CSE467: Project Phase 3 - Building the GPU Core and Instruction Stores

Vincent Lee, Mark Wyse, Mark Oskin

Winter 2015
Design Document Due: Saturday February 7th, @ 11:59PM
Design Reviews: Monday February 9th 2:30-4:30PM
Due: Tuesday February 24th @ 11:59PM
Last Updated: February 1, 2015

1 Introduction

The objective of this project phase is to implement the GPU core which will be responsible for performing the programmable segments of the graphics pipeline. As outlined in the previous phase of the project, the GPU core will interface to the work queues sourced by the CPU and eventually generate the pixels to be displayed at the output. The base instruction set architecture is the same as the previous checkpoint, however if you modified or augmented the instruction set you will have to make adjustments accordingly. By the end of this phase of the project, you should be able to execute the assembly code for the programmable portions of the graphics pipeline on the GPU core and display the teapot on the screen. This checkpoint is probably going to be the most intensive phase of the project so we highly recommend you start early.

2 The GPU Core

2.1 Instruction Set Architecture

The base GPU instruction set architecture is reproduced below in Figure 1. If you have augmented the ISA or modified the ISA, this is less relevant and should be modified accordingly.

Again, the "n" following each instruction is reserved for which predicate register should be used for that instruction. If the "/n" is omitted, the instructino will execute unconditionally.

2.2 Register Set

The GPU has 15 general purpose 32 bit integer registers (r0-r14), one "machine status register," 3 predicate registers (p1-p3), and does not support branches. All instructions are executed conditionally based on an optionally specified predicate. Predicate 0 should be hardwired to TRUE and is the default predicate used for instructions when no other predicate is specified. Register 15 (r15) is special and comprises a machine status register. Note that if you added additional stages to the graphics pipeline, you will have to expand the queue source field to eat up some of the padding which currently isn't in use. The encoding for this register looks like:

[instruction_count_16][processor_number_8][padding_2][queue_source_2][predicates_4]

Note that the predicate register set does not necessarily have to be part of the general purpose register set and can be instantiated separately from the general purpose registers in the design.

Instruction	Register Description	RTL Specification
LDGPMEM/n	*sourcereg, targetreg	targetreg ← Mem[sourcereg]
STGPMEM/n	source1reg, *source2reg	Mem[source2reg] ← source1reg
MUL/n	src1reg, src2reg, targetreg	targetreg ← src1reg * src2reg
ADD/n	src1reg, src2reg, targetreg	targetreg ← src1reg + src2reg
SUB/n	src1reg, src2reg, targetreg	targetreg ← src1reg - src3reg
SRL/n	src1reg, #, targetreg	targetreg ← src1reg >> src2reg
SLL/n	src1reg, #, targetreg	targetreg ← src1reg << src2reg
AND/n	src1reg, src2reg, targetreg	targetreg ← src1reg & src2reg
NOT/n	src1reg, targetreg	targetreg ← ˜ src1reg
XOR/n	src1reg, src2reg, targetreg	targetreg ← src1reg ⊕ src2reg
OR/n	src1reg, src2reg, targetreg	targetreg ← src1reg | src2reg
NAND/n	src1reg, src2reg, targetreg	targetreg ← ˜(src1reg & src2reg)
LI/n	targetreg, #	targetreg ← # (signed immediate)
SETLT/n	src1reg, src2reg, predictateregN	if (src1reg < src2reg) predicateregN ← 1; else predicateregN ← 0
STOREQ/n	srcreg	queue[srcreg][255:0] ← {r7, r6, r5 ... r0} //little endian
STOREQI/n	#	queue[#][255:0] ← {r7, r6, r5, ... r0} //little endian
END/n		

Figure 1: GPU Instruction Set

2.3 Implementing Memories

Depending on how large your memory blocks need to be, or how many read or write ports are required, you will need to either build your own memories or generate them using the Quartus tool. Note that there are several different types of memories which usually have different features and timings. Under the "Basic Functions" list, go to "On Chip Memory" to find the list of different memories that the Quartus tool can generate for you. There are several options here but the memory blocks you want to generate can be built with one of the RAM or ROM blocks. These memories will use the on board hardened memory cores on the FPGA of which there are approximately several megabits. If you need more than this, you will need to use one of the memory controllers to talk to the larger memory chips on the board. You can find these controllers under the "Memory Interfaces and Controllers" list and selecting the appropriate controller you want. These controllers will require you to add top level contraints as they are talking to a module off chip; the pin constraints and pin outs can be found in the manual for the DE1 SoC.

2.4 Implementing Work Queues

To implement work queues, you will have to use the Quartus core generator to produce the appropriate FIFO IP blocks. To do this look for the FIFO block in the core generator tool. If you only need one type of FIFO, all you need to do is run the generator once and instantiate the resulting module multiple times. The width and depth of the FIFO for these work queues depends on your design requirements. For each stage of the graphics pipeline, you will need at least one work queue to feed into it (again this depends on your design).

2.5 Implementing Instruction Stores

The instruction stores contain the programs that will execute on the GPU. For each stage of the graphics pipeline, you will need to build one instruction store of appropriate size depending on the size of your code.

The memory size depth should be a power of 2 greater than your code size. Also note that each instruction store will need to be programmable so you will need to have one read and one write port. These instruction stores will multiplex to the GPU depending on the priority GPU controller. Eventually you will want to make this instruction store programmable from the ARM core; how you accomplish this is up to you.

3 Design Documents

All teams will be required to submit a design document prior to design reviews. As with all large projects, it's always good to have a battle plan before you go to war with the FPGA. For this particular project phase, we expect a detailed block diagram showing each of the components for your GPU core. There is no required structure for the design document but it must address the following:

- A high level overview of what you expect the system to achieve
- Define the modules you need, their interface signals, and the role of each of the interface signals
- A detailed set of diagrams showing how module definitions and how they are connected with respect to other modules
- A high level block diagram detailing the connections between the HPS FIFO, GPU core, instruction stores, work queues, and display infrastructure from the previous checkpoint
- A detailed block diagram showing each of the components in your GPU core. This should include any arithmetic primitives such as adders, multipliers, and shifters, memories, pipeline registers, the general purpose registers, multiplexors, control units, and module interfaces. We expect that the design of this module will follow a control and datapath design paradigm, though you are free to choose otherwise.
- An explanation of how the priority control for the work scheduler will operate and what signals from each of the modules it will require
- An explanation of your testing strategy - both unit and integration tests. A test plan for each module should be presented along with your integration testing strategy for appropriate components of the design. In particular, we require a detailed testing plan for the GPU core.
- A detailed explanation of how you expect to implement the programmable istores and program them from the ARM HPS.

Design documents will be graded on clarity, conciseness, and effort. Note we are not looking for perfect optimal solutions as there are multiple solutions for this project. We really just want you to think through the solution and put it on paper so that you have a plan when you go to build the design.

The design document must be at least 4 pages in length and be submitted to the Catalyst dropbox as a PDF document. The more detailed the design, the more productive your design review will be. It is in your best interest to write a good design document as it will allow us to provide more detailed feedback.

All design documents must be submitted as a PDF to the catalyst dropbox. If you submit a non-PDFied document, we will require you to resubmit it.

4 Design Reviews

All teams are required to schedule a design review with one of the TAs in addition to submitting a design document. During the design review, you will be asked to identify the problem components, present your solution, and justify why your solution is sufficient. You are free to reuse your design document, or bring additional presentation materials when explaining your solution. Design reviews will be 15 minutes long and be graded based on your teams ability to effectively communicate your solution.

5 Demonstration

You will be required to schedule a demonstartion with your TA to show a working solution. During this demonstration, you will need to show that your implementation works by displaying the teapot on the screen. We expect you to be able to do this by logging in to the ARM core on the FPGA and launching some piece of software to communicate the teapot data to your design over the HPS-FPGA link. If your implementation is not working, you will need to prove to us which components of the design work properly. This can be done either by showing us an alternate but convincing demonstration or by executing test harness simulations.

6 Code Submission

You are required to submit a copy of your code to the Catalyst dropbox. This submission must be a tar file labeled as *project*_1_$*name*_1_$*name*_2.*tar* where *name*_ and *name*_2 are the last names of each team member. If you are working alone, omit the second name. At the top level of this submission, you must also include a README file which contains the first and last names, student ID, and uw.edu emails of each team member. Only one member of the team is required to submit the tarball of the code.

7 Tips and Tricks

As always some tips and advice which may or may not be useful when working on this checkpoint:

1. Simulate everything before you deploy to the board. Debugging a bad design live on the FPGA is the nuclear option. Almost everything can be resolved and debugged using simulation tools. As bad as the simulation tools are, the live debugging tools are infinitely worse.
2. Perform very thorough testing of the GPU core. Getting any CPU or GPU core right is very challenging. One broken instruction renders the whole core useless so good test coverage is key that will save you painful hours or days of debugging.
3. Use simple test cases to get you off the ground and hit corner cases. When simulating a large design, it is much easier to verify things if you use simple targeted test cases as opposed to the more complicated ones. Think of the more complicated test cases as a sort of final integration test.
4. Use Verilog macros when defining constants like the instruction encodings. This will make your verilog code easier to read and debug instead of staring aimlessly at a bunch of 1s and 0s.
5. Hardware compilations cycles are costly. The designs will take a while to compile as they get bigger so you really only want to fire off the minimum number of compilations necessary.
6. Testing, testing, testing... Hardware verification can be either a tedious task at best, or scar you for life... In most cases, more testing inevitably leads to fewer frustrating hours in the lab...