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