Homework 5 answers, Winter 2014

1. 
   a. The paper is trying to make the wakeup/select logic non-atomic. If it is atomic you are stuck with either a long clock period or losing the ability to execute dependent instructions back-to-back.

   b. 
      i. Reg read is reading from the PRF. In class we had the values stored in the RS, here the values are being read as the instruction is issued to the execution units.
      ii. The tag from the XOR is being broadcast on the bus during select/broadcast.
      iii. In the execute/bypass stage the instruction is executed. If data needed by the instruction is coming from an EX unit that just finished in the last cycle, that it will grab the data (bypassing the register file) from the EX unit at that time.

   c. Points given for nearly anything that seems on-point.

2. 
   \[ \text{L.D } F0,0(R1) \quad ; \quad F0=\text{Mem}[R1+0] \]
   \[ \text{(p1)} \quad \text{DADD } R1,R2,R3 \quad ; \quad \text{if (p1) then } R1=R2+R3 \]
   \[ \text{(p2)} \quad \text{DSUB } R5,R1,R4 \quad ; \quad \text{if (p2) then } R5=R1-R4 \]

   The first instruction must use the M-Unit while the other two can use either the M-unit or the I-unit. There also must be a stop between the second and third instruction to insure correctness. If it is known at compile time that p1 and p2 cannot both be true then no stop is needed. Also if there is a dependency with the next bundle then a stop is needed at the end of this bundle.
   - Template 0 can be used if at compile time p1 and p2 are known not to both be true and this bundle is dependent on the next bundle.
   - Template 1 can be used if at compile time p1 and p2 are known not to both be true and this bundle is dependent on the next bundle.
   - Template 2 can be used if at compile time p1 and p2 are not known not to both be true and this bundle is not dependent on the next bundle.
   - Template 3 can be used if at compile time p1 and p2 are not known not to both be true and this bundle is dependent on the next bundle.
   - Template 8 can be used if at compile time p1 and p2 are known not to both be true and this bundle is dependent on the next bundle.
   - Template 9 can be used if at compile time p1 and p2 are known not to both be true and this bundle is dependent on the next bundle.

   From a grading view point, as long as you identify the MII issue and the potential need for a stop bit between the 2\textsuperscript{nd} and 3\textsuperscript{rd} instruction, we’ll give you full credit.
3a.

\[
\begin{align*}
A &= \text{mov } rx, \text{MAX-2} \\
B &= \text{ld } r3, (r1)++ \\
C &= \text{ld } r4, (r2)++ \\
D &= \text{st } r5, (r0)++
\end{align*}
\]

3b.

If, in the original C code, MAX is less than 3 the software-pipelined loop will behave incorrectly.

4a. Tricky question.

Let’s first remove the 5M BILs/sec. So 66M BRL, BRILs and BWLs. We know that only BRLs and BRILs can cause an eviction and that 20% of all evictions cause dirty data. Thus \((\text{BRL+BRIL}) \times 1.2 = 66\text{M}\). So \(\text{BRL+BRIL}=55\text{M}\). Further we know that \(\frac{1}{4}\) of all transactions are caused by stores. We now know that \(\text{BRL+BRIL+BIL}=60\text{M}\) and that \(\text{BRIL+BIL}\) must be 15M (\(1/4\) of 60). So we get:

- 5 million BILs/sec (given)
- 10 million BRILs/sec
- 45 million BRLs/sec
- 11 million BWLs/sec

4b.

E state would reduce the number of times we need to do a BIL (BILs happen when the data is in the S state and we write to it, if it were in the E state we’d not need to do a BIL as we can go from E to M silently).

4c.

This will cause many more lines to be in the M state, thus creating more dirty evictions. Thus the number of BWLs will go up. At the same time, the number of BILs will go down. Though the question didn’t ask, it is very unlikely this will be a net positive.

5a.

An advanced load is one that has been moved above a (possibly conflicting) store. We are worried about the store(s) the load was moved above writing to the same address (thus making our early load incorrect). The way it works is fairly similar to the way a load-lock, store-conditional scheme works. When we do an advanced load (say \(\text{ld8.a}\)) the address for the load is put into the ALAT. Any conflicting store to that same address will cause the ALAT to clear that entry. When we do a check (either \(\text{ld.c}\) or \(\text{chk.a}\)) then the hardware checks to see if the address is still in the ALAT. If it isn’t in the ALAT we either a) redo the load (for a \(\text{ld.c}\)) or call some fix-up code (for the \(\text{chk.a}\)).

5b.

A speculative load is one that has been moved above a branch. We are worried about the load throwing an exception when the load wasn’t supposed to have been executed. They way this works is that if the load does throw an exception, the exception is deferred and
the load’s destination register is labeled as junk using a “NaT” (Not a Thing) bit. A NaT bit gets attached to any instruction that uses a “NaT-ed” register as a source register. Then, if the load was supposed to occur we handle the exception once a “chk.s” instruction associated with that load (or register down-stream from the load) is checked.

5c. The primary advantage a compiler has over the hardware is one of scope. Not only does the compiler have an effectively infinite reordering window, it also has access to the source code. From that, certain information can be divined—information which the translation to machine code would have lost. An example would be two memory accesses being to two different arrays and thus being known to not-overlap (in certain languages anyways).

There are a large number of examples of the two working together. One would be that the hardware (on the Alpha and most computers) provides a separate instruction for a “call” and a “return” even if those two do exactly the same thing. The software than uses its view into the source code to identify calls and returns. The hardware then uses this information to better predict branches (using a RAS for example). That’s hardware providing an instruction to the software so it can help the hardware with branch prediction. Similar examples would be prefetching instructions (instructions which just move data into the cache without doing a load to a register) and branches which the compiler identifies if the branch is likely to be taken or not.

Another set of examples come from IA-64. For the advanced load there the hardware is providing a fair bit of support for hoisting loads above a branch (NaT bits, additional instructions, etc.) so the compiler can better optimize execution. Similar IA-64 examples would be speculative loads, register rotation and the like.