RAMP Blue in RDL

by Jue Sun

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor J. Wawrzynek

Research Adviser

(Date)

* * * * * * *

Professor D. Patterson

Second Reader

(Date)

Contents

List of Figures

1 Introduction

To improve system performance, the industry and the academic worlds have turned away from high speed single-core systems. Despite this shift, there has been little exploration of multi-core systems with more than 64 processors. The Research Accelerator for Multiple Processors (RAMP) project was started to enable research on multi-core operating systems, compilers, debuggers, programming languages and scientific libraries through the creation of one or more multi-core simulators built in FPGAs [2].

The desire for very fast, nearly interactive, and cycle accurate simulators led to the choice of a hardware based simulator. ASICs have long turnaround time and are very inconvenient to debug, not to mention having a huge capital outlay. As silicon technology improves, it has become possible to squeeze many small embedded processors on to a Field-Programmable Gate Array (FPGA) without losing the convenience of reconfigurability.

The goal of RAMP Blue, as developed by Alex Krasnov, Andrew Schultz, and Pierre-Yves Droz, was to build a message passing multi-core system as an example of the simulation and emulation possibilities. The goal of this work has been to port RAMP Blue in to the RAMP Description Language (RDL) in order to begin the process of converting it from a multi-core system implemented in FPGAs to a simulator (of a multi-core system) implemented in FPGAs.

1.1 Background of RAMP Blue

Figure 1 shows the Berkeley Emulation Engine (BEE2) platform on which RAMP Blue [6] was developed. Each board has 5 FPGAs, 4 DRAM slots per FPGA, high-speed inter-chip links and Multi-Gigabit Transceiver (MGT) links [1]. The initial goal was to build a system with 16 BEE2 boards, allowing a total of 1024 processors in the system [3].

Each BEE2 board would use the center FPGA as a control FPGA allowing users to program the other four FPGAs: the user FPGAs. The user FPGAs are connected in a ring via inter-chip links and each of them connects directly to the control FPGA. Each user FPGA also has 4 MGT links that can be

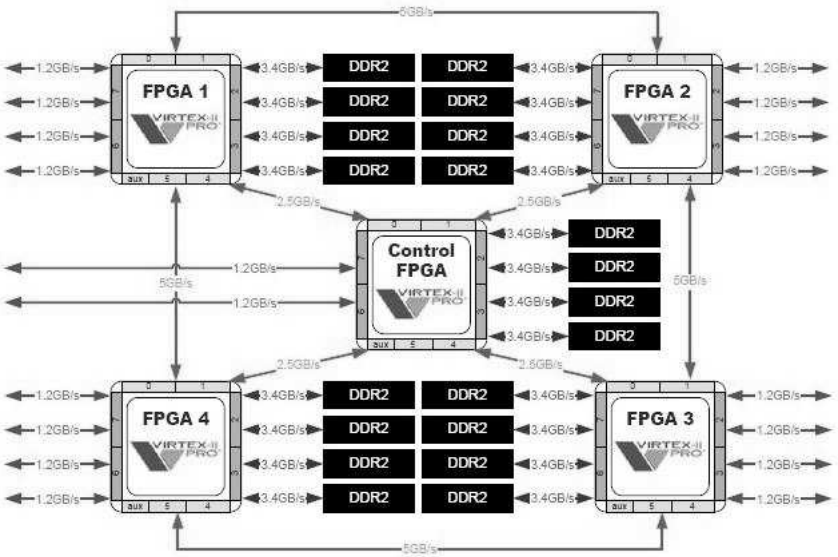

Figure 1: BEE2 Architecture [1]

used to connect to other BEE2s. Therefore, every BEE2 board can be directly connected to every other board using the MGT links in a 16 board system.

The Xilinx MicroBlaze [4] was chosen for the initial implementation of RAMP Blue, instead of, for example, the PowerPC or Leon. The MicroBlaze is an embedded processor developed by Xilinx, making it compatible with Xilinx FPGAs and CAD tools. In addition, it has simple point-to-point interfaces, the Fast Simplex Link (FSL), and requires few FPGA resources.

Figure 2 shows the RAMP Blue per-FPGA architecture. Each FPGA contains: 4 MicroBlazes, 2 console switches, 2 console serializers, 1 network switch, 2 memory arbiters and controllers, and a shared floating point unit (FPU). Only 4 MicroBlazes are used in this configuration, but similar structures with 2-8 MicroBlazes can be built.

The architecture does not employ buses except for the Processor Local Bus (PLB) connecting each MicroBlaze to its timer and interrupt controller. Instead RAMP Blue uses point-to-point connections. This decision was made because RDF only allows point-to-point connection between system component.

The two console networks manage the communication between all the MicroBlazes and the control FPGA. A crossbar switch on each user FPGA

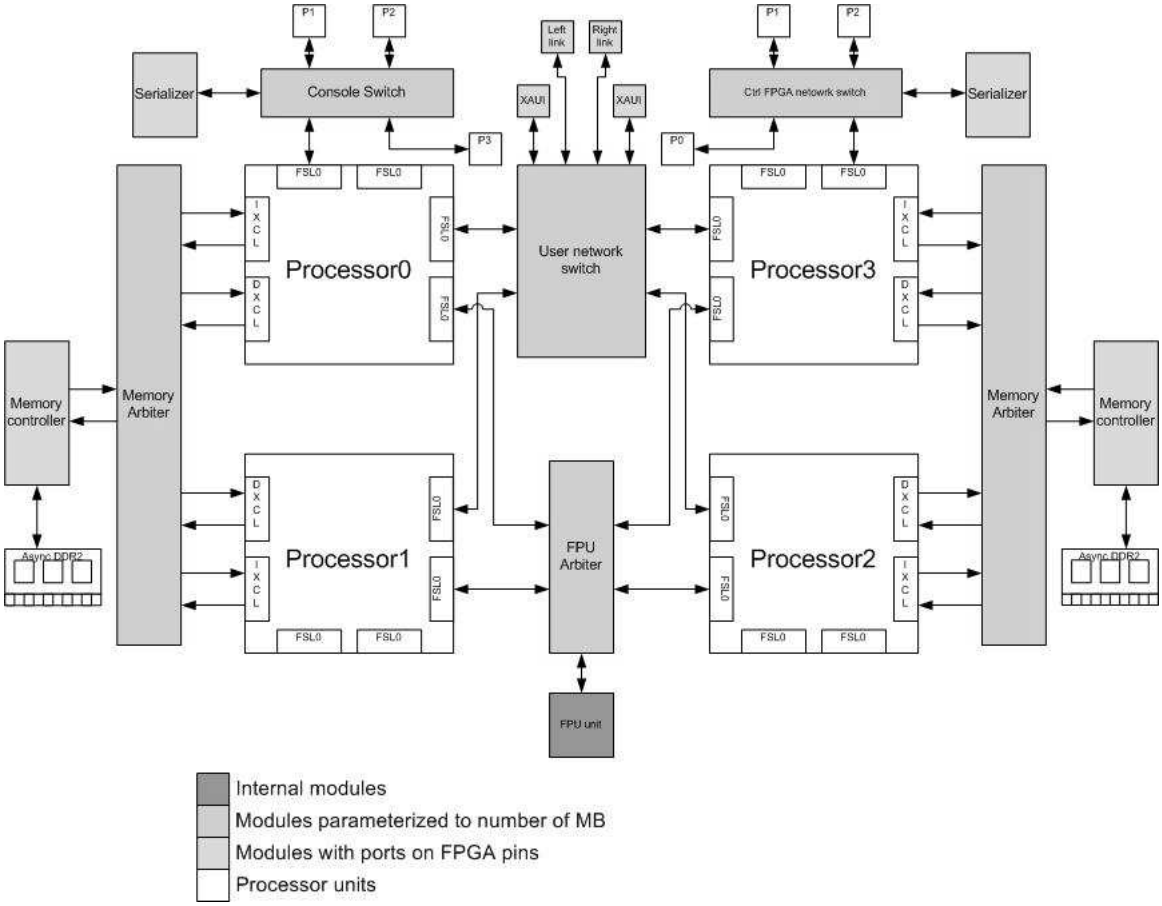

Figure 2: RAMP Blue Architecture for each FPGA - 4 Processor Configuration

connects all the MicroBlazes, the left and right inter-chip links and the 10 gigabit Ethernet Attachment Unit Interface (XAUI). XAUI is a standard network access layer allowing point-to-point communication protocol between two boards. The memory arbiter regulates both the data and instruction memory access to the memory controller. The multi-port FPU is shared among all the MicroBlazes. The XAUI infrastructure, DDR infrastructure, inter-chip module, clock generating modules, and reset generating modules are not shown in Figure 2. They are not the core components in the architecture, but are needed to provide clocks, resets and to interface the system to the FPGA pins.

Each MicroBlaze runs uClinux [5], which maintains most of the functionality in a traditional Linux kernel except support for fork as a result of the lack of MMU on the MicroBlaze. RAMP Blue can run the UPC (Unified Parallel C) benchmarks with Global-Address-Space Networking (GASNet), allowing us to measure the performance of the system [6].

The MicroBlaze processors use FSL-based network interfaces making the uCLinux network driver the system's bottleneck, as FSL write operations are slow. A current project is working to implement a direct memory access (DMA) interface to the network, which would eliminate the overhead associated with the UDP/IP stack and the uClinux network stack [6].

1.2 Background of RAMP Description Language

The RAMP description language (RDL) [7] is being developed to describe hardware based simulators. The focus of RDL is on the standardization and parameterization of such complex, cycle-accurate distributed simulators, with support for performance modeling and technology independent mappings. The goal of this work has been to create a decoupled machine model of RAMP Blue in RDL.

The RAMP Design Framework (RDF), the formal model for RAMP simulators which RDL supports, is structured around loosely coupled *units* connected via well defined *channels*. According to RDF, the inter-unit communications are latency insensitive to allow performance research through the parameterization of channel timing models. However, the channels are always point-to-point, lossless, unidirectional, strictly typed, and guaranteed to deliver messages in order.

At this point, some terminology should be defined. The system being emulated is the *target* system, and the machine performing the emulation is the *host* system. In the case of RAMP Blue, the target system is the multi-core RAMP Blue system, and the host is a collection 8 or 16 of BEE2 boards. The target system is composed of units which communicate over channels. Each unit is implemented on a *platform*, in this case an FPGA, and each channel is implemented by a *link*, an arbitrary and platform dependent communications medium. The unit of data transfer over channels is the *message* which is strictly typed in RDL. RDL also provides for time dilation by separating the *host clock* and the *target clock*. The host clock is the physical clock whereas the target clock is abstract and distributed, and regulated by `start` and `done` signals in at each unit.

Informally, there are two types of units in RAMP Blue, a regular unit

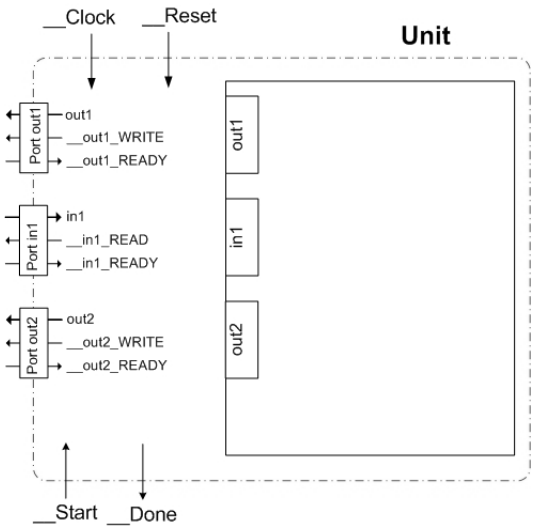

Figure 3: An Example Unit

and an IO unit. A regular unit interface consists of channel interfaces and four other signals: `Reset`, `Clock`, `Start`, `Done`. There are two types of channel interfaces on units – input and output ports. An input port consists of `x_Ready`, `x_Read`, and `x_Data`, whereas an output port consist of `x_Ready`, `x_Write`, and `x_Data`. Figure 3 shows an example of a regular unit. An IO unit is similar to a regular unit except that it is allowed to have other signals besides port signals to interface to peripherals. In other words, it contains signals which escape the RDL message passing formalism to interface with physical hardware. Figure 10 shows an example of an IO unit.

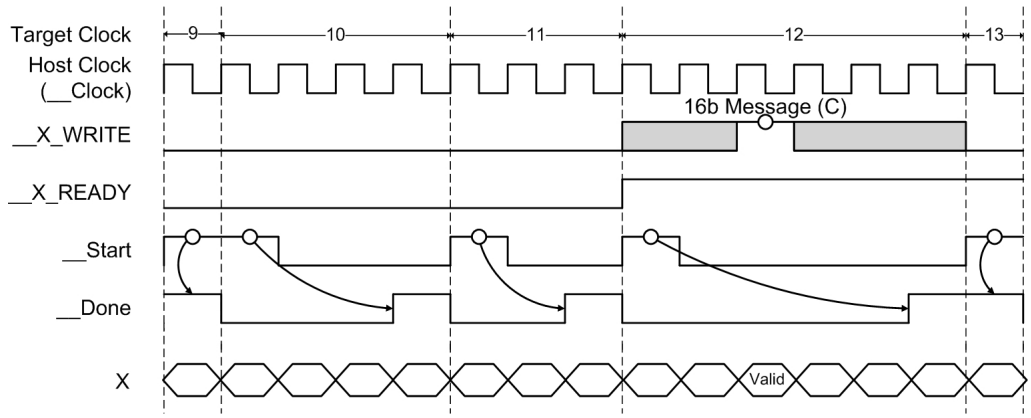

Figure 4: RDL timing(courtesy of Greg Gibeling)

Figure 4 shows a timing diagram for an output port including the `start` and `done` signals used for dilation of the target clock cycles. The `start` signal

indicates the start of a target cycle, and the `done` signal indicates the end of a target cycle. Only one or zero message is allowed to transfer through each channel in each target cycle. The `x_Ready` signal indicates whether the channel is ready to accept a message; the `x_Write` (and `x_Read`) signals indicate to the channel that the data is to be written (or read).

An adapter is needed to assemble all the Verilog modules from the original RAMP Blue system into a unit and to interface between the original signals and the RDL port interfaces. Specifically, the adapters have a few functions: convert data from the original modules into messages, convert messages to data, target cycle firing (signaling start and done), and to interface the original module's control signals to the RDL port interface. The adapter also must make sure that the unit is latency independent. For an example, if the original module interface involves a protocol in which a continuous burst of data is required, the adapter must provide buffer for the data so that the unit itself is latency independent.

1.3 RAMP Blue in RDL

The original development of RAMP Blue and the RDL compiler (RDLC) was conducted in parallel, thus RDL could not be used to develop RAMP Blue. Instead, RAMP Blue was developed using Xilinx's embedded system platform (EDK) tools since many of the components used in the system are easily instantiated in it. This project aims to adopt the Ramp Design Framework and to describe the current RAMP Blue system in RDL, enabling it's use as a fully qualified RAMP system.

The RDL RAMP Blue system is the first RAMP system to use RDL, making it a candidate reference design for new RAMP designers. Also, the process of implementing RAMP Blue in RDL has been a source of necessary feedback to the RDL developers.

Abstractly, converting a point-to-point and latency insensitive design to an RDL design is fairly straightforward. However, this conversion involves changing the tool flow from EDK to a RDL-compatible tool and debugging any possible errors introduced by these changes. The work is conceptually simple, but demanding in an engineering sense.

2 Design

RAMP Blue was designed with full knowledge that it would need to conform to the RDF model. For instance, all the major components of the system are linked together with point-to-point connections. Therefore, no major structural changes to RAMP Blue were needed. However, some logic was needed to string interfaces together since the original module interfaces are not quite the same as the RDL port interfaces. Further, some modules are grouped together inside one single RDL unit, and others are broken into a multiple units. The architecture is shown in Figure 5.

In this design, each unit's target cycle matches its host cycle. In each unit, the signal `Done` is asserted whenever `Start` is asserted, thus each host cycle is actually a target cycle. This can be changed according to the emulation requirements of the system. For an example, if we wished the memory controller to emulate the ability to respond to a read request every target clock cycle, the target clock cycle can be extended to many host clock cycles allowing the controller to completes its read operation. Such time dilation can be done if, after receiving the `Start` signal, the unit's `Done` signal is not asserted until memory controller is done servicing the request.

2.1 Processor

The MicroBlaze processors are grouped together with their Block RAM (BRAM) and PLB peripherals. The BRAMs are used to load boot code, and the PLBs are used to connect to a timer and an interrupt controller. They are connected to the MicroBlaze using complex non-point-to-point interfaces, and therefore grouped together as a single unit. Figure 6 shows a processor unit's configuration.

The processor unit has no IO pins, but it does have six pairs of input and output ports connecting to the following: instruction memory, data memory, network, two console networks, and a floating point unit (FPU). The input and output ports are interfaced to the instructional Xilinx CacheLink (ixcl), data Xilinx CacheLink (dxcl) and fast simplex link (FSL) interface of the MicroBlaze processor.

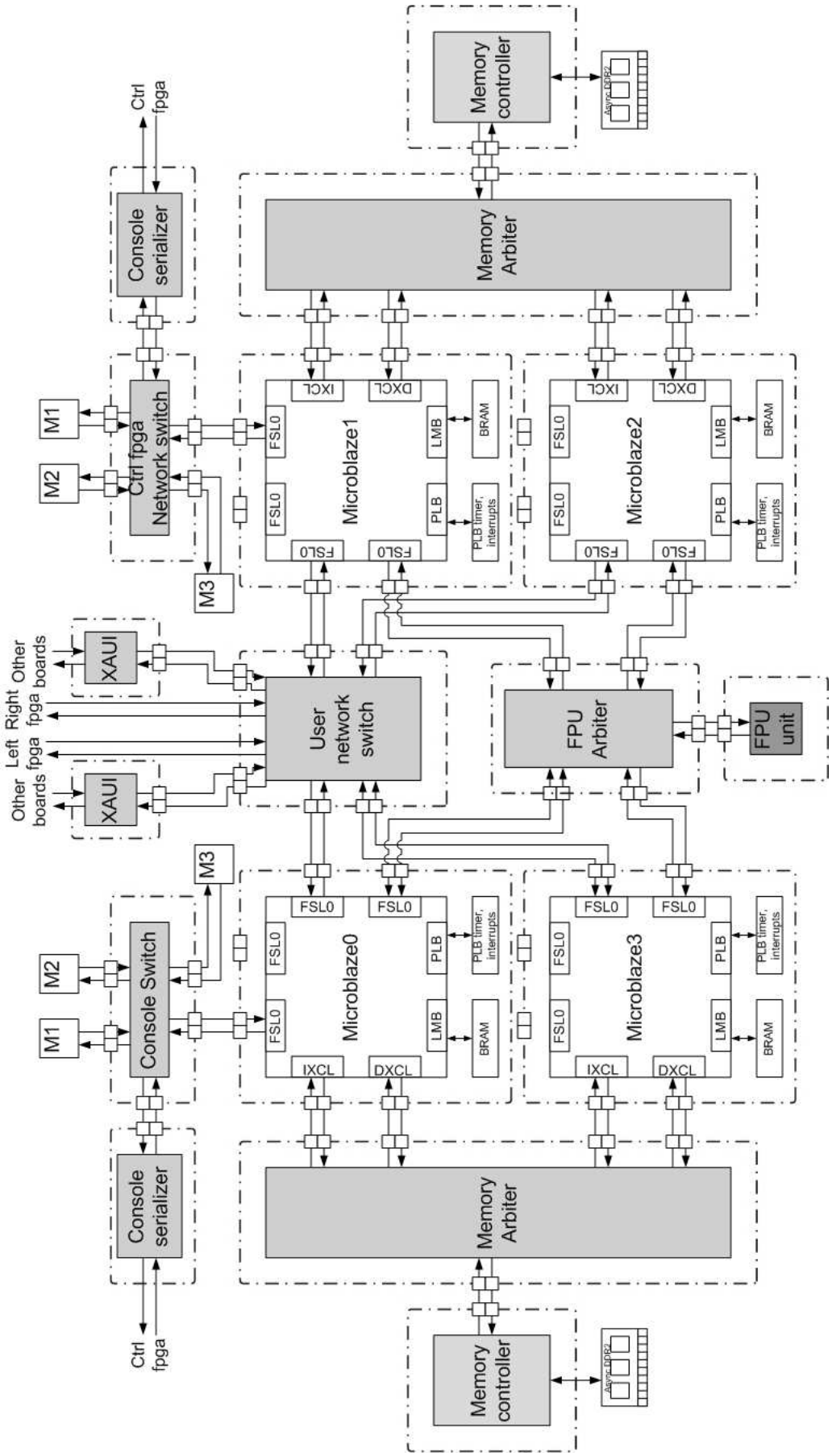

Figure 5: RAMP RDL Overall Architecture

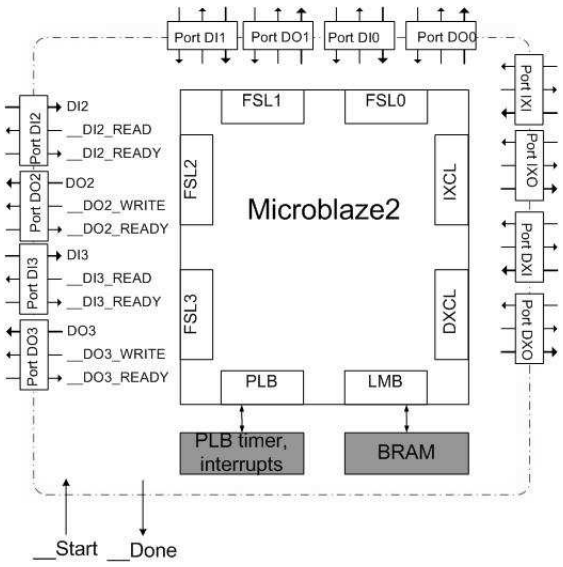

Figure 6: Processor Unit

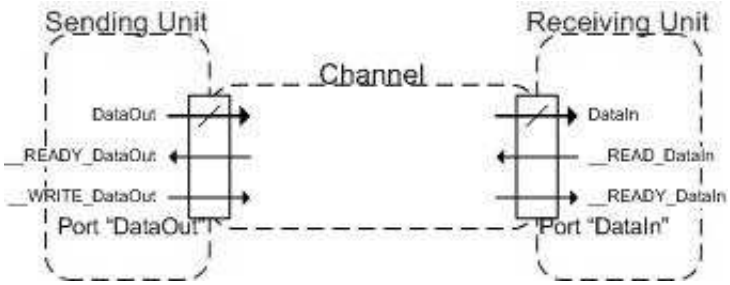

Figure 7: RDL Channel Signals(courtesy of Greg Gibeling)

The XCL and FSL interfaces are similar, as they both have 32 data bits and 1 control bit (the control bit can be written by the software to convey extra information such as beginning and ending of a packet, or whether the memory request is a read or write). Thus, all the ports of this unit carry 33 bit messages. Simple AND gates and OR gates are needed to convert FSL's `exists` / `full` / `read` / `write` signals to RDL port signals. Figure 7 shows the RDL interface and Figure 8 shows the FSL interface. Figure 9 shows the simple conversion between the two interfaces.

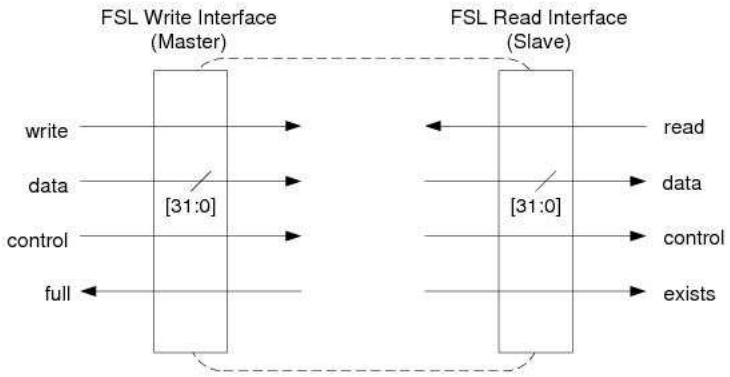

Figure 8: FSL Interface Signals

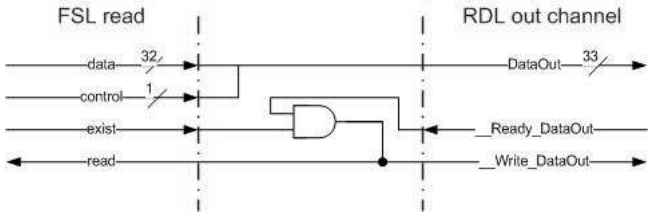

Figure 9: FSL Interface to RDL Interface

2.2 Memory

There are three different memory components in this system: the memory arbiter, the DDR2 memory controllers and the DDR2 infrastructure. The arbiter and the controller are separate units, but the DDR2 infrastructure is instantiated outside of any RDL unit.

Originally, the memory arbiter interacted with the processor through the XCL interface. An arbiter with 8 port-pairs can arbitrate between four processor cores, with each core using one port-pair for instruction-side and one for data-side memory. The arbiter integrates those signals and exchanges information with the memory controller through two channels, with each channel bundling the memory address, tag, data command, and bit width information. In RDL, those signals combine to form a structured message to be sent across channels.

The current memory units do not use burst mode, thus the data sent between memory arbiters and memory controllers does not have to be continuous. If memory modules that support the burst feature are used in this

project, the adapter would need to be changed to support the burst operation. To keep the modules latency insensitive, buffers would need to be added to the adapter so that the data is buffered during a burst operation.

2.3 Network

The cross-bar network switch connects the MicroBlaze processors, XAUI cores, and the left and right links together. Originally, the network switch connected to the MicroBlazes processors via the FSL interface; now they communicate through channels. A XAUI unit is an IO unit, since each XAUI interfaces to the MGT links of the FPGA. It is connected to network using channels and its original interface has been interfaced to the RDL port. The network switch itself is also an IO unit, since it has IO connections to the left and right links, with the external signals connecting to the inter-chip module.

Ideally, the XAUI cores and inter-chip links should be implemented as RDL links and therefore completely abstracting the inter-FPGA communications. However, the necessary RDL link generators are not ready in RDLC2 yet. When those features are available, the design can be easily modified to use them.

The original network switch module is complex, since its interface includes data ports, virtual channels and drop signals, making it difficult to describe as an RDL unit. Instead, the network buffers interfacing with these signals are placed around each switch interface (Figure 10) and the RDL unit contains the network switch and buffers. Finally, the RDL port signals are interfaced to the FSL signals of the network buffers.

2.4 Console Switches

Each FPGA has two console networks, one for debugging and one for networking between the user FPGA and the control FPGA. Each console network uses a console switch and a console serializer. The console switch is a unit used to communicate to all the MicroBlaze processors and it originally interfaced to the processors via FSL. The serializer is an IO unit that connects to the console switch through a channel and serializes and de-serializes the informa-

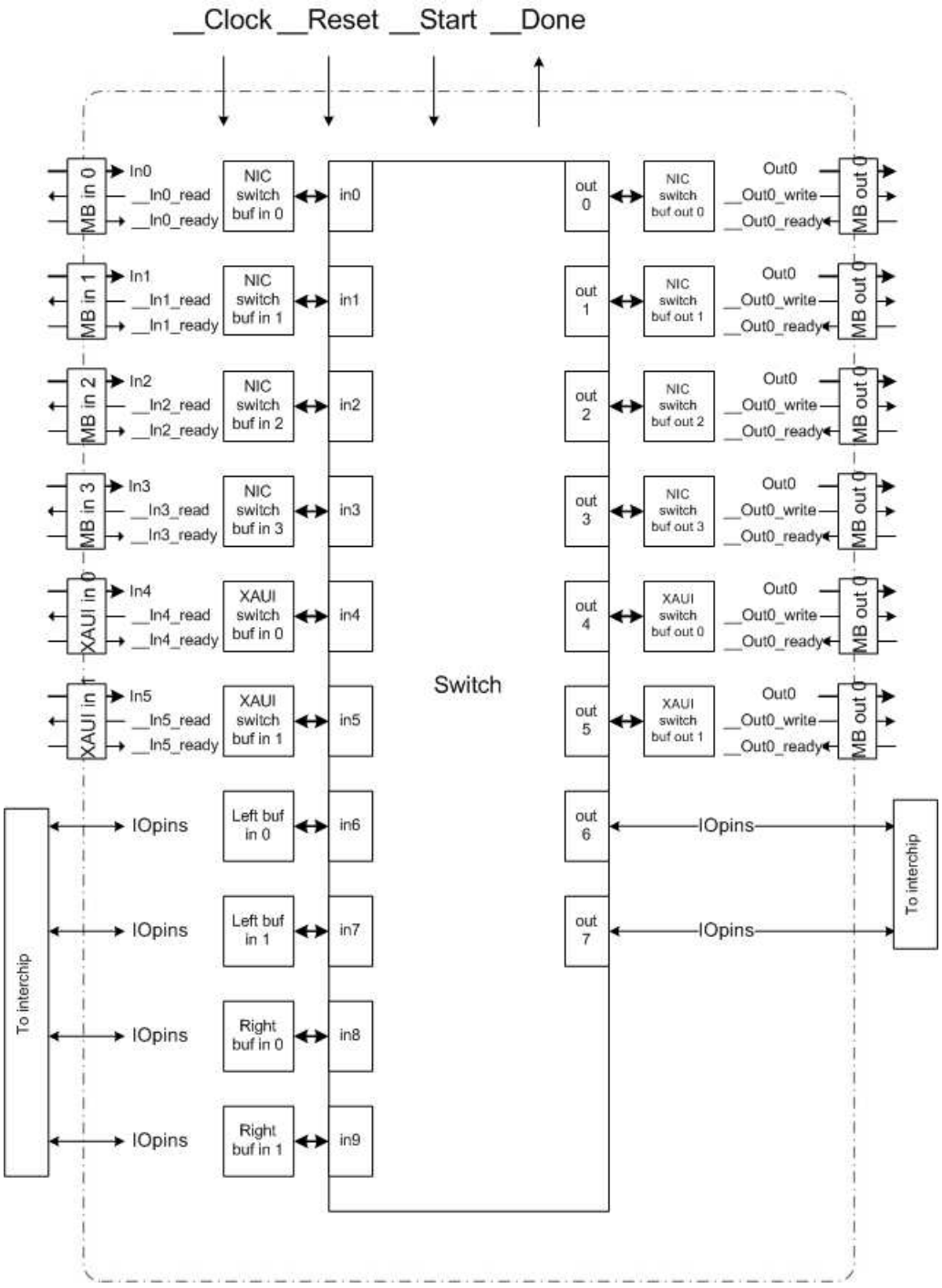

Figure 10: The Switch Unit

tion consolidated by the console switch. The IO signals on the serializer unit go through the inter-chip modules to connect to the high speed links between the control FPGA and the user FPGA.

Similarly to the XAUI, the serializer along with the inter-chip module should be implemented as an RDL link once the appropriate link generator is available.

2.5 Floating Point Unit

The multi-port floating point unit has port-pairs (interfaced to the original FSL signals) to communicate with MicroBlazes. The arguments and operation are transferred by the input ports, and the output ports transfer the results back to the processors. The FPU unit arbitrates requests from a variable number of processors, and sends those requests to one of four Xilinx floating point macros. Originally, plans were made to separate the FPU arbiters from the FPU cores, but the logic is well integrated and this separation would result in inefficiencies. Therefore, the FPU and arbiter are integrated into a single RDL unit.

2.6 Top FPGA

A top RDL unit contains the major component of the system. An RDL file instantiates and appropriately connects the RDL units with channels. RDLC compiles the RDL files and generates HDL files that assemble the system. The files generated also include RDL wrapper files for the units and HDL describing the links. The IO unit's IO signals become the input and output signals of the top level HDL module.

However, there are some modules in the original system that do not fit into the channel-based RDF communication model, such as the clock and reset generation module, the inter-chip modules, the DDR2 infrastructure module and the XAUI infrastructure module. A `top_fpga` file instantiates the top level unit generated by RDLC and connects its IO signals to these modules. Figure 11 shows the top FPGA configuration.

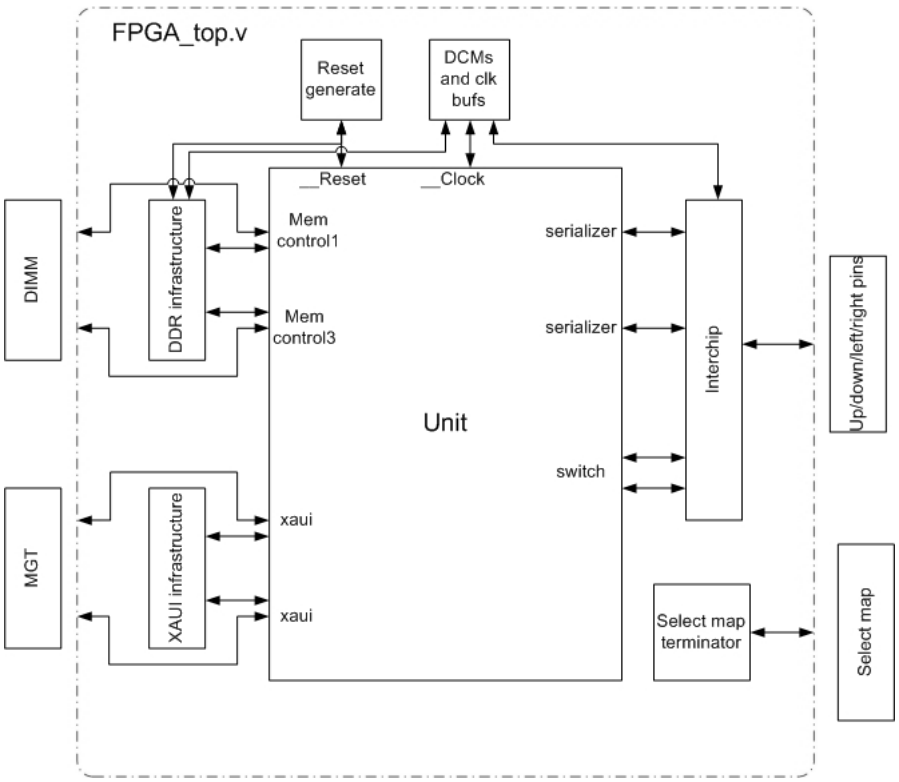

Figure 11: Top FPGA Configuration

3 Experience

The RDL version of RAMP Blue is a straightforward adaptation of the original RAMP Blue system with little change to the high-level architecture. Despite this, making it work required significant effort. This section documents the experience of the changing tool flow, using RDL, and adapting non-RDL HDL to implement RDL units.

3.1 Tool Flow Changes

The original RAMP Blue system was developed on the XPS-EDK platform. EDK is a Xilinx tool flow made specifically for embedded processors on Xilinx FPGAs. However, since RDLC is not yet compatible with the EDK tool flow, much time was spent converting the project to the ISE tool flow. In the end, using a combination of both tool flows was helpful.

The following describes the design steps along with the tool flow for RAMP

Blue in RDL:

- Divide the system into units and write an adapter for each unit with port declarations. As mentioned in Section 1.2, the adapters interface the signals of the original module to the RDL port interfaces. The RDL compiler helps with the unit module declaration by generating an empty Verilog module shell with the port and IO signals.
- Write the RDL file that instantiates units and connects them together using channels. Run the file design through the RDLC "map" command, to generate a folder containing Verilog files that instantiate units and channels.
- Start an ISE project and add the following files:
  - All the RDLC generated Verilog files under the output folder
  - All the files used to construct modules under each unit
  - Verilog files that instantiate top_fpga, clocks, reset, infrastructures, inter-chip modules
  - Blackboxes (ngc, ngd) files needs to be copied under the ISE project directory
  - A ucf file for timing constraints on the design
  - A file, `system.bmm` – copy the file under ISE project folder, select the top Verilog file, then right click on 'implement design' in tool flow, select properties, and in the box 'Other NGDBuild command line option', type '-bm system.bmm')
  - A file, `processor.mhs` that specifies the processor and its peripherals (see Section 3.2)

In the process of using RDL and ISE, a few work around had to be used due to bugs in ISE and features lacking in RDLC2. They are listed below:

- ISE has trouble detecting black boxes if the black box's port specification is written in Verilog. Therefore, a VHDL adapter sometimes needs to be created so that ngc files can be included.

- EDK uses Tool Command Language (TCL) script for parameterization, but it is rather inconvenient to use TCL in an ISE flow. All the modules are converted from using TCL to using the Verilog generate statement. However, this change is not mandatory as TCL can still be run to generate HDL files for ISE tool flow.

- VHDL libraries are used by some modules. It is easy to specify the library in the pcores of a EDK project, but in ISE, a new library needs to be created and added with files.

- Some IO units are used more than once. For an example, there are two FSL serialisers. However, RDL does not allow two IO units that have the same IO pins (of course). Therefore, the IO units are declared to be different, but after generating the HDL files they are changed to instantiate the same unit anyways. Our finding of this problem has resulted in a redesign of how RDLC3 handles IO.

It is quite inconvenient to implement the design in ISE, since a lot of processor-related components are better matched to use EDK. The components are presented as pcores, a file format specific to EDK that includes HDL files, interface declarations and parameterizations. A top pcore-instantiating MHS file is also used. Unfortunately, RDL is not well integrated with the EDK tool flow, so we used Xilinx ISE Foundation instead. ISE is commonly used for systems without an embedded processor, and does not support EDK's MHS file format and pcores. This means that cores that are fully specified in VHDL or Verilog will have an easier transition to using RDL, but cores developed to run on a specific environment with special formatted files will need to be compiled independently.

One may feel that system level tools such as EDK provides an easy way to connect components together by simply specify the bus connections (in MHS file for EDK), thus using RDL without EDK is a regrettable loss. However, a RDF system is point-to-point and bus-free, thus there is no need for easy bus-connection. In fact, an RDL file is quite equivalent to an MHS file in EDK, as it is a top level file that specifies connections between well defined unit ports.

Our problems and concerns relating to EDK and RDL have resulted in design changes in the next revision of the RDL tools: RDLC3. Part of the

goal of this work was to explore the use of RDL and find necessary changes, such as these.

3.2 Adding Processor Unit files

Because processor unit components are originally formatted as pcores with VHDL libraries components, they are harder to instantiate in ISE. This problem can be sidestepped by using a black box (files that had been synthesized from HDL files to netlists such as ngc files) in the ISE project. This gives us a quick and easy way to test modules without having to track down and organize their source code for ISE.

Besides "black boxes", there are two ways of adding the processor unit's components to an ISE project. We can use an EDK project to generate the submodules in ISE, or add the MicroBlaze's HDL files along with black boxes and VHDL libraries.

The following describes how to use a EDK project for the processor unit:

- Right click in the source box of the project navigator, choose "new source".
- Choose "embedded processor" at the bottom left of the popped up screen.
- Import the `system.mhs` file that instantiate MicroBlaze, BRAM, PLB peripherals and their appropriate controllers.
- In the ISE project, add a VHDL black box adapter file that details the interface signal of the module, and instantiate that module.

This way, the pcores are automatically imported into the ISE project via EDK. Upgrading to new versions of pcores or changing a pcore's parameterization is easy through EDK. However, each MicroBlaze processor uses an RLOC property that is dependent on a parameter being unique to guarantee that they will be mapped to different tiles on the FPGA. Since an EDK project does not allow any parameterization on its top MHS file, the RLOC would need to be manually changed to different locations in the constraint (UCF)

file. Unfortunately, this would hardwire the location of the MicroBlaze processor and would not allow the tools to map the processors to the optimal locations. It is also very difficult to manually figure out the correct RLOC locations.

Another way to incorporate MicroBlaze processors into an ISE project is to add all the files in the MicroBlaze pcores manually:

- Right click in the source box of the project navigator, choose "new source".
- Choose "VHDL library" on the left side of the popped up screen, and the name of the library is "microblaze_v4_00_a".
- In the library box, right click on microblaze_v4_00_a, and click on "add source", and add files that matches `*pkg*.vhd`.
- In the work library, add the rest of the MicroBlaze files.
- Other processor peripherals can be added in similar manner, or they can be presented as a black box.
- Instantiate the MicroBlaze core, but make sure that a different string is passed to each MicroBlaze instance's "C_INSTANCE" parameter.

Passing a different string into the C_INSTANCE parameter of the MicroBlaze processor for each instantiation would allow the tools to map the processors to different locations. This allows optimal mapping, but is not a clean implementation since this method does not use the original pcore format.

3.3 RDL Experience

The compiler used for the RDL units was the RAMP Description Language Compiler version 2 (RDLC2) and it implements most of the basic features. The RAMP Blue RDL file used features such as IO pin definitions, different link generators, structured messages, arrays of channels/units, and so on.

We encountered few problems using RDLC2. It was lacking some features that would make RAMP Blue RDL file more elegant and a few pieces of

functionality. Those features have been promised in RDLC3, mainly as a result of this project. Due to the RDF (and RDL) compatible interfaces chosen in the original RAMP Blue implementation, using RDL did not introduce any new functional problems. In short, using RDLC2 was a pleasant experience.

3.4 Cores

This section discusses the experience of converting each module to be an RDL unit.

Once the system was divided up into units, it took a little work to convert the original modules to RDL units. For non-IO units, the interface only needs to consist of channel interfaces, `clock`, `reset`, `start` and `done`. If a design is point-to point and latency independent, as RAMP Blue was, it takes only a few lines of code to convert the original interface to a RDL port interface. Figure 9, shows how easily the FSL interface can be converted to the RDL interface.

Because no buses, other than PLB, are used in the original RAMP Blue design, most of the changes in interface are straightforward. In most cases only a few gates need to be added and data signals need to be grouped together. If the interface is more complicated and latency dependent, then interfacing the two protocols requires a state machine and some registers to support the latency insensitivity requirement of RDF (note that RDL has no such requirement).

The width of input and output signals are specified by parameters because cores may have different number of ports depending on the number of MicroBlaze processors. Unfortunately, Verilog and VHDL does not support 2D arrays for input and output signals; therefore all the data signals are collapsed into a 1D array at every module interface and re-constructed into a 2D array inside the modules. In TCL, each set of input and output signal can be generated with a different name, but for generate statements, collapsing the arrays seems to be the solution. Fortunately, RDLC2 supports for 2 dimensional arrays, and can generate RDL unit shells that contain the collapsed 2D array declaration and wire reconstruction to help speeding up the coding process.

4 RDL Impact on Design

RDL aims to have as little impact on the design as possible and at the same time provide uniform interfaces to all units, timing accurate simulation and so on. However, given the limited number of link generator plugins available for RDLC2, with none of them support full timing models, the design is limited and this affected the performance of the channels.

In this section, three types of RDL links are described: single register link, wire link, dual register link. The single register links were used during development. However, these links slow down the performance of the system, thus dual register link and wire link were used in attempt to recover this performance. RDLC2 allows user to use different plug-ins to describe different link implementations, separating the system description from the link implementation.

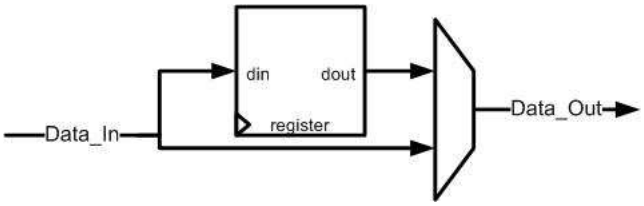

Figure 12: Wire Register Link

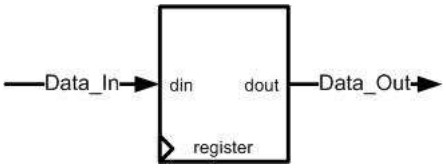

Figure 13: Single Register Link

4.1 Resource Utilization

The resources usage for RAMP Blue after adopting RDF depends on the link type. Three different types of links were tested for their resource utilization: single register link (Figure 13), double register link (Figure 14) and a wire link (Figure 12).

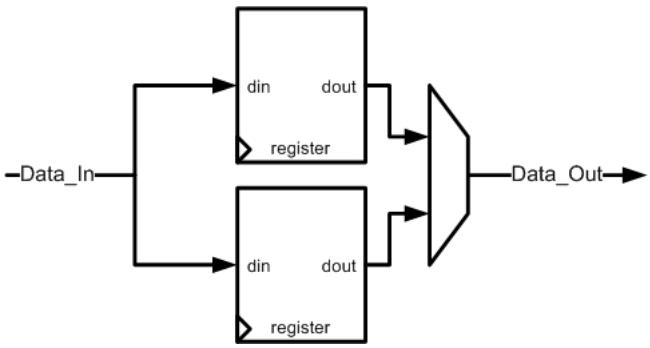

Figure 14: Dual Register Link

The resources shown are flip flops, LUTs and slices occupied. Each slice in a Virtex-II Pro consists of 2 flip flops and 2 LUTs. Therefore, an increase usage in either flip flops or LUTs can cause the number of slices to go up. An occupied slice may also simply be used for routing purpose and have none of its resources used. Thus, it is more accurate to look at the flip flop and LUTs usage.

Each link has at least one register per data bit and one register for control signals. Thus, flip flop usage increases. The number of 4-input lookup tables (LUTs) used increased slightly, because most of the interface change involves adding a few gates. Table 1 shows a summary of resource utilization and table 2 shows a proportionate comparison of the utilization for easier reading.

System	Slice Flip Flops	4-input LUTs	Occupied Slices
Non-RDL RAMP	24,931	33,864	25,571
Wire Link RDL RAMP	29,921	39,528	28,895
Single Register RDL RAMP	29,921	36,520	27,850
Double Register RDL RAMP	33,089	39,708	29,244

Table 1: Resource Utilization

System	Slice Flip Flops	4-input LUTs	Occupied Slices
Non-RDL RAMP	1.00	1.00	1.00
Wire Link RDL RAMP	1.20	1.16	1.12
Single Register RDL RAMP	1.20	1.08	1.09
Double Register RDL RAMP	1.33	1.17	1.14

Table 2: Normalized Resource Utilization

As shown in Table 2, the LUT count for the double register link and wire link increases dramatically since more logic is needed to multiplex between

the two sources of data (the two registers for double register link and the one register and one direct wire in wire link) in the links.

The increase in flip flop usage is consistent with the total amount of data bit in each channel. From the processor units, there are 8 pairs of channels for the network switch, 16 pairs of channels interfacing to the memory arbiter, 8 pairs of channels for the FPU, and 16 pairs of channels to the console switches. Each of these channels is 33 bits wide. Between the memory arbiter and the memory controller, there are two channels, each 227 bits and 176 bits wide. They are composed of many memory data signals, such as memory data, address and tag. There is a pair of channels that are 36 bits wide between the console switches and the serializers. The message consists of the data, control signal and 3 bits indicating which processor the data belongs to. Thus, in total, there are 102 channels with a total of 4046 bits of data in those channels. This number is quite consistent with the increase in number of flip-flops used for each system.

Usually, if the system is not concerned with the speed of emulation, using a single register channel is the optimal choice. Flip-flops in FPGAs are usually plentiful for most designs, and a single register channel has the simplest control logic, hence the least impact on number of LUTs used and the critical path of the system.

The increase in resource utilization for even the most basic single register link is due to a few reasons. First, the system is overly rigorous. The original RAMP Blue system uses back pressure in most of its modules to ensure that if a module writes, the receiving module is guaranteed to receive that data. However, when adopting RDF, we have chosen to ignore this assumption. Therefore, buffer space is needed in each channel for the case which the receiving unit is not ready to read when the sending unit writes.

This decision was made so that the system will be valid according to RDF and future units added into the system will not have to rely on the original RAMP Blue assumptions. The resource utilization could be reduced if the module interfaces were changed to work with the port interface better, and if the units abandon their internal back pressure buffering. This was not done in this project as many of the modules were still under development, and changing these modules would make it hard to keep the system updated.

Another alternative to reduce resource utilization would be to use direct wires without registers for the link, but this breaks the RDF assumptions, and thus decreases the research value of the resulting system.

Despite the 15% increase in resource utilization in certain designs, a system using eight MicroBlazes is still able to fit within a Virtex-II Pro FPGA. The most contended resource in the system is LUT usage; therefore using a single register channel will help fitting a design with more MicroBlazes into Virtex-II Pro. For other systems, the increase in resource utilization will depend on the number of channels and number of data bits in the system.

4.2 Performance

We used the UPC benchmarks to measure the performance of the RAMP Blue system. The performance varies depending on the link (and channel) type used. The same three links used to test resource utilization were used to test the performance of the system.

The default link in RDLC2 is a single register link. In this configuration, there is one register that buffers the data between the output port and the input port. Therefore, the channel has two states: empty and full and one cycle of latency in the forward and reverse directions. For each message transmitted, there is one cycle delay on each channel. Even if the data is transmitted back-to-back, it would be slowed down by this channel since only one data can pass through every two host clock cycles. Thus, the simulation performance of the system drops significantly.

The double register links did not increase performance much, as shown in Table 3. This is because the assumption that there are many back-to-back data transmission across the channels is incorrect. The performance of the system is limited by the slow network, which is in turn limited by the software driver writing into the FSL interface. Since the MicroBlaze processor uses the FSL port for network interfacing, packets are usually broken into single 32-bit data words moving through the network and that still incurs one delay per data in each channel. Therefore, performance of a system with dual register channels still lags behind the original RAMP Blue performance.

Finally, the last channel tested was the wire link. It is a single register link

with a straight through connection. Therefore, if an output port is sending data over, it could be transmitted straight to the input port, assuming that the input port is ready to accept data. If the input port does not read data in the same clock cycle, then the data is saved in the register. This creates a zero forward and reverse latency channel.

However, due to the timing issues, the wire link channel proved hard to implement. In the original implementation, there exists a path without registers running through two FSL interfaces on both the console switches and MicroBlaze processor. The total delay from these two paths turned out to be the critical path of the RDL system. The non-RDL system meets timing constraints as does the RDL version that used registers in the channels (thus decoupling the two paths). However, with wire link, the logic delay introduced by several channels chains together, causing the critical path to lengthen. The critical path of the system shot up from 10ns to around 15ns. As a work around, the performance is checked at a half the original clock rate to prove that a system with wire links can have performance equivalent to the original system, with half the clock rate.

System	cg-S	ep-S	is-S	mg-S
non-RDL RAMP	1.0	1.0	1.0	1.0
Wire Link RDL RAMP	1.0	1.0	1.0	1.0
Single Register RDL RAMP	.76	.38	1.0	.82
Double Register RDL RAMP	.82	.41	1.0	.86

Table 3: UPC benchmark performance ratios

Table 3 shows the performance ratio of RAMP Blue in RDL with wire link, single register link and double register link compared to the original RAMP Blue system. As mentioned before, RAMP Blue using the wire link channel has the same performance as the non-RDL RAMP Blue system and systems using registered channels see a drop in performance due to channel latency, exactly as one would expect.

Designers must be careful with their choice of RDL link. The type of link (and channel) used impacts the performance, resource utilization and critical path of RAMP Blue, exactly as one might expect. A more complicated link such as wire link can add a significant amount of delay on the critical path, causing the design to fail on timing. A simple channel like the single register channel uses fewer resources, but impact clock cycle performance negatively.

An extremely simple wire link without registers will allow the system to eliminate most of the resource overhead incurred, but it would violate the RAMP design framework. The impact on performance and resource utilization is quite natural and independent of RDL. However, RDL allowed us to quantify this more easily than otherwise possible.

4.3 Development Overhead

The total time taken for the conversion of RAMP Blue to RDL was about 4 to 5 months with one person working on it half time.

At least 4 to 5 weeks of were spent on solving the problem of porting the project from an EDK tool flow to ISE/EDK/RDL tool flow and dealing with tool flaws. Much of that information had been recorded in this paper, so one going through a similar conversion would have some examples to follow.

Understanding the system and interface of each module took around 3 to 4 weeks. Writing adapters took up another 4 to 6 weeks, with most of the time spent on integrating modules into units, interfacing the RDL port logic to original interfaces, and debugging the Verilog. However, after the initial TCL version, development of the non-TCL version, which relied on RDL parameterization, only took 2 weeks even though all the adapters needed to be rewritten. This is because some modules are grouped together to one single unit and that work had been done in the TCL version. Also, familiarity with the system helped speed up the process.

Only a few days were spent writing the RDL, but it was under constant improvement as new RDL features were learned. Better documentation of RDL and RDLC2 would have helped this process along.

It is expected that another person working on a similar problem would spend less time if each module's interface were well documented (RAMP Blue was not always so), the system used a point-to-point interface and each transmission of data between units were accompanied by `valid` and `ack` signals. If the same tool flow could used in the RDL and non-RDL versions, the original was developed with RDL more in mind, or with a RDL-compatible tool, the design time and resources used would be far less.

5 Future Work

There are many improvements possible to make the RDL design of RAMP Blue more useful and closer to its ideal implementation.

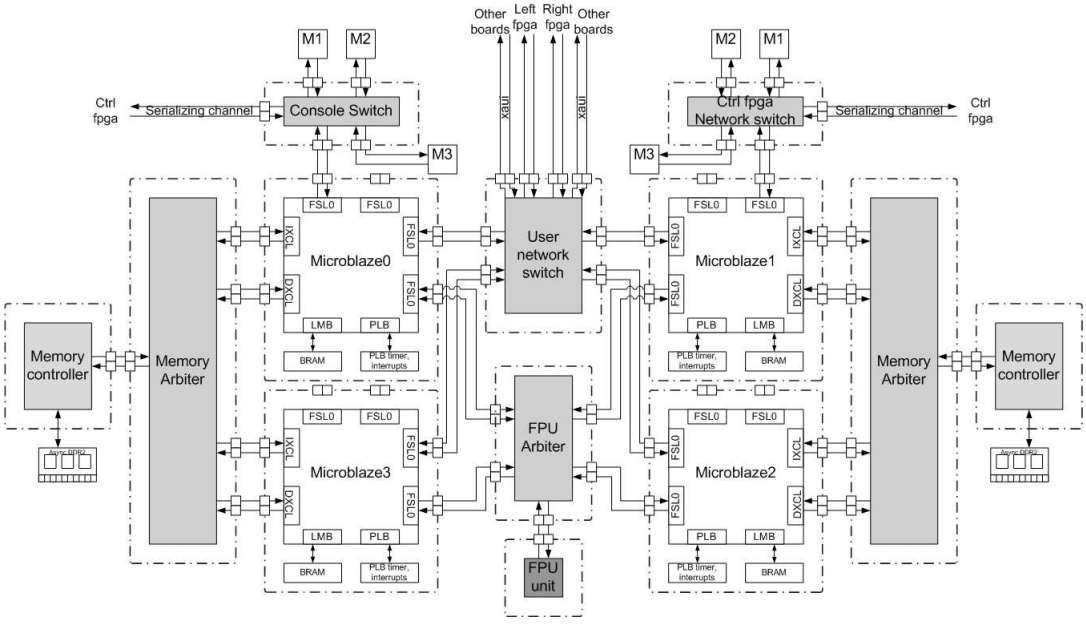

Figure 15: Ideal RAMP Implementation

First, some modules such as the XAUI cores are currently implemented as units in the design. However, these should be implemented as channels between network switches as their sole purpose is to carry data through the MGTs. Therefore, the XAUI unit should be a link, and the target system should not be aware that it is actually implemented using a XAUI module. Similarly, modules such as serializers can be replaced by a parameterized link generator that can serialize and de-serialize data. Figure 15 shows an ideal implementation of RAMP Blue.

Similarly, some buffers can be shifted into the channels around the network. By adjusting channel buffer depth and width, performance can be tuned. Currently, the network switch unit contains the network buffers, but they should be implemented by the channels. This is a limitation of the link generator plugins available for RDLC2.

Although the RAMP Blue port interfaces are simplified such that there are only three signals per port on a module, these interfaces are still not stan-

dardized in every module. For example, the memory arbiter unit's output port to the memory controller consists of `data`, `WRITEdata`, `READYdata`. However, the definition of a data signal is not defined rigidly, and the sequence of what to transmit in each target cycle is not part of an RDL specification. Therefore, in order to share units among designers, interface design is still an important and non-trivial exercise. In other words RDL helps codify a design, but a human designer is still needed.

Since the host clock mirrors the target clock in our implementation, the system also does not yet perform time dilation. More interesting performance measurements could be derived if time dilation were available in some of the units. This is an important piece of functionality provided by RDL, and would turn RAMP Blue from an simple implementation into a simulator.

6 Conclusion

This report details the process of converting RAMP Blue from an HDL/EDK project into an RDL based project, including the changes made to the hardware and the tool flow.

We have documented the effect on the resource utilization and performance as a result of our limited adoption of RDL. As a result of adopting the RAMP Design Framework and avoiding modifications to the original Verilog, resource utilization increases by up to 15 percent, and the system can achieve around 80-100% of the original RAMP Blue performance depending on the choice of link implementation. The resource utilization could be lowered by combining RDL channels with the original modules more efficiently.

In conclusion, a RAMP Blue system has been ported successfully into RDL using RDLC2. Currently, an up-to-date RAMP Blue system using RDL is implemented and successfully runs UPC benchmarks. It is similar to the original RAMP Blue system – the software used is exactly the same and the hardware matches very closely. Although it took some time to complete this exercise, a lot of the time was spent dealing with the tools and understanding the original module interfaces.

This project should serve as a reference design for those using RDL, pro-

viding an example of the basic features of RDL and helping others to convert their design to RDL. During the course of this work, a number of suggestions for RDLC3 have been found, but by large RDLC2 was quite sufficient.

7 Acknowledgment

This project would not be possible without the support of the RAMP Blue team. I would like to thank my advisor Professor John Wawrzynek for his leadership, advice and support during this project. I would also like to thank Professor Dave Patterson for his suggestions and support.

I would like to thank Greg Gibeling, who developed RDL and RDLC, for providing detailed project guidance, as well as help with RDL and RAMP Blue. I would like to thank Alex Krasnov, who ported uCLinux and debugged the original RAMP Blue system, not to mention providing generous help with the original hardware and development and debugging environment. The original RAMP Blue system was developed by Andrew Schultz, Alex Krasnov, and Pierre-Yves Droz.

Lastly, I would like to thank Professor Wawrzynek, Professor Patterson and especially Greg Gibeling for commenting and proof-reading this paper.

References

[1] Chen Chang, John Wawrzynek, and Robert W. Brodersen. BEE2: A High-End Reconfigurable Computing System. 2005. IEEE Design and Test of Computers, March/April 2005 (Vol. 22, No. 2).

[2] Arvind, Krste Asanovic, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, David Patterson, Jan Rabaey, and John Wawrzynek. RAMP: Research Accelerator for Multiple Processors - A Community Vision for a Shared Experimental Parallel HW/SW Platform. Technical Report UCB/CSD-05-1412, 2005.

[3] Andrew Schultz. RAMP Blue: Design and Implementation of a Message Passing Multi-processor System on the BEE2. Master's thesis, 2006.

[4] Xilinx. *MicroBlaze Processor Reference Guide.* http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf.

[5] John Williams. MicroBlaze uClinux Project Home Page. http://www.itee.uq.edu.au/ jwilliams/mblaze-uclinux/.

[6] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre-YvesDroz. RampBlue: A Message-Passing Many Core System in FPGAs. 2007. Submitted for IEEE Micro 2007.

[7] Greg Gibeling, Andrew Schultz, and Krste Asanovic. The RAMP Architecture and Description Language. Technical report, 2005.