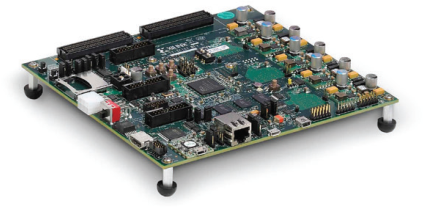


|  |
| --- |
| Hardware DLL |
| Real Time Partial Reconfiguration Management of FPGA by OS |



|  |
| --- |
| Submitted by Alon Reznik and Anton Vainer |
| Supervised by Oz Shmueli and Ina Rivkin |

Bi‑Semestrial Student Project | 2014 ‑ 2015 | Final Report (Part 1)

Table of Contents

[1.0 Introduction 1](#_Toc424301595)

[2.0 Partial Reconfiguration 2](#_Toc424301596)

[2.1 Benefits 2](#_Toc424301597)

[2.2 Limitations 3](#_Toc424301598)

[2.3 Example: Video Decoding Hub 3](#_Toc424301599)

[3.0 Project Objective 7](#_Toc424301600)

[4.0 Project Overview 8](#_Toc424301601)

[4.1 Generating Hardware Accelerators 8](#_Toc424301602)

[4.2 Integrating the Hardware DLL Module 8](#_Toc424301603)

[4.3 Partially Reconfiguring the FPGA in Real Time 9](#_Toc424301604)

[5.0 Creating the Toolkit 10](#_Toc424301605)

[5.1 Hardware Accelerator Template 10](#_Toc424301606)

[5.2 Temporary FPGA Configuration 11](#_Toc424301607)

[5.3 Partitioned FPGA Configuration 11](#_Toc424301608)

[6.0 Using the Toolkit 12](#_Toc424301609)

[6.1 Hardware Accelerator Template 12](#_Toc424301610)

[6.2 Automation Scripts 13](#_Toc424301611)

[6.3 Hardware Accelerator 13](#_Toc424301612)

[7.0 FPGA Management System (XilliX) 14](#_Toc424301613)

[7.1 API functions 15](#_Toc424301614)

[8.0 Booting Hardware DLL from an SD Card 17](#_Toc424301615)

[8.1 Zynq Boot Image 17](#_Toc424301616)

[8.2 SD Card 18](#_Toc424301617)

[9.0 Design Considerations 19](#_Toc424301618)

[9.1 Choosing the Right Evaluation Board 19](#_Toc424301619)

[9.2 Selecting a Suitable AMBA interconnect 19](#_Toc424301620)

[9.3 Deciding How to Partition the FPGA Configuration 20](#_Toc424301621)

[9.4 Assigning Address Spaces to Reconfigurable Partitions 20](#_Toc424301622)

[9.5 Accessing Hardware Accelerators from Linux 21](#_Toc424301623)

[10.0 Performance Analysis 22](#_Toc424301624)

[11.0 Current System Limitations and Future Development 23](#_Toc424301625)

[12.0 References 24](#_Toc424301626)

[Appendix A. Terminology 26](#_Toc424301627)

[Appendix B. User Guide 38](#_Toc424301628)

[Appendix C. Project Construction Guide 39](#_Toc424301629)

Table of Figures

[Figure 1: Block Diagram of a non‑reconfigurable VDH Design 4](#_Toc424301630)

[Figure 2: Block Diagram of a conventionally reconfigurable VDH Design 5](#_Toc424301631)

[Figure 3: Block Diagram of a Partially Reconfigurable VDH Design 6](#_Toc424301632)

[Figure 4: Flow Diagram of Hardware DLL: Project Overview 9](#_Toc424301633)

[Figure 5: Flow Diagram of Hardware DLL: Creating the Toolkit 10](#_Toc424301634)

[Figure 6: Flow Diagram of Hardware DLL: Using the Toolkit 12](#_Toc424301635)

[Figure 7: Flow Diagram of Hardware DLL: FPGA Management System (XilliX) 14](#_Toc424301636)

[Figure 8: Base Addresses of the Reconfigurable Partitions 16](#_Toc424301637)

[Figure 9: Offset Addresses of a Hardware Accelerator 16](#_Toc424301638)

[Figure 10: Flow Diagram of Hardware DLL: Booting Hardware DLL from an SD Card 17](#_Toc424301639)

[Figure 11: PS/PL Diagram of Hardware DLL 19](#_Toc424301640)

# 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.

# 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.

## 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 amount 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.

## 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]](#footnote-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.

## 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 input channels for receiving up to 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 known video codecs, whose index is transmitted on the channel’s control bus. The following are 3 possible FPGA‑based VDH designs.

VDH Design 1: Non‑Reconfigurable Design

High efficiency can be achieved by implementing a single VDH configuration with all 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 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 channels lose some data every time a new video stream is sent on one of them. Moreover, there are 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. | Figure 2: Block Diagram of a conventionally reconfigurable VDH Design |
| Cons   * Loses data from all channels. * Wastes memory. |

VDH Design 3: Partially Reconfigurable Design

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

While a channel’s decoder is being replaced, the other 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 and/or increase, the weaknesses of the first two designs become more and more prominent, whereas the later design suffers less.

# 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 a Zynq boot image.
* Prepare automation scripts and a user guide.
* Use the scripts to generate custom HWAs from the template.
* Build a PetaLinux module to access the PL from user space.
* Integrate the module into a test and demo application.
* Cross‑compile the application for the embedded system.
* Boot the device and analysing the system’s performance.
* Write conclusions and recommendations for future development.

# 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.

## 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.

## Integrating the Hardware DLL Module

To enable the functionality of the FPGA management system, programmers include the header file of the Hardware DLL module in their source code. This header file contains the FPGA management system’s API, so 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 cross‑compile the integrated applications on the host platform (PC) for the target platform (board).

## Partially Reconfiguring the FPGA in Real Time

End users copy the generated HWAs and the cross‑compiled applications to an SD card along with the Hardware DLL boot files, insert the SD card into the board, and power it on. Once the FPGA is initialized and PetaLinux is booted, end users can instantiate processes by executing the copied 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 |

# 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.

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

## Hardware Accelerator Template

The HWA template is a Vivado high level synthesis (HLS) project that contains a single CPP file with 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.

## 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, which was added from Vivado’s default IP catalog, consists of a Zynq block with DDR and fixed I/O ports, a reset block, and an advanced extensible interface (AXI) block with 16 master ports. A 64KB address space was assigned to each dummy module, and therefore 16 address bits were assigned to each AXI I/O port. After synthesizing the IDE project, the FPGA configuration was ready to be partitioned.

## 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 17 blocks, 16 of which were marked as reconfigurable. The static partition was 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.

# 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.

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

## Hardware Accelerator Template

Software developers start by replacing the content and parameters of the dummy function with those of the targeted function. They convert the parameters into I/O signals and bundle them into a bus adapter along with the interface signals. After synthesizing the HLS project, software developers should verify the utilization estimates of their IP core against the utilization constraints of the RPs, which are listed in the user guide. 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.

## 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 and batch files, so the potential for human error is significantly reduced.

### fix\_16.tcl

Since the automation scripts are used in Vivado’s non‑project batch flow, the IP core’s low‑level HDL files are explicitly included in its top‑level HDL file. The address width of the IP core’s AXI I/O ports is then extended to 16 bits, to match the AXI I/O ports of the FPGA configuration.

### make\_dcp.tcl

The IP core’s top‑level HDL file is resynthesized and saved as a Vivado project checkpoint, which is loaded into all of the RPs in the partitioned FPGA design. The design is then reimplemented and verified against the original, to confirm partition pins, partial routing, and AXI I/O compatibility. After the verification is passed, the RMs, which meet their RPs’ utilization constraints, are saved as partial BIT files.

### make\_bin.bat

Unlike partial BIN files, partial BIT files do not include configuration data for the PR controller, so they are converted into partial BIN files by PROM file generator (PROMGen). Each resulting partial BIN file contains all of its RM’s information; hence RMs and partial BIN files are synonyms.

## Hardware Accelerator

Software developers copy the RMs (partial BIN files) to a folder called “RM library”. 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.

# FPGA Management System (XilliX)

The FPGA management system, affectionately named “XilliX”, is a user space module (ADT) that equips applications with essential PL tools. Since the host platform differs from the target platform, XilliX was developed as a Vivado SDK application project, which targets the ZC702 hardware platform and the Linux OS software platform. XilliX can load HWAs into the FPGA, map their address spaces, and use the returned pointer to access their interface and I/O registers.

XilliX’s header file (XilliX.h) contains the module’s 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 cross‑compiled by Vivado SDK, to generate the application’s ELF file.

|  |
| --- |
| Figure 7: Flow Diagram of Hardware DLL: FPGA Management System (XilliX) |

## API functions

### void xillix\_initialize (void);

Initializes XilliX by creating 16 files to represent the 16 RPs, and storing them on the SD card. To indicate that the RPs are empty, a negative Boolean is written into each file. It is followed by 2 zero‑valued integers, which would later denote the number of input and output parameters respectively.

### int xillix\_load (const char\* hwa\_repo\_path, const int input\_param\_num, const int output\_param\_num);

Searches the given HWA’s repository for an RM that fits an available RP. If a match is found, the RM replaces its RP’s content, a positive Boolean and the HWA’s number of I/O parameters replace the respective file’s content, and the utilized RP’s index is returned. Otherwise, a negative value is returned. The actual PR process is done in 2 steps. First, the FPGA controller is notified that the next bitstream it receives is a partial one. This is done by writing “1” into the “is\_partial\_bitstream” proc file. Then, the partial BIN file’s content is written into the “xdevcfg” proc file, and thus the HWA is loaded. The loaded HWA remains idle until activated.

### void xillix\_activate (const int rp\_idx, const long\* input\_params);

Activates a loaded HWA by sending it its input parameters. The RPs’ base addresses were automatically generated during the instantiation of the HLS modules in the FPGA configuration, and the registers’ offset addresses were automatically generated during the exportation of the custom HWAs’ RTL from the HWA template. Therefore, the HWAs’ register addresses can be easily calculated and mapped into the application’s virtual memory. The HWA’s input parameters are written directly into the mapped memory, and the “start” bit of the interface register is turned on.

|  |  |
| --- | --- |
| Figure 8: Base Addresses of the Reconfigurable Partitions | Figure 9: Offset Addresses of a Hardware Accelerator |

### bool xillix\_check\_result (const int rp\_idx);

Checks whether the HWA finished its operation. The indexed RP’s memory is mapped into the application’s virtual memory, the “done” bit of the interface register is read, and its Boolean value is returned.

### void xillix\_get\_result (const int rp\_idx, long\* output\_params);

Returns the HWA’s output parameters after it finishes its operation. The indexed RP’s memory is mapped into the application’s virtual memory. When the “done” bit of the interface register turns on, the HWA’s output registers are read, and their values are returned via the given pointer.

### void xillix\_unload (const int rp\_idx);

Replaces the content of the file, whose index is given, by a negative Boolean and 2 zero‑valued integers. The actual RM remains in its RP.

### void xillix\_terminate (void);

Terminates XilliX by deleting the 16 files, which represent the 16 RPs. It does not change any PL configuration.

# Booting Hardware DLL from an SD Card

All the files that are necessary to boot PetaLinux on a Zynq‑7000 SoC are found in repositories called “Zynq releases”[[2]](#footnote-2). However, Hardware DLL uses a custom FPGA configuration, which requires a custom Zynq boot image.

|  |
| --- |
| Figure 10: Flow Diagram of Hardware DLL: Booting Hardware DLL from an SD Card |

## Zynq Boot Image

The Zynq boot image is a BIN file that contains information on how to boot the board. This information is stored in 3 files:

First stage boot loader (fsbl.elf) – Loads the second stage boot loader and the initial FPGA configuration.

Second stage boot loader (u‑boot.elf) – Initializes the PS.

Initial FPGA configuration (init.bit) – Initializes the PL.

The first and second stage boot loaders were obtained from the Zynq release that matches Vivado’s version. The initial FPGA configuration is the full BIT file that was generated in section ‎5.3. When all 3 components of the Zynq boot image were ready, they were packaged into a BIN file by the Zynq boot image creation tool in Vivado SDK.

## SD Card

Boot Files:

Zynq boot image (BOOT.bin) – Initializes the PS and the PL.

Linux kernel image (uImage) – Container for the Linux OS kernel files.

Linux root file system (uramdisk.image.gz) – The mounted file system.

Device tree blob (devicetree.dtb) – Describes the hardware to the OS.

All of the Hardware DLL boot files, except the Zynq boot image, were obtained from the Zynq release that matches Vivado’s version.

Applications and HWAs are added to the SD card by end users.

The board’s UART port is connected to a PC, which runs PuTTY terminal emulator. When it boots from the SD card, the board initializes its PS and PL according to the Zynq boot image data. After the startup process is finished, an emulated PetaLinux terminal is opened by PuTTY. End users mount the SD card and start the applications. The associated RM libraries are “dynamically linked” by the applications at run time.

# Design Considerations

|  |
| --- |
| Figure 11: PS/PL Diagram of Hardware DLL |

## Choosing the Right Evaluation Board

The Xilinx ZC702 evaluation board was chosen for this project, because it is one of the 3 boards that are supported by the Zynq releases (the other two boards are the ZC706 and the ZedBoard). Pre‑compiled kernels of Arch Linux, Ubuntu, and even Android are also available for the ZC702 board, so it can be used in future project development. Besides, the Zynq‑7000 PR reference design[[3]](#footnote-3) is built specifically for the ZC702 board, making it the obvious choice for Hardware DLL.

## 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[[4]](#footnote-4), which was also used as a reference design for the project.

## Deciding How to Partition the FPGA Configuration

The Zynq‑7000 SoC has master AXI ports, and each AXI interconnect has master AXI ports; therefore the SoC can have up to 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 , so a system with HWAs (for example) would require just of RM storage memory.

Due to the geometric design of the FPGA fabric and the structural distribution of its resources, different amounts of 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.

## Assigning Address Spaces to Reconfigurable Partitions

According to table 6 of the SoC’s overview[[5]](#footnote-5) and table 4‑1 of its technical reference manual[[6]](#footnote-6), the PL can access of the address space, which is supported by the SoC. However, since Hardware DLL prioritizes design simplicity over maximal resources utilization, only were assigned to each RP ( in total). The SoC’s registers have (32 bit) address spaces, so each HWA can have up to registers. HWAs require register for their interface signals, registers for their interrupt signals, and registers for each 32 bit I/O data signal. Therefore, the maximum amount of integer type parameters (32 bit I/O data signals) per HWA is .

## Accessing Hardware Accelerators from Linux

As explained in slide 6 of the embedded Linux workshop[[7]](#footnote-7), building custom drivers for PetaLinux 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 module 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.

# Performance Analysis

Three algorithms tested without any improvements using the default HLS settings.

The results:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| accelerator | time SW | time HW | difference | % saved | inputs | notes |
| Fibonacci | 3.15004 | 1.234562 | 1.915478 | 60.81 | 123456789 |  |
| prime | 0.393612 | 2.03907 | -1.64546 | -418.04 | 1234 |  |
| GCD |  |  |  |  |  | algorithm to fast to compare |

Fibonacci algorithm was solved by HLS in a way that benefits HW, from the results we can see that the HW is 60% faster.

Prime algorithm is match faster on SW that on HW (HW is 418% slower). This might be because HLS default solution is not optimized well for HW or the algorithm itself is faster on SW.

GCD algorithm got results in single micro seconds on both SW and HW regardless of input size so it's not comparable.

# Current System Limitations and Future Development

As the first part of 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. While some features will be added and improved in the second part of the project, the following are 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 other OSs (Ubuntu, Android, etc.) and other boards (ZedBoard, Parallella, etc.).

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.

Batch files and Tcls 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, and perhaps even the custom HWAs’ IP cores. 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.

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:

1. DS190: Zynq‑7000 All Programmable SoC – Overview, Xilinx. October 8, 2014
2. UG585: Zynq‑7000 All Programmable SoC – Technical Reference Manual, Xilinx. November 19, 2014
3. UG821: Zynq‑7000 All Programmable SoC – Software Developers Guide, Xilinx. June 13, 2014

Vivado IDE/SDK:

1. UG994: Vivado Design Suite User Guide – Designing IP Subsystems Using IP Integrator, Xilinx. October 16, 2014
2. UG995: Vivado Design Suite Tutorial – Designing IP Subsystems Using IP Integrator, Xilinx. October 1, 2014
3. 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:

1. UG902: Vivado Design Suite User Guide – High-Level Synthesis, Xilinx. October 1, 2014
2. UG871: Vivado Design Suite Tutorial – High-Level Synthesis, Xilinx. November 10, 2014
3. 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:

1. 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
2. UG909: Vivado Design Suite User Guide – Partial Reconfiguration, Xilinx. November 19, 2014
3. UG947: Vivado Design Suite Tutorial – Partial Reconfiguration, Xilinx. October 1, 2014
4. 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:

1. Embedded Linux on Zynq using Vivado [Online]. Available: <http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-embedded-linux-zynq.html>
2. Zynq Releases [Online]. Available: <http://www.wiki.xilinx.com/Zynq+Releases>
3. Zynq‑7000 Partial Reconfiguration Reference Design [Online]. Available: <http://www.wiki.xilinx.com/Zynq+7000+Partial+Reconfiguration+Reference+Design>

Future Development

1. 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
2. O. P. Mencer, Liquid Circuits – Automated Dynamic Hardware Acceleration of Compute-Intensive Applications. Dept. of Computing, Imperial College London
3. 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.

1. User Guide

A file named how\_to.docx is included.

1. Project Construction Guide

A file named: “Make it yourself guide.docx” is included.

1. Xilinx – Vivado Design Suite User Guide: Partial Reconfiguration (UG909) [↑](#footnote-ref-1)
2. <http://www.wiki.xilinx.com/Zynq+Releases> [↑](#footnote-ref-2)
3. <http://www.wiki.xilinx.com/Zynq+7000+Partial+Reconfiguration+Reference+Design> [↑](#footnote-ref-3)
4. <http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-high-level-synthesis-flow-zynq.html> [↑](#footnote-ref-4)
5. Xilinx – Zynq‑7000 All Programmable SoC: Overview (DS109) [↑](#footnote-ref-5)
6. Xilinx – Zynq‑7000 All Programmable SoC: Technical Reference Manual (UG585) [↑](#footnote-ref-6)
7. <http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-embedded-linux-zynq.html> [↑](#footnote-ref-7)