Hardware Implementation of TRaX Architecture

Thesis Project Proposal

Tim George

I. Project Summery

The hardware ray tracing group at the University of Utah has designed an architecture for rendering graphics using ray tracing called Threaded Ray eXecution (TRaX). This architecture contains processor cores and shared functional units that are used to process a scene. Additionally, each core uses an L1 cache and the cores share an L2 cache. My thesis project is to create a hardware implementation of the TRaX architecture on a virtex2Pro field-programmable gate array (FPGA).

Currently, the TRaX architecture is being tested using a software simulation of the architecture. This simulation has shown that the architecture is able to render different scenes using ray tracing. Additionally, simulation of the ray tracing architecture provides the ability to relatively quickly test how changes in the architecture will affect the performance of rendering a scene. However, simulation only provides part of the testing needed to show that the architecture will work as intended. For this reason, the TRaX group needs someone to create a hardware implementation of the architecture.

The main goal of the project is to create a working implementation of the TRaX architecture on hardware. The project will start with learning about how the architecture works so that I can accurately translate it to hardware. Next, each piece of the architecture will be separated out into blocks of hardware that can be independently coded in verilog. These verilog representations will be individually tested to verify that each does what it is intended to do. Each block of hardware will then be combined to create a larger piece of the architecture. This process will be continued until there is a working hardware implementation of the TRaX architecture that can accurately display a scene using the ray tracing method.

Figure 1 shows how the TRaX architecture is organized. The TRaX architecture is made

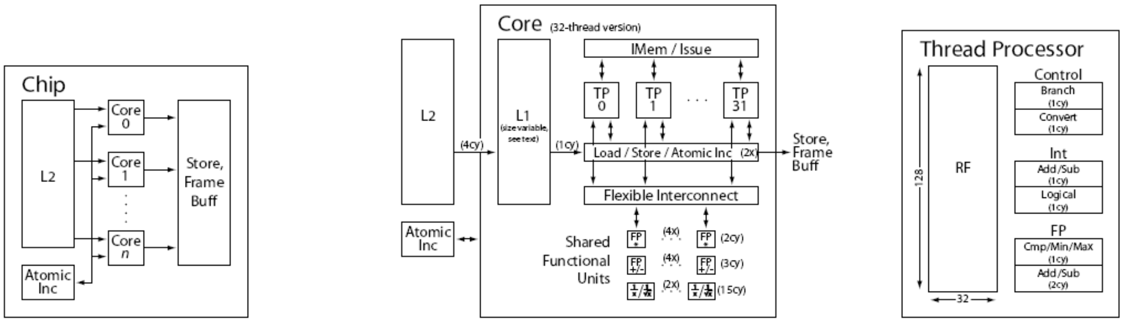

Figure 1: Block diagram of the TRaX architecture [1]

up of one or more processing cores [1]. These cores contain an L1 cache, one or more thread processing units, and shared functional units. The L1 cache stores previously accessed memory for faster access later. Each processing unit reads and processes an instruction. There are four types of shared functional units, floating point add/subtract, floating point multiply, floating point inverse, and floating point inverse square root. These functional units are shared between the cores. The overall chip is made up of one or more of these cores along with an L2 cache and a frame buffer for storing results. Each of these blocks of hardware will need to be implemented in verilog for the project.

This thesis project is taking the TRaX architecture and creating a hardware version of it. I will learn about the architecture so I understand how each piece works. Next I will take what I learned and use it to create independent blocks of hardware. I will combine the blocks of hardware to create a complete chip. Then I will synthesize the chip onto hardware so it can show the architecture in a working prototype.

II. Tasks

a. Cache/Memory:

Initially, a direct mapped cache will be written in verilog and tested to make sure it works. The cache is tested using a simulation and also synthesized to the board. Next, the DDR memory controller for a Virtex 2 Pro will be researched. This research includes looking up different available memory controllers and determining which controller is best for the project. There are two main ways this controller could be created. The first is to create a controller that directly allows the DDR memory to interface with the cache. Another method is to have the intellectual property generator in the Xilinx embedded design kit generate a memory controller. The second method requires the use of a mircoblaze processor which will take up additional space on the board. Then the memory controller will be synthesized onto the board for testing. After a working memory controller has been created, the cache can be added so the two can be

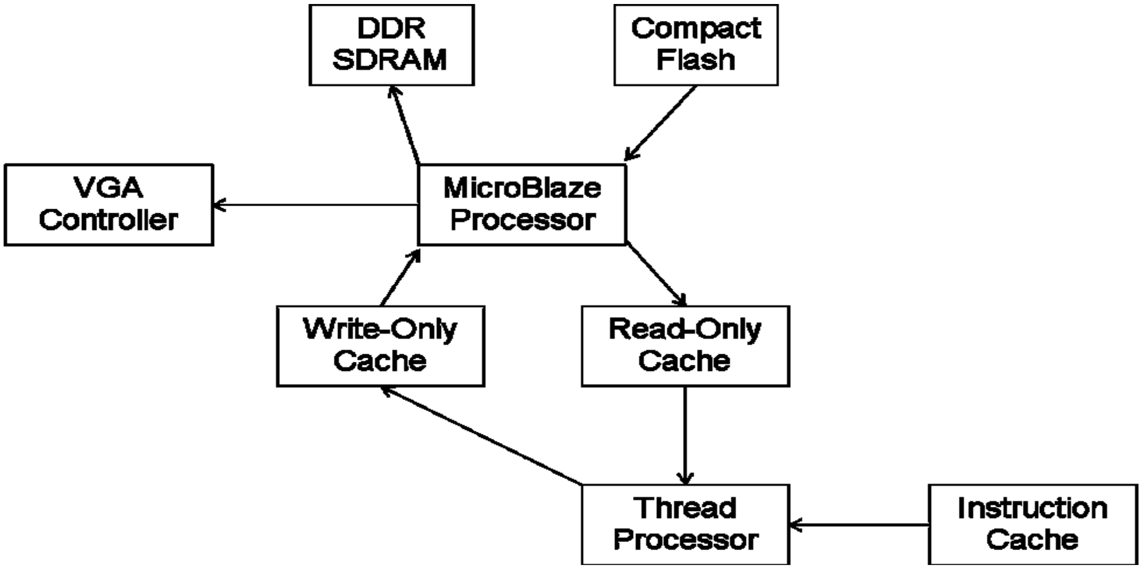

Figure 2: Memory block diagram and data flow.

tested together. This piece can then be set aside until the rest of the hardware has been created. In addition to the DDR memory, there is a compact flash slot on the board. The compact flash slot can be used to store the information about the scene that is processed by the ray tracer. This is better than using just the DDR for the scene because there is no way to preload the DDR with data. In order to use the compact flash a controller will need to be written to provide access to the card. The microblaze processor used for the DDR SDRAM also has a built in compact flash controller so it can also be used for the compact flash. Because the read speed of the card is much slower than the processor will need information, another cache will be needed to provide quicker access to recent information. Data about the scene will be loaded onto the card prior to its use.

Another aspect of memory that needs to be created is the instruction memory. Initially, this may consist of having simple block RAM set aside for the memory. There is a risk that there is too much ray tracing code; this risk could mean there may need to be interaction with the DDR memory to allow the instruction cache to update during runtime.

Figure 2 shows how the memory will be structured and the flow of data through the processor. The DDR SDRAM will be used to store the frame buffer for the ray traced scene. The DDR SDRAM is interfaced with the thread processor through a write-only cache. This cache is write-only because each of the rays being calculated are independent the results of previous rays do not need to be accessed. Also, because the cache is write-only, it only takes 1 cycle for the processor to store data. Data for the scene are stored into the compact and does not need to be changed so it is accessed through a read-only cache. Lastly, the read-only instruction cache provides the thread processor with the program instructions.

b. Shared Functional Units:

These functional units are shared by processing threads for functions that require too much logic for each thread to have its own. There are four different types of shared functional units, an inverse floating point square root, a floating point inverse, a floating point multiply, and an inverse multiply. Each of these functional units needs to be written in verilog. Additionally, each unit can be created and tested separately due to their modular nature.

c. Thread Processor:

The thread processor consists of a register file along with the logic for executing instructions. Besides instructions that use the functional units there are multiple instructions internal to the thread processor. These instructions include, branch, convert, integer add/subtract, logical, floating point compare, floating point min/max, and floating point add/subtract. The

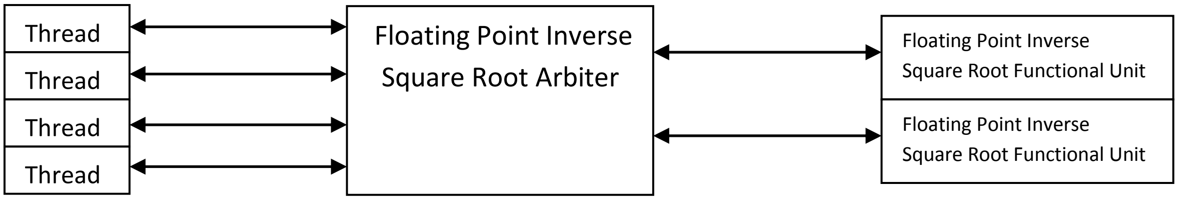
Thread
Thread
Thread
Thread
Floating Point Inverse Square Root Arbiter
Floating Point Inverse Square Root Functional Unit
Floating Point Inverse Square Root Functional Unit

Figure 3: Interfaces between threads and functional units.

logic for these instructions will need to be written in verilog. Additionally, the overall design of how the hardware will be written will need to be created.

d. Combine Components:

Once the above components have been created, they will need to be combined to create a working ray tracer. Initially, one of each of the pieces can be used to get the project to work, but the overall design of the TRaX architecture would require multiple thread processors interfacing with multiple functional units. The interaction between these pieces would need arbiters to allow the pieces to interact efficient and effective. Finally, multiple multi-threaded cores can be placed in tandem to create even better parallelism.

e. Multi-Threaded:

The TRaX architecture works well because of its multi-threaded nature and because ray tracing the primary rays is highly parallel and independent. This is why there will need to be multiple threads created within each core. These multiple threads will share the caches and the functional units. Because the threads will be sharing the same bus lines between the caches and the functional units, there will need to be arbiters added to direct the flow of data between the many points. Figure 3 shows how the threads will interact with the arbiters and the functional units.

In addition to arbiters for the caches, throughput can be increased even further by adding banks to the caches. For example, if a cache with one bank will only allow a single read per cycle, a cache with two banks can allow two reads simultaneously during a single cycle. An issue with banking the caches is that each bank can only access information stored within it. This issue means that in order to achieve the best throughput the data needs to be organized so each thread needs data out of a different bank.

f. Other Tasks:

When the TRaX processor is working, it would be nice to have a way to show the picture it processed. This would require that a VG A controller be created and able to display the contents of a frame buffer to a monitor. The frame buffer would have 9-bits for each pixel because that is what development board supports. Because initially the hardware will be processing a static image, the VGA controller would be disabled until the processing of the scene is complete. This will keep the VGA controller from blocking access to the memory and allow the data from the thread processors to be stored in the frame buffer. Once the scene is

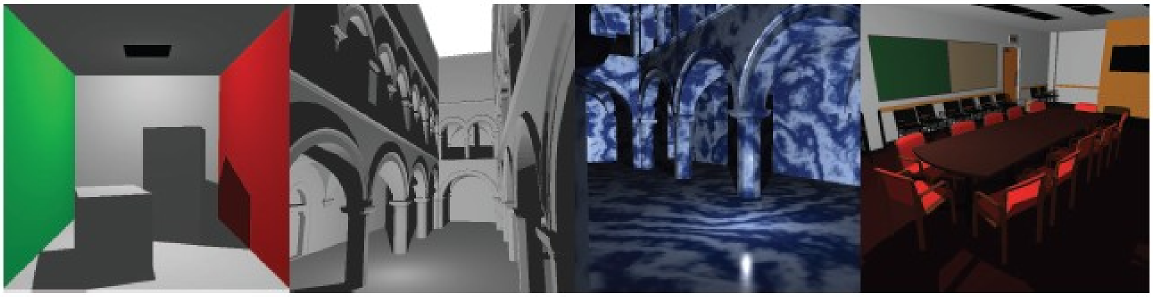

Figure 4: Test renderings created by the TRaX simulator [1]. These are also examples of what the hardware will display once it is completed.

completely processed, the thread processors can go idle.

III. Completed Work

a. Memory Structure

The first work that was done on the project was to figure out how the memory structure will be built. First the DDR SDRAM controller needed to be created. There were a couple different prebuilt DDR SDRAM controllers for the Virtex II board on opencores.org. I was unable to get any of the controllers to synthesize onto the board.

The other option for a DDR SDRAM controller is to create a microblaze processor and have the rest of the hardware interface the memory through the microblaze processor. The Xilinx Embedded Design Kit is used to create a microblaze processor for the FPGA. Using the EDK I was able to create a microblaze processor that has a built in DDR SDRAM controller. This will allow the hardware to have access to the DDR SDRAM.

Another piece of the memory structure is the compact flash memory. Research on the compact flash for the board looked like it would be easier to use the microblaze processor to interface with it. Using the EDK a compact flash controller was added to the microblaze processor. This will allow the hardware to interface with the microblaze processor to have access to the compact flash memory.

b. FPGA Usage Estimates

The next thing was done was each of the modules for the hardware were synthesized so I could figure out if the design would fit within the size of the board. The microblaze processor was created that had both access to the DDR SDRAM and compact flash. Each of the shared functional units was also created. They were created using the Xilinx IP core generator. The integer multiply, floating point multiply, and floating point square root all have can be generated using the core generator. Floating point inverse was estimated by creating a floating point division unit. The floating point inverse may be able to be smaller if an inverse unit is used instead of a division unit. Floating point inverse square root was estimated by using a floating point square root and a floating point division unit.

Results for the estimates can be found in table 1. This table shows that the amount of logic that will be used to create a single threaded TRAX processor. It also shows that the single threaded design will fit on the Virtex II FPGA. Additionally, there is enough room left over so that a multiple threaded, multiple core design can fit.

In addition to the logic estimates I estimated the amount of block ram that would be used to make sure there would be enough memory for the design. The results for the estimates can be found in table 2. This table shows that there is enough room for the memory that is required for the design.

IV. Remaining Work/Testing

a. Memory

Now that the microblaze processor has to be created, an interface needs to be designed and created so the rest of the hardware can access the memory. In addition to the interface, a L1 cache needs to be designed and created to store reads from the compact flash. Also a write buffer needs to be designed and created for writes to the DDR SDRAM.

The memory will be tested by putting data on the compact flash and having the microblaze processor read it. Using the EDK to debug the processor the data can be looked at to make sure it being read correctly. To test the DDR SDRAM, data will be written to the memory and then read back and check to see if it is the same. Cache will be tested using the simulator in Xilinx ISE. A simulation of reading information from memory and checking to see if the cache provides the correct result will test the cache. The final testing will be to test the all of the pieces of the memory together. Information will be read from the compact flash and sent to the cache. Next the data will be written to the write buffer and finally to the DDR SDRAM. A final read from the DDR SDRAM will be compared to the initial data on the compact flash card to determine if the memory works.

b. Thread Processor

Pieces of the thread processor will each be created and tested individually. Each of the pieces will be written in Verilog using ISE. All of the pieces will be first tested using the ISE simulator and then synthesized to the board and tested on the board. One piece of the processor is the register file. It will be created using block ram and will be tested by storing data in a register and reading from the same register and then checking the data to make sure it is the same. Another piece is the ALU that will handle the instructions that are do not have to do with memory or shared functional units. A simulation will test each of the instructions with different values and check the results. Also it will be synthesized to make sure it will work on the board. The decoder is another piece of the processor. It will be tested using a simulation to make sure it gives the correct return values for each of the instructions. These pieces will be connected together and then tested to make sure they work together correctly.

c. Shared Functional Units

Each of the shared functional units will be created and tested individually. They will be simulated with different input values and the output values will be checked for accuracy. Next they will be synthesized and tested on the board.

d. VGA Controller

The VGA controller will be built on the microblaze processor. Software on the processor will read data from the DDR SDRAM and the pass it to the VGA controller. It will be tested by putting data on the compact flash that the microblaze processor will read and then write the data into the DDR SDRAM. Then the VGA controller will read in data out of the memory and display it on a monitor. If the picture in the data matches the picture on the monitor, then the VGA controller works correctly.

Table 1: FPGA Usage Estimates

Module	LUTS	Slices	Flip Flops	18X18 bit multiplier
Integer Multiply	791	414	64	1
FP Multiply	183	187	295	4
FP Inverse (Using Division)	812	776	1386	0
FP Square Root	514	452	840	0
FP Inverse SR	1326	1228	2226	0
MicroBlaze Processor with SDRAM DDR Controller and Flash Controller	3403	2311	2481	3
FPGA Total:	27392	13696	27392	136
Remaining	20363	8328	20100	128

Table 2: Memory Usage Estimates

Module	Blocks Used (18Kb)	Kilobytes
MicroBlaze	4-32	8-64
128 32-bit Registers	1	.5
Write Buffer	1	2
Data Cache	32	64
Instruction Cache	1	2
FPGA Total:	136	262
Remaining	97-69	184-138

V. Initial Schedule

Overall the goal will be to make incremental progress over the course of the next year and a couple months. The end goal is to have a working hardware prototype of the TRaX ray tracing written in verilog and running off of a FPGA. Additionally, during the course of the 09-10 school year, work will need to be done on the requirements for an undergraduate thesis. Details will be added to the schedule once I have figured them out. Table 3 shows a timeline for the project.

End of May 2009:

At this point I would like to have an initial cache and memory interface working. The initial cache would be a direct mapped cache that would interface with the onboard DDR module. Also the compact flash controller should be written. This initial memory progress should be tested and set aside for later use. At this point the overall design and implementation of the memory should be finalized.

End of June 2009:

Initial project planning documentation and presentation should be completed. Additionally, work should be started on the thread processor.

Middle of summer 2009:

The thread processor should be completed at this point. Additionally, work should be made with having the thread processor interact with the memory. The thread processor should be able to run simple code that does not involve the yet to be created functional units.

End of summer 2009:

Both the functional units and thread processor should be completed so that putting the pieces together can get started. Work should begin on having the thread processor use the functional units.

Middle of fall semester 2009:

All of the components for a single threaded implementation should be complete. A VGA controller should be completed so that the single thread can process a scene and the results can be viewed in close to real time.

End of fall semester 2009:

Work should begin on a generic bus arbiter that can be used on the many bus lines with few changes. Additionally, there should be a good idea for how the multi-threaded work will be completed.

Middle of spring semester 2009:

The multi-threaded hardware should be working so that a scene can be processed and viewed via the VGA controller. The documentation should be being brought together.

End of spring semester 2010:

Final documentation should be completed. There should be a working prototype of TRaX hardware ray tracer implemented in verilog on a Virtex 2 Pro board.

VI. Materials

Here is a list of materials that will be used for the project.

Hardware:

- Virtex II-Pro development board
- 256MB DDR SDRAM
- Monitor
- Compact Flash Card (Size to be determined)

Software:

- Xilinx Embedded Design Kit
- Xilinx ISE

	Project Research	Documentation and Presentation	Memory Structure	Thread Processor	Combine Memory and Thread Processor	Shared Functional Units	Combine Shared Fuctional Units with rest of modules	VGA Controler	Render Scene and Display Result	Multiple Threads	Multiple Cores	Final Documentation and Results
4/19/2010												🟧
4/12/2010												🟧
4/5/2010												🟧
3/29/2010												🟧
3/22/2010												🟧
3/15/2010											🟩	🟧
3/8/2010											🟩	🟧
3/1/2010											🟦	
2/22/2010											🟦	
2/15/2010											🟦	
2/8/2010											🟦	
2/1/2010										🟩	🟦	
1/25/2010										🟩	🟦	
1/18/2010										🟩	🟧	
1/11/2010										🟦		
1/4/2010										🟦		
12/28/2009										🟦		
12/21/2009										🟦		
12/14/2009										🟦		
12/7/2009										🟦		
11/30/2009										🟦		
11/23/2009										🟦		
11/16/2009										🟦		
11/9/2009										🟦		
11/2/2009										🟦		
10/26/2009									🟩	🟧		
10/19/2009								🟩	🟩			
10/12/2009								🟦				
10/5/2009								🟦				
9/28/2009							🟩	🟧				
9/21/2009							🟦					
9/14/2009							🟦					
9/7/2009						🟩	🟧					
8/31/2009						🟦						
8/24/2009						🟦						
8/17/2009						🟦						
8/10/2009						🟦						
8/3/2009					🟩	🟧						
7/27/2009					🟦							
7/20/2009					🟦							
7/13/2009				🟩	🟧							
7/6/2009				🟦								
6/29/2009				🟦								
6/22/2009				🟦								
6/15/2009				🟦								
6/8/2009				🟦								
6/1/2009			🟩	🟧								
5/25/2009			🟦									
5/18/2009			🟦									
5/11/2009		🟧	🟦									
5/4/2009		🟧	🟦									
4/27/2009		🟧	🟧									
4/20/2009	🟧	🟧										
4/13/2009	🟧											
4/6/2009	🟧											

KEY:	
Research and Documentation	🟧
Development	🟦
Testing and Checkpoint week	🟩

Table 3: Project Schedule

References

[1] J. Spjut, D. Kopta, E. Brunvand. S. Boulos, and S. Kellis. TRaX: A Multi-Threaded Architecture for Real-Time Ray Tracing. In Symposium on Application Specific Processors, 2008.