

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
December 2013

Copyright © 2013 by Nimit Nigania

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: December 2013

*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 do 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. We will also discuss some example systems which were built using this tool chain along with some results.

ACKNOWLEDGEMENTS

First of all, I would like to express deepest sense of gratitude to my advisor Dr. Hyesoon Kim whose energy, enthusiasm and constant support inspired me from the beginning till the end of the 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 had proved invaluable during the course of the project. I would not have been able to finish 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 lab.

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

TABLE OF CONTENTS

DEDICATION	iii
PREFACE	iv
ACKNOWLEDGEMENTS	v
LIST OF TABLES	viii
LIST OF FIGURES	ix
I INTRODUCTION	2
1.1 Motivation	2
1.2 Organization	3
II DESIGN OVERVIEW AND TOOL CHAIN	5
2.1 System Overview	5
2.2 CHDL	7
2.3 FPGA board and Development Tools	8
III SYSTEM DESIGN	11
3.1 System Components	11
3.1.1 Harp ISA	11
3.1.2 Core	12
3.1.3 Cache	18
3.1.4 Memory Controller	21
3.2 Putting it together: A Multicore and multilane GP-GPU system	24
IV MEASUREMENT RESULTS AND ANALYSIS	27
4.1 Board and Test Environment	27
4.2 Results	28
4.3 Prototyping a big.LITTLE Heterogeneous Harp System	31
V FUTURE WORK AND CONCLUSION	33
5.1 Related Work	33

5.2	Future Work	34
5.3	Conclusion	35
	REFERENCES	36

LIST OF TABLES

1	Micro-Benchmarks Studied	12
2	Core Configuration.	28
3	Single Core 1-Lane Performance	29
4	Logic Utilization Of A Single 1-Lane Core	29
5	Dual Core 1-Lane Performance	29
6	Logic Utilization Of A Dual 1-Lane Core	29
7	8-Lane SIMD Core Performance	30
8	Logic Utilization Of A Single 8-Lane SIMD Core	31
9	Heterogeneous Harp System Performance	32
10	Logic Utilization Of big.LITTLE Harp Systems	32

LIST OF FIGURES

1	Basic system components	6
2	Simple Counter in CHDL and Verilog	8
3	Basic ALU in CHDL	9
4	Altera Stratix III Board [14]	10
5	Harp Core Pipeline	13
6	MMU or LoadStore Unit Interface	15
7	MMU Components	16
8	Cache Interface Signals for Core and Memory Sides	18
9	Cache Design	19
10	Test Setup for Memory Controller IP and Internal Block Diagram . .	22
11	DDR cache side signals for read/write operation [2]	23
12	Possible System Designs	24
13	Heterogeneous System with 1 Big and 6 Little Cores	26

FPGA	Field Programmable Gate Arrays
ASIC	Application Specific Integrated Circuit
GPGPU	General Purpose Graphics Processing Units
IP	Intellectual Property
SIMD	Single Instruction Multiple Data
MMU	Memory Management Unit
PLL	Phase Locked Loop
RAM	Random Access Memory
ROM	Read Only Memory
MSHR	Miss Status Handling Register

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 time 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 HDL(Hardware Description Language) for the actual design phase. If we can combine these two phases of architecture exploration and design, then 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 implementation of the design they are very fast to run tests and simulations before actually taping out an ASIC. FPGAs are used standalone also 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 which has say a billion instructions running on the simulator like MacSim which runs at generally 50 kips(kilo instructions per second) takes around 5 hours where as running on the FPGA (~ 200 MHz) might take us 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 till about ~ 500 MHz where as an ASIC can run at much higher speed. The modern day FPGA tools make prototyping (implementation and debugging) a relatively easy task as compared to ASIC flow hence this approach was taken.

The aim of this work was to create a tool chain to prototype general purpose graphics processing unit 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 quickly. Most of the work in 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 with building simple cores and then moved on to make bigger cores with SIMD support. Then we demonstrated systems with multiple cores and also heterogeneous systems consisting of a 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 which was prototyped. It also discusses about the various tools used and also gives an overview of CHDL which is

the programming language 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 skipped to keep the descriptions brief. After discussing about the system components we then discuss about how we went on to integrate all these components and create example designs.

Chapter 4 shows the simulation setup along with some of the results obtained. We also show the resource consumption by 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 a MIPS based ISA with support for several customizations which will be discussed in the next section. The key features are predication support, configurable instruction width, no. 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 very simple design which we have used just to demonstrate different components. Each of these components will be discussed separately in the next section. We will also look at the different possible systems we can design in Chapter 4. The core which 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 build a system which uses a new kind of memory system (example HMC [10]) all we would have to do is to replace the DDR2 controller which we are using right 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.

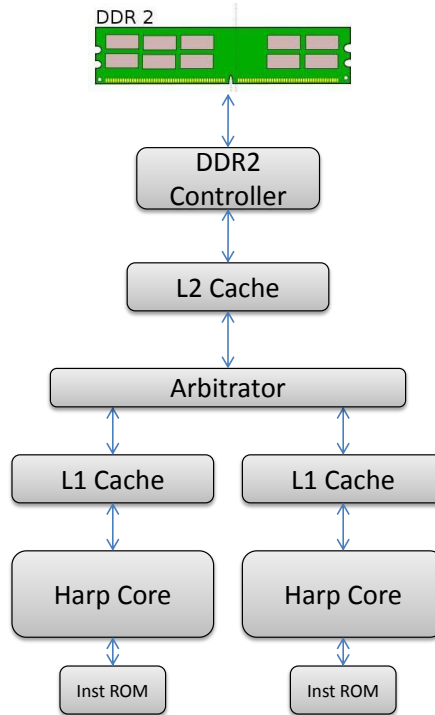


Figure 1: Basic system components

As we can see in Figure 1, our design supports multiple cores and each core can support SIMD hence creating a GPGPU or a 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 to HARP translator which would allow us to run more general commercially available applications.

2.2 *CHDL*

CHDL is the custom environment which was 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 the simulation flow. CHDL is implemented as a source to 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 as to how 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 will discuss some examples later which 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 of 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. [17] shows that a 1M-gate design requires about 300k lines of RTL. Whereas, using higher level of abstraction can reduce that by 7x-10x [7].

There are some down sides of using this high level synthesis approach though. Since we can describe lot of complex functionality using this approach hence 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. [7] gives a good overview of HSL (High Level Synthesis) frameworks and discusses about 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 and 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 bool 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.

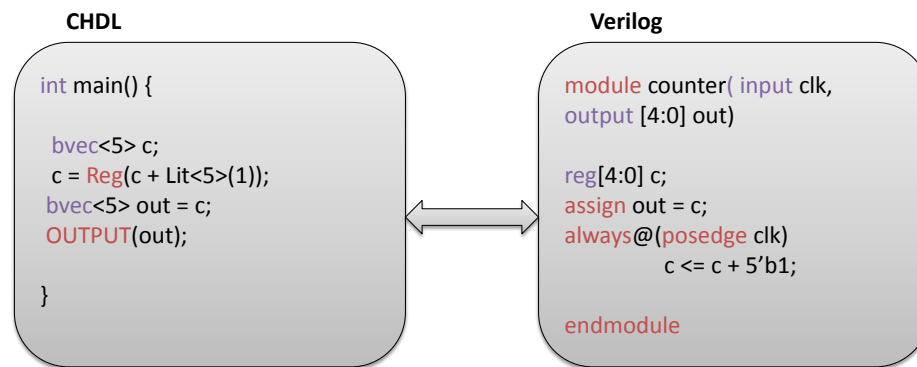


Figure 2: Simple Counter in CHDL and Verilog

Similarly if we want to implement basic operations we can directly use statements like that 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

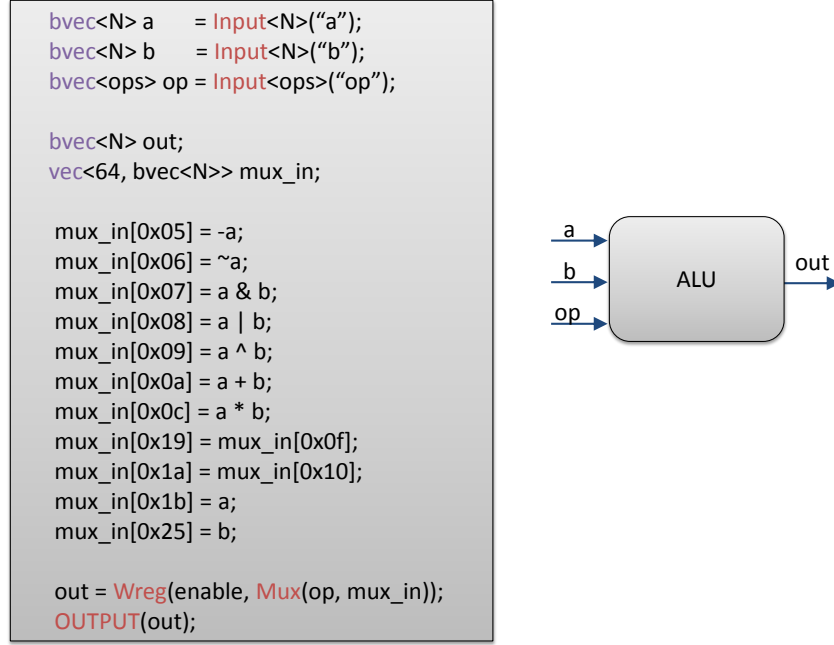


Figure 3: Basic ALU in CHDL

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 will be discussed in the 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.

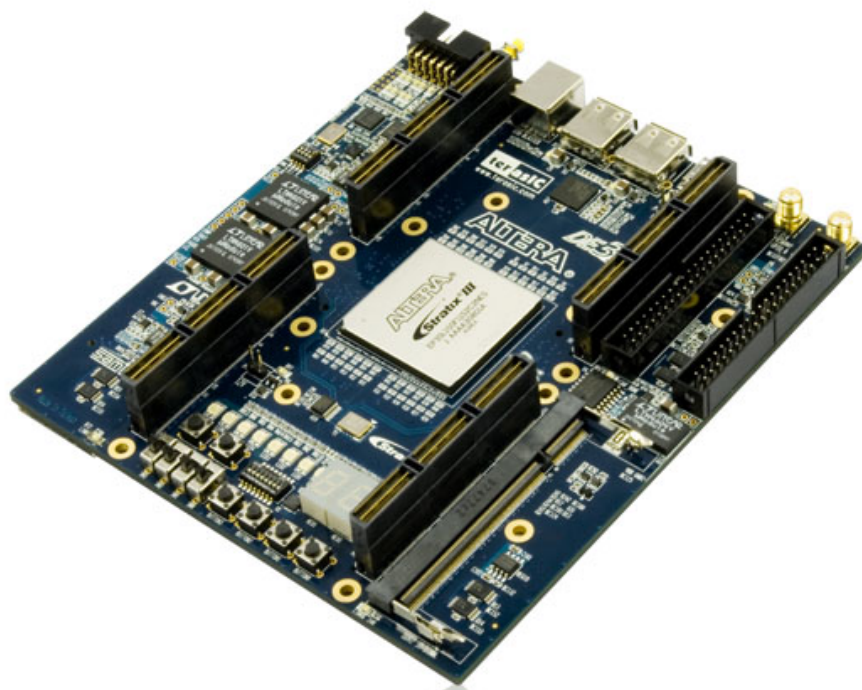


Figure 4: Altera Stratix III Board [14]

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 to also 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 similar fashion. That is, the most significant bit of each instruction indicates if it's predicate or not, 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 much 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 to not only test the overall design but also to get an idea of the performance. These are very naive

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

Table 1: Micro-Benchmarks Studied

applications compared to some of the complex benchmarks which are available out there but these do provide a good credibility to 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 the design of a tool which 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 also in its current version 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 will give 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 of the stages is implied from their names and showed 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

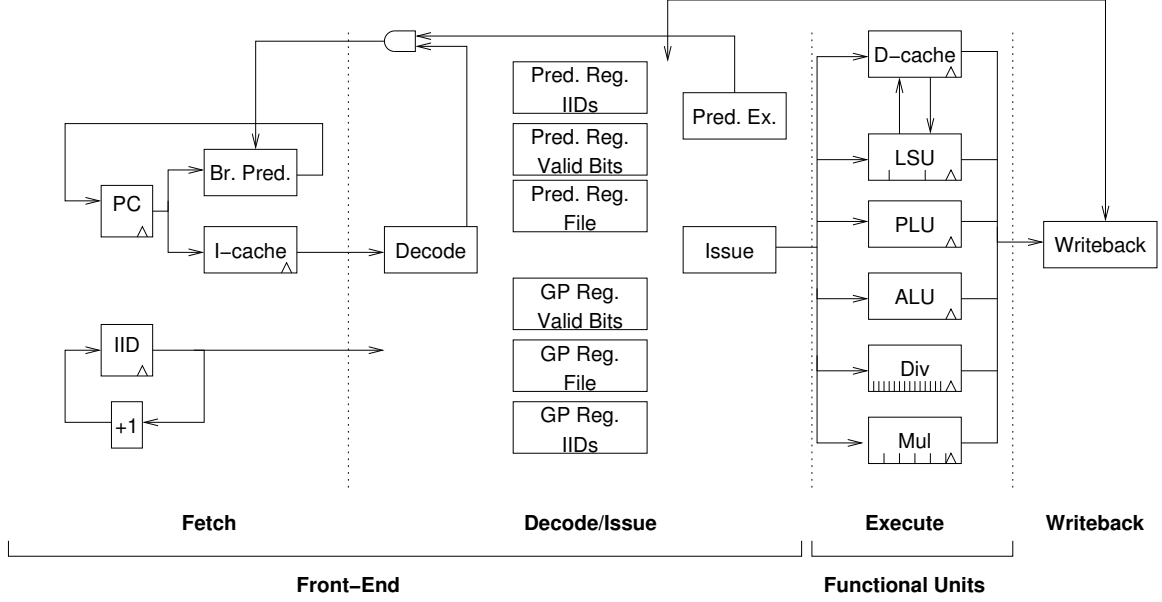


Figure 5: Harp Core Pipeline

history based) branch predictor and a BTB (branch target buffer). The PHT (pattern 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 and this is used to determine which instruction is responsible to update 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 a 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 will be 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 will discuss more about the SIMD support in the next part and the later sections will 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 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 register 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 were instantiated as many times as 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 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 as will be discussed in the next section. Also for SIMD support changes had to be done 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.

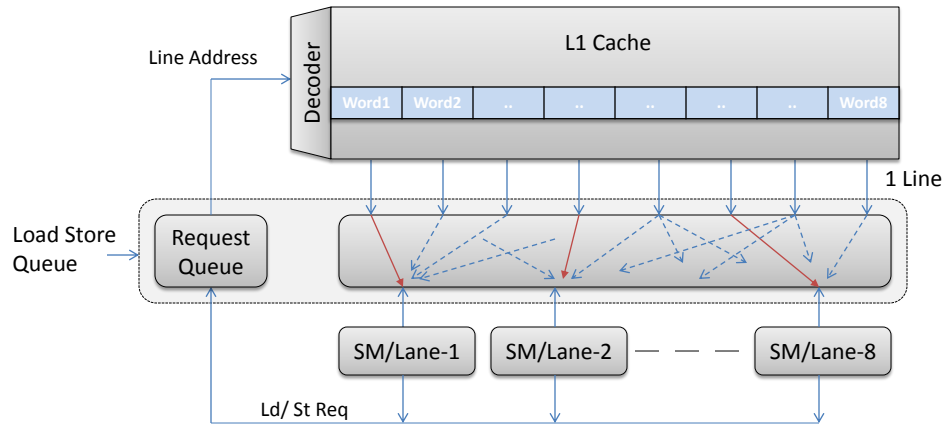


Figure 6: MMU or LoadStore Unit Interface

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 talks about 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 4 major components as can be seen in Figure 7:

- Front End
- Load Queue
- Store Queue

- Cache Interface logic

We will now briefly discuss the functionality of each of the components for a SIMD Harp core. This SIMD support requires the address/data granularity of a cache line between MMU and cache. The design in Figure 7 is an example of an 8-lane SIMD core along with an 8 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.

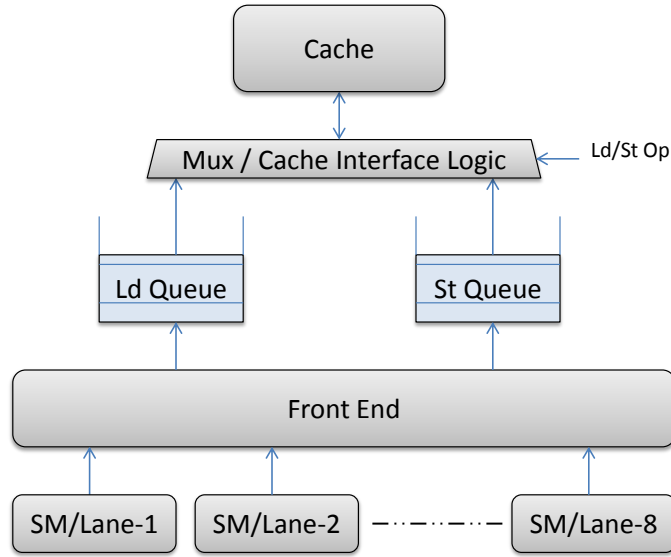


Figure 7: MMU Components

Front End: 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 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 same

instruction (in case of un-coalesced) are followers. The leader checks if data for all lanes are loaded by waiting till 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 LSF (Load Store Forwarding) 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 done to line we make the load entry wait until the store entry commits.

Store Queue: The store queue gets store data along with 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 then we send the data along with the valid/mask bits for each word to the cache. When both load and store queue 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 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 2-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 had 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

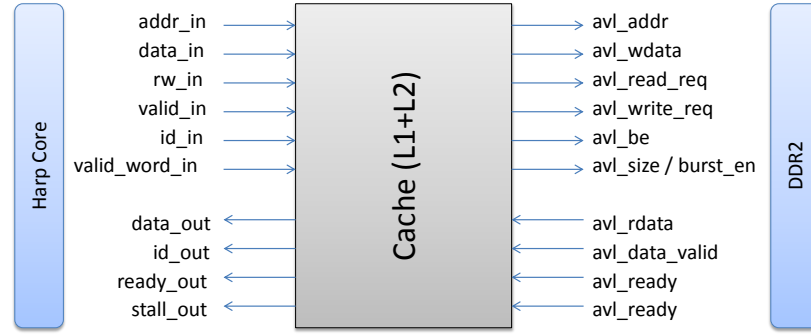


Figure 8: Cache Interface Signals for Core and Memory Sides

in Figure 8. The cache is parameterizable where we can configure the cache line size (32 bytes used for our design), address/data width, MSHR entries and also make data input/output granularity to be 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 do tag read and write in the same clock cycle. This clock is fed in the top level system block and is generated via a PLL. The L1 cache is a direct mapped cache and the L2 is 4-way set associative cache. We will now discuss the main building blocks of the L1 cache, L2 is designed in the 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 :

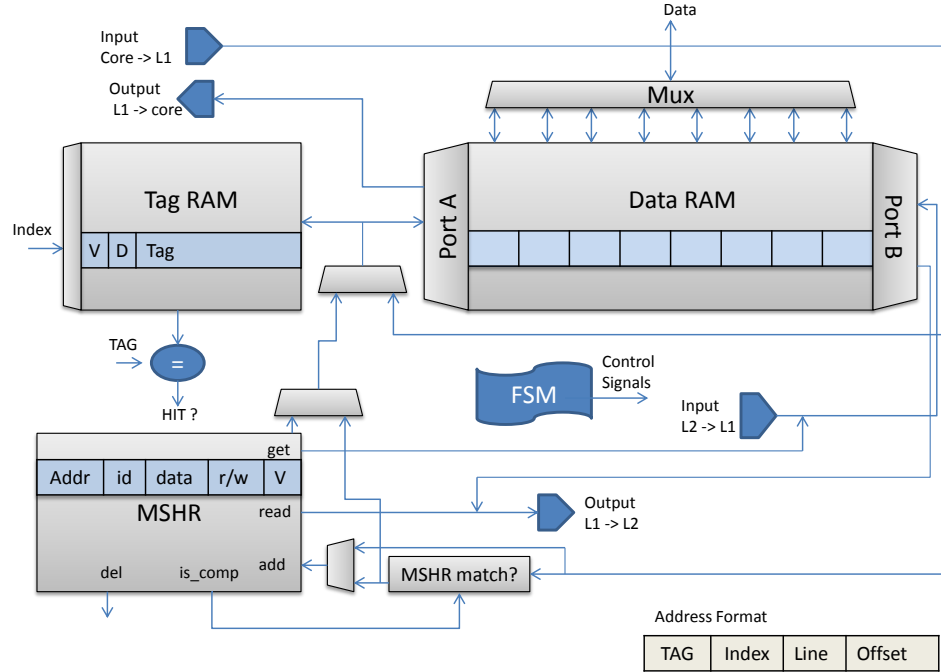


Figure 9: Cache Design

Data/Tag ram: These storage elements hold the data and also tags along with dirty and valid bits. The actual data and tags are stored in eight 2-ported 32-bit (word) RAMs ($8 \times 32 = 256$ bit cache line). We need 2 ports as we need to service 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 which miss in the cache and need to be wait till the cache lines 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 case, a second request to the same waiting cache line happens 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 whenever 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: This is used for the multi-core versions of our system where each core has a private L1 cache and they share a 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 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 (1st 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 (2nd cycle of 2x clock) and don't need to reply anything to the core.

Read/Write Miss operation: In 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 then it sends the request to the L2 and waits for the data. The L1 cache in the mean time 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 MSHR 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 here [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 the Altera's MegaWizard [1] which automatically generates the IP along with some example files for simulation purposes. Although it might seem straight forward but creating this IP with the correct parameters set was very critical. Many times the RTL simulation (using a system verilog model for a DDR2 memory) of generated controller IP was found to be working but it failed on the board because one of the timing parameters was set incorrectly. Even using 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

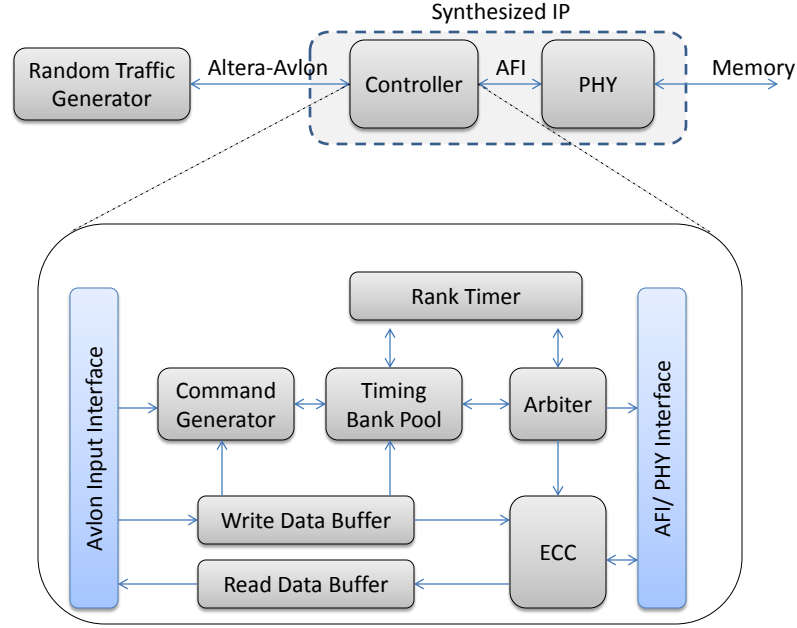


Figure 10: Test Setup for Memory Controller IP and Internal Block Diagram

example DDR2 IP project from the FPGA board vendor (Terasic). Also since the number of pins which need to be connected to the DDR2 were significant they all had to be set very carefully again using the parameter obtained from 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 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 but we can’t reuse all of Altera’s propriety IP. 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

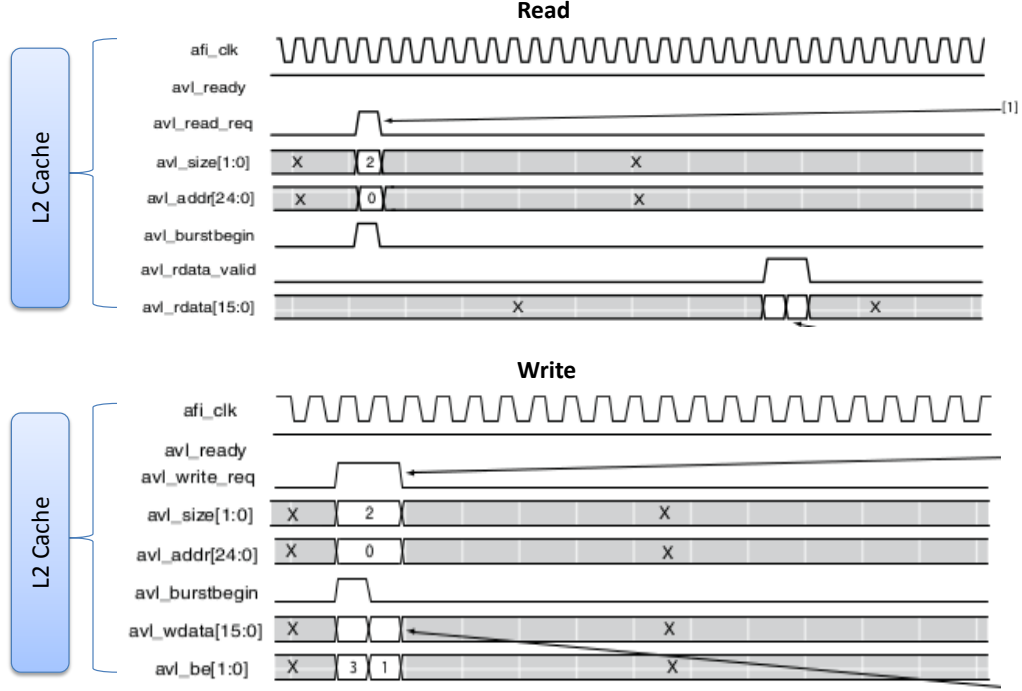


Figure 11: DDR cache side signals for read/write operation [2]

a thorough test to make sure the DDR2 runs for the parameters set on the board. We first tested the controller in RTL using a DDR2 memory model and then on the FPGA board. Once DDR2 was tested then we started integrating it with 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 where as the DDR2 just returns the data along with valid bits on a first come first serve basis as can be seen in the DDR2 wave diagrams in Figure 11. This wrapper is mainly a FIFO based structure which stores the request along with the id of incoming request coming from the cache and sends the data returned from 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 which was designed and tested on the FPGA board. We saw in previous sections in this chapter as to 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. This although might sound simple to start with but was a fairly time consuming task as it was through this process that many hidden bugs were discovered.

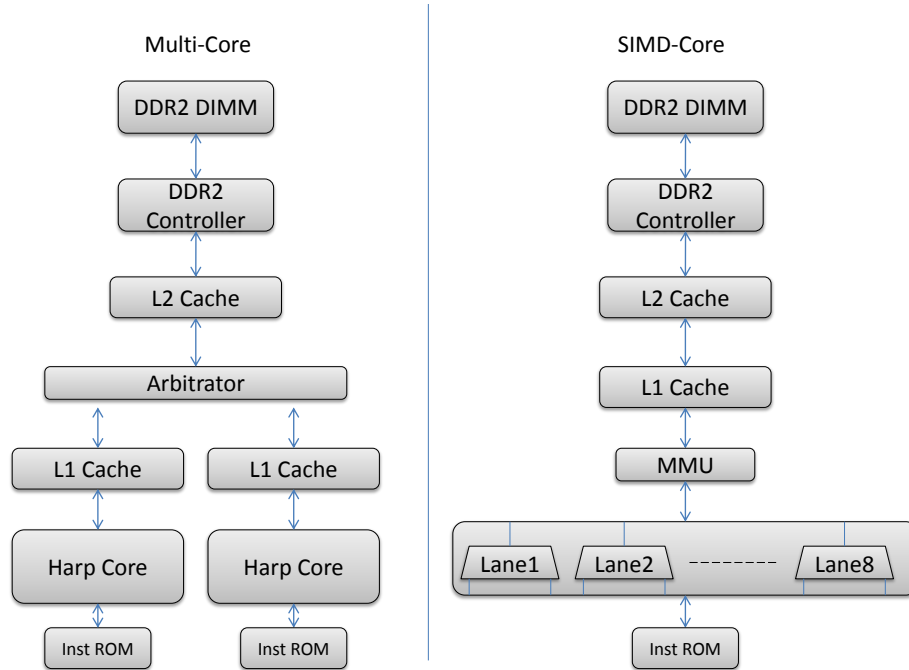


Figure 12: Possible System Designs

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 verilog testbench, our tests were to a good extent thorough and helped us discover bugs in the very blocks which we had tested earlier individually.

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 big component by a dummy model to isolate issues. Next, we synthesize and do gate level simulation, this is hard to debug as signal names are changed or synthesized away and it takes a long time to reach end of simulation. The best way to debug issues at this stage was to fix the warnings which were thrown by Altera's Quartus tool and this for us fixed many bugs in the design. Although many issues including the errors due to DDR2 timing issues were hard to fix and took a lot of time. After this stage, we test the system on the board by downloading the FPGA programming binary on the board. If it does not run even now then the reasons which I came across were mainly due to issues in the reset logic or the clock or the DDR timings 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, in case 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 it 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 multiple 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 lot of un-coalesced, coalesced and broadcast memory requests to make sure if

the complex memory management unit was working properly or not. Once the single core (one lane or SIMD) was tested on the board, then it was easy to create multiple core versions by instantiating more instances of the core with each core running its own separate application. Figure 12 shows one such system, as we can see 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 multi-core 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 we tried to run different combination of applications on each of the core accessing different data in the main memory.

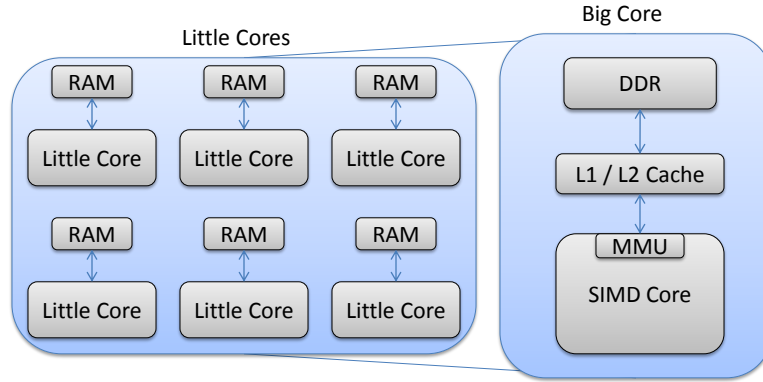


Figure 13: Heterogeneous System with 1 Big and 6 Little Cores

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 heterogeneous 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 very less FPGA resource. The results from the FPGA experiments for the systems above are discussed in the next section along with details of our heterogeneous 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, 7-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 benchmarks 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. Number of cycles can be easily measured on the simulator but to measure the actualy 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 7-segment display on the board.

4.2 Results

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

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

Table 2: Core Configuration.

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 its performance for simple micro-benchmarks.

Single 1-Lane Harp Core:

First we ran the benchmarks on a simple 1-lane Harp Core. For the benchmarks ran, 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 1-Lane Harp Core

Next we instantiate two of the 1-lane Harp cores above to form a multi-core system. Table 5 shows the performance and Table 6 shows the logic utilization of this system.

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 3: Single Core 1-Lane Performance

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 4: Logic Utilization Of A Single 1-Lane Core

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 5: Dual Core 1-Lane Performance

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)

Table 6: Logic Utilization Of A Dual 1-Lane Core

We can clearly see in Table 5 the benefits of using multiple cores on performance. Also we can see how by using a simple design of the core and having 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 8 cores on the FPGA device. Since there is no sharing and the applications are not memory intensive the performance scales linearly with 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 8-Lane Harp Core

We also design and run an 8 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 intensive 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 of this reported value as the same instruction is executed across all the eight 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.

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

Table 7: 8-Lane SIMD Core Performance

Using SIMD does give us performance benefits as can be seen in Table 7. We also dont see a big difference in the performance of coalesced and uncoalesced array sum applications due to the fact that we have a non-blocking cache and a load store queue. For the above tests a maximum of 4 requests are being served simultaneously but can

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)

Table 8: Logic Utilization Of A Single 8-Lane SIMD Core

be easily increased by changing the parameter for MSHR entries and the FIFO size for cache-DDR interface.

The aim of all the above experiments was not only to show obvious benefits due to SIMD or multiple cores. We can always choose to build a system and select an application to show really good performance benefit. But rather the aim of this chapter was to establish credibility and show the flexibility 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 examples of a 1 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.

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 9: Heterogeneous Harp System Performance

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)

Table 10: Logic Utilization Of big.LITTLE Harp Systems

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

A lot of literature can be found which demonstrate a framework to translate C/C++ code or higher abstractions to HDLs like Verilog/VHDL for hardware design. But not much work exists which demonstrate using the same source code for software simulation as well as hardware prototyping for architectural exploration purposes, some summary of prior work can be found here [7]. [9] proposes a new programming language inspired from C/C++. [15] focuses on creating RTL for floating point algorithms written in C. [16] does compile time optimization of the application to find regions of code which can be accelerated on FPGAs and it then generates VHDL for only those regions. [12] proposes something very much similar to SystemC with not much flexibility to integrate it with regular C++ or verilog models. [5] talks about how to generate variable pipelined functional units from high level abstractions of HDL. [8] discusses hardware/software codesign and simulation infrastructure for embedded systems. A good design of an example system using these translation tools was shown in [4] and this work is very much similar to our work. But again the research does not focus on architecture flexibility and exploration, instead in looks into designing a complex system in the least time with their own 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. They were either too application focussed or involved

a big learning curve for the programmer, which has prevented broad acceptance of these HSL (High Level Synthesis) frameworks as they missed out on showcasing its impact on future systems. [7] gives a good overview of current HLS frameworks but again is bit more inclined towards the hardware synthesis part. In this work we took a look at the broader picture. We thought given we have something which translates C/C++ code to HDL and also be able to use it for our software simulation what kind of possible architectures can be easily explored. The work mentioned in this thesis was focussed on the FPGA flow side of things but the overall aim of the whole project is still focussed towards future possible architecture exploration.

5.2 Future Work

Since this work is part of a multi-year and 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, branch divergence using mask registers will be one important feature which 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 were all in Harp assembly but to allow us to run of the shelf CUDA or OpenCL applications a software translator tool is being worked upon which 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 (Hyper Memory Cube) [10]. Building these systems would require only isolated changes to the existing design. For example to use a 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 prototyping flow with software simulation infrastructure will allow us to explore future architectures at different levels of granularity. 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 few different systems which 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, CEFPP’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., COOPERBALLS, 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.