from `.v.pyv/h.pyv`. For more information see Simulation Manual
- **test run** - compiles a simulation model of a chip and runs simulation of a specified test using **sims** script. For tests which are using UART, **protosyn** passes and argument specifying UART baud rate (board frequencies 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*)

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

Table 5: Steps of **protosyn** run for some of its options

Among used defines there are:	DDR	BRAM	SD	UART
PITON_FPGA_SD_BOOT PITON_FPGA_MC_DDR3 (default)	addr[63] == 0 except UART	–	addr[63] == 1 except UART	UART addr
PITON_FPGA_BRAM_TEST	–	all except UART	–	UART addr
PITON_FPGA_BRAM_BOOT PITON_FPGA_MC_DDR3 (default)	addr[63] == 0 except UART	addr[63] == 1 except UART	–	UART addr

Table 6: Memory address spaces for different prototype configurations

and **bram_map** module for mapping from physical to BRAM addresses (see Section 4.2.2)

- **project creation** - generates Vivado project for a specified board type from a provided tcl script using Vivado command line. **Overrides a previous project**
- **implementation** - runs prototype implementation for a targeted development board down to bitstream generation

In the Table 5 shown which steps are run depending on extra **protosyn** options.

5.4 Memory Address Spaces for Different Prototype Configurations

During simulation of a chip, where memory is implemented using Verilog PLI, all memory requests are directed at the same place. However, for a prototype, final destination of a request is defined by configuration of **mem_io_splitter** and **uart_boot_splitter** (see Figure 1).

In Table 6 shown memory address spaces for DDR memory, BRAM memory, SD cards and UART 16550 depending on used defines are shown:

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 a controller for DDR memory, reset is passed to a processor, `io_ctrl_top` module (see Figure 1), `mem_io_splitter` and `clk`, reset logic. After a predefined delay for reset control logic, an interrupt packet is sent to Core0 through NOC1. This packet resets a core and it starts fetching data from `0xffff_f000_0020` address.

In the case there is a memory controller, system reset is used for `MC_TOP` module and a processor. Rst control logic is waiting until memory controller finishes calibration of its DDR interface and reset for application side of memory controller is deasserted. After both these conditions are met, reset is generated for `io_ctrl_top` and the part of rst logic responsible for sending the first interrupt packet to the processor.

6.2 Assembly test execution

Assembly tests are stored in BRAM (see Section 4.2.2). The result of an assembly test can be either checked in the terminal (for those which have an output) or PC can be checked using build-in FPGA Logic Analyzer. See Section 7.2 on how to set up debug cores for Xilinx FPGAs.

6.3 Running an Open Boot from BRAM

The size of Open Boot is 1MB, which makes it easy to store in a BRAM (using corresponding defines for prototype configuration). It uses UART 165500 model to communicate with a user and uses processor clock frequency to correctly set up divisor latch for UART. Precompiled coefficient files for Open Boot are located in directories, corresponding to development boards: `$DV_ROOT/tools/src/proto/<board name>/obp.coe`.

6.4 Booting OS from an SD card

Micro SD card allows to store both Open Boot and OS image on it. Open Boot requires processor clock frequency to be able correctly set up divisor latch for UART. Precompiled binary files are located **TODO**. 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 C.

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 is `fpga_top`. It generates clock and 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 SYTHESIS of a design
- after synthesis finished, expand OPEN SYNTHESIZE 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

8.1 Source Files and Scripts

Some source files used for chip simulation require changes for OpenPiton prototype. They have `.xlx.v` extension and located in sub-folders of both `$DV_ROOT/verif/env` and `$DV_ROOT/design/chip`.

Source code for `protosyn` script is located in `$DV_ROOT/tools/src/proto` directory along with a script for mapping an assembly test into BRAM - `map_mem_map.py`.

Board specific files are located in `$DV_ROOT/tools/src/proto/<board name>` folder. Each of them includes `<board name>_piton_project.tcl` script used by Vivado to create a project for targeted board. `.xdc` file contains pin and timing constraints. `obp.coe` contains a precompiled version of Open Boot configured for a frequency default for a targeted board (see Table 3). `ip_cores` folder contains a set of folders with `.xci` configurations of IP cores used by a prototype.

This copy is copied to *\$MODEL_DIR* directory by a **protosyn** script. So, if you want to add project configuration for a board, you need to update files in *\$DV_ROOT/tools/src/proto/<board name>*, which will be used by **protosyn** to create Vivado project during its next run.

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 and copies files from a corresponding folder *\$DV_ROOT/tools/src/proto/<board name>* into it. In addition, it copies all files from *\$DV_ROOT/tools/src/proto/common* into this directory.

Inside a created folder, *<board name>_piton* is a working directory for Vivado. Its structure is a default one used by Vivado.

protosyn_logs is an output folder for **protosyn** script and contains logs for each of its step. These logs along with Vivado default logs allow should be used for tracking **protosyn** progress and errors.

A protosyn manpage

Usage: protosyn -b <board type> [options]

Required Options:

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

Additional options:

-h, --help	Print this usage message
--bram-test <test name>	Name of an assembly test to be mapped into a BRAM
--bram-boot	Implement design with OpenBoot mapped to a BRAM
--no-ddr	Implement design without DDR memory
--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

B 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_mmc` 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 5.1

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 HyperVisor matches the system clock in the design. If necessary, regenerate *obp.coe* file and `bram_16384x512` IP core.

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

C 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 `.bin` 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!

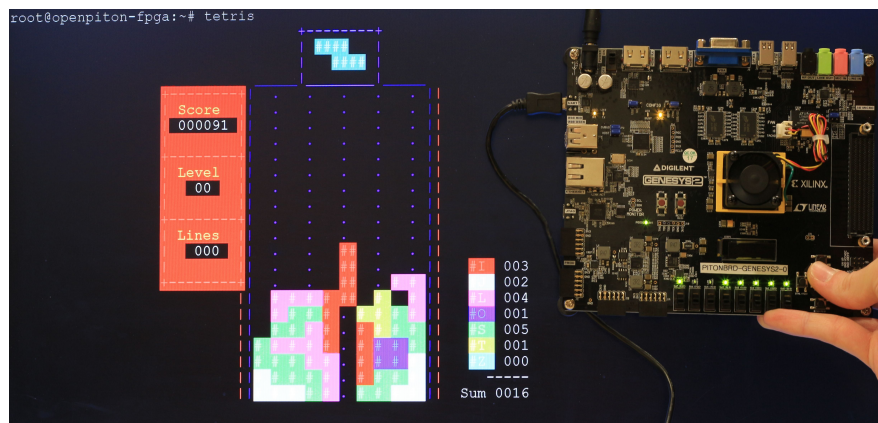


Figure 3: Genesys2 board running Tetris on full stack Debian Linux

D Generating an SD-Bootable Image

This appendix explains in detail how to generate an SD-bootable image 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.

D.1 Building a Ramdisk from Scratch

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

D.1.1 Initialization

First, create an empty ramdisk. Simply execute the following instruction, replacing `{size}` with the desired size of your ramdisk in megabytes.

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

Next, create a Sun disk partition in this file. Run `parted` and enter the following commands in order to navigate its menus.

```
sudo parted ramdisk
mklabel
sun
mkpart
ext3
0
{size}M
q
```

Save the VTOC from the front of the ramdisk.

```
dd if=ramdisk of=vtoc count=1
```

Then, make the file system.

```
sudo mke2fs -j ramdisk
y
```

D.1.2 Mounting and Filling the Disk

Next, use a loopback device to mount the ramdisk.

```
mkdir mnt
sudo mount -o loop 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 conv=notrunc count=1
```

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

D.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 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
```

D.3 Creating an SD Image

With a PROM file and a ramdisk ready, you can generate a bootable SD image in a few steps.

First simply run the following bash script. It takes three positional arguments: the PROM file, the ramdisk file, and the desired output file. The script places the PROM file at the start of the image and then adds enough bytes to offset the ramdisk to 0x1000000 bytes above the PROM file. Finally, it reverses the byte order in the file to configure it to run on OpenPiton.

```
#!/bin/bash

cp "$1" "$3"
HVSIZE=$(stat -c%s "$3")
RDISKOFF=16777216
RDISKSIZE=$(stat -c%s "$2")
```

```
ADDSIZE=$(( $RDISKOFF-$HVSIZE ))  
truncate -s +"$ADDSIZE" "$3"  
cat "$2" >> "$3"  
objcopy -I binary -O binary --reverse-bytes=8 "$3" "$3"
```

Once an image file has been generated, 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 OpenPiton’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.