# CSE467 Project Phase 1: Introducing the GPU Instruction Set

Vincent Lee, Mark Wyse, Mark Oskin

Winter 2015
Due Sunday, January 25th @ 11:59PM
Demos Due Monday, January 26th @ 4:00PM

## 1 Change Log

Due date pushed back again to Sunday night due to the project infrastructure being on fire.
RTL descriptions for each GPU instruction were added to Figure 2.
Checkpoint deadline has been extended to January 22nd and demonstration due date moved to Monday @ 4:00PM.

## 2 Introduction

In this phase of the project, we begin to more closely examine which components of the graphics pipeline will live in the programmable graphics processing unit software and what functionality will live in hardened RTL cores. As indicated in the previous phase of the project, our graphics pipeline has a decent number of hand tunable parameters which the programmer may want to alter between executions. If the entire pipeline was a hardened processor, we would have to recompile the design every time we wanted to change one of these parameters which is a terrible solution at best. The solution is to make these parameters programmable by moving functionality into more general purpose programmable GPU cores. The objective is to convert these software programmable components which you built in the previous phase of the project, and implement the equivalent program for our custom GPU instruction set.

**Starting from this checkpoint onwards, you are allowed to work in pairs.** Tag teams will not be made permanent until the next project phase after which unless a catastrophe occurs you will not be allowed to change partners. You are not allowed to form teams larger than two people. Please post/check on the class forum if you are looking for a partner.

## 3 GPUs Late 1990s - Current

GPUs changed rapidly in the late 1990s. The first thing that happened is the transformation and lighting steps were replaced by programmable processors. The first card that did this was from 3Dfx (long since defunct). These cards had a handful of VLIW processors on them for this step. The rest of the rendering pipeline remained in fixed function logic. The next step in the evolution of GPUs was the migrate from fixed-point arithmetic to floating point and eventually to IEEE compatible floating point. After this the rest of the pipeline (except the z-buffer updates) was moved to programmable logic. Here a variety of architectures were and are used. Some designs, such as those from Intel use a straightforward multicore with 80 CPUs. Those from NVidia and AMD use more specialized processors that employ a SIMD-like execution approach with Warps. It is this style of GPU that we will be building.

```
if size(z-buffer) >= maxsize(z-buffer) - N then
  idle()
end if
if size(z-buffer) < 1/2 maxsize(z-buffer) and not_empty(queue3) then
  execute_from(queue3)
else

  if size(queue3) < 1/2 maxsize(queue3) and not_empty(queue2) then
    execute_from(queue2)
  else

    if  size(queue2) < 1/2 maxsize(queue2) and not_empty(queue1) then
      execute_from(queue1)
    else

      if  size(queue1) < 1/2 maxsize(queue0) and not_empty(queue0) then
        execute_from(queue0)
      else
        idle()
      end if
    end if
  end if
end if
```

Figure 1: Priority scheduling pseudocode

# 4 The 467 Programmable GPU

The GPU you will build for this class is really a flexible parallel execution engine. It happens to be specialized for graphics work but it could really do anything. Logically the architecture has 10 dedicated memories, 4 work queues, 1 z-buffer work queue, 4 instruction memories, and 1 general purpose memory.

However, you will find it is more efficient and perfectly reasonable to implement this as 4 memories (work-queue, z-buffer queue, code, and general purpose memory). The goal of the GPU is to keep the z-buffer queue full. So to do this, the GPU selects new work to do with a priority scheduler which is pretty simple to describe in Figure 1.

The CPU interface dumps work into queue0. A chunk of work can create new work and place it on queue0 through queue3. The difference between the work queues is the priority at which they execute and the code that is executing for the queue's work items. Each work queue has a corresponding program that executes on its items. However, each of these programs are executed on the same programmable processor using the same instruction set. You have complete freedom to design this instruction set as you see fit, but we strongly, and we mean strongly encourage you to start from the following instruction set shown in Figure 2. Note that the instruction set listed in Figure 2 is a base ISA and that our simulator supports several other instructions which you can find by looking at the 467gpu.c and 467gpu.h files.

## 4.1 GPU ISA

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 is 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. The encoding for this register looks like:

| 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 2: GPU Instruction Set

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

As mentioned before, the GPU uses work queues to hold data. These work queues are 256 bits wide (note: this is very wide and may require that you use multiple cycles to read/write them!). When an item is removed from a work queue and scheduled for execution the GPU is re-initialized in the following way: registers 0-7 are initialized with the item from the work queue, all other registers are initialized to 0, and all predicates are set to TRUE.

The GPU Instruction set is shown in Figure 2. You will be writing code in this assembly language. If at any time you are confused as to the RTL encoding, please take a look at the 467cpu.c file which contains the source code for the model of the GPU ISA.

There are no branches in this ISA, which drastically simplifies the design. Without branches we do not need to worry about WARP divergence. The ISA does have general purpose load and store instructions and queue writing instructions. These instructions will create structural hazards in your design and necessitate that multiple cycles be required to implement them.

Since there are no branches, you are probably wondering how do you loop? The answer is an interesting hack: you do one (or more) iterations of the loop and then create a new work item of the remaining work and place it back on the queues for further execution. When writing looping code, you should start out by doing just one iteration and placing the rest back on the work queue. Once you are confident your assembly works you can try "unrolling" it by executing multiple iterations at a time and using predicated execution. Unrolling code is great, except the work queues have a bounded size and it is critical they never fill up entirely. This is checked for in the first line of the priority code in Figure 1. You get to choose the size of work queues (we suggest 128 entries or more), so think carefully about queue sizing and unrolling. You may not be ale to unroll loops as much as you would like to!

After writing the assembly code, it must be transformed into the actual bits the processor can understand. To simplify this task we have written a small assembler for you to use. The assembler is designed to be easy for you to hack and add new instructions, change the instruction encoding, etc. The provided assembler supports the ISA in Figure 2.

The GPU instruction encoding that we provide to you can be found in the 467gpu.h file. The current

encoding format is shown in Figure 3.

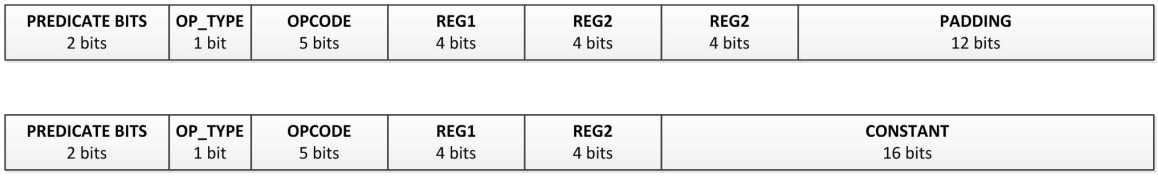


Figure 3: Encoding Format for the CSE467 GPU

Note that there are two encoding styles: one for three register encodings, and one for two register encodings and a constant. The GPU you build will only understand the binary encoding so eventually you will need to generate the appropriate binary, ASCII binary, or ASCII hexadecimal files.

## 4.2 Example: Transformation

This is likely an unfamiliar execution model to most of you, so consider the following example. The first step in the GPU pipeline is to multiply all of the points of a triangle by the transformation matrix. The CPU sends the triangle to the GPU by encoding each point in z / alpha space (x,y,z,a) where each entry of the four-tuple is a 16-bit fixed point value. Each point requires a total of 64-bits. Therefore, each triangle (3 points) requires 192-bits. The remaining 64-bits of the work item can be used to optionally specify texture or lighting information. Feel free to add textures and lighting after you have the basic GPU architecture working.

```
# Note this code is not the most efficient. It s not super important to be
# highly optimized here. It s probably a lot more important to be correct!
XOR r8,r8,r8 # zero out r8
OR r0, r0, r8 # copy r0 into r8
ANDI r8, 0xffff, r8 # mask off the upper 16 bits
XOR r9, r9, r9 # zero out r9
OR r0, r0, r9 # copy r0 into r9
SRLI r9, 16, r9 # shift down r9 so the upper 16 bits are now the lower 16
XOR r10,r10,r10 # zero out r10
OR r1, r1, r10 # copy r1 into r10
AND r10, 0xffff, r10 # mask off the upper 16 bits
XOR r11, r11, r11 # zero out r11
OR r1, r1, r11 # copy r1 into r11
SRLI r11, 16, r11 # shift down r11 so the upper 16 bits are now the lower 16
# At this point r8, r9, r10 and r11 each contain 16-bit values for (x,y,z,a)
# These values are multiplied by the transofrmaiton matrix
# Once the multiply is done, re-pack all the values into the registers r0-r7 and
# send the work to the next stage
LI r8, 1
STOREQ r8
END
```

Figure 4: Triangle transformation assembly code example

These CPP macros can be your poor-mans subroutines. So there are no branches, how do you loop? The answer is an interesting hack: you do one (or more) iterations of the loop and then create a new work item and place it back on the queues for further execution. You should start out by doing just one iteration and

```
#define UNPACK(src, t1, t2) \
xor t1, t1, t1 ; \
or src, src, t1 ; \
andi t1, 0xffff, t1 ; \
xor t2, t2, t2 ; \
or src, src, t2 ; \
srli t2, 16, t2
```

Figure 5: UNPACK macro

```
UNPACK(r0, r8, r9)
UNPACK(r1, r10, r11)
```

Figure 6: Transformation with macros

then placing it back on the work queue. Once you are confident your assembly works you can try unrolling it by doing multiple iterations (note predicated execution!). Other notes: work queues have a bounded size. This size should be fairly large (choose 128 entries or more for example). But is critical that queues never entirely fill up. This shouldnt happen thats what the first line in the priority code ensures. But it also means you cant unroll the loops as fully as you might like. Think about it some to see how much you can unroll!

# 5 Using the GPU Simulation Infrastructure

In order to execute the graphics processing functionality that you implemented in the previous checkpoint, we will have to put the equivalent assembly programs into the GPU instruction memories for each stage of the graphics pipeline. For our programmable GPU, we will execute the transformation, lighting, projection, and rasterization phases in the GPU (hence the four work queues). The z-buffer and framebuffer will remain as hardened logic cores which you will implement in the next phase.

We have provided an example implementation of the transformation phase in transform.s. This file contains an implementation of the first stage of the graphics pipeline which you will need to adapt to plug in to your simulation. You are responsible for writing the remaining instruction streams for the lighting, projection, and rasterization stages. We encourage you to take a look at it to get a handle on what the assembly file may look like and to understand the basic work flow when using the infrastructure.

We also provide a debugger, assembler, and disassembler for the GPU. You can obtain these files by downloading the cse467.zip file from the course website. The archive contains C source files and a Makefile; as usual, to build the binaries which include the simulator (among other things) simply execute make inside the top level. There will be three executables that compile: a debugger (467db), an assembler (467as), and a disassembler (467disasm).

## 5.1 Using the Assembler

The assembler (467as) is used to compile an assembly file using the instruction set we provide to the binary which will be executed on your GPU. A sample assembly file transform.s is already completed for you. In order to compile the assembly file, we must first expand macros using the *cpp* tool then pipe the resulting file into the assembler. For the transform.s file you should be able to run the following to see a hexadecimal dump of the resulting binary in ASCII form.

```
cpp < transform.s | ./467as -d debug.sym
```

In this example the -d flag tells the assembler to write the debug symbols into a file called debug.sym. Note that this hex dump is not a binary image and cannot be fed into the assembler. There are other dump formats that are supported by the assembler. A more complete explanation of usages are outline in Figure 7.

| Option | Flag | Description |
|---|---|---|
| Input File | -i [file] | Specifies the input file to the assembler. The file specified should be the result of the macro expansion from cpp. |
| Output File | -o [file] | Specifies the destination file for the result of the assembler. If no output file is specified, the result from the assembler is dumped to standard out. |
| Debug Symbols | -d [file] | Specifies the destination for the debug symbols generated during assembly. These will be useful when running the debugger. |
| ASCII Hex Dump | -a | Directs the assembler to output an ASCII readable hexadecimal representation of the resulting binary encodings of the assembly file. |
| ASCII Binary Dump | -b | Directs the assembler to output an ASCII readable binary representation of the resulting binary encodings of the assembly file. |
| Binary | -m | Directs the assembler to output the raw binary encodings of the assembly file. This file is the binary image required by the debugger. |
| Help | -h | Displays some usage information |

Figure 7: Usage Summary for Assembler

For each of your assembly files you will eventually want to produce a binary file from the assembler to run in the debugger. In later checkpoints, it will be useful to have the ASCII readable dumps for later Verilog RTL simulations.

## 5.2 Using the Debugger

The debugger (467db) is used to run and simulate binaries generated by the assembler. It is intended to work for the base instruction set architecture and will need some modifications if you wish to modify the ISA. To use the debugger, you will need the binary file and the debug symbols you generated from the assembler.

A complete explanation of the command line usage can be found in Figure 8. The debugger is a stripped down version of gdb and supports similar commands; you can access the usage message by entering "help" or "h" into the debugger. By default the debugger is set to run interactively however if you wish to provide it a batch file and run non-interactively you can use the -i flag to specify an input batch file. Usage messages for each command for the debugger are returned if the command is not executed properly.

## 5.3 Modifying the Instruction Set Architecture

As mentioned before, you are not limited to the base instruction set architecture we provide for the GPU. If you would like to augment or modify the instruction set architecture you are free to do so. To get our GPU simulator to support these new instructions, you will have to modify the 467gpu.c and 467gpu.h files. These files contain the constants and data structures used to define the GPU ISA. Depending on the type of instructions you wish to add, you may or may not need to modify the 467db.c and 467db.h files which contain the source code for the debugger.

| Option | Flag | Description |
|---|---|---|
| Binary Image | -b [file] | Specifies the raw binary file generated by the assembler to be debugged. This should be the result of the -m option from the assembler. |
| Debug Symbols | -d [file] | Specifies the debug symbol file that should be used. This should be the same as the file specified by the -d option in the assembler. |
| Source File | -s [file] | Ignore this option. |
| Binary Memory Image File | -m [file] | Specifies an ASCII file of the binary memory image that should be preloaded prior to running the target binary for debugging purposes. |
| ASCII Memory Image File | -a [file] | Specifies an ASCII memory image file that should be preloaded prior to running the target binary for debugging purposes. |
| Input Command File | -i [file] | Specifies a batch file of commands that should be executed by the debugger. In other words, the debugger commands specified by the file will be executed automatically for you before displaying an interactive debug terminal. The debugger will automatically terminate after executing all of the commands in the command batch file. |
| Continue Debugger After Input Command File | -c | Prevents the debugger from automatically exiting the after reading an input command batch file. After executing the batch file commands, the debugger will switch to interactive mode and display a debugger prompt. |
| Output File | -o [file] | Specifies the output file to write the results of the debugger. |
| Help | -h | Display the help message |

Figure 8: Usage Summary for Debugger

# 6 Recommended Steps

## 6.1 Converting to Assembly Instructions

To use our simulator infrastructure, we are first going to have to convert the graphics pipeline algorithm from whatever language you used in the first checkpoint into the GPU instruction set. Recall we need to generate 4 GPU programs for the transformation, lighting calculation, screen projection, and rasterization. Note that the transformation program is already mostly done for you. Your job will be to build the remaining 3 GPU programs for the lighting calculation, screen projection, and rasterization. These three programs should each stand on their own as they will be stored in three separate instruction buffers on the GPU. Recall communication will eventually have to work but we recommend you first write the core functionality and resolve where you need loops before writing code that implements the communication. Note that you will also have to decide how you are going to implement fixed point arithmetic and resolve division operations to conform to the GPU ISA.

## 6.2 Interfacing to Work Queues

To communicate between pipeline stages, the work queues are to be allocated in memory as software FIFOs. To do this, we simply need to allocate some data structures such as a head and tail pointer, and a data buffer. It is up to you to decide how you want to implement your software FIFOs and which addresses they live in. In order to simulate FIFO state between graphics pipeline stages, you will need to pipe the memory image between stages. A memory dump can be generated using the dump command, and specifying the address and length of the dump after the target address in the debugger. A memory image can be loaded into the debugger by using the -m or -a options.

### 6.3 Verifying the Result

We recommend that you script calls to the debugger using the -i option to automate the simulation. You will have to inspect the contents of the memory to confirm the resulting output. We recommend you build a simple script at the end of your simulation pipeline to take care of the z-buffering, framebuffer, and PGM image writing. Again, you should be able to see the teapot image in the PGM file at the end of the day.

## 7 Tips and Tricks

As always, a set of landmines to watch out for and potentially helpful tips when implementing your assembly instructions:

- Avoid getting bogged down in tedious manual labor. We recommend that you automate as much of the debugging process as possible. Basically this means scripting as much of the debug process as possible with shell scripts or Makefiles.
- Unit test each stage of the graphics pipeline in isolation before attempting to put anything together.
- Use #define macros to build poor-man's abstraction. It's always better than no abstraction.
- Implementing the loopings for the rasterization phase can be tricky. There are one or more loops depending on your implementation which may require some amount of state to pass between iterations. Think about the job encoding for jobs that feed into the stage to account for jobs that may queued as a resulting of loops.

## 8 Specifications and Demonstration

Your assignment for this phase of the project is to adapt the model you wrote in the previous phase to the GPU instruction set architecture we present in this document. Yes, that means you need to write the assembly code to implement each of the programmable portions of the GPU and simulate the end to end graphics pipeline. The implementation should plug in to the simulator we provide for you so that you know that the assembly and binaries will work when we execute them on the GPU. You do not need to implement the z-buffer, framebuffer, and PGM file writing logic using GPU instructions; you may use external scripts or programs to perform these steps.

You will be required to provide a live demonstration for your working simulation. All this consists of is showing us the resulting PGM file with different lighting vectors, and transformation matrices, and a quick comparison of the results from your GPU simulation and the software model from the previous phase of the project. In the event that your demonstration does not work correctly, we require you to be able to prove to us which components of the system design work and explain to us why the rest is broken. In addition to the demonstration, you are required to submit your source files in a tarball to the course dropbox. At the top level of your submission folder, you must include a README file which contains the first and last name, SID, and email of each team member. The README should also contains instructions on how to build, compile, and run your code.