Parallel Systems Lab
Department of Electrical Engineering

TECHNION
Israel Institute
of Technology

High Speed Digital Systems Lab
Department of Electrical Engineering

Hardware DLL

Real Time Partial Reconfiguration Management of FPGA by OS

Submitted by Alon Reznik

Supervised by Oz Shmueli

BI-SEMESTRIAL STUDENT PROJECT | 2014 - 2015 | FINAL REPORT (PART 2)

Table of Contents

Table of Figures

1.0 Introduction

Due to the physical constraints preventing frequency scaling, the early 2000s saw the classic single-core processor being replaced by the more power efficient and better performing multi-core processor. The increased performance was not derived from faster clock speeds, but rather from the ability to process multiple CPU instructions simultaneously. This boost in computing power is only available to those programs that are amenable to parallel computing, and it is ultimately limited by Amdahl's law.

Parallelization is hardly new to computer science. Another prominent and much more time-tested method of providing concurrency to computer systems is hardware acceleration. Implemented on an FPGA/ASIC, fixed-function hardware accelerators (HWAs) can perform computationally intensive tasks separately from the general-purpose CPU; thus reducing its workload and increasing the system's overall performance. However, being task-specific, HWAs lack the versatility and flexibility of a general-purpose processor, so programs that were not explicitly designed to take advantage of a certain HWA will not be able to utilize it.

Although the concept of reconfigurable computing has existed since the 1960s, slow progress in silicon technology prevented any commercial units from being manufactured until 1991. Since then, many innovations have been made in FPGA technology. One of them in particular, partial reconfiguration (PR), might be more significant to parallel computing than previously thought.

2.0 Partial Reconfiguration

PR is a modern FPGA feature that is gaining popularity among hardware developers. It is a modular design methodology for dynamic FPGA-based systems, which requires the use of advanced software tools in a complex design flow. Nevertheless, understanding both the theoretical and the technical aspects of PR is essential for this project, and this chapter provides the necessary background.

2.1 Benefits

Unlike conventional reconfiguration, PR doesn't require the FPGA to be held in reset while an external controller reloads it with a complete design. Instead, it allows for some parts of the design to continue operating while an external/internal controller reloads other parts of the design into predefined reconfigurable partitions (RPs). Each PR session changes the content of a single RP, so only the targeted RP has to be held in reset during the reconfiguration.

PR is often used to save space for multiple FPGA configurations by storing their common parts as a single static configuration and their different parts as multiple reconfigurable modules (RMs). Partial BIT files (bitstreams of synthesized RMs) are much smaller than full BIT files (bitstreams of synthesized FPGA configurations); therefore synthesizing numerous RMs is more memory efficient than synthesizing the same number of FPGA configurations. The static configuration is the constant part of the PR design, whereas the RMs can be loaded and reloaded on demand.

A consequent advantage of PR over conventional reconfiguration is its speed. Since partial BIT files are much smaller than full BIT files, RPs can be reconfigured much faster than the FPGA. Reducing time overhead improves the system's responsiveness and reduces the risk of losing data during reconfiguration.

Lower FPGA requirements also result in power efficiency, reduced heat generation, and of course lower cost.

2.2 Limitations

Current technology limitations prevent RMs from being interchangeably between RPs, because the interface between static and reconfigurable logic, called "partition pins", is unique to each RP. Partition pins are identifiable by an address that is written into the partial BIT files during synthesis, so every partial BIT file is associated with a single RP. This association ensures that the PR controller can load RMs into their target RP, but it also prevents RMs from being loaded into other RPs.

It is physically impossible to load a large RM (that requires a lot of resources) into a smaller RP (that has fewer resources than required), so before synthesizing IP cores into RMs, developers have to make sure that their IP cores meet the targeted RPs' resource utilization constrains. However, present software tools can only estimate resource utilization, and these estimates do not include routing resources; therefore resource optimization is currently impossible. Consequently, RPs are prone to internal fragmentation, because a small RM loaded into a larger RP leaves some of the RP's resources unutilized. Unfortunately, the software tools do not support sub partitioning, so multiple RPs cannot be joined to accommodate a large RM, and a large RM cannot be split over multiple RPs.

Due to the physical structure of the FPGA itself, PR designs can be very tricky and hard to implement. The FPGA's manufacturer provides PR design rules and guidelines[1], but the implementation's success is ultimately determined by the developer's experience, which is gained through trial and error. For example, there are limited routing resources along the edges (and especially in the corners) of RPs, so clustering the partition pins there is likely to result in routing congestion.

2.3 Example: Video Decoding Hub

With so many limitations it is easy to wrongfully overlook the benefits of PR designs over conventionally configurable ones. Perhaps these benefits are best explained by the video decoding hub (VDH) example, which is a common example of a device that can be better implement as a PR design on an FPGA. The VDH has N input channels for receiving up to N encoded video streams at the same time, and it must be able to decode all of them simultaneously. Each video stream is encoded by one of M known video codecs, whose index is transmitted on the channel's control bus. The following are 3 possible FPGA-based VDH designs.

[1] Xilinx – Vivado Design Suite User Guide: Partial Reconfiguration (UG909)

VDH Design 1: Non-Reconfigurable Design

High efficiency can be achieved by implementing a single VDH configuration with all M decoders per channel. Each channel is always ready to receive an encoded video stream and convey it to its corresponding decoder. Therefore, the FPGA never requires a reconfiguration, and no data is ever lost.

The configuration is not scalable, which implies that it has to be implemented on an expensively oversized FPGA. In addition, all $N \cdot M$ decoders are constantly active, consuming power and generating heat.

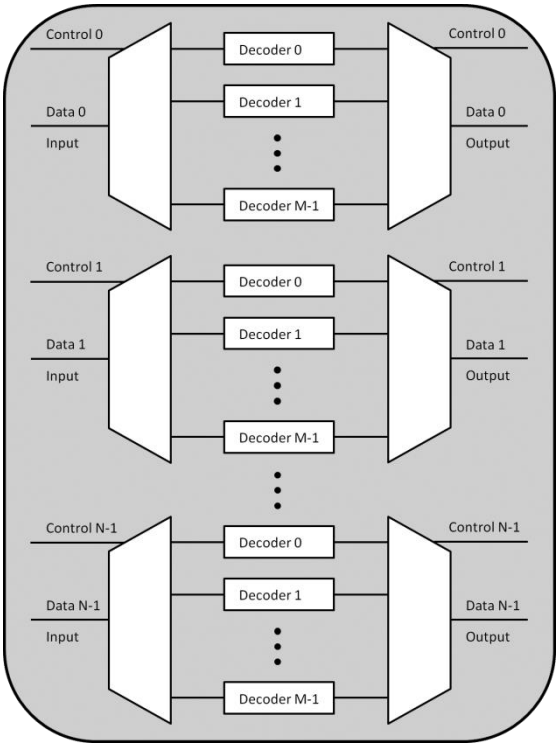

Figure 1: Block Diagram of a non-reconfigurable VDH Design

Pros

- Does not lose any data.
- Saves memory.
- Simple and easy to implement.

Cons

- Requires many FPGA resources.
- Wastes power.
- Inclined to overheat.

VDH Design 2: Conventionally Reconfigurable Design

By implementing multiple VDH configurations with only one decoder per channel, it is possible to greatly reduce the size of each configuration and consequently the size of the required FPGA. A smaller FPGA is cheaper, more power efficient, and generates less heat.

The VDH is inactive during FPGA reconfiguration, so all N channels lose some data every time a new video stream is sent on one of them. Moreover, there are M^N different VDH configurations that have to be synthesized individually and stored on non-volatile memory; hence the design itself is unscalable.

Pros

- Requires few FPGA resources.
- Saves power.
- Uninclined to overheat.
- Simple and easy to implement.

Cons

- Loses data from all channels.
- Wastes memory.

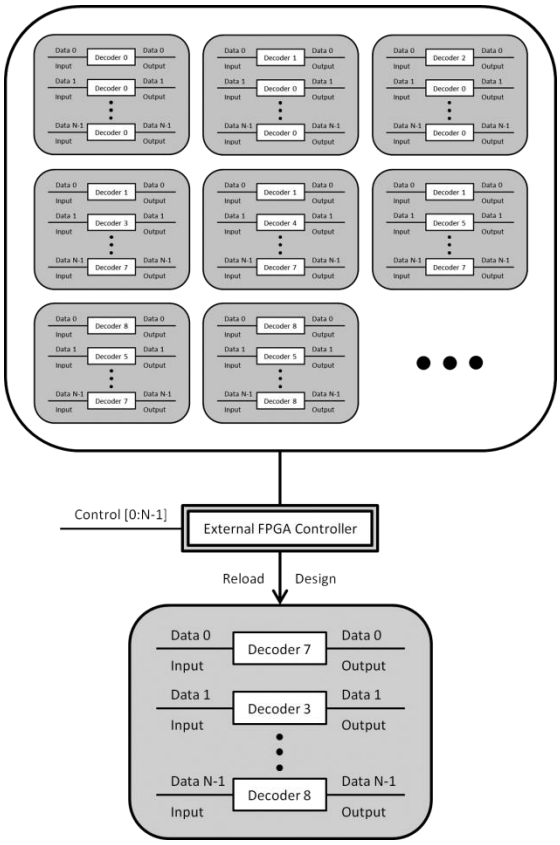

Figure 2: Block Diagram of a conventionally reconfigurable VDH Design

VDH Design 3: Partially Reconfigurable Design

Scalability can be improved by reducing the design's memory requirement. The N channels are predefined as RPs on a single static logic configuration, which is then synthesized into a full BIT file; and the M decoders are predefined as RMs, which are then synthesized into partial BIT files. The RMs are synthesized per RP, so a total of $N \cdot M$ partial BIT files and one full BIT file are generated in the process. Nonetheless, compared with M^N full BIT files, the memory requirement is significantly reduced.

While a channel's decoder is being replaced, the other $N - 1$ channels remain operational, so their data is not lost in the process. Furthermore, the reconfigured channel loses less data due to PR being faster than conventional reconfiguration. Decoders can be removed from unused channels; thus reducing power consumption and heat generation even further.

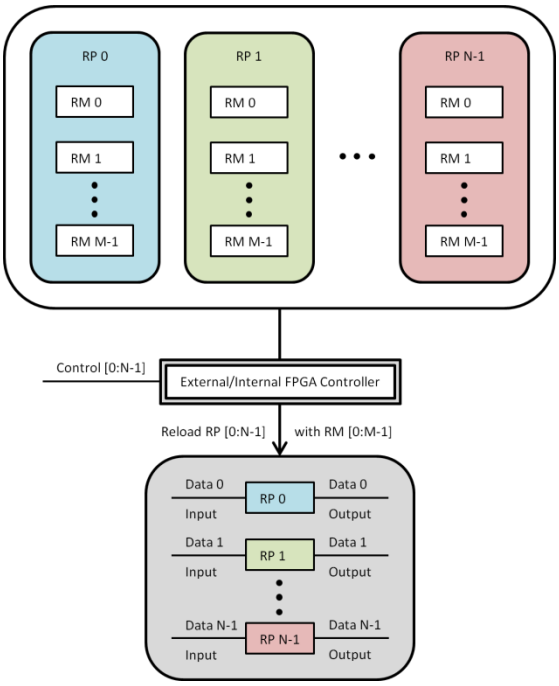

Figure 3: Block Diagram of a Partially Reconfigurable VDH Design

Pros

- Requires few FPGA resources.
- Saves memory.
- Saves more power.
- Even less inclined to overheat.

Cons

- May lose some data from the reconfigured channel.
- Intricate and hard to implement.

As shown in the example above, PR not only reduces the design's requirements, but it also keeps them low by mitigating the costs of additional channels and codecs. As N and/or M increase, the weaknesses of the first two designs become more and more prominent, whereas the later design suffers less.

3.0 Project Objective

When Linux OS is required to execute multiple processes, it enqueues them in a variant of a round-robin queue, because each CPU core may execute only one process at a time. An embedded FPGA allows the processing system (PS) to extent its processing power from the CPU to the programmable logic (PL). HWAs can be loaded to the PL, receive input data from the PS, process it, send the output data back to the PS, and then be unloaded from the PL. In this procedure one may find some resemblance to Microsoft's DLL concept; hence the term "Hardware DLL".

Of course, the PL may only contain as many HWAs as its resources allow it. Hardware DLL's solution to this problem is based upon the virtual memory paging technique. HWAs (pre-synthesized RMs) are loaded to the PL only when suitable resources (large enough empty RPs) are available. After sending their output data back to the PS, HWAs are unloaded from the PL to free the resources that they previously utilized.

Applications that were integrated with Hardware DLL can request HWAs to be loaded to the PL. If denied, the applications can try again until suitable PL resources become available, or they can send their data to be processed by the CPU. Otherwise, the applications send their data to be processed by the HWAs on the PL. Then, they can wait for the HWAs' output to become available, or they can perform other tasks while the HWAs are busy processing their data.

Project Goals:

- Make a generic HWA template.
- Design an embedded system configuration.
- Partition the FPGA resources.
- Implement the designed configuration.
- Create the device boot image.
- Prepare automation scripts and a user guide.
- Use the scripts to generate custom HWAs from the template.
- Build an application to manage the PL from user space.
- Integrate the application into a test and a demo.
- Compile the application on the embedded system.
- Boot the device and analysing the system's performance.
- Write conclusions and recommendations for future development.

4.0 Project Overview

The Two Faces of Hardware DLL:

- Hardware DLL is a toolkit that enables programmers with limited knowledge in hardware development to generate HWAs from their software, and to integrate these HWAs back into their software with relative ease.
- Hardware DLL is an FPGA management system that runs on an embedded Linux OS in real time and accelerates predefined targeted functions, while balancing the CPU's workload with the available hardware resources on the FPGA.

4.1 Generating Hardware Accelerators

Hardware development is fundamentally different from software development, so programmers may find it very difficult to create HWAs for their software. Hardware DLL provides a generic HWA template, automation scripts, and a user guide. With this toolkit (and the Vivado design suite) programmers can easily synthesize C/C++ functions into IP cores and IP cores into RMs. In fact, HWAs generated by the Hardware DLL toolkit are simply hardware implementations of the original C/C++ functions, which are designed to fit into predefined RPs on the FPGA.

4.2 Integrating the Hardware DLL Management Application

To enable the functionality of the FPGA management system, programmers include the header file of the Hardware DLL management application in their source code. This header file contains an API, through which programmers can load/unload HWAs and convey their I/O by invoking the respective API functions from their source code. Calling a targeted function directly executes its software implementation on the CPU, whereas calling it via the API executes its hardware implementation on the FPGA (provided that the resources are available). When programmers finish modifying their source code, they flash an SD card with the device boot image and copy the generated HWAs to the SD card along with their applications. They compile the integrated applications on the target platform (board) and give the SD card to end users.

4.3 Partially Reconfiguring the FPGA in Real Time

End users insert the SD card into the board, connect the board to a network, and turn it on. Once the Zynq SoC is initialized and Ubuntu is booted, end users can connect to a terminal via the network and instantiate hardware accelerated processes by executing suitable applications. When a process requests the FPGA management system to service targeted functions, it checks for available RPs and assigns them to those functions whose HWA has a matching RM. These RMs are loaded into their respective RPs and service the targeted functions that they are associated with. When a process terminates its accelerated functions, it requests the FPGA management system to remove their HWAs and free the RPs. End users are unaware of any system changes, only of the increased application performance.

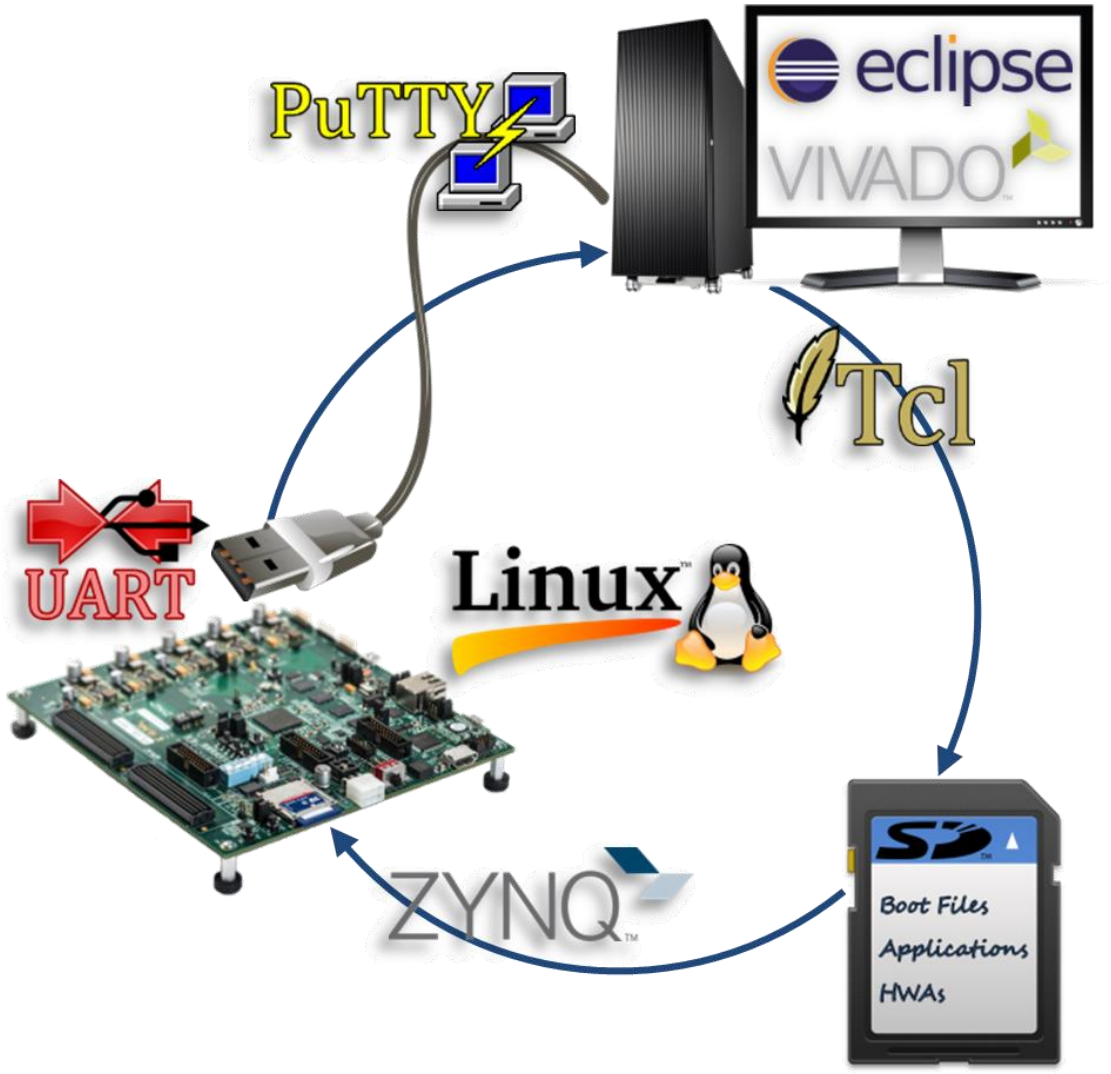

Figure 4: Flow Diagram of Hardware DLL: Project Overview

5.0 Creating the Toolkit

Consistency is an essential requirement of any automated process, and PR is not an exception. The key to RM-RP compatibility is consistent partition pins and partial routing placement, which Hardware DLL ensures by enforcing strict naming conventions. To assimilate these conventions into the toolkit, a dummy HWA was built and discarded, but its framework was kept. The toolkit reuses this framework to generate custom HWAs, which inherit the dummy HWA's conventions and consequently its partition pins and partial routing placement. The toolkit is built automatically by sourcing the "build.tcl" script.

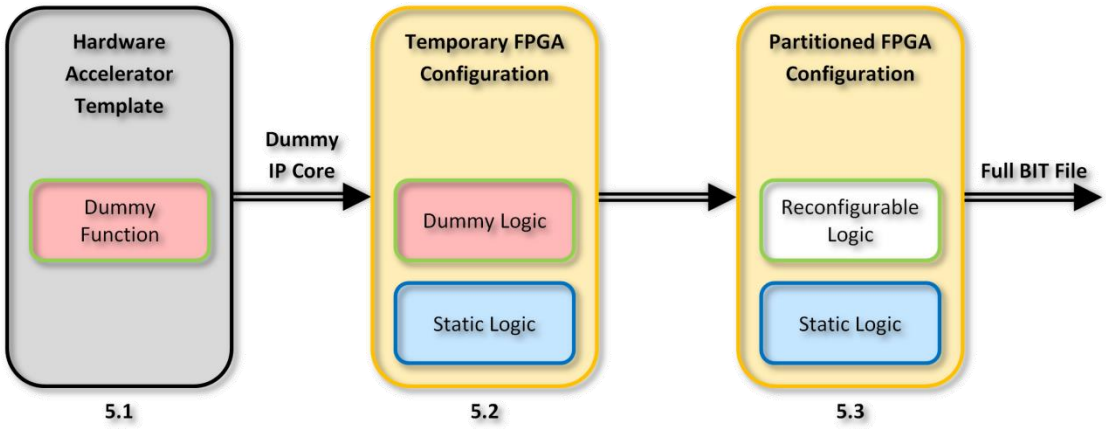

Figure 5: Flow Diagram of Hardware DLL: Creating the Toolkit

5.1 Hardware Accelerator Template

The HWA template is a Vivado high level synthesis (HLS) project that contains settings and a dummy function. Since it is used only as a placeholder for targeted functions, the dummy function's content is irrelevant. The dummy function's parameters were converted into I/O signals and bundled into a bus adapter along with the interface signals (start, valid, done, and idle). After synthesizing the HLS project, the dummy function's RTL (dummy IP core) was exported in an IP-XACT format.

5.2 Temporary FPGA Configuration

The temporary FPGA configuration is a Vivado IDE project that contains dummy logic and static logic. The dummy logic consists of 16 dummy modules, which were imported from the HWA template. The static logic consists of an advanced extensible interface (AXI) and a BRAM, which were added to an existing Parallella block design. A 4KB address space was assigned to each dummy module, and an 8KB address space was assigned to the BRAM. After synthesizing the IDE project, the FPGA configuration was ready to be partitioned.

5.3 Partitioned FPGA Configuration

The partitioned FPGA configuration is a checkpoint in the Vivado IDE project that contains reconfigurable logic and static logic. The reconfigurable logic consists of 16 RPs, which were customized for the dummy HWA. The static logic was directly derived from the temporary FPGA configuration. The synthesized floorplan was partitioned into 16 blocks, which were marked as reconfigurable. The non-partitioned resources were assigned to the static logic, and each RP was assigned to a dummy module. Implementing the configuration caused partition pins and partial routing to be placed automatically; thus making the dummy modules reconfigurable. The dummy RMs were removed, leaving behind only partition pins and partial routing. The final configuration was saved as a Vivado project checkpoint and a full BIT file.

6.0 Using the Toolkit

The toolkit is used by software developers to generate RMs from targeted functions in their source code. Unlike the user guide, which follows step-by-step instructions, this chapter provides insight and general guidelines. HWAs are built automatically by sourcing the "ide.tcl" script in their targeted function directory.

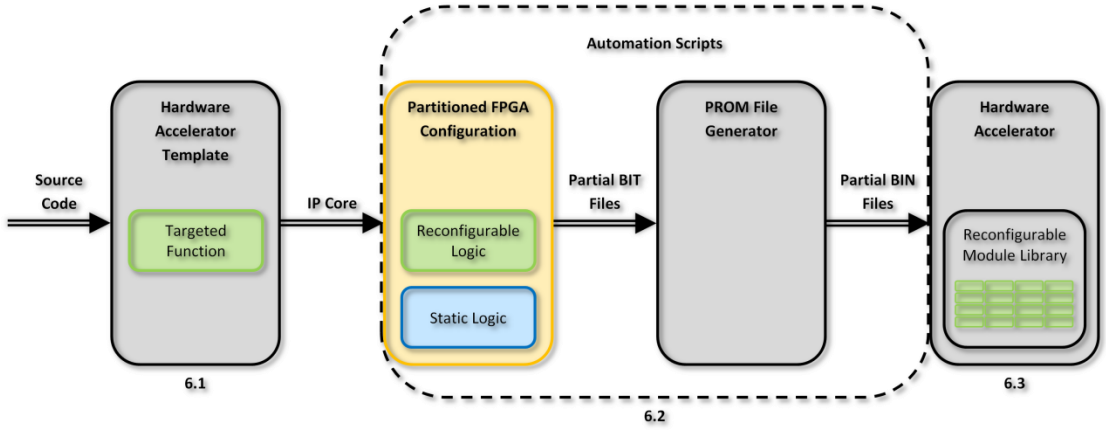

Figure 6: Flow Diagram of Hardware DLL: Using the Toolkit

6.1 Hardware Accelerator Template

Software developers copy two Tcl scripts to the targeted function's directory and replace the scripts' settings with those of the targeted function. After sourcing the script which synthesizing the HLS project, software developers should verify the utilization estimates of their IP core against the utilization constraints of the RPs. An IP core, whose utilization estimates do not meet the utilization constraints of certain RPs, is unlikely to produce partial bitstreams for those RPs. Furthermore, as pointed out by the "Limitations" section, utilization estimates are inaccurate. Even if the IP core meets the utilization constraints of all RPs, the actual utilization of the resulting RMs may exceed their RPs' utilization constraints, in which case they will also not produce partial bitstreams. Nevertheless, Hardware DLL highly recommends checking the IP core's utilization estimates before exporting the targeted function's RTL in a Verilog file format.

6.2 Automation Scripts

HWA generation is quite a repetitious process and a potential source of human error. Some of its steps (6.1 and 6.3) require human interaction, but most of the tedious labour is done behind the scenes by Tcl scripts, so the potential for human error is significantly reduced.

hls.tcl

- Opens the HLS project and generate the IP core.
- Copies the Verilog files from the IP core to the targeted function's directory.
- Replace the value of the address width with the dummy's.

ide.tcl

- Sources "hls.tcl".
- Synthesizes the IP core and writes the synthesized checkpoint.
- Implements the synthesized IP core as reconfigurable modules and writes the implemented checkpoint.
- Verifies the reconfigurable modules' compatibility and writes their partial bitstreams.
- Writes the reconfigurable modules' partial binaries.

6.3 Hardware Accelerator

The RMs (partial BIN files) are created in the library directory. Henceforth, all the data necessary to describe the HWA is located in its RM library, so the HWA and its RM library also become synonyms.

7.0 FPGA Management System (HWDLL)

The FPGA management system, affectionately named “HWDLL”, is a user space module (ADT) that equips applications with essential PL tools. Since the host platform differs from the target platform, HWDLL was developed as a Vivado SDK application project, which targets the Linux OS software platform. HWDLL can load HWAs into the FPGA, map their address spaces, and use the returned pointer to access their interface and I/O registers.

HWDLL’s header file (hwdll.h) contains the API functions, which can load/unload HWAs and convey I/O between loaded HWAs and running applications. Software developers include this header file in their source code, and replace direct calls to targeted functions with the desired sequence of calls to API functions. The integrated software is then compiled on the target platform.

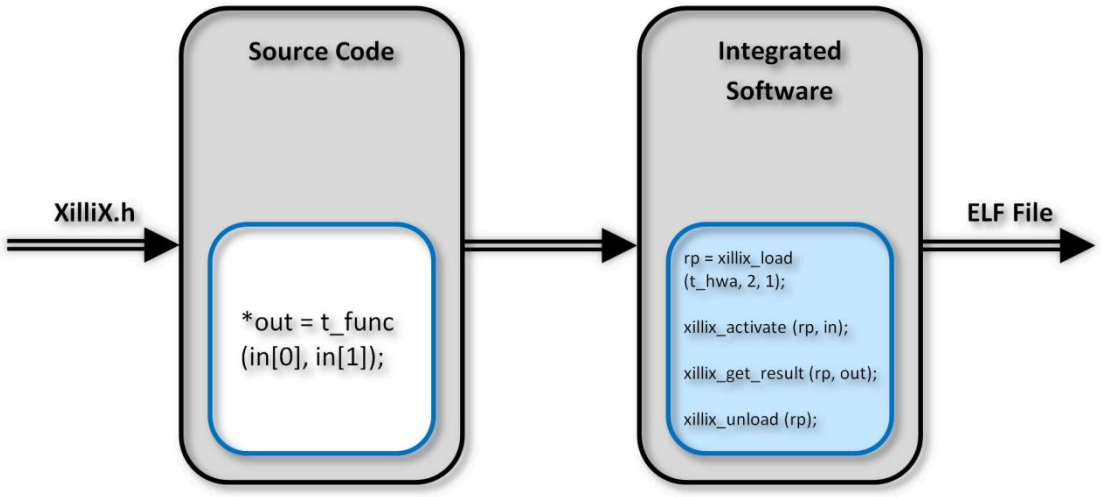

Figure 7: Flow Diagram of Hardware DLL: FPGA Management System (HWDLL)

7.1 API functions

int hwdll_create (void)

Creates the Hardware DLL data structure: Reads entries from the catalog file. Locates the available reconfigurable modules. Allocates data substructures and initializes them with the gathered data.

void hwdll_destroy (void)

Destroys the Hardware DLL data structure: Deallocates data substructures. Unloads all of the hardware accelerators from the FPGA.

int hwdll_load (int id)

Loads a hardware accelerator into a reconfigurable partition. The hardware accelerator's identifier is passed as an argument.

void hwdll_unload (int idx, bool force)

Unloads a hardware accelerator from a reconfigurable partition. Active hardware accelerators can be unloaded by force. The reconfigurable partition's index is passed as an argument.

int hwdll_load_list (const int* id_list, int* idx_list, int list_size)

Loads a list of hardware accelerators into reconfigurable partitions. A pointer to the list of hardware accelerator identifiers is passed as an input argument. A pointer to the list, in which the utilized reconfigurable partition indexes will be stored, is passed as an output argument.

void hwdll_unload_list (const int* idx_list, int list_size, bool force)

Unloads a list of hardware accelerators from reconfigurable partitions. Active hardware accelerators can be unloaded by force. A pointer to the list of reconfigurable partition indexes is passed as an argument.

int hwdll_read (int idx, int* out_list);

Reads a hardware accelerator's output. Clears the hardware accelerator's activity status. The reconfigurable partition's index is passed as an argument. A pointer to the list, in which the output parameters will be stored, is passed as an output argument. The list's size must be the same as the number of output parameters for the hardware accelerator, as stated in the catalog file. Unread output parameters are set as zeros in the list, and as "invalid" in the record file entry.

int hwdll_write (int idx, const int* in_list)

Writes a hardware accelerator's input. The reconfigurable partition's index is passed as an argument. A pointer to the list of input parameters is passed as an input argument. The list's size must be the same as the number of input parameters for the hardware accelerator, as stated in the catalog file.

int hwdll_probe (int idx)

Probes a hardware accelerator for its activity status. Clears the hardware accelerator's activity status. The reconfigurable partition's index is passed as an argument.

void hwdll_log (void)

Writes the Hardware DLL data structure's content to the log file.

8.0 Design Considerations

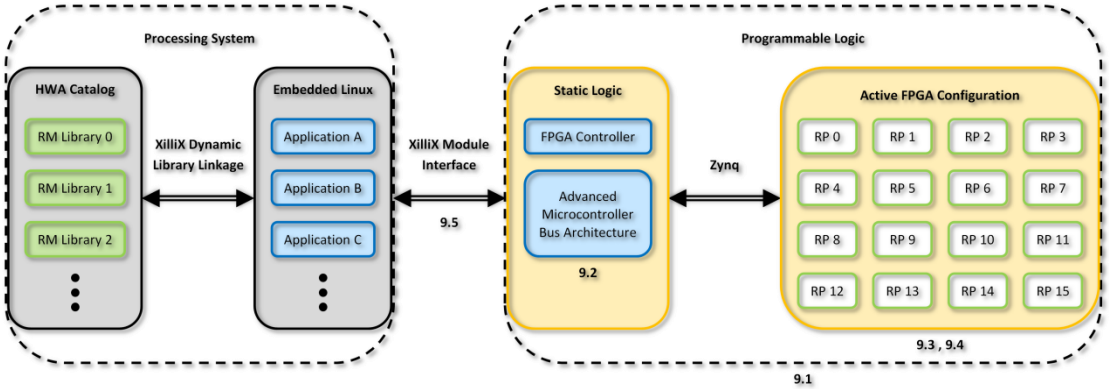

Figure 8: PS/PL Diagram of Hardware DLL

8.1 Choosing the Right Board

The Parallella board was chosen as a successor of the Xilinx ZC702 evaluation board for this project, because it has a Zynq SoC and a 16-core Epiphany processor, so it can be used in future parallel computing development. Besides, Parallella ha a readily available boot image, which comes with an Ubuntu OS.

8.2 Selecting a Suitable AMBA interconnect

Since Hardware DLL's objective is performance enhancement, data transaction speed is paramount; therefore the system's PS/PL bus must not become its bottleneck. AXI-Lite was selected to interconnect between the PS and the PL, because it is the fastest and the simplest of the 3 AXI types to work with (both in design and usage). The additional features of AXI-Full and AXI-Stream are superfluous for the purposes of this project. Furthermore, AXI-Lite is a key component in lab 4 of the HLS workshop[2], which was also used as a reference design for the project.

[2] http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-high-level-synthesis-flow-zynq.html

8.3 Deciding How to Partition the FPGA Configuration

The Zynq-7000 SoC has 2 master AXI ports, and each AXI interconnect has 64 master AXI ports; therefore the SoC can have up to $2 \cdot 64 = 128$ concurrent AXI slave peripherals (in this case, RPs). Additional AXI slave peripherals require linking multiple AXI interconnects in a hierarchical concatenation, which is not an elegant solution, and luckily there was no need for it. Hardware DLL prioritizes design simplicity over maximal resources utilization, so only one AXI interconnect was used in the embedded system configuration, and it has just 16 ports for slave peripherals.

HWA concurrency can be increased by dividing the PL into smaller partitions, but additional RPs require additional partition pins and partial routing, which reduce the RPs' sizes even further. Moreover, since HWAs have to be synthesized per RP, more RMs and consequentially more memory would be required. Still, the size of a typical partial BIT file is approximately $128KB$, so a system with 50 HWAs (for example) would require just $50 \cdot 16 \cdot 2^{17} = 100MB$ of RM storage memory.

Due to the geometric design of the FPGA fabric and the structural distribution of its resources, different resources are available to different RPs. Fortunately, this is one drawback that Hardware DLL managed to turn into a feature. To save resources and reduce internal fragmentation, differently sized RPs are used to accommodate differently sized RMs. Since resource optimization is technologically impossible anyway, the sizes of individual RPs were chosen empirically.

8.4 Assigning Address Spaces to Reconfigurable Partitions

According to table 6 of the SoC's overview[3] and table 4-1 of its technical reference manual[4], the PL can access $2GB$ of the $4GB$ address space, which is supported by the SoC. However, since Hardware DLL prioritizes design simplicity over maximal resources utilization, only $4KB$ were assigned to each RP ($16 \cdot 4 = 64KB$ in total). The SoC's registers have $4B$ (32 bit) address spaces, so each HWA can have up to $\frac{4KB}{4B} = \frac{2^{12}}{2^2} = 1024$ registers. HWAs require 2 registers for their interface signals, 3 registers for their interrupt signals, and 2 registers for each 32 bit I/O data signal. Therefore, the maximum number of integer type parameters (32 bit I/O data signals) per HWA is $\left\lfloor \frac{1024-(2+3)}{2} \right\rfloor = 509$.

[3] Xilinx – Zynq-7000 All Programmable SoC: Overview (DS109)
[4] Xilinx – Zynq-7000 All Programmable SoC: Technical Reference Manual (UG585)

8.5 Accessing Hardware Accelerators from Linux

As explained in slide 6 of the embedded Linux workshop[5], building custom drivers for Embedded Linux is a complicated and time-consuming task. There is, however, a quick workaround, which the Hardware DLL module employs. By mapping chunks of PL memory into their virtual memory, processes are able to access HWAs directly from user space. Unlike a conventional device driver, the Hardware DLL management system cannot handle interrupts or prevent simultaneous access (mutual exclusion), but it is portable and very simple, which fits well with the project's design priorities. Building a proper device driver was left as a challenge for future project developers.

[5] http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-embedded-linux-zynq.html

9.0 Current System Limitations and Future Development

As a bi-semestrial student project, Hardware DLL had its share of technical difficulties and tactical compromises. Though the main objective has been achieved, many peripheral features had to be omitted. Still, Hardware DLL was designed with the foresight for future project development. The following features were left as a challenge for other students.

- As mentioned in the "Accessing Hardware Accelerators from Linux" section, Hardware DLL employs a quick workaround that allows applications to access HWAs directly from user space. The lack of a kernel device driver prevents Hardware DLL from handling the HWAs' interrupt signals and accessing the same HWAs from different applications. Building a proper device driver for an embedded Linux OS is an essential step towards enabling these features. It would also be nice to see this driver adapted for Android OS.

- On its own, a kernel device driver is not enough to enable the desired feedback from the PL to the PS. This requires a slave AXI interconnect to relate the interrupt signals from the HWAs to Zynq. Therefore, a slave AXI interconnect has to be added to the FPGA configuration, and a bus adapter for an AXI-Lite master has to be created when generating HWAs from the generic HWA template.

- Scripts have the potential to automate the entire design flow of Hardware DLL. Combined with an options menu (and a user friendly GUI), automation scripts can build partitioned FPGA configurations for different boards, Zynq boot images and other boot files for different OSs. Another option that can be added to the menu is a choice of target functions to be multiplexed into a single IP core in order to reduce the RPs' internal fragmentation.

- The target and the host platforms can be merged by adding a video DMA core to the FPGA configuration, connecting a monitor to the board, and installing the Vivado design suite on the board's OS. Then, on-board applications would be able to use the automation scripts to generate HWAs in real time and load/unload them on demand.

10.0 References

General:

[1] *Parallel Computing* [Online]. Available: http://en.wikipedia.org/wiki/Parallel_computing

[2] *Hardware Acceleration* [Online]. Available: http://en.wikipedia.org/wiki/Hardware_acceleration

[3] *Reconfigurable Computing* [Online]. Available: http://en.wikipedia.org/wiki/Reconfigurable_computing

[4] *Xilinx Glossary* [Online]. Available: http://www.xilinx.com/company/terms

Zynq-7000:

[5] DS190: *Zynq-7000 All Programmable SoC – Overview*, Xilinx. October 8, 2014

[6] UG585: *Zynq-7000 All Programmable SoC – Technical Reference Manual*, Xilinx. November 19, 2014

[7] UG821: *Zynq-7000 All Programmable SoC – Software Developers Guide*, Xilinx. June 13, 2014

Vivado IDE/SDK:

[8] UG994: *Vivado Design Suite User Guide – Designing IP Subsystems Using IP Integrator*, Xilinx. October 16, 2014

[9] UG995: *Vivado Design Suite Tutorial – Designing IP Subsystems Using IP Integrator*, Xilinx. October 1, 2014

[10] *Advanced Embedded System Design on Zynq using Vivado* [Online]. Available: http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-adv-embedded-design-zynq.html

Vivado HLS:

[11] UG902: *Vivado Design Suite User Guide – High-Level Synthesis*, Xilinx. October 1, 2014

[12] UG871: *Vivado Design Suite Tutorial – High-Level Synthesis*, Xilinx. November 10, 2014

[13] *High-Level Synthesis Flow on Zynq using Vivado HLS* [Online]. Available: http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-high-level-synthesis-flow-zynq.html

Partial Reconfiguration:

[14] D. Koch, “Partial Reconfiguration on FPGAs – Architectures, Tools and Applications” in *Lecture Notes in Electrical Engineering (LNEE)*, vol. 153. New York, NY: Springer, 2013. DOI: 10.1007/978-1-4614-1225-0

[15] UG909: *Vivado Design Suite User Guide – Partial Reconfiguration*, Xilinx. November 19, 2014

[16] UG947: *Vivado Design Suite Tutorial – Partial Reconfiguration*, Xilinx. October 1, 2014

[17] *Partial Reconfiguration Flow on Zynq using Vivado* [Online]. Available: http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-partial-reconfiguration-flow-zynq.html

Embedded Linux:

[18] *Embedded Linux on Zynq using Vivado* [Online]. Available: http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-embedded-linux-zynq.html

[19] *Zynq Releases* [Online]. Available: http://www.wiki.xilinx.com/Zynq+Releases

[20] *Zynq-7000 Partial Reconfiguration Reference Design* [Online]. Available: http://www.wiki.xilinx.com/Zynq+7000+Partial+Reconfiguration+Reference+Design

Future Development

[21] T. Drahonovsky, M. Rozkovec and O. Novak, “Relocation of reconfigurable modules on Xilinx FPGA” in *Proceedings of the 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS 2013)*. Karlovy Vary, CZ: IEEE, 8-10 April 2013. DOI: 10.1109/DDECS.2013.6549812

[22] O. P. Mencer, *Liquid Circuits – Automated Dynamic Hardware Acceleration of Compute-Intensive Applications*. Dept. of Computing, Imperial College London

Appendix A. Terminology

A

Address

The identification of a storage location, such as a register or a memory cell.

AMBA

Advanced Microcontroller Bus Architecture. An on-chip communications standard for high performance 32 bit and 16 bit embedded microcontrollers.

API

Applications Programming Interface. A set of software libraries, developed by a particular software vendor, that allows third party software programs to interface with programs from that vendor.

ASIC

Application-Specific Integrated Circuit. An integrated circuit customized for a particular use (versus general-purpose use). For example, a chip designed solely to run a cell phone for a specific manufacturer is an ASIC. Either a full-custom circuit in which every mask is defined by the user, or a semi-custom circuit (gate array) where only a few masks are defined.

AXI

Advanced eXtensible Interface protocol. A bus protocol that is targeted at high performance, high clock frequency system designs and includes a number of features that make it very suitable for high-speed sub-micron interconnect.

B

BIT file

A bitstream file.

BITGen

BIT file generator. A program that produces a bitstream for Xilinx device configuration. BITGen takes a fully routed native circuit description (NCD) file as its input and produces a configuration bitstream.

Bitstream

A stream of data that contains location information for logic on a device, that is, the placement of configurable logic blocks (CLBs), input/output blocks (IOBs), 3-state buffer (TBUFs), pins, and routing elements. The bitstream also includes empty placeholders that are filled with the logical states sent by the device during a readback. Only the memory elements, such as flip-flops, RAMs, and CLB outputs, are mapped to these placeholders, because their contents are likely to change from one state to another. When downloaded to a device, a bitstream configures the logic of a device and programs the device so that the states of that device can be read back.

Block

A group of one or more logic functions. A schematic or symbol sheet. There are four types of blocks:

- A Composite block indicates that the design is hierarchical. A composite block is a symbol representing an underlying schematic or netlist.
- A Module block is a symbol with no underlying schematic. A module block is also referred to as a primitive.
- A Pin block represents a schematic pin.
- An Annotate block is a symbol without electrical connectivity that is used only for documentation and graphics.

C

Configuration

- A complete design that has one Reconfigurable Module for each Reconfigurable Partition. There might be many Configurations in a Partial Reconfiguration FPGA project. Each Configuration generates one full BIT file as well as one partial BIT file for each Reconfigurable Module.
- The process of loading design-specific bitstreams into one or more devices to define the functional operation of the logical blocks, their interconnections, and the chip I/O.

Constraints

Specifications for the implementation process. There are several categories of constraints: routing, timing, area, mapping, and placement constraints. Using attributes, you can force the placement of logic (macros) in CLBs, the location of CLBs on the chip, and the maximum delay between flip-flops. PAR does not attempt to change the location of constrained logic.

Core

A predefined function such as a processor or a bus interface that is typically licensed from the software developer. Cores can be implemented directly in silicon, either in fixed logic or programmable logic devices, and saves chip designers time during product development. Synonymous with Intellectual Property.

D

Design

A netlist (elaborated RTL or synthesized), a constraint set, and a target device. Each project netlist can support multiple designs using different constraints or devices.

Design Implementation

The actual implementation of the design from low-level components expressed in bits. This is different from the functional specification of the design, which refers to the definition of the design or circuit function.

Device

An integrated circuit or other solid-state circuit formed in semiconducting materials during manufacturing. Each Xilinx architecture family contains specific devices.

E

ELF File

Executable and Linkable Format file.

Evaluation Kit

A product that bundles a specific set of Xilinx Targeted Design Platform components into a single orderable product.

F

Floorplanning

- The process of choosing the best grouping and connectivity of logic in a design.
- The process of manually placing blocks of logic in an FPGA where the goal is to increase density, routability, or performance.

Flow

An ordered sequence of processes that are executed to produce an implementation of a design.

FPGA

Field Programmable Gate Array. A class of integrated circuits pioneered by Xilinx in 1984. An integrated circuit device or “programmable platform” that can be programmed in the field after being manufactured, providing electronic product manufacturers with additional design flexibility. Unlike application-specific chips, FPGAs allow engineers to make changes very late in the design cycle and even upgrade products with new functionality after manufacture. FPGA applications include fast counters, fast pipelined designs, register intensive designs, and battery powered multi-level logic.

H

HDL

Hardware Description Language. A language that describes circuits in textual code. The two most widely accepted HDLs are VHDL and Verilog. HDL describes designs in a technology independent manner using a high level of abstraction.

I

I/O

Input/Output. The physical connections, and the various electrical standards, for getting signals on and off a chip.

Implementation

The mapping, placement and routing of a design. A phase in the design process during which the design is placed and routed.

Instance

One specific gate or hierarchical element in a design or netlist. The term “symbol” often describes instances in a schematic drawing. Instances are interconnected by pins and nets. Pins are ports through which connections are made from an instance to a net. A design that is flattened to the lowest level constituents is described using primitive instances.

Instantiation

The act of placing a symbol that represents a primitive or a macro in a design or netlist.

IP

Intellectual Property. A function or algorithm that can be implemented in programmable logic with a defined interface (input, output, and control) and that behaves deterministically based on this interface. IP can be delivered as source code or as an encrypted netlist. Synonymous with Core.

Interconnect

Silicon in programmable logic that is devoted to connecting memory elements on the chip to create a logic circuit.

I/O Port

User I/Os that are assigned to physical package pins. Each I/O signal is defined as a port.

M

Mapping

The process of assigning a design's logic elements to the specific physical elements that actually implement logic functions in a device.

N

Netlist

A text description of the circuit connectivity. It is basically a list of connectors, a list of instances, and, for each instance, a list of the signals connected to the instance terminals. In addition, the netlist contains attribute information.

P

PR

Partial Reconfiguration. A method of modifying a subset of logic in an operating FPGA design by downloading a partial bitstream.

Partition

A logical section of the design, defined by the user at a hierarchical boundary, to be considered for design reuse. A Partition is either implemented as new or preserved from a previous implementation. A Partition that is preserved maintains not only identical functionality but also identical implementation.

Partition Pin

The logical and physical connection between static logic and reconfigurable logic. Partition pins are automatically created for all Reconfigurable Partition ports.

Partitioning

The process of splitting a single design among multiple devices.

Pblock

Physical Block. A Pblock is defined in the Vivado design suite during floorplanning. Traditionally, a single or group of logic instances are assigned to a Pblock. The Pblock can have an area, such as a rectangle defined on the FPGA device, to constrain the logic. Pblocks may be specified with specific RANGE types to contain various types of logic only (such as SLICE, RAM/MULT, and DSP). Pblocks can be defined with multiple rectangles to enable non-rectangular shapes to be created, such as 'L' shaped and 'T' shaped.

Pin

- A symbol pin, also referred to as an instance pin, is the connection point of an instance to a net.
- A package pin is a physical connector on an integrated circuit package that carries signals into and out of an integrated circuit.

PL

Programmable Logic in the Zynq-7000 All Programmable SoC. Equivalent to the FPGA in the 7 series devices.

Placing

The process of assigning physical device cell locations to the logic in a design.

PROM file

One or more BIT files (bitstreams) formed into one or more datastreams. The file is formatted in one of three industry-standard formats: Intel MCS86 HEX, Tektronics TEKHEX, or Motorola EXORmacs. The PROM file includes headers that specify the length of the bitstreams as well as all the framing and control information necessary to configure the FPGAs. It can be used to program one or more devices.

PROMGen

PROM file generator. A Xilinx program that formats a BITGen-generated configuration bitstream (BIT file) into a PROM format file. The PROM file contains configuration data for the FPGA.

PS

Processing System. The processor portion of the Zynq-7000 All Programmable SoC.

R

Reconfigurable Computing

A methodology of using programmable logic devices in a system design such that the hardware based logic can be changed to perform various tasks. Benefits include the use of fewer components, less power, and the flexibility that bring about. Also allows networked equipment in the field to be upgraded or repaired remotely.

Reconfigurable Logic

Any logical element that is part of a Reconfigurable Module. These logical elements are modified when a partial BIT file is loaded. Many types of logical components can be reconfigured such as LUTs, flip-flops, BRAM, and DSP blocks.

RM

Reconfigurable Module. The netlist or HDL description that is implemented within a Reconfigurable Partition. Multiple Reconfigurable Modules will exist for a Reconfigurable Partition.

RP

Reconfigurable Partition. An attribute set on an instantiation that defines the instance as reconfigurable. The Reconfigurable Partition is the level of hierarchy within which different Reconfigurable Modules are implemented. Tcl commands such as opt_design, place_design and route_design detect the HD.RECONFIGURABLE property on the instance and process it correctly.

Reference Design

A technical blueprint of a system that contains essential elements intended for others to copy, enhance and modify for a specific application.

Register

A digital circuit that stores bits (1s and 0s).

Routing

The process of assigning logical nets to physical wire segments in the FPGA that interconnects logic cells.

RTL

Resistor Transistor Logic.

S

Scalable Optimized Architecture

Describes the fact that all 7 series FPGA device families, from low-end to ultra-high end, are built with the same core building blocks of logic, memory, DSP, clocking, etc.

Script

A series of commands that automatically execute a complex operation such as the steps in a design flow.

SoC

System-on-Chip. A chip that holds the necessary hardware and electronic circuitry (programmable logic, memory, processing, peripheral interfaces, clocking, and I/O) for a complete system.

Synthesis

A process that starts from a high level of logic abstraction (typically Verilog or VHDL) and automatically creates a lower level of logic abstraction using a library containing primitives.

Static Logic

Any logical element that is not part of a Reconfigurable Partition. The logical element is never partially reconfigured and is always active when Reconfigurable Partitions are being reconfigured. Static logic is also known as Top-level logic.

Static Design

The part of the design that does not change during partial reconfiguration. The static design includes the top level and all modules not defined as reconfigurable. The static design is built with static logic and static routing.

T

Targeted Design Platform

A Xilinx-specific term that describes the integration of five key components into a common development and run-time environment for FPGA designs, including:

- Design tools supporting different design methodologies.
- Boards.
- Intellectual property cores.
- FPGA silicon devices.
- Targeted reference designs.

Targeted Design Platforms enable software and hardware designers alike to leverage common design methodologies, development tools, and run-time platforms. This allows them to spend less time developing the infrastructure of an application and more time building differentiating features into the end application.

Tcl

Tool Command Language. A scripting language used for rapid prototyping, scripted applications, graphical user interfaces, and testing.

U

UART

Universal Asynchronous Receiver-Transmitter.

V

Verification

The process of reading back the configuration data of a device and comparing it to the original design to ensure that all of the design was correctly received by the device.

Appendix B. Project Construction Guide

- Go to: https://www.parallella.org/quick-start
- Follow the instructions to flash an SD card with “Ubuntu 14.04 Parallella (Zynq7020) headless image”.

- Extract “hwdll.zip” to <extraction directory>. This author recommends a short pathname, such as “C:\hwdll”, because some branches of the hwdll directory tree are quite long. For more information on the subject, please read: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365247%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396#maxpath
- Open Vivado 2015.4, type “cd <extraction directory>” in the Tcl console, and hit enter. From now on, the <extraction directory> will be referred to simply as “.”.
- Type “source ./scripts/build.tcl” in the Tcl console and hit enter. Hit it as hard as you can and go make some tea. The script runs about 20-25 minutes on a very powerful 2014 PC (i7 CPU, 32GB RAM, etc.).

Don’t close the Tcl console yet, because you might what to compare your console’s log with the one in “./logs/build”.

By the time you finish drinking your tea, the “.” directory should look like this:

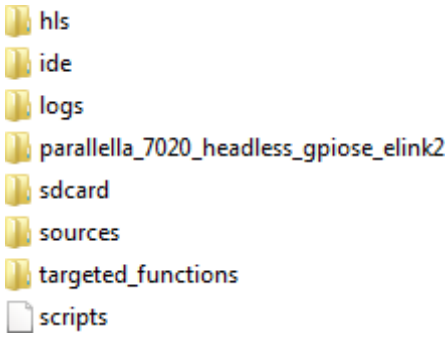

Figure 9: "." Directory

parallella_7020_headless_gpiose_elink2

Source Parallella Project

hls

Hardware Accelerator Template

ide

Temporary FPGA Configuration

Appendix C. Hardware Accelerator Generation Guide

The name of your targeted function is referred to as <name>.
Your targeted function's prototype should look like this: int targeted_function (int input_000, int input_001, … int* output_000, int* output_001, …)
Inputs should come before outputs. Examples can be found in: "./targeted_functions/<name>/targeted_function.c"

- Copy the source file of <name> to: "./targeted_functions/<name>/targeted_function.c"
- Copy "./targeted_functions/numbers/hls.tcl" and "./targeted_functions/numbers/ide.tcl" to "./targeted_functions/<name>/hls.tcl" and "./targeted_functions/<name>/ide.tcl" respectively.
- Open the scripts and modify their settings (under "Settings" section) to fit your HWA.
- Open Vivado 2015.4 and type the following commands in the Tcl console:
  cd .
  source ./targeted_functions/<name>/ide.tcl
- If "place_design" fails, copy "./targeted_functions/multiplier/ide.tcl" to "./targeted_functions/<name>/ide.tcl", modify settings, and source ./targeted_functions/<name>/ide.tcl again.

Don't close the Tcl console yet, because you might what to compare your console's log with the one in "./logs/numbers" or "./logs/multiplier".

Remember that "place_design" will fail if you IP core's utilization estimates do not meet the utilization constraints of certain RPs.

Your RM library directory is: "./sdcard/hwdll/library/<name>". Open "./sdcard/hwdll/catalog" and add an entry line to describe your HWA to hwdll. Make sure you read the "Macro Definitions" section in "./sdcard/hwdll/hwdll.h and the "./sdcard/hwdll/README" file before you read the instructions in the catalog file.

Extract "hwdll_complete.zip" to use as a reference of the complete project and its directory tree.

Appendix D. Software Guide

Macro Definitions

The header file "./sdcard/hwdll/hwdll.h" contains macro definitions, which can be modified to suit your settings. Though these definitions are self-explanatory, you might wonder where they came from. Read the "./sdcard/hwdll/README" file first.

RP_OFFSET and RP_RANGE: Open the block design of "./ide/ide.xpr" and go to the "Address Editor" tab. Notice that reconfigurable_module_00 has an offset address - this is the RP_OFFSET. Also notice that each reconfigurable module has a range – this is the RP_RANGE. If you read the "./sdcard/hwdll/README" file, as I'm sure you did, you know that $4K = 0x1000$.

IO_OFFSET and IO_RANGE: Open "./hls/<name>/impl/ip/drivers/ targeted_function_v1_0/src/xtargeted_function_hw.h". Notice that the address of the first register which is not a control, interrupt, or return signal is $0x18$ – this is the IO_OFFSET. Also notice that each of the I/O signals, which follow IO_OFFSET, is assigned with two 32-bit registers (one for control, and one for data) – hence IO_RANGE.

PR_FILE, CFG_FILE, and MEM_FILE: Turn on Parallella and connect to its Ubuntu terminal, which has a standard Unix-like filesystem. PR_FILE, CFG_FILE, and MEM_FILE are the relevant pathnames in Ubuntu 14.04. Though unlikely, our OS may have different pathnames.

Compiling Hardware DLL

- When you're done generating HWAs, copy the content of "./sdcard" to the SD card, which you flashed earlier.
- Insert the SD card into Parallella, connect Parallella to the network via its Ethernet adapter, and turn it on.
- Use your favourite SSH client to access Parallella's Ubuntu terminal from a PC. This author recommends "MobaXterm".
- Open "./scripts" and type the commands under "Static IP Address" into the terminal.
- Restart your SSH client with the static IP address and type the commands under "HWDLL Self-Test" into the terminal.

Remember that Hardware DLL needs superuser permissions to access the PR_FILE, CFG_FILE, and MEM_FILE files.