FPGA PROTOTYPING OF CUSTOM GPGPUS

A Thesis
Presented to
The Academic Faculty

by

Nimit Nigania

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School Of Computer Science

Georgia Institute of Technology
May 2014

FPGA PROTOTYPING OF CUSTOM GPGPUS

Approved by:

Professor Hyesoon Kim, Advisor
The School Of Computer Science
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
The School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Saibal Mukhopadhyay
The School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: Jan 2014

To my family

and friends.

PREFACE

Prototyping new systems on hardware is a time-consuming task with limited scope for architectural exploration. The aim of this work was to perform fast prototyping of general-purpose graphics processing units (GPGPUs) on field programmable gate arrays (FPGAs) using a novel tool chain. This hardware flow combined with the higher level simulation flow using the same source code allowed us to create a whole tool chain to study and build future architectures using new technologies. It also gave us enough flexibility at different granularities to make architectural decisions. We will also discuss some example systems that were built using this tool chain along with some results.

ACKNOWLEDGEMENTS

First, I would like to express my deepest gratitude to my advisor, Dr. Hyesoon Kim, whose energy, enthusiasm, and constant support inspired me from the beginning till the end of this project. I am indebted to her for her ideas, valuable guidance and the encouragement throughout.

I would like to thank various faculty members of Georgia Institute of Technology from whom I have benefited as a student. I express my sincere gratitude to Chad Kersey and Syed Minhaj Hassan, whose support proved invaluable during the course of the project. I would not have been able to finish the project in time if not for their help with the building blocks and design tools. Last but not least, I would like to thank Nagesh, Jaekyu, Jaewoong, Pranith, JooHwan, Hyojong, and Dilan, my friends in the HPArch lab for their support and the friendly atmosphere in the lab.

Finally, I dedicate this thesis to my parents, my brother, and Neha, who make even my smallest success meaningful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Motivation

The increasing complexity of current and future systems have made the task of a designer difficult and time consuming. There has also been an increased focus on the time taken to market these designs. These factors have contributed to a growing need for a higher level of design abstraction in order to increase design productivity. A typical system design cycle involves first building performance and functional models using C/C++/SystemC and then using hardware description language (HDL) for the actual design phase. If we can combine these two phases of architecture exploration and design, we can significantly decrease our design time.

To address the above issues, we are building a fast FPGA prototyping tool chain to explore various GPGPU-based architectures. The tool chain allows us to use the same code written in C++ to be translated to Verilog and also used for our simulation framework (SST [13]). Field programmable gate arrays (FPGAs) have been used to prototype hardware designs, for emulation and as accelerators. Since they are hardware implementations of the design, they run tests and simulations very fast before actually taping out an application specific integrated circuit (ASIC). FPGAs are also used as standalones to implement highly parallel and configurable application-specific designs. Having this emulation platform will allow us to run full-feature applications, which are generally the problem when running software simulations. For example, to run an application on the simulator (MacSim used in this study [11]), we first need to trace the application and then run the trace on the simulator. Since the simulator is slow, we trace only a small portion (hot loop) of the application, whereas on the FPGA we can run the whole application without having to generate traces. For an application that has, say, a billion instructions, running it on simulator like MacSim, which runs at generally 50 kips (kilo instructions per second), takes around five hours, whereas running it on the FPGA (2̃00 MHz) might take only a few seconds. The biggest difference or downside when running applications on an FPGA rather than an ASIC is the speed of the FPGA logic; for example, the Altera Stratix III FPGA used for this work can run only until about 5̃00MHz, whereas an ASIC can run at a much higher speed. The modern-day FPGA tools make prototyping (implementation and debugging) a relatively easy task compared to ASIC flow, hence this approach was taken.

The aim of this wok was to create a tool chain to prototype general-purpose graphics processing units on FPGAs. This hardware flow combined with the higher-level simulation flow using the same source code creates this whole tool chain to study future architectures, using new technologies. Most of the research has focused on either the simulation flow or the hardware flow. Having this flow for research would allow us to explore hardware designs and show results on the real prototype. In this work we demonstrate how we were able to create a full system on an FPGA and quickly try out various options. We created different systems using our parameterizable building blocks in a plug-and-play fashion. We started by building simple cores and then moved on to make bigger cores with single instruction multiple data (SIMD) support. Then, we demonstrated systems with multiple cores and also heterogeneous systems consisting of a large/big core and multiple small cores. All this helped us meet our goal of quickly prototyping complex systems.

1.2 Organization

The thesis is organized as follows:

Chapter 2 gives an overview of the overall system that was prototyped. It also discusses the various tools used and also gives an overview of CHDL, which is the programming language library used to write most of the system design code.

Chapter 3 describes each of the system components in more detail. We mainly cover some details of the ISA used, the design of the core, the cache, and the memory controller. Many obvious micro-architecture details have been omitted to keep the descriptions brief. After discussing the system components, we then discuss how we integrated all of these components and created example designs.

Chapter 4 shows the simulation setup along with some of the results obtained. We also show the resource consumption of the design on the FPGA board.

Chapter 5 discusses related work. It also concludes the work and discusses future directions.

CHAPTER II

DESIGN OVERVIEW AND TOOL CHAIN

2.1 System Overview

The aim of this work was to design a custom GP-GPU as a starting point to explore future designs. Since we wanted the whole system to be customized, a custom ISA called “HARP” was used for this work. The ISA is an MIPS-based ISA with support for several customizations, which are discussed in the next section. The key features are predication support, configurable instruction width, number of general-purpose and predicate registers, and vector support.

As for the design, the main components are the core, the cache, and the memory controller, as can be seen in Figure 1. Figure 1 is a simple design we used just to demonstrate different components. Each of these components is discussed separately in the next section. We will also look at the different possible systems we can design in Chapter 4. The core that implements the main pipeline and the memory management unit was written in “CHDL” which is our custom high-level synthesis tool.The cache was written in Verilog and the memory controller IP was generated using Altera’s FPGA tools (Quartus II, [1]). Even though we have built a system using a particular configuration, we can use this flow to try many different scenarios. For example, if we want to built a system that uses a new kind of memory system (example HMC [10]), all we would have to do is to replace the DDR2 controller we are using now with the IP of the new memory controller and everything else will remain the same. We can also add new instructions to the ISA and add support to the core (like support for multiple warps similar to commercial GPUs), keeping everything in the uncore part unchanged.

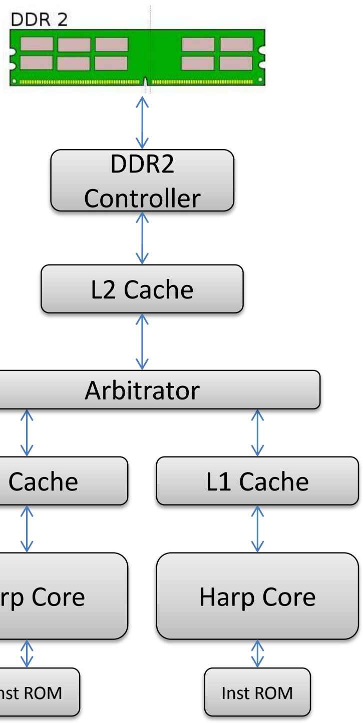


Figure 1: Basic system components

As we can see in Figure 1, our design supports multiple cores, and each core can support SIMD, thus creating a GPGPU or an SIMD CPU as one would like to call it. To enable support for multiple cores, we have also designed the arbitrator, which sends requests from the private caches of each core to the shared cache. Also similar to the way GPGPUs are designed, our design does not support coherence among the private caches of each of the cores. So, the programmer should be aware of this fact and must write code accordingly. For the current work, most benchmarks are written in HARP assembly but one of the future tasks as part of this project is to design a CUDA/OpenCL-HARP translator, which would allow us to run more general commercially available applications.

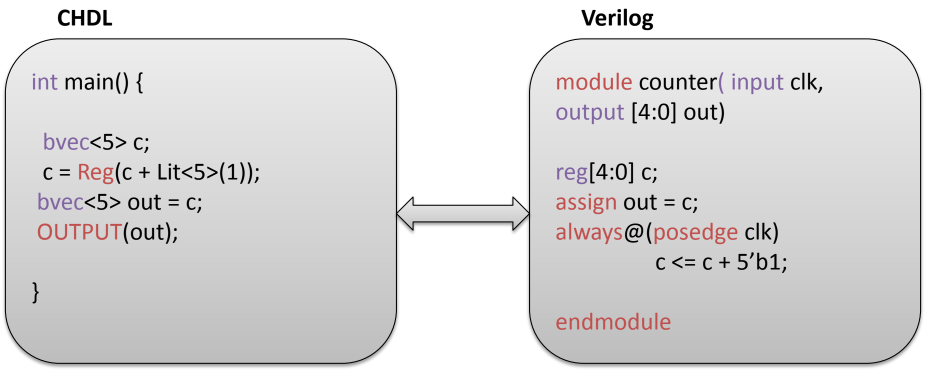
2.2 CHDL

CHDL is the custom environment used to generate HDL (hardware description language) using C++. It has similarities to System C and other HSL (high level synthesis) languages [7]. We used CHDL so that, as previously mentioned, the same code written in C++ can be used for the FPGA flow as well as for the simulation flow. CHDL is implemented as a source-source translator, which translates high level C++ to Verilog HDL. It offers a set of C++ libraries to support correct translation of code to its Verilog equivalent. Even though we write in C++, we still have to think to some extent about what the generated Verilog code will look like. For example, one thing to keep in mind is that assignments to wires (represented as a vector of Boolean) are continuous and we can assign the wire only once in our code. We discuss some examples later that give a good overview of this framework in terms of its ease of use and the restrictions it places on the programmer.

There are several advantages to taking the CHDL approach; as we write code in C++ it becomes easy to describe complex functions using simple code as compared to Verilog. Also, since we already have a library of commonly used functions, we can write code quickly; for example we have implementations of multiplexers, decoder, encoders, state machines etc. This allows us to significantly reduce our code density. The author in [17] shows that a 1M-gate design requires about 300k lines of RTL, whereas using a higher level of abstraction can reduce that by 7x-10x [7].

There are some down sides to using this high-level synthesis approach though. Since we can describe a lot of complex functionality using this approach, it might not always generate the most optimal Verilog code. We can expect the Verilog compiler to optimize the code and remove redundancies, but we would be able to generate better code if we write directly in Verilog. The authors in [7] give a good overview of HSL (high level synthesis) frameworks and discuss the quality of HDL generated. Another downside of using this approach is that it is very hard to debug, as the signal/variable names are lost in the code generated and we have to specify signals using special debug statements if we want to preserve the name to help us in debugging. It is also hard to fix or improve a critical path in the design to improve the performance.

Figure 2 shows a simple example of code written using CHDL compared with its Verilog equivalent. The Verilog code generated from CHDL is about 106 lines compared to about 10 lines in Verilog. As can be seen, this is a code for a simple 5 bit counter. The Verilog code is implied; as for the CHDL version, we represent the counter as a vector of boolean with 5 bits. The "Reg" statement translates to "always@(posedge clk) output <= input;" in Verilog. The assignments in CHDL are similar to the "assign" statement in Verilog.



```
int main() {

  bvec<5> c;
  c = Reg(c + Lit<5>(1));
 bvec<5> out = c;
 OUTPUT(out);

}
```

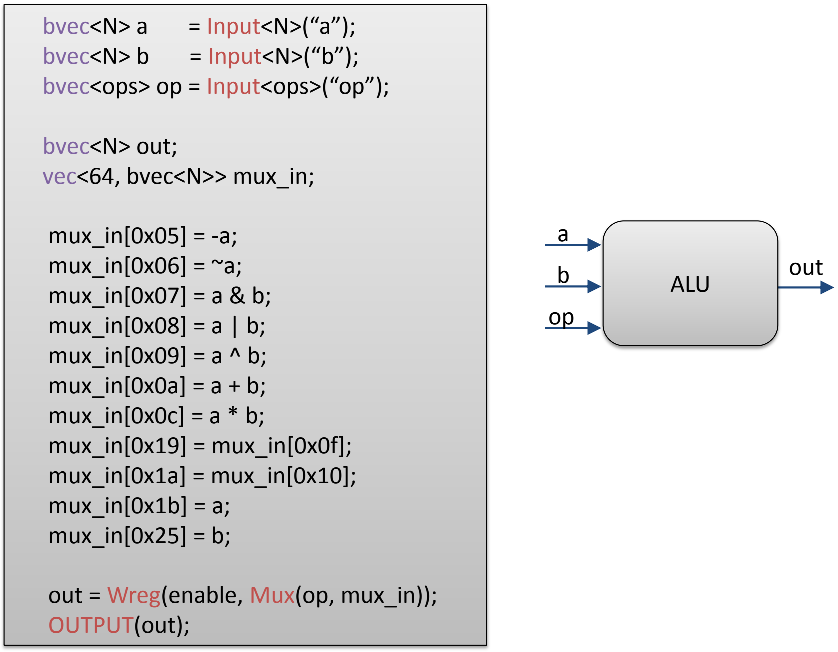
```
module counter( input clk,
output [4:0] out)

reg[4:0] c;
assign out = c;
always@(posedge clk)
            c <= c + 5'b1;

endmodule
```

Figure 2: Simple counter in CHDL and Verilog.

Similarly, if we want to implement basic operations we can directly use statements like those shown in Figure 3. This figure shows the implementation of a basic ALU where the final output is that coming out from the multiplexer, which selects the correct output based on the "op" signal, which acts as the selector.

2.3 FPGA board and Development Tools

The FPGA board used for our experiment is a Terasic DE3 board. It uses an Altera FPGA for prototyping. The FPGA used is a Stratix III 3SL150 FPGA with 142,000



```
bvec<N> a      = Input<N>(“a”);
bvec<N> b      = Input<N>(“b”);
bvec<ops> op = Input<ops>(“op”);

bvec<N> out;
vec<64, bvec<N>> mux_in;

 mux_in[0x05] = -a;
 mux_in[0x06] = ~a;
 mux_in[0x07] = a & b;
 mux_in[0x08] = a | b;
 mux_in[0x09] = a ^ b;
 mux_in[0x0a] = a + b;
 mux_in[0x0c] = a * b;
 mux_in[0x19] = mux_in[0x0f];
 mux_in[0x1a] = mux_in[0x10];
 mux_in[0x1b] = a;
 mux_in[0x25] = b;

 out = Wreg(enable, Mux(op, mux_in));
 OUTPUT(out);
```

Figure 3: Basic ALU in CHDL.

logic elements (LEs). The DE3 board also has support for DDR2 RAM, leds, 7-segement display, and connectors to stack multiple boards. Figure 4 shows a picture of the FPGA board used for this work.

We used Altera's FPGA tools for development purposes. The Quartus 11.3 tool [1] was used to synthesize the HDL and program the FPGA device via USB. We used ModelSim for the RTL and gate-level simulation experiments. As for other tools used, we also used open source tools like “iVerilog” [18] to compile our Verilog code and dump wave signals to a file, which we then viewed using “gtkwave” [6]. The DDR2 was used along with Altera's DDR2 memory controller IP. More about the DDR2 controller is discussed in Chapter 3. The Quartus tool with a built-in IP generator called “Megacore wizard” was used to generate the memory controller IPs along with other commonly used IPs like PLLs for managing clocks.

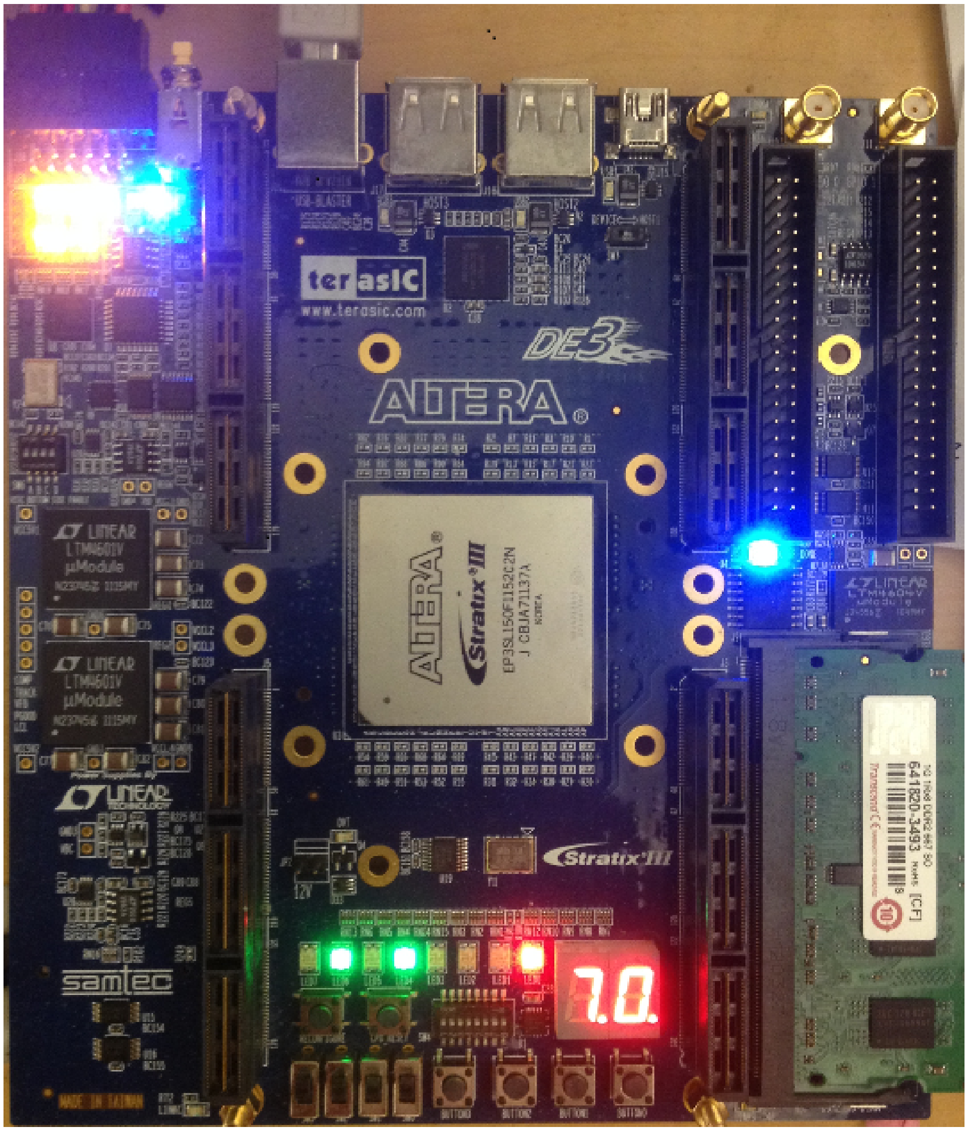


Figure 4: Altera Stratix III Board [14].

CHAPTER III

SYSTEM DESIGN

3.1 System Components

3.1.1 HARP ISA

The ISA used as part of this work was a custom RISC-based ISA. The ISA was developed under the name **HARP** which stands for Heterogeneous Architecture Research Project. The main features of the ISA are: full predication, SIMD support, and customizability. Many features of this ISA are customizable like the vector width, instruction length, number of general-purpose and predicate registers etc. The main reason to do this was to add support for new instructions for future architectures and also to allow adapting the ISA quickly to various architectural configurations. There are several types of instructions, depending on the number of arguments (register, immediate, predicate registers etc.), but all instructions are encoded in a similar fashion. That is, the most significant bit of each instruction indicates if it's predicate or not, the next field specifies the predicate register, the next field stands for the opcode, and so on.

The HARP ISA is also supported by its tool chain called "Harptool" which acts as an assembler, linker, and emulator. For this work the benchmarks used were written directly in HARP Assembly, which is very similar to RISC-based assembly programs. These benchmarks, written with the aim for testing and performance purposes, are listed in Table 1

We varied the input size (array size for array sum, matrix size for matrixmul, input range for sorting) and ran it on our design, which was used not only to test the overall design, but also to get an idea of the performance. These are very naive

Table 1: Micro-Benchmarks studied.

Single Lane Benchmarks	
Array Sum	Sum 240 numbers
Sieve of Eratosthenes	Finding prime numbers between 1 to 100
Bubble sort	Sort Numbers 0-9 using bubble sort
Matrix multiplication	Multiply two 8x8 matrices
Multi-Lane Benchmarks	
Array Sum	Sum 240 numbers (coalesced and un-coalesced version)
Matrix multiplication	Multiply two 8x8 matrices

applications compared to some of the complex benchmarks that are available, but these do provide good credibility for the design if not the performance as of now. One of the main focuses of writing these applications was to stress corner cases. Part of the future work is to design a tool that can translate CUDA or OpenCL code to HARP Assembly, once that is in place it will allow us to do more thorough testing.

The HARP ISA in its current version alos supports only a single simple console I/O to help debug and display the output. Any store instruction at an address 0x80000000 causes the HARP core to send the required data to the display (7-segment display or LEDs or VGA).

3.1.2 Core

3.1.2.1 Core Pipeline

The base HARP core design used as part of this work was a design written completely in CHLD. This section provides an overview of the base design of the core, which was modified for use as part of this work. The HARP core is an in-order issue and out-of-order completion pipelined processor. The pipeline stages are mainly Instruction Fetch, Decode/Register File access/Issue, Execute and WriteBack stage. The function of each stage is implied from its name and is shown in Figure 5.

The instruction fetch stage reads the instruction stored in the instruction ROM and passes it to the next stage. To improve performance, there is also a GHB (global history based) branch predictor and a BTB (branch target buffer). The PHT (pattern

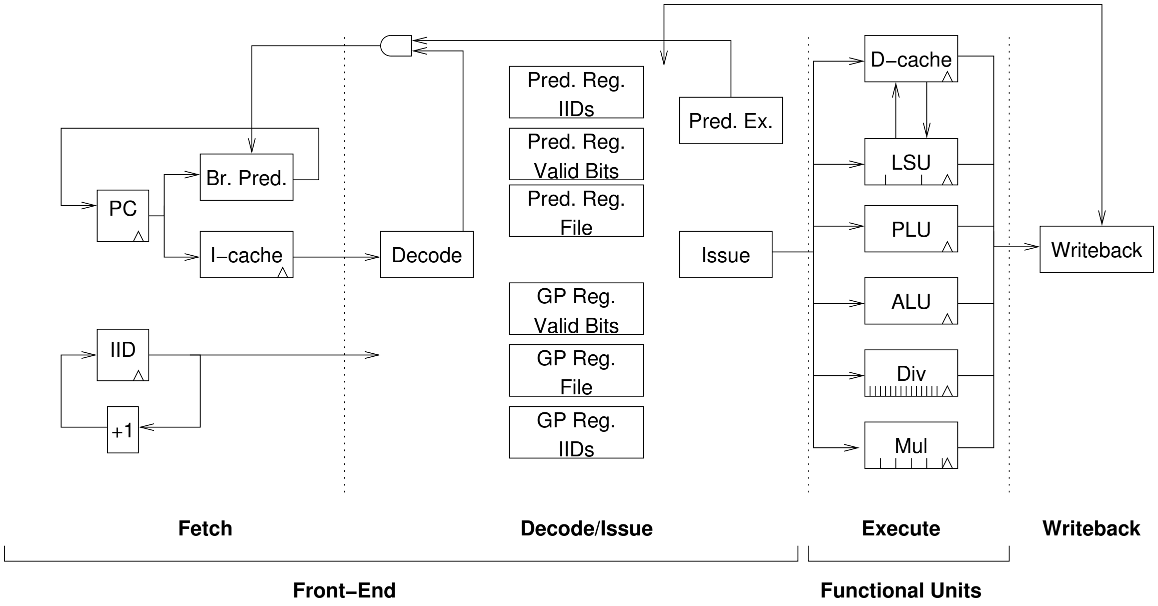


Figure 5: HARP core pipeline.

history table) of the GHB and BTB is indexed by XORing the PC and the branch history. The target branch address is obtained from the BTB and the direction from the PHT (pattern history table). Next, the decode stage decodes the instruction and reads all the registers required for each instruction. It also checks for dependencies and stalls the pipeline if the registers it needs are not valid, as the responsible instructions have not updated them yet. To handle dependencies, the design assigns a unique Instruction ID (IID) to each instruction, which is used to determine which instruction is responsible for updating the register file to its latest value and set the valid bit. So, each instruction updates the IID file for the register it writes to with its own IID and also sets the valid bit to zero before proceeding to the execute stage. The predication logic is also implemented in the decode stage. We convert the instruction to an NOP, depending on the value of the predicate register. There is no support for data forwarding in this design right now; it has been left for future work.

The execute stage has all the ALU units to implement the functions offered by the ISA. Additional modules like faster ALU units, say for example fast pipelined multiply, divide, or floating point operations, can be written separately in Verilog and integrated with our current design as and when needed. This stage also has the memory management unit (MMU) to handle memory accesses, which are discussed later. The write-back stage writes back the updated data to the register files and updates the state of the registers (set the valid bit) so the dependent instructions can now progress.

Factors such as SIMD width, number of registers, ROM size, and data/address width can be varied. We discuss more about the SIMD support in the next part and later sections discuss more about the memory management unit. We also allow for a version of the core without any cache support but with each lane having a small RAM and a ROM on its own. This variation is just to explore different architecture possibilities.

3.1.2.2 SIMD support

Writing code in CHDL allowed us to easily extend our core design for SIMD support. The whole register file was instantiated as many times as the number of lanes, hence converting each register to a vector register. For a given vector instruction, the same operation was done for each word in the vector register. Once all the vector registers are read, they are sent to the execute stage. If the instruction involved an immediate, then the same immediate was used for all the lanes.

In the execution unit each of the ALU units was instantiated as many times as the number of lanes and their corresponding inputs were passed on from the decode stage. Everything else (checking for dependencies logic etc.) remained mostly the same as in the case of a single lane (branch outcomes, predication outputs were determined only by the first lane). As of now, the core does not support any mask operations, so the same operation is done on all the words of the vector register. Adding support for masking via mask registers and also support for branch divergence will be part of the future work. The only big change in order to support SIMD instructions was done in the memory management unit. To enable this, a complex load store queue, shown in Figure 6, was designed and is discussed in the next section. Also, for SIMD support, changes had to be made in the cache, as the L1 cache now has to send and receive data at cache line granularity rather than word granularity; also the cache had to support masked writes in case of un-coalesced accesses.

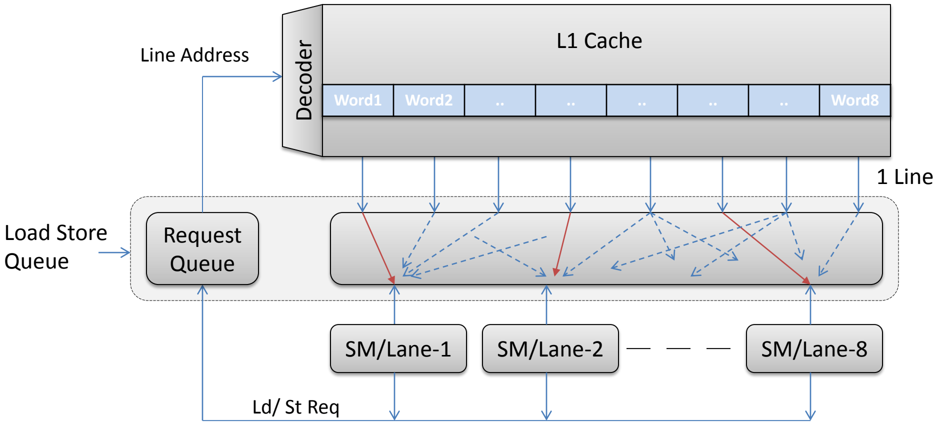


Figure 6: MMU or Load-Store unit interface.

3.1.2.3 Memory management unit (MMU)

The whole memory management unit and the cache were among more complex portions of the design. This section discusses the memory management unit/load-store queue/coalescing unit used in the execute stage of the core to send requests to the cache subsystem. The MMU can be divided into four major components, as seen in Figure 7:

- Front End
- Load Queue
- Store Queue
- Cache Interface logic

We now briefly discuss the functionality of each of the components for an SIMD HARP core. This SIMD support requires the address/data granularity of a cache line between the MMU and cache. The design in Figure 7 is an example of an 8-lane SIMD core along with an eight word cache line, so all lanes can together request one whole cache line every cycle. We also have a design variation without the MMU and blocking memory requests to save on logic resources.

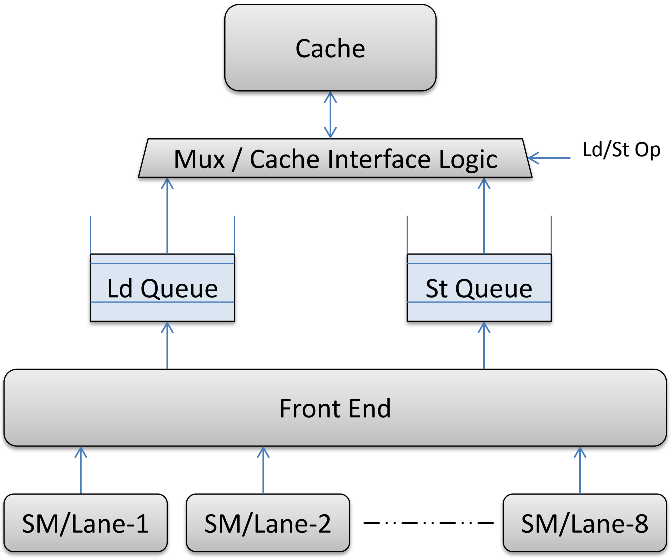


Figure 7: MMU components.

Front End: The front end receives the request from the core with addresses and data for each lane. It then tries to pack all lane requests made to the same cache line together and forms one request to be sent to the load/store queue. Hence, each entry in the load/store queue corresponds to a cache-line. For an un-coalesced request, the front end keeps creating as many new entries as unique cache lines. Along with the cache line address, each entry also keeps information about the lanes requesting or writing that cache line along with their corresponding word offsets in the cache line.
Load Queue: All load requests are passed on to this queue. The first request for an instruction is called the leader and all the following requests derived from the same instruction (in case of un-coalesced) are followers. The leader checks if data for all

lanes are loaded by waiting until its followers return the required data. The follower writes the data it receives from the memory to the corresponding lane data slot in the leader entry. Once data for all lanes are loaded (indicated by a loaded bit for each lane), we mark the entry as done and ready to retire. The load requests can be serviced in any order. We also enable load store forwarding (LSF) feature where, if an incoming load request sees data for the corresponding cache line in the store queue, then it directly gets the data from the store queue entry and is marked as done. The current implementation is conservative in the sense, it gets data forwarded only when the whole cache line is being written by a store queue entry. If only a partial store is made to a cache line, we make the load entry wait until the store entry commits.

Store Queue: The store queue gets the store data along with the cache line address from the front end. It then checks if there is a pending load entry for the same address; if there is, then we make this request wait until the matching load request is serviced to handle WAR (write after read) hazard. Once the store request is ready to commit, we send the data along with the valid/mask bits for each word to the cache. When both load and store queues want to send their requests to the cache, we give higher preference to requests from the load queue.

Cache Interface logic: This part of the MMU takes each pending entry from the load or store queue and sends it to the cache. There is a lot of switching logic involved mainly to handle requests from the store queue entry. This is because we allow for cross lane stores wherein each lane can write to any word in a cache line. We also have a broadcast logic built in, which allows multiple lanes in the same request to ask for any/same word in the cache line.

3.1.3 Cache

We use a two-level non-blocking, write-back, single-cycle latency cache for this design written completely in Verilog. A CHDL implementation was not written mainly because for the FPGA prototyping part we already have a base implementation of the cache design in Verilog and for the software simulation part we already have a cache design tightly integrated with our simulation infrastructure. We still have a CHDL design in the works. The input and output signals to the cache can be seen in Figure 8. The cache is parameterizable, where we can configure the cache line size (32 bytes used for our design), address/data width, and miss status handling register (MSHR) entries and also make data input/output granularity to be a word or cache line (for SIMD). Since we have designed a single-cycle latency cache, we need an extra 2x clock to allow us to perform tag read and write in the same clock cycle. This clock is fed in the top level system block and is generated via a phase locked loop (PLL). The L1 cache is a direct mapped cache and the L2 is 4-way set associative cache. We now discuss the main building blocks of the L1 cache; L2 is designed in a similar way except it uses a random cache line replacement policy and extra tag/data RAM modules for each extra way. Figure 9 shows the internal design of the cache with each of the components briefly discussed below.

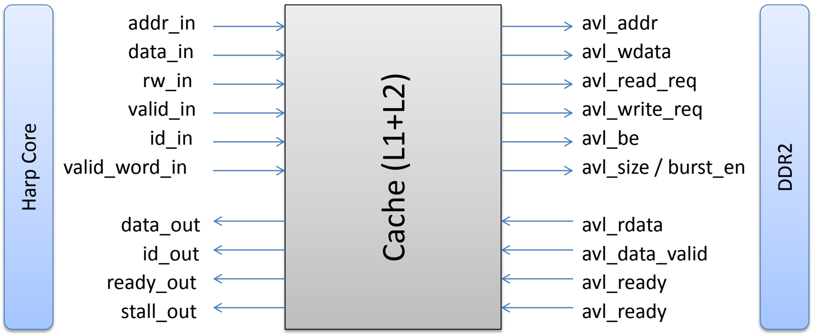


Figure 8: Cache interface signals for core and memory sides.

Data/Tag ram: These storage elements hold the data and tags along with dirty and valid bits. The actual data and tags are stored in eight two-ported 32-bit (word) RAMs (8x32 = 256 bit cache line). We need two ports ,as we need to service the

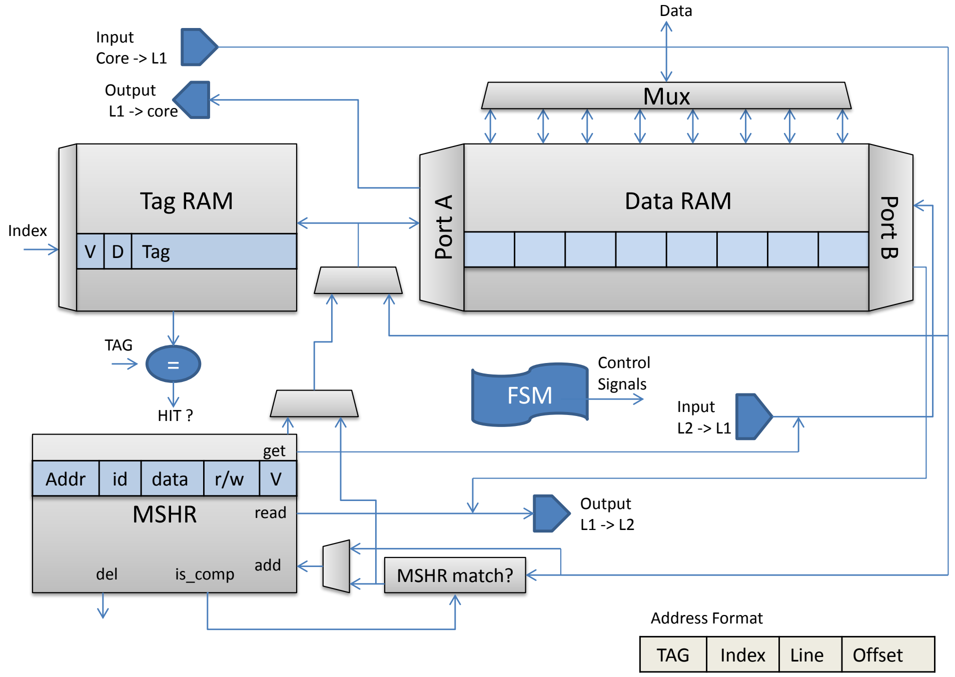


Figure 9: Cache design.

request coming from the core as well as lower level memory (L2 cache).

MSHR (Miss Status Handling Register): Since we implemented a non-blocking cache, we need an MSHR. This block stores the requests that miss in the cache and need to wait until the cache line it misses on is serviced by the lower level memory. This stores the data, addresses, read/write, and core request-id for each read/write operation. In the case, where a second request to the same waiting cache line occurs, we stall the cache and store the new request in a separate data structure. This new request is serviced only after the matching entry in the MSHR is serviced. A future extension to improve the performance will be to allow some kind of a piggy-back feature where, instead of stalling on the same cache line, we keep adding the requests to the same MSHR entry and service them all at once (though this will save us only a few cycles).

Non-blocking FSM (finite state machine): This state machine controls the overall operation of the cache. Since we are handling requests from the core and lower-level caches, the FSM keeps track of when a new request arrives from either side. Depending on the request, it issues control signals to the MSHR or the data/tag RAM to update their state or generates a stall signal to tell the core to stop sending more requests.

Arbiter: The arbiter is used for the multicore versions of our system, where each core has a private L1 cache and they share an L2 cache. The arbiter gets requests from L1 caches of all the cores and sends only one cache line request to the L2. It appends the core-id to the request-id coming from each core so it can return the data to the appropriate core when it is returned from the L2. Various arbitration schemes can be used, but for this work we used a priority encoder. The arbiter, like other components, is completely configurable allowing it to instantiate as many L1 caches as possible with only a small change in the Verilog code (more signals need to be added at the input port, as Verilog does not allow using a parameterized array of an input signal).

To get a better understanding of how these blocks interact to service simple load and store requests, we discuss a few simple cases below.

Read/Write Hit operation: When a new request arrives in the cache, we read the tag RAM to see if there is a match (first cycle of the 2x clock); if there is a hit, we then send the data from the data RAM to the core in the next cycle along with asserting the valid out bit for a load request. For a store request we update our data and tags (second cycle of 2x clock) and don't need to reply anything to the core.

Read/Write Miss operation: In the case of a cache line miss, the FSM allocates a new entry in the MSHR for this request. If the L2 cache is not stalled, it sends the request to the L2 and waits for the data. The L1 cache in the meantime can receive new requests if there is free space left in the MSHR. Each cache miss is treated in a similar way unless the new request is for the same line waiting in the MSHR. In this case, we stall the L1 cache and wait until the matching MSHR entry is freed. Once the data from L2 arrives, it directly updates the cache data and tag RAM. Then, the FSM gets the MSRH entry responsible for the miss and issues it again, making it a cache hit this time. Once issued, the MSHR entry is freed and the data along with valid output is sent to the core.

3.1.4 Memory Controller

The memory controller supported by the DE3 FPGA board used as part of this work is a DDR2 memory controller with an operating frequency ranging from 125 MHz - 533 MHz. We used a 1GB DDR2 RAM for this work running at 125MHz due to frequency limitations set by other components of the system. The DDR2 is connected on the board via 200 pins for clock and control signals coming from the FPGA. Figure 10 shows the top view of the memory controller block; more details can be found in [2]. The command generator receives the signals from the user side and passes it to the timing bank pool. The timing bank pool then checks for signal timings and if valid data is present in the data buffers. Then, it passes the request to the arbiter, which finally passes the command forward. The rank timer maintains rank-specific information to maintain correct functionality.

The DDR2 controller IP was generated using Altera's MegaWizard [1], which automatically generates the IP along with some example files for simulation purposes. Although it might seem straightforward but creating this IP with the correct parameters set was critical. Many times, the RTL simulation (using a system Verilog model for a DDR2 memory) of a generated controller IP was found to be working, but it failed on the board because one of the timing parameters was set incorrectly. Even the timing presets present in the Altera tool for the actual DDR2 DIMM used in the board were not correct. The correct parameters were later obtained from an example DDR2 IP project from the FPGA board vendor (Terasic). Also, since the number

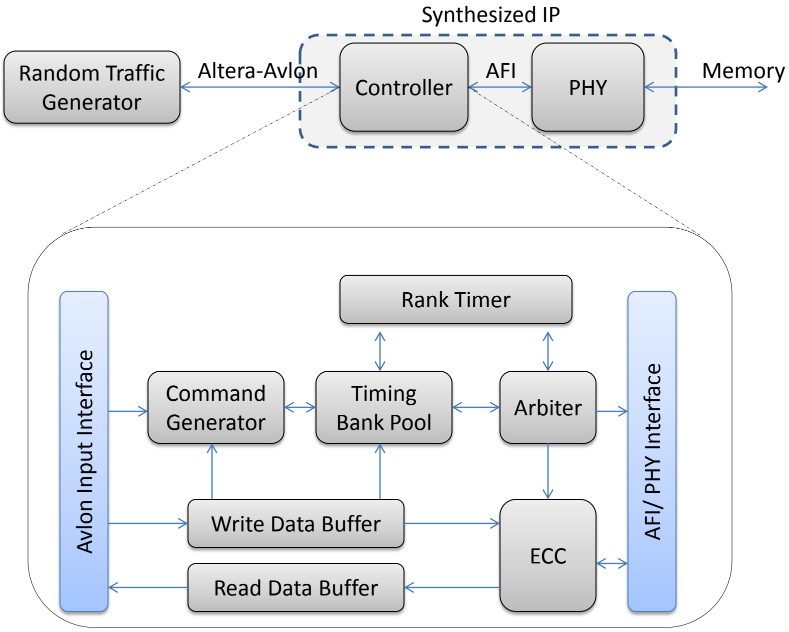


Figure 10: Test setup for memory controller IP and internal block diagram.

of pins that need to be connected to the DDR2 was significant, they all had to be set very carefully, again using the parameter obtained from the board vendor, which was also a confusing part as Altera by default recommended another configuration. The incorrect pin and parameter configurations were the main reasons behind failures seen initially. We found that many DDR2 IP parameters like the frequency and the row open/close policy were configurable via Altera's IP generator. Though we can customize the IP RTL itself, we can't reuse all of Altera's propriety IPs. Different scheduling policies like FRFCFS etc. can be tried and can be one research direction going forward for different types of system architectures.

Figure 11 shows the timing diagrams for requests being sent to the memory controller. The user side signal uses Altera's Avlon interface [2], and this is the prefix ('avl') given to these signals. Before using the controller with the rest of the system, it had to be tested using a random traffic generator (generated along with IP). It does a thorough test to make sure the DDR2 runs for the parameters set on the board.

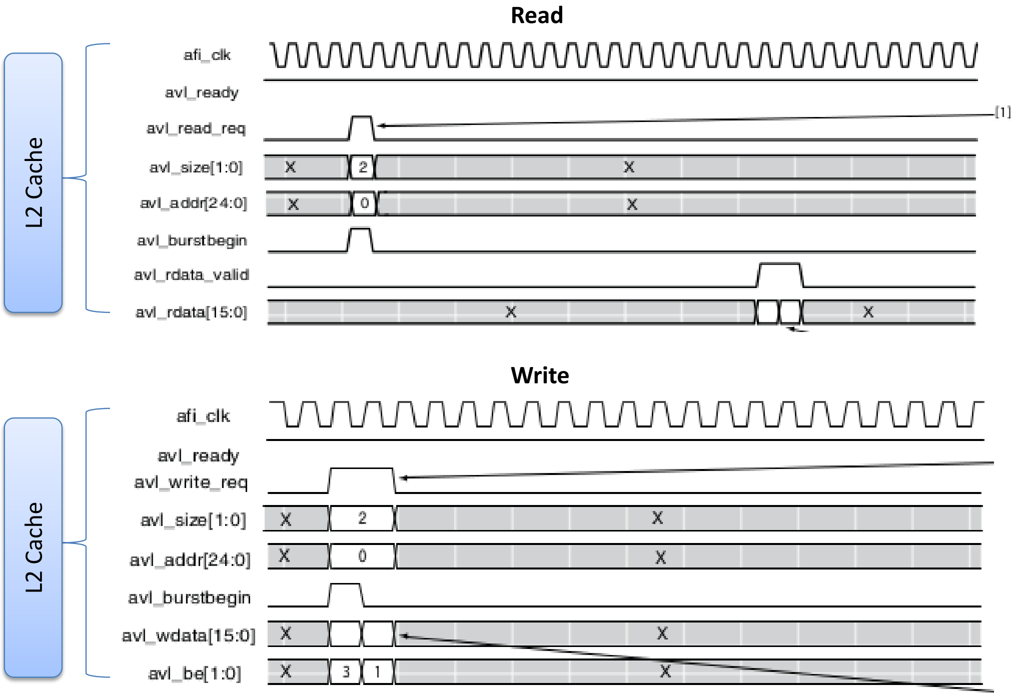


Figure 11: DDR cache side signals for read/write operation [2].

We first tested the controller in RTL using a DDR2 memory model and then on the FPGA board. Once DDR2 was tested, we started integrating it with the rest of the system. Since the cache interface is slightly different from the user interface signals of the memory controller, a memory controller wrapper block was written to interface them. The main difference was that the cache identifies each request via a unique ID, whereas the DDR2 just returns the data along with valid bits on a first-come first-served basis, as can be seen in the DDR2 wave diagrams in Figure 11. This wrapper is mainly a FIFO-based structure that stores the request along with the ID of incoming request coming from the cache and sends the data returned from the main memory back to the cache along with the original request id.

3.2 Putting it together: A Multicore and multilane GP-GPU system

Figure 12 shows the final system that was designed and tested on the FPGA board. We saw in previous sections in this chapter how each of the components, namely the SIMD core, MMU, cache, and the memory controller, was designed. Now the aim of the work was to integrate each of these components to make the overall system. Although this might sound simple to start with, it was a fairly time-consuming task, as it was through this process that many hidden bugs were discovered.

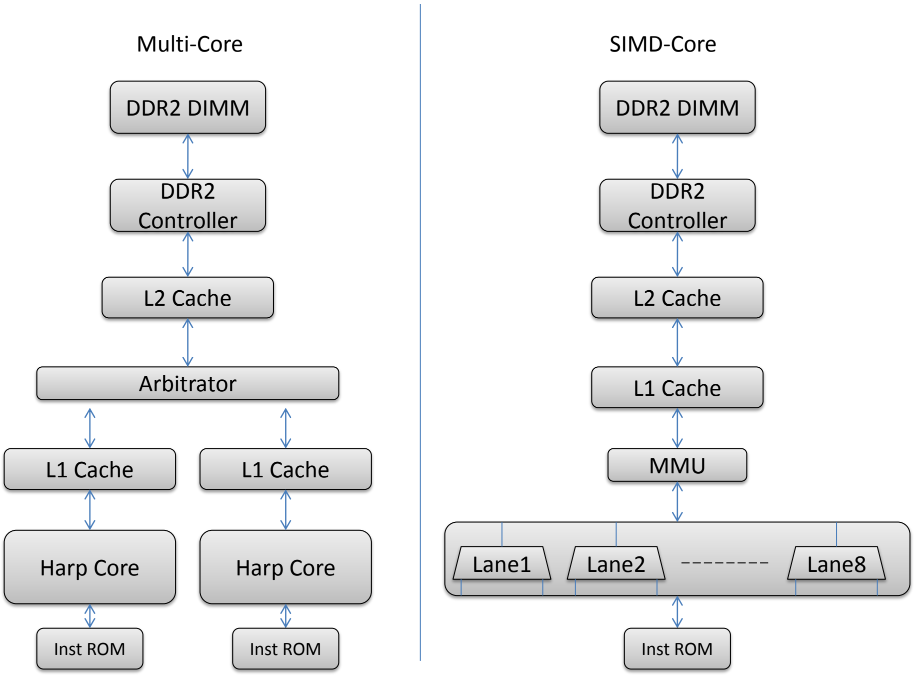


Figure 12: Possible system designs.

We started by testing each module before integrating them; this was done by writing separate testbenches (in Verilog) for each of the Verilog blocks. This led us to know if we were able to get the basic functionality out of the desired block. As many blocks were complex and had many inputs/outputs, it was hard to test them. Once basic building blocks were tested reasonably, we started integrating the whole system. To check overall functionality we started by testing a single lane HARP core system and made sure this was working and meeting all the timing constraints. For a system of this magnitude, it is generally hard to test all the corner cases and even now there might be some hidden bugs in the design. Since we tested our final system by writing applications rather than creating stimuli or a Verilog testbench, our tests were, to a large extent, thorough and helped us discover bugs in the very blocks which we had tested earlier individually.

The first step was to do RTL simulation of the whole system and make sure it was passing. Many functional bugs can be easily found and fixed at this stage. At the RTL stage, we can also replace each large component with a dummy model to isolate issues. Next, we synthesize and perform gate-level simulation; this is hard to debug, as signal names are changed or synthesized away and it takes a long time to reach the end of simulation. The best way to debug issues at this stage was to fix the warnings that were given by Altera's Quartus tool and this for us fixed many bugs in the design. Many issues, including the errors due to DDR2 timing issues, were hard to fix and took a lot of time. After this stage, we tested the system on the board by downloading the FPGA programming binary on the board. If it does not run even after this then the reasons we came across were mainly due to issues in the reset logic or the clock or the DDR timing parameters. One good practice is to use PLLs to generate cleaner clocks with less jitter. Most of the problems in my design were fixed using the above process. If one is still not able to isolate issues, then the next step is to use hardware debug IPs (Altera Signal Tap Analyzer), which can monitor signals on board and display them on the screen. This method was also used to debug some issues in the DDR2 as part of the test phase.

Once a single-lane version of the whole system was working on the board, we then tested a multi-lane version of the core using different tests to stress most of the commonly used blocks as shown in Figure 12. For example, we wrote test cases for doing a lot of un-coalesced, coalesced, and broadcast memory requests to determine whether or not the complex memory management unit was working properly. Once the single core (one lane or SIMD) was tested on the board, it was easy to create multi-core versions by instantiating more instances of the core with each core running its own separate application. Figure 12 shows one such system; as can be seen we have two cores, each having its own private L1 cache and sharing an L2 cache. As mentioned previously, we don't have support for coherence as of now. The extra component needed to make this multicore system was the L1 cache arbitrator, which schedules requests from different cores to the L2. For this work, a priority arbiter was used, but other possibilities can also be explored. To test it, we tried to run different combinations of applications on each of the cores accessing different data in the main memory.

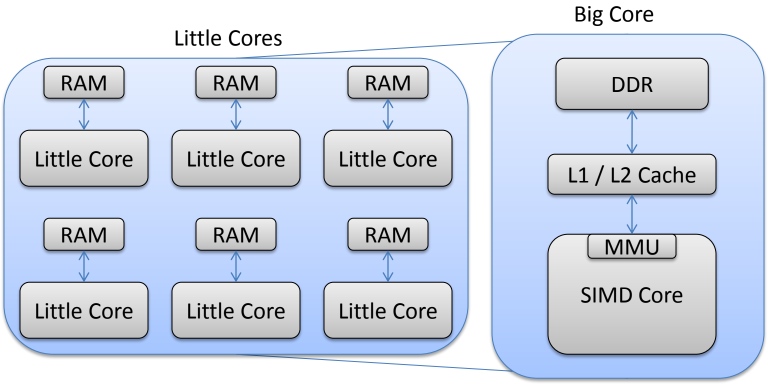


Figure 13: Heterogeneous system with one big and six little cores.

After testing the above systems, we tried to create novel heterogeneous systems from our basic building blocks, as shown in Figure 13. This was possible because of our tool chain and parameterized building blocks. We were able to test a heteregeneous system consisting of a complex big SIMD core and multiple smaller cores [3]. The bigger core has its own cache and memory subsystem (with the DDR) and the smaller cores have their own memory without any cache, hence consuming fewer FPGA resources. The results from the FPGA experiments for the systems above are discussed in the next section along with details of our heteregeneous system.

CHAPTER IV

MEASUREMENT RESULTS AND ANALYSIS

4.1 Board and Test Environment

The FPGA board used for the studies and experiment is a Terasic DE3 board. It uses an Altera FPGA for prototyping. The FPGA used is a Stratix III 3SL150 FPGA with the following properties: 142,000 logic elements (LEs), 5,499K total memory Kbits, 384 18x18-bit multipliers blocks and 736 user I/Os. The DE3 board also has support for DDR2 RAM, USB, leds, seven-segement display, and connectors to stack multiple boards. Figure 4 shows a picture of the FPGA board used for this work. We use a 1GB DDR2 DIMM. The benchmarksn as mentioned earlier in Table 1 were all written in HARP assembly.

To measure the performance, we calculated IPC (Instructions Per Cycle). The number of instructions was calculated using the Harptool emulator. The number of cycles can be easily measured on the simulator but, to measure the actual number of cycles on the FPGA board we created hardware counters in our design. We start counting when all components are initialized and stop when we reach the end of our benchmark. To indicate correctness we display three signals on the board to indicate test complete, pass, and fail for the core. The checksum logic for the pass signal is defined for each benchmark in our test module, say, for example we check the final sum of 'array sum' application with the known answer. We also have two more signals to show DDR2 controller initilization pass/fail. The number of cycles information from the hardware counter is displayed on the seven-segment display on the board.

4.2 Results

The basic parameters for the core and cache are set as shown in Table 2.

Table 2: Core configuration.

Core Configuration	
FPGA Device:	StratixIII (EP3SL150F1152C2)
FPGA flow tool:	Quartus II 13.0.1
Core Operating Freq:	62.5 MHz
DDR2 Operating Freq:	125 MHz
Instruction Width:	32 bits
L1 / L2 cache:	16KB / 128KB
General Registers:	16 32-bit Regs
Predicate Registers:	16
SIMD Lanes:	8
Instruction ROM:	1024KB

After creating the basic building blocks of our system, we tried to run performance and functional tests on a few example designs. Below we discuss three versions of the HARP Core and their performance for simple micro-benchmarks.

Single One-Lane HARP Core:

First we ran the benchmarks on a simple one-lane HARP core. For the benchmarks run, we did not see much benefit of using a complex load store queue, as the applications had a dependent instruction following a load instruction, which would stall the pipeline until the dependent load is serviced. Hence, to make the design simple we removed the complex load/store queue so the core now sends blocking requests to the memory subsystem, though independent instructions can still keep flowing in the pipeline. Table 3 shows the performance and Table 4 shows the logic utilization of this system. The low IPC is because almost all applications have a chain of dependent instructions.

Dual One-Lane HARP Core:

Next we instantiate two of the one-lane HARP cores above to form a multi-core system. Table 5 shows the performance and Table 6 shows the logic utilization.

Table 3: Single core one-lane performance.

Benchmark	Instructions	Cycles	IPC	Description
Array Sum:	2912	9598	0.3034	Sum 240 numbers
Sieve of Eratosthenes:	1611	5504	0.2926	Prime numbers in 1 to 100
Bubble sort:	799	2593	0.3081	Sort 0-9
Matrix multiplication:	6082	19855	0.3063	Multiply 8x8 matrices

Table 4: Logic utilization of a single one-lane core.

Logic Utilization	
FPGA Logic Utilization:	22%
Combinational ALUTs:	13% (14,850 / 113,600 ALUTs)
Dedicated Logic Registers:	12% (14,850 / 113,600 ALUTs)
Memory ALUTs:	3% (1,499 / 56,800 ALUTs)
Total block memory bits:	27% (1,505,792 / 5,630,976)
Total pins:	25% (189 / 744)

Table 5: Dual core one-lane performance.

Core-1 Benchmark	Core-2 Benchmark	Instructions	Cycles	IPC
Array Sum	Sieve of Eratosthenes	4523	9598	0.4712
Bubble Sort	Sieve of Eratosthenes	2410	5504	0.4378
Array Sum	Matrix multiplication	8994	19857	0.4529
Matrix multiplication	Matrix multiplication	12164	19857	0.6125

Table 6: Logic utilization of a dual one-lane core.

Logic Utilization	
FPGA Logic Utilization:	30%
Combinational ALUTs:	19% (21,945 / 113,600 ALUTs)
Dedicated Logic Registers:	17% (19,027 / 113,600 ALUTs)
Memory ALUTs:	3% (1,499 / 56,800 ALUTs)
Total block memory bits:	29% (1,647,104 / 5,630,976)
Total pins:	25% (189 / 744)

We can clearly see in Table 5 the benefits on performance of using multiple cores. Also, we can see that by using a simple design of the core and having a slightly more complex cache design helps us to keep the logic utilization to the minimum (scalable design). Given the above logic utilization we can easily instantiate more than eight cores on the FPGA device. Since there is no sharing and the applications are not memory internsive, the performance scales linearly with the number of cores for the above applications when we replicate the application across multiple cores, as we can see in the results for matrix multiplication.

Single SIMD eight-Lane HARP Core:

We also designed and ran an eight lane SIMD core on the FPGA board. We used simple applications initially to test the complex coalescing unit. Once that was done we tried to run a compute-internsive matrix multiplication code on the board. Comparing the performance numbers we can clearly see the advantage of using SIMD. The effective IPC would be about 8x this reported value, as the same instruction is executed across all the lanes. This comes at the cost of much higher logic utilization due to the coalescing unit and the duplication of ALU blocks and register files for the SIMD core. Table 7 shows the performance and Table 8 shows the logic utilization of this system.

Table 7: Eight-lane SIMD core performance.

Benchmark	Instructions	Cycles	IPC	Description
Matrix Multiplication:	929	2881	0.3224	Multiply matrices of size 8x8
Coalesced Vector Sum:	399	1068	0.3735	Sum 240 numbers
Un-Coalesced Vector Sum:	399	1300	0.3069	Sum 240 numbers

Using SIMD does give us performance benefits, as can be seen in Table 7. We also don't see a big difference in the performance of coalesced and uncoalesced array sum applications because we have a non-blocking cache and a load store queue. For the above tests, a maximum of four requests are being served simultaneously but can be easily increased by changing the parameter for MSHR entries and the FIFO size

Table 8: Logic utilization of a single eight-lane SIMD core.

Logic Utilization	
FPGA Logic Utilization:	54%
Combinational ALUTs:	37% (42,321 / 113,600 ALUTs)
Dedicated Logic Registers:	24% (27,517 / 113,600 ALUTs)
Memory ALUTs:	3% (1,499 / 56,800 ALUTs)
Total block memory bits:	27% (1,505,792 / 5,630,976)
Total pins:	25% (189 / 744)

for the cache-DDR interface.

The aim of all the above experiments was not only to show obvious benefits resulting from SIMD or multiple cores; we can always choose to build a system and select an application to show really good performance benefit. Rather, the aim of this chapter was to estabilish credibility and show the felxibility offered by the whole design. Depending on the resource constraints and performance goals, we can create a heterogeneous system where we have the choice to select the type of core (simple/complex/SIMD) and the number of cores.

4.3 Prototyping a big.LITTLE Heterogeneous HARP System

The above section demonstrates some of the conventional systems that can be designed using our tool set. In order to highlight the flexibility offered in quickly prototyping new systems, we tested a heterogeneous system consisting of a big SIMD core along with many small cores. This system has similarities to the big.LITTLE systems proposed by ARM [3]. This system also fits the description of a possible future system consisting of an HMC. An HMC has layers of memory stacked on top of a logic layer, which can be an ideal place to implement tiny processors for computing close to the memory. The vault in an HMC is one vertical division with its own logic and memory stack. Hence, a vault can be one such tiny core. In our system, the smaller cores simulate a vault. They all have their own memory close to them, alleviating the need for a cache. All the small cores work independently with a big SIMD core, which is running a compute-intensive application like matrix multiplication. Since there can be many such cores, we look at the example of one big core along with two, six, and eight little cores. Table 9 shows the performance of various heterogeneous systems tested and Table 10 shows the logic utilization of these systems.

Table 9: Heterogeneous HARP system performance.

Core Configuration	Total Instructions	Cycles	IPC	Description
1 Big - 2 Little:	2527	2881	0.8771	Big core: matmul Little cores: bubble sort 0-9
1 Big - 6 Little:	5723	3102	1.8444	Big core: matmul Little cores: bubble sort 0-9
1 Big - 8 Little:	7321	3041	2.4074	Big core: matmul Little cores: bubble sort 0-9

Table 10: Logic utilization of big.LITTLE HARP systems.

FPGA Resource	1 Big - 2 Little	1 Big - 6 Little	1 Big - 8 Little
FPGA Logic Utilization:	62%	80%	88%
Combinational ALUTs:	44%	58%	65% (73,590/113,600)
Dedicated Logic Registers:	27%	31%	33% (37,649/113,600)
Memory ALUTs:	3%	1%	1% (302/56,800)
Total block memory bits:	27%	28%	28% (1,590,474/5,630,976)

The above results showcase the benefits of having these kinds of heterogeneous systems as one of the many possible options. We can also easily see how scalable and flexible our tool chain is in designing these systems for future research.

CHAPTER V

FUTURE WORK AND CONCLUSION

5.1 Related Work

Much literature can be found that demonstrates a framework to translate C/C++ code or higher abstractions to HDLs like Verilog/VHDL for hardware design. But not much work exists that demonstrates using the same source code for software simulation as well as hardware prototyping for architectural exploration purposes; a summary of prior work can be found here [7]. In [9] the author proposes a new programming language inspired by C/C++. The work presented in [15] focuses on creating RTL for floating point algorithms written in C. The authors in [16], do compile time optimization of the application to find regions of code that can be accelerated on FPGAs and then generate VHDL for only those regions. Something very similar to SystemC was presented in [12], with not much flexibility to integrate it with regular C++ or Verilog models. The authors in [5] talk about how to generate variable pipelined functional units from high-level abstractions of HDL. A hardware/software codesign and simulation infrastructure for embedded systems was presented in [8]. A good design of an example system using these translation tools is shown in [4] and this work is very similar to our work. But, again, the research does not focus on architecture flexibility and exploration; instead it looks into designing a complex system in the least amount of time using the author's framework. Our work allows us to use a general open source simulation framework SST [13] to do more high-level coarse grained architecture exploration.

Most of the prior work focussed too deeply on generating optimal HDL code from C/C++ code and showed results on the FPGA comparing it with original Verilog/VHDL implementations. These works were either too application focussed or involved a major learning curve for the programmer, which has prevented broad acceptance of these HSL frameworks, as they missed out on showcasing its impact on future systems. A good overview of current HLS frameworks can be found in [7] but again it is a bit more inclined towards the hardware synthesis part. In this work we took a look at the broader picture. Given that we have something that translates C/C++ code to HDL and can also be used for our software simulation, we wondered what kind of possible architectures could be easily explored. The work mentioned in this thesis focused on the FPGA flow side of things, but the overall aim of the whole project is still focused on future possible architecture exploration.

5.2 Future Work

Since this work is part of a multi-year, multi-person project, many extensions are planned for this work. As for changes in the core part of the design, changes like adding support to handle multiple warps, and branch divergence using mask registers will be one important feature that will make this design truly comparable to modern-day GPGPUs. Also a feature like data forwarding can be added if we find that making the core more CPU like will be beneficial for certain kind of systems.

The applications written right now are all in HARP assembly, but to allow us to run of the shelf CUDA or OpenCL applications, a software translator tool is being worked on that can generate HARP assembly from these binaries.

The next major part of the future work is to explore future systems using new memory technologies like the HMC (Hybrid Memory Cube) [10]. Building these systems would require only isolated changes to the existing design. For example, to use an FPGA board with HMC, we would only have to generate a new memory interface IP for the HMC (as the controller is integrated in the logic layer of the

HMC) and use this with the rest of the system.

5.3 Conclusion

To conclude, we showcased a tool chain and possible designs to allow quick prototyping of GPGPU designs on real hardware. Integrating the FPGA protyping flow with software simulation infrastructure will allow us to explore future architectures at different levels of granulatiry. By creating a parameterized design we can change many aspects of our design to affect performance under given design constraints. The flexibility offered by CHDL was also shown, which allowed us to easily add more features to our system if needed, like SIMD support. We were able to see benefits of the SIMD version of the core using a coalescing unit over the single lane version. This work also discussed a few different systems that were emulated on hardware with only minor changes to the base design flow.

REFERENCES

[1] Altera Corp., "Altera Quartus II Design Software." http://www.altera.com/products/software/sfw-index.jsp. [Online; accessed 1-Aug-2013].

[2] Altera Corp., "Altera ddr and ddr2 sdram controller compiler user guide." http://www.altera.com/literature/ug/ug_ddr_sdram.pdf, 2009. [Online; accessed 1-Nov-2013].

[3] ARM Holdings, "big.LITTLE Processing with ARM Cortex-A15 and Cortex-A7." http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, 2011. [Online; accessed 1-Dec-2013].

[4] Barroso, L. A., Gharachorloo, K., and Ravishankar, M., "Managing complexity in the piranha server-class processor design," in *In 2nd Workshop on Complexity-Effective Design held in conjunction with the 27th International Symposium on Computer Architecture*, 2001.

[5] Ben-Asher, Y. and Rotem, N., "Synthesis for variable pipelined function units," in *System-on-Chip, 2008. SOC 2008. International Symposium on*, pp. 1–4, 2008.

[6] Bybell, A., "Gtkwave 3.3 wave analyzer user's guide." http://gtkwave.sourceforge.net/gtkwave.pdf, 2013. [Online; accessed 1-Dec-2013].

[7] Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z., "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[8] Esmaeilzadeh, H., Moghimi, A., Ebrahimi, E., Lucas, C., Navabi, Z., and Fakhraie, S. M., "Dcim++: a c++ library for object oriented hardware design and distributed simulation," in *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pp. 4 pp.–1286, 2006.

[9] Grelck, C., "Single assignment c (sac) high productivity meets high performance: High productivity meets high performance," in *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, (Berlin, Heidelberg), pp. 207–278, Springer-Verlag, 2012.

[10] Hybrid Memory Cube Consortium, "Hybrid memory cube specification 1.0." http://hybridmemorycube.org/files/SiteDownloads/HMC_Specification%201_0.pdf, 2013. [Online; accessed 1-Dec-2013].

[11] Kim, H., Lee, J., Lakshminarayana, N. B., Sim, J., Lim, J., and Pho, T., "Macsim: A cpu-gpu heterogeneous simulation framework."

[12] Liao, S., Tjiang, S., and Gupta, R., "An efficient implementation of reactivity for modeling hardware in the scenic design environment," in *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, (New York, NY, USA), pp. 70–75, ACM, 1997.

[13] Rodrigues, A. F., Hemmert, K. S., Barrett, B. W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., Cooper-Balls, E., and Jacob, B., "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 37–42, Mar. 2011.

[14] Terasic, "De3 user manual." `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=260&PartNo=4`. [Online; accessed 1-Dec-2013].

[15] Tripp, J., Peterson, K., Ahrens, C., Poznanovic, J., and Gokhale, M., "Trident: an fpga compiler framework for floating-point algorithms," in *Field Programmable Logic and Applications, 2005. International Conference on*, pp. 317–322, 2005.

[16] Villarreal, J., Park, A., Najjar, W., and Halstead, R., "Designing modular hardware accelerators in c with roccc 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pp. 127–134, 2010.

[17] Wakabayashi, K., "C-based behavioral synthesis and verification analysis on industrial design examples," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, (Piscataway, NJ, USA), pp. 344–348, IEEE Press, 2004.

[18] Williams, S. and Baxter, M., "Icarus verilog: Open-source verilog more than a year later," *Linux J.*, vol. 2002, pp. 3–, July 2002.