Name ______________________

EE 7700-1

Final Examination

Friday, 4 May 2007 to Friday 11 May 2007

Problem 1	______	(30 pts)
Problem 2	______	(30 pts)
Problem 3	______	(10 pts)
Problem 4	______	(15 pts)
Problem 5	______	(15 pts)
Exam Total	______	(100 pts)

Alias ______________________

Good Luck!

Problem 1: For the problems below consider the loop starting at address `0x11fac` in gzip, and the loop's execution on the machine models that are being used in class. The loop assembler is here for your convenience, it will be needed to solve several problems. Note that `bcs` is equivalent to branch less than unsigned. [30 pts]

```
fill_window$deflate+57
 0x11fac  mov  0, %o4
 0x11fb0  cmp  %o1, %i3
 0x11fb4  add  %o3, 1, %o0
 0x11fb8  bcs  +3i -> 0x11fc4
 0x11fbc  mov  %o0, %o3
 0x11fc0  sub  %o1, %i3, %o4
 0x11fc4  sth  %o4, [ %o5 ]
 0x11fc8  add  %o5, 2, %o5
 0x11fcc  cmp  %o0, %i3
 0x11fd0  bcs,a  -9i -> 0x11fac
 0x11fd4  lduh  [ %o5 ], %o1
```

(*a*) Find a place where execution of the loop is fetch limited despite branches being correctly predicted.

☐ Run ID:

☐ Segment:

☐ Tag (at bottom of window):

(*b*) What is the largest fetch/decode width for which the original loop will be fetch limited in regions without branch mispredictions? Explain. (The answer can be any fetch/decode width, not just the ones simulated for class (2, 4, 8, 16)).

Problem 1, continued:

(*c*) Devise a frame for the loop, the frame should improve performance both on fetch- and execute-limited regions. Don't unroll the loop (the frame should cover just one iteration).

Use assert instructions of the form `assert.CM REG,REG`, where `CM` is a comparison such as `lt` (less than), `ltu` (less than unsigned), `le` (less than or equal to), `eq`, `ne`, `gt`, etc., and `REG` is a register. For example,

```
assert.gt %i1, %o0   ! Assertion fails if %i1 is not greater than %o0.
```

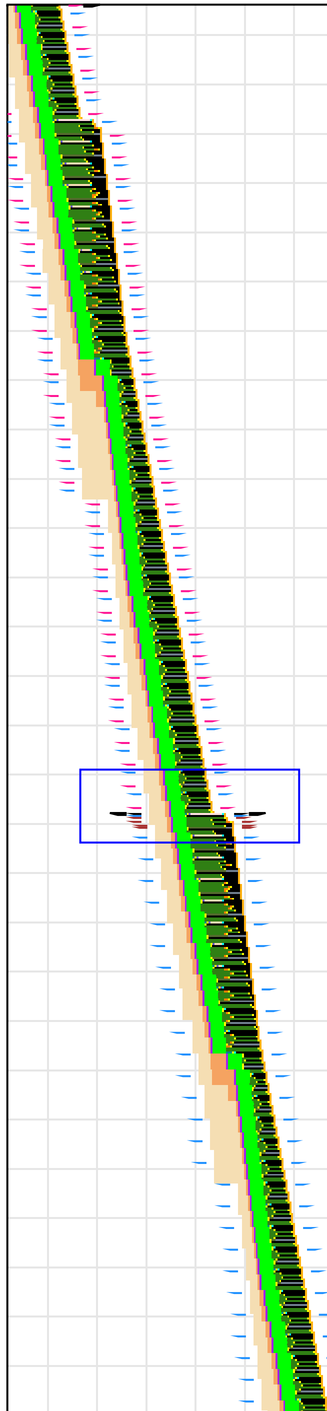
- Don't unroll the loop.
- Other than `assert` use only SPARC V8 instructions.
- Consider regions without branch mispredictions and cache misses.

☐ Frame:

☐ What is the speedup using this frame on a 2-way superscalar system like sm.98.114? Explain.

☐ What is the speedup using this frame on a 16-way superscalar system like sm.98.126? Explain.

Problem 2: Consider the execution of the loop in sm.98.122 segment 63 which is commit limited due to L1 cache misses (L2 cache hits). [30 pts]



Snip Serial 2

```
fill_window$deflate+67
 00011fd4   lduh  [ %o5 ], %o1  {[prev[26334]]}  {[0x919fc]}
fill_window$deflate+57
 00011fac   mov  0, %o4  {0x0}      Redef +5 0x11fc0
 00011fb0   cmp  %o1, %i3
 00011fb4   add  %o3, 1, %o0
 00011fb8   bcs  +3i -> {0x11fc4 fill_window$deflate+63}
 00011fbc   mov  %o0, %o3
 00011fc0   sub  %o1, %i3, %o4
 00011fc4   sth  %o4, [ %o5 ]  {[prev[26334]]}  {[0x919fc]}
 00011fc8   add  %o5 , 2,  %o5
 00011fcc   cmp  %o0, %i3
 00011fd0   bcs,a  -9i -> {0x11fac fill_window$deflate+57}
 00011fd4   lduh  [ %o5 ], %o1  {[prev[26335]]}  {[0x919fe]}
fill_window$deflate+57
 00011fac   mov  0, %o4  {0x0}      Redef +5 0x11fc0
 00011fb0   cmp  %o1, %i3
 00011fb4   add  %o3, 1, %o0
 00011fb8   bcs  +3i -> {0x11fc4 fill_window$deflate+63}
 00011fbc   mov  %o0, %o3
 00011fc0   sub  %o1, %i3, %o4
 00011fc4   sth  %o4, [ %o5 ]  {[prev[26335]]}  {[0x919fe]}
 00011fc8   add  %o5 , 2,  %o5
 00011fcc   cmp  %o0, %i3
 00011fd0   bcs,a  -9i -> {0x11fac fill_window$deflate+57}
 00011fd4   lduh  [  %o5  ], %o1   {[prev[26336]]}  {[0x91a00]}
fill_window$deflate+57
 00011fac   mov  0, %o4  {0x0}      Redef +5 0x11fc0
 00011fb0   cmp  %o1 , %i3
 00011fb4   add  %o3, 1, %o0
 00011fb8   bcs  +3i -> {0x11fc4 fill_window$deflate+63}
 00011fbc   mov  %o0, %o3
 00011fc0   sub  %o1 , %i3,  %o4
 00011fc4   sth  %o4 , [ %o5 ]   {[prev[26336]]}  {[0x91a00]}
 00011fc8   add  %o5, 2, %o5
 00011fcc   cmp  %o0, %i3
 00011fd0   bcs,a  -9i -> {0x11fac fill_window$deflate+57}
 00011fd4   lduh  [ %o5 ], %o1  {[prev[26337]]}  {[0x91a02]}
fill_window$deflate+57
 00011fac   mov  0, %o4  {0x0}      Redef +5 0x11fc0
```

Time 98,739,284

Grid 25 insn X 25 cyc

Problem 2, continued:

(*a*) Insert a prefetch instruction in the original code to avoid the cache misses. Choose the prefetch instruction so that the target line arrives just in time for the target load whenever the line is not in the L1 cache but is in the L2 cache. A prefetch instruction is similar to an ordinary load, it looks like a load that writes register `g0` (the dummy register). Unlike ordinary loads, a prefetch can commit before its data arrives.

```
fill_window$deflate+57
 0x11fac  mov  0, %o4
 0x11fb0  cmp  %o1, %i3
 0x11fb4  add  %o3, 1, %o0
 0x11fb8  bcs  +3i -> 0x11fc4
 0x11fbc  mov  %o0, %o3
 0x11fc0  sub  %o1, %i3, %o4
 0x11fc4  sth  %o4, [ %o5 ]
 0x11fc8  add  %o5, 2, %o5
 0x11fcc  cmp  %o0, %i3
 0x11fd0  bcs,a  -9i -> 0x11fac
 0x11fd4  lduh  [ %o5 ], %o1
```

(*b*) Would the modified code run faster or slower on the configuration in sm.98.122? (Assume, of course, that the prefetch instructions operate correctly and do not create misses elsewhere.)

(*c*) Would the modified code run faster or slower on a 4-way superscalar system that was otherwise identical to sm.98.122? (Note: sm.98.116 and sm.98.115 are *not* such systems.) Explain.

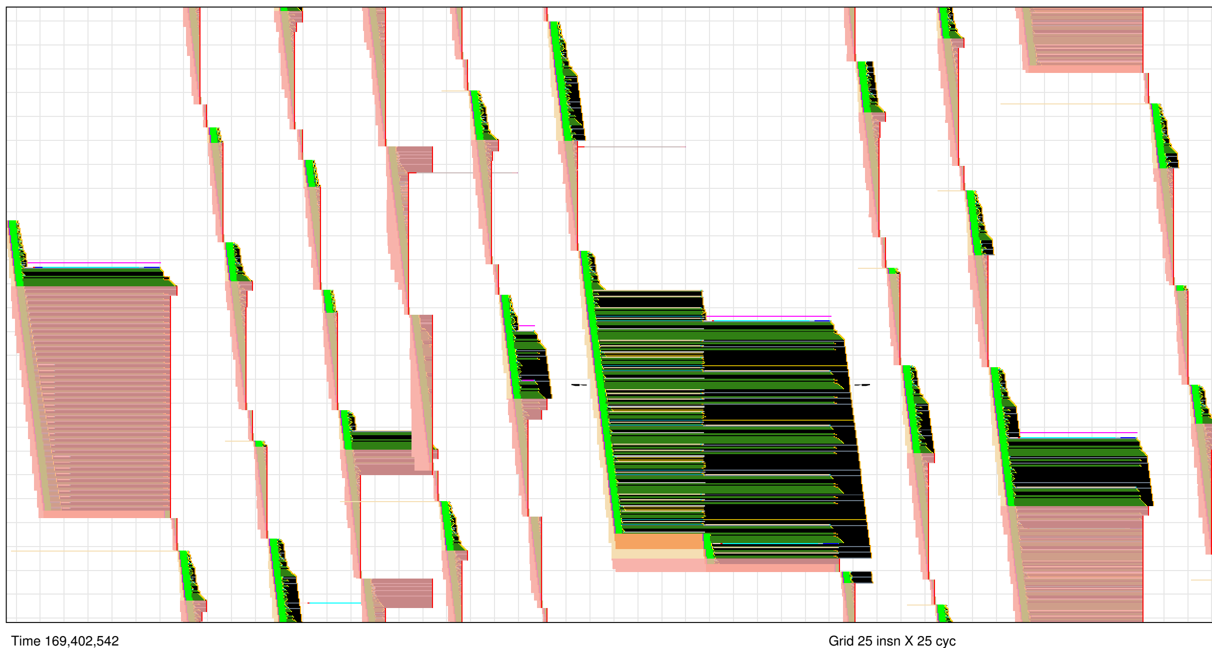
Problem 2, continued: Continue to consider the loop starting at address `0x11fac` in gzip.

(*d*) On which configurations could pre-execution for this loop hurt performance? Assume that cache misses are a problem that pre-execution solves. Explain.

(*e*) How can value prediction improve execution of this loop beyond what is possible with pre-execution? Explain.

(*f*) For this loop, which is better, pre-execution or hardware prefetch? Explain.

Problem 3: Consider the execution of code in Segment 120 of sm.98.056, which shows a run of gap on an 8-way superscalar system. [10 pts]



(*a*) Is there any region in this segment that is commit limited? Explain, use an example if necessary.

(*b*) Suppose there were no branch mispredictions in this segment, the ROB could be any finite size you choose, and that execution beyond this segment is similar. Would the execution be fetch limited or execute limited? Explain, stating any assumption made. (A correct answer would have to discuss and possibly assume behavior of some stick-out-like-a-sore-thumb load instructions.)

Problem 4: Consider a processor that had rePlay and also had a reconfigurable functional unit that could perform operations roughly as complex as an integer add (which, given carry-look ahead, is not exactly simple), the unit would be programmed in advance, in this case by rePlay at the time of frame construction. When rePlay is constructing a frame it will fuse groups of simple instructions into a single custom instruction and prepare the reconfigurable hardware for it.

Back to Segment 120 of sm.98.056. It seems the branch at `0x3de0c` is responsible for over half of the mispredictions in this segment. [15 pts]

(*a*) How could this reconfigurable rePlay use the reconfigurable hardware to lessen the impact of mispredictions of this branch? Show enough detail to answer this and the next part, the entire frame does not have to be shown.

☐ Show instructions that will be fused.

☐ Describe any other relevant details.

(*b*) Estimate how many cycles would be saved over the execution of the benchmark due to this frame.

Problem 5: Once again consider Segment 120 of sm.98.056. The branch at `0x3de34` seems to be responsible for most of the rest of the mispredictions in this segment. This branch, `0x3de34`, is for a loop that iterates for a specified number of iterations, the number of iterations can vary. As can be seen from the local history for this branch the number of iterations is small:
`tttttntntntntttntnttntnttntttntntttntntttttttntntntntntttttntnt`

Unlike some other branches with irregular behavior, it seems that because the number of iterations is known this one ought to be predicted better. [15 pts]

(*a*) Describe how such a predictor might work. *Hint: It might be make use of hardware that can find a backward slice.*

Consider such a predictor, and assume that it is able to perfectly make use of the number of iterations (once that information is available).

(*b*) Find an instance in which the predictor would save many cycles.

☐ Provide the tag of the branch (see bottom of window when hovering over an instruction).

☐ Indicate how many cycles are saved.

(*c*) Find an instance in which the predictor would not save any cycles.

☐ Provide the tag of the branch (see bottom of window when hovering over an instruction).

☐ Briefly indicate why none are saved.