LSU EE 7700-1 **Homework 1** Solution **Due: 3 March 2010**

To complete this assignment follow the setup instructions from the course Web page. The setup instructions bring you to the point where you can compile the cpu-only examples but for this assignment that won't be necessary. (The setup will facilitate the use of Emacs to view the source and assembler code.)

In this assignment the PSE (pronounced see) visualization program will be used to analyze the execution of a SPARC v8plus program (our lighting demo).

For instructions on using PSE visit the EE 4720 procedures page,
`http://www.ece.lsu.edu/ee4720/proc.html`*, and look for the PSE instructions.*

For documentation on the SPARC instructions see
`http://www.ece.lsu.edu/ee4720/doc/JPS1-R1.0.4-Common-pub.pdf`*.*

Datasets, source code, and assembly code for the timing analysis versions of the demo-4-lighting.cc have been placed in `/home/faculty/koppel/pub/gpm/2010/hw01` on the ECE Linux/Sun filesystem.

To view a dataset you could issue a command like.

```
pse /home/classes/ee7700/com/2010/hw01/demo-4-lighting-normal.ds &.
```

To save typing you might create a symbolic link to the dataset directory and use that:

```
cd ~                                          # Or place link elsewhere.
ln -s /home/faculty/koppel/pub/gpm/2010/hw01  # Do this just once.
pse hw01/demo-4-lighting-normal.ds &          # Run PSE in background.
emacs hw01/demo-4-lighting.cc  &              # View source using Emacs.
```

The dataset `demo-4-lighting-normal.ds` shows execution on a four-way superscalar CPU with a 64-entry reorder buffer, two floating-point units, and predict one block (branch) per cycle and fetch two non-contiguous regions per cycle. The processor roughly matches the Fujitsu SPARC 64 VI.

The source code and assembler are also in the hw01 directory.

Problem 1: *Note: This problem is here to check out the software setup and for familiarization. It won't be graded.* The code demonstrating 3D rendering, demo-4-lighting.cc, is in directory /home/faculty/koppel/pub/gpm/2010/hw01(on the ECE Linux/Sun systems), along with the code in SPARC assembly language (demo-4-lighting.s) and a record of its execution on a simulated computer, in file demo-4-lighting-normal.ds.

Load the source and assembler code into the editor of your choice, and display the dataset using PSE. If there are any difficulties with this step ask questions.

In PSE, view a segment PED graph (the one showing instruction execution and assembly code). You should be able to see assembler on the right-hand side.

(*a*) Find a source code line reference in the assembler file, such as demo-4-lighting.cc:712. Then find the corresponding line in the source code file (just jump to the appropriate line number) and in the assembler file (search for "! 712" for example). Use your favorite editor, Emacs with the class setup is recommended.

Problem 2: The dataset file records execution over several frames. Using PSE find the fraction of time (for a frame) spent in each section.

Prepare a table with a row for each section (make up a name, perhaps using comments in the source file). Each row should have:

- A section name.

- The number of cycles in the section.
- The percentage of time for that frame needed for the section.
- The execution rate during the section (in insn per cycle).
- A quick judgment on why the execution was below the peak (or "at peak" if execution was 4 IPC). Possible reasons are: branch misprediction, cache misses, low local ILP.

Numbers appear in table below. The numbers under the "Start" columns are in thousands. The table was constructed by looking for stable regions in the PSE overview graph. Placing the mouse cursor over the first segment in a stable region provides the cycle and instruction numbers appearing in the first two columns below (on PSE they appear at the bottom of the window).

To identify what was going on in that phase the source line numbers displayed in PSE (upper-right) were used to find the source code in the file demo-4-lighting.cc. Some sections were easy, such as for coordinate transformations. To figure out what the "FB Clear" section was doing one needed to look in the frame buffer simulator code. The rasterize section appeared, based on the changing behavior seen in the PSE overview, to be multiple sections, but the line numbers show the different behaviors are from the same code, that doing rasterization. The differences are due to the projected triangle size and perhaps whether the primitive is obscured.

The Pct column shows the percentage of time in that section. The IPC column shows execution rate in IPC.

	----Start----		--Duration--			
	Cycle	Insn	Cycle	Insn	Pct	IPC
FB Clear	14960	33242	160	417	2.3	2.61
ZB Clear	15120	33659	180	707	2.6	3.93
VTX Copy	15300	34366	200	752	2.9	3.76 (closer to 4.)
Coord to eye	15500	35118	220	323	3.2	1.47
Normal to eye	15720	35441	300	274	4.3	0.91
Lighting	16020	35715	660	649	9.5	0.98
Trans to window	16680	36364	240	356	3.5	1.48
Rasterize	16920	36720	4980	12492	71.8	2.51

Note: Cycles and insn counts are in thousands.

None of the sections achieved the peak IPC of 4 that the hardware is capable of. The reasons for following short are described below, these are more detailed than the three reasons given in the assignment, students were not expected to provide this level of detail.

Section limiters: FB Clear: Hardware. The hardware is capable of sustaining execution of up to two memory instructions (store in this case) per cycle.

ZB Clear: Close to peak. Note that the FB clear and ZB clear sections do almost the same thing, initializing an area of memory to some value. Each writes four characters per pixel (of the frame buffer), and each takes roughly the same amount of time to do it. The FB clear code is more efficient so it reaches the hardware limit on two stores per cycle. If the hardware could handle more than two stores per cycle the FB clear code would be much faster than the ZB clear code.

VTX Copy: At peak. The range of cycles shown for vtx copy includes a bit of the coord-to-eye section, so it makes vtx copy look a bit slower.

Coord to eye: Low local ILP. In particular, the code for homogenizing a coordinate puts a divide and multiply in the critical path. The 56 instructions needed to transform a vertex takes 14 cycles to fetch, and the 64-entry ROB can hold little more than one iteration. The time needed to for the division and multiplication takes 21 cycles, enough time to fetch 1.5 iterations, since the ROB can't hold it fetch stalls and so execution is below the peak.

Normal to eye: Low local ILP. Here the normal is normalized after transformation, requiring a square root. This is even more time consuming than the divide needed for homogenization so execution rate is even worse.

Lighting: Low local ILP. One culprit is the square root needed to determine the distance from the light to the vertex.

Trans to window: Low local ILP. Similar to reasons for Coord-to-eye.

Rasterize: During the first part of this section ILP is high. Later it drops due to branch misprediction. When ILP is high it is limited to a small extent by local ILP (some madd instructions). Later, when projected primitives are smaller, it suffers from division needed to set up scan lines and by branch misprediction at scan line ends.

Problem 3: Locate the code that transforms normals and coordinates to eye space, and determine the following for each loop (coordinate loop and normal loop):

(*a*) Find the average number of cycles per vertex and the total number of instructions per vertex. (Per vertex means per coordinate for the coordinate loop and per normal for the normal loop.)

Coordinate transformation: A loop iteration consists of 56 instructions. (This can be found by finding the insn count between occurrences of some instruction, such as the divide instruction that appears once per loop iteration.) To find the average time per vertex measure the time over a large number of vertices. One easy way to do that is to divide the number of instructions by the execution rate (IPC) from the last problem. That gives $56/1.47 = 38.1$ cycles.

Normal transformation: A loop iteration is composed of 46 instructions, so the average time is $46/0.91 = 50.5$ cycles.

(*b*) Provide the number of instructions in each category below:

- Graphical Computation: Floating-Point (used to operate on coordinate or normal).
- Graphical Data Movement: Load or Store of vertex-related data.
- Overhead: Instructions needed to compute addresses, determine whether at end of list of vertices, and other non-graphical tasks.

Instructions in the coordinate transform loop are labeled below. An O in the first column is for overhead, a D is for data movement, and a G is for graphical computation. There are 20 for graphical computation, 17 for graphical data movement, and 19 for overhead.

```
// Coordinate transformation

O  sll     %i1,2,%l6
O  sub     %l6,%i1,%o3
O  sll     %o3,2,%i0
O  sub     %i0,%i1,%o1
O  sll     %o1,2,%g3
O  addcc   %l7,%g3,%l6
O  be,pn   %icc,.L77003469
O  add     %i1,1,%l3
O  prefetch        [%l6+512],2
O  prefetch        [%l6+524],2
O  prefetch        [%l6+512],2
O  prefetch        [%l6+520],2
D  ld      [%fp-2164],%f31
O  st      %l3,[%fp-332]
D  ld      [%l6+4],%f30
D  ld      [%g3+%l7],%f5
D  ld      [%l6+8],%f6
D  ld      [%l6+12],%f7
G  fmuls   %f30,%f23,%f1
G  fmuls   %f30,%f19,%f2
```

```
G  fmuls   %f30,%f20,%f4
G  fmuls   %f30,%f21,%f0
G  fmadds  %f5,%f31,%f1,%f24
G  fmadds  %f10,%f5,%f2,%f25
G  fmadds  %f13,%f5,%f4,%f3
G  fmadds  %f16,%f5,%f0,%f26
G  fmadds  %f17,%f6,%f24,%f27
G  fmadds  %f9,%f6,%f25,%f29
G  fmadds  %f12,%f6,%f3,%f31
G  fmadds  %f15,%f6,%f26,%f4
G  fmadds  %f18,%f7,%f27,%f28
D  st      %f28,[%g3+%l7]
D  ld      [%l6],%f3
G  fmadds  %f8,%f7,%f29,%f30
D  st      %f30,[%l6+12]
D  ld      [%l6+12],%f1
G  fmadds  %f11,%f7,%f31,%f2
G  fmadds  %f14,%f7,%f4,%f0
D  st      %f0,[%l6+4]
D  st      %f2,[%l6+8]
D  ld      [%l6+4],%f24
D  ld      [%l6+8],%f5
G  fdivs   %f22,%f1,%f6
G  fmuls   %f5,%f6,%f26
G  fmuls   %f24,%f6,%f25
D  st      %f25,[%l6+4]
D  st      %f26,[%l6+8]
G  fmuls   %f3,%f6,%f27
D  st      %f27,[%g3+%l7]
D  st      %f22,[%l6+12]
O  ld      [%fp-332],%g3
O  ld      [%fp-340],%g4
O  ld      [%fp-348],%l7
O  cmp     %g3,%g4
O  bne,a,pt        %icc,.L900001151
O  ld      [%fp-332],%i1
```

Instructions in the normal transform loop are labeled below. An O in the first column is for overhead, a D is for data movement, and a G is for graphical computation. There are 17 for graphical computation, 13 for graphical data movement, and 16 for overhead.

```
// Normal transformation

O  sub     %l5,%l3,%o0
O  sll     %o0,2,%i2
O  sub     %i2,%l3,%l4
O  sll     %l4,2,%o7
O  addcc   %l7,%o7,%o4
O  be,pn   %icc,.L77003483
O  add     %l3,1,%o5
O  prefetch        [%o4+544],2
O  prefetch        [%o4+552],2
```

```
O  st      %o5,[%fp-332]
D  ld      [%o4+36],%f0
D  ld      [%o4+32],%f25
D  ld      [%o4+40],%f4
D  ld      [%i3+%lo(___const_seg_900000801)],%f5
G  fmuls   %f0,%f15,%f1
G  fmuls   %f0,%f14,%f2
G  fmuls   %f0,%f13,%f24
G  fmadds  %f11,%f25,%f1,%f26
G  fmadds  %f7,%f25,%f2,%f27
G  fmadds  %f9,%f25,%f24,%f3
G  fmadds  %f10,%f4,%f26,%f6
D  st      %f6,[%o4+36]
D  ld      [%o4+36],%f21
G  fmadds  %f12,%f4,%f27,%f28
D  st      %f28,[%o4+32]
D  ld      [%o4+32],%f30
G  fmadds  %f8,%f4,%f3,%f29
D  st      %f29,[%o4+40]
D  ld      [%o4+40],%f20
G  fmuls   %f21,%f21,%f16
G  fmadds  %f30,%f30,%f16,%f17
G  fmadds  %f20,%f20,%f17,%f18
G  fsqrts  %f18,%f19
G  fdivs   %f5,%f19,%f31
G  fmuls   %f20,%f31,%f23
G  fmuls   %f21,%f31,%f22
D  st      %f22,[%o4+36]
D  st      %f23,[%o4+40]
G  fmuls   %f30,%f31,%f0
D  st      %f0,[%o4+32]
O  ld      [%fp-332],%l3
O  ld      [%fp-340],%g4
O  ld      [%fp-348],%l7
O  cmp     %l3,%g4
O  bne,pt  %icc,.L900001150
O  sll %l3,2,%l5
```

Problem 4: Continue considering the coordinate and normal transformation code.

(*a*) Find an example of where this code (either the coordinate or normal transformation) can be improved. Show the existing code and show your improved version. *Hint: There is at least one example that does not require in-depth knowledge of how the transformation or vertex list code works.*

The coordinate transform loop is shown below with eliminated instructions marked with an X, followed by a reason (sometimes a reason is given for a group of adjacent instructions).

The first group of instructions can be eliminated by just adding on the size of a pVertex element to compute the address of the next vertex to load. Instead the code multiplies an index by the size of a pVertex, but does so using multiple adds and shifts.

The compiler issues four prefetches when one would do.

There are many cases where values are unnecessarily loaded and sometimes stored and loaded. (Yes, the optimization flag was used!)

The suggestions below would eliminate 19 instructions, bringing the vertex loop instruction count to $46 - 19 = 27$. In once case an eliminated store/load pair is on the critical path, look for the CX to the right of the instruction. Since they are on the critical path their elimination saves one cycle per iteration. (If the store and load each took one cycle, their elimination would save two cycles. However the hardware links them, saving a cycle.)

```
! Coordinate transform code.

O  sll     %i1,2,%l6       X Eliminate, these.
O  sub     %l6,%i1,%o3     X Instead just add size of vertex
O  sll     %o3,2,%i0       X to vertex pointer and compare to
O  sub     %i0,%i1,%o1     X pre-computed end address.
O  sll     %o1,2,%g3       X
O  addcc   %l7,%g3,%l6
O  be,pn   %icc,.L77003469
O  add     %i1,1,%l3
O  prefetch        [%l6+512],2
O  prefetch        [%l6+524],2  X Eliminate (one sufficient)
O  prefetch        [%l6+512],2  X Eliminate
O  prefetch        [%l6+520],2  X Eliminate
D  ld      [%fp-2164],%f31
O  st      %l3,[%fp-332]        X Eliminate, sufficent regs.
D  ld      [%l6+4],%f30
D  ld      [%g3+%l7],%f5
D  ld      [%l6+8],%f6
D  ld      [%l6+12],%f7
G  fmuls   %f30,%f23,%f1
G  fmuls   %f30,%f19,%f2
G  fmuls   %f30,%f20,%f4
G  fmuls   %f30,%f21,%f0
G  fmadds  %f5,%f31,%f1,%f24
G  fmadds  %f10,%f5,%f2,%f25
G  fmadds  %f13,%f5,%f4,%f3
G  fmadds  %f16,%f5,%f0,%f26
G  fmadds  %f17,%f6,%f24,%f27
G  fmadds  %f9,%f6,%f25,%f29
G  fmadds  %f12,%f6,%f3,%f31
G  fmadds  %f15,%f6,%f26,%f4
G  fmadds  %f18,%f7,%f27,%f28
D  st      %f28,[%g3+%l7]
D  ld      [%l6],%f3
G  fmadds  %f8,%f7,%f29,%f30
D  st      %f30,[%l6+12]          CX Unnecessary reg movement.
D  ld      [%l6+12],%f1           CX
G  fmadds  %f11,%f7,%f31,%f2
G  fmadds  %f14,%f7,%f4,%f0
D  st      %f0,[%l6+4]            X Unnecessary register movement.
D  st      %f2,[%l6+8]            X
D  ld      [%l6+4],%f24           X
```

```
D  ld      [%l6+8],%f5              X
G  fdivs   %f22,%f1,%f6
G  fmuls   %f5,%f6,%f26
G  fmuls   %f24,%f6,%f25
D  st      %f25,[%l6+4]
D  st      %f26,[%l6+8]
G  fmuls   %f3,%f6,%f27
D  st      %f27,[%g3+%l7]
D  st      %f22,[%l6+12]
O  ld      [%fp-332],%g3          X Have enough regs, no
O  ld      [%fp-340],%g4          X aliasing.
O  ld      [%fp-348],%l7          X
O  cmp     %g3,%g4
O  bne,a,pt        %icc,.L900001151
O  ld      [%fp-332],%i1          X Have enough regs.
```

(*b*) Estimate the speedup of your code (the average cycles per vertex before and after your change).

See above.

Problem 5: Consider just the code that transforms normals to eye space. The execution rate for this code (and other code too) would be improved by a larger ROB.

(*a*) Estimate the minimum size of the ROB needed for full-speed execution by measuring the computation critical path: a sequence of instructions in which successive instructions wait for the previous one. (These stand out and can be verified using PSE's ability to highlight dependencies.) Note that if the critical path is C cycles then during its execution the CPU would fetch $4C$ instructions and the ROB would approximately need space to hold them. Use this fact to estimate the minimum size needed.

The critical path length is about 99 cycles, from the first insn in the loop to the completion of the last fmuls. In those 99 cycles 396 instructions could be fetched, of which about 44 are for the vertex being computed and so space is needed for at least $396 - 44 = 352$ instructions.

(*b*) Explain why the modified code below might improve performance on this system. Optional: Estimate by how much the modified code will improve performance based upon critical path measurements above.

```
// Before
while ( pVertex* const v = vtx_list.iterate() )
  {
    v->normal *= nte;
    v->normal.normalize();
  }

// After
while ( pVertex* const v = vtx_list.iterate() )
  v->normal *= nte;
while ( pVertex* const v = vtx_list.iterate() )
  v->normal.normalize();
```

The code above might improve performance by shortening the time per loop iteration.