EE 4720 Homework 5 Due: 5 December 2001

Problem 1: An ISA has a character size of $c = 9$ bits (one more than most other ISA's!) and a 30-bit address space (A). An implementation has a bus width of $w = 72$ bits and has no cache. Show how $2^{20} \times 36$ memory devices can be connected to implement the entire address space for this implementation. Show only the connections needed for loads. Show the alignment network as a box. Label inputs and outputs and be sure to specify which address bits are being used. The solution will require many memory devices so use ellipses ($\cdots$) between the first and last of a large group of items.

Problem 2: The program below computes the sum of an array of doubles and also computes the sum of the characters in the array. The system uses a direct-mapped cache consisting of 1024 lines with a line size of 256 bits.

```
void p3(double *dstart, double *dend)
{
  double dsum          = 0.0;
  int csum             = 0;
  double *d            = dstart;               // sizeof(double) = 8 characters
  unsigned char *c     = (unsigned char *) dstart;     // A character is 8 bits.
  unsigned char *cend = (unsigned char *) dend;
  int dlength          = dend - d;
  int clength          = cend - c;

  while( d < dend ) dsum += *d++;

  if( ! LAST_PART ) flush_the_cache();  // Removes all data from the cache.

  while( c < cend ) csum += *c++;
}
```

When the procedure is called none of the data in the array is cached. When answering the questions below consider only memory accesses needed for the array (double or character). Assume that the number of iterations is some convenient number, except zero of course.

Note: Though they both access the same amount of data the number of iterations of the two loops are different. The first while loop is equivalent to: `for(i=0; i<dlength; i++) dsum = dsum + dstart[i];`

(a) What is the hit ratio for the first while loop? Assuming the cache is flushed (emptied) between the two while loops, what is the hit ratio for the second while loop?

(b) Consider a single-issue (one-way) statically scheduled system in which the pipeline stalls on a cache miss. The cache miss delay is 1000 cycles. Roughly how does the time needed to execute the two loops compare? Assume that when there's a cache hit the time needed for one iteration is the same for both loops.

(c) Consider a single-issue (one-way) dynamically scheduled system with perfect branch and branch target prediction, a non-blocking cache, and a reorder buffer that can hold sixteen iterations of the while loops. The miss delay is still 1000 cycles however assume that for cache misses there is an initiation interval of one cycle so that the data for misses at $t = 0$ and $t = 1$ will arrive at $t = 1000$ and $t = 1001$, respectively. *Now* how do the two loops compare?

(*d*) Suppose the cache is *not* flushed before the second while loop executes. What is the smallest value of `dlength` (dee, not cee) for which the hit ratio of the second loop is less than 1.0?

Problem 3: The SPARC V9 program below adds an array of integers.
(See `http://www.ece.lsu.edu/ee4720/samv9.pdf` for a description of SPARC V9.) Except for `prefetch` these instructions (or similar ones) have been covered before. The `prefetch` instruction is used to avoid the type of cache misses suffered by the program in the previous problem. It is like a `nop` in that it does not modify registers or memory, however like a load instruction, it moves data into the cache. As used below it will fetch data that will be needed ten iterations later. The data will be moved in to the cache (if not already present) but not in to a register. Ten iterations later the `ldx` instruction will move the data in to register `%l2`. Unlike loads, `prefetch` instructions never raise an exception. If the address is invalid or there is another problem the `prefetch` instruction does nothing, so there is no danger in prefetching, say, past the end of an array.

Unlike for a load that misses the cache, a statically scheduled processor would not stall on a prefetch miss. (There'd be no point in that!)

```
! Reminder: In SPARC assembler the destination register is on the right side.
LOOP:
  ldx [%l1], %l2            ! Load extended word (64 bits, same size as reg)
  prefetch [80+%l1], 1      ! Prefetch from address 40+%l1, type 1
  add %l1, 8, %l1
  subcc %l4, %l1, %g0       ! %g0 = %l4 - %l1. (%g0 is zero register.) Set cc.
  bpg LOOP,pt               ! Branch if condition code >0, predict taken
  add %l3, %l2, %l3         ! Branch delay slot.
```

(*a*) In the code above the prefetch *distance* is ten iterations. What is the problem with the distance being too large or too small?

(*b*) Suppose SPARC V9 did not have a `prefetch` instruction. Explain how `ldxa` could be used as a prefetch. Show a replacement for `prefetch` in the program above.

The `ldxa` and similar instructions include an *address space identifier* (ASI) which specifies which address space to load or store from. The ASI can be specified with an immediate or the `%asi` register. See the architecture manual. Normal loads and stores use the `ASI_PRIMARY` address space. `ldxa` lets you specify a different one. A load from a particular address in two different address spaces may load from two different memory locations or may load the same memory location in different ways. For example, an ordinary load of an address, `ldx [%l1], %l2`, would load an integer using big-endian ordering. But a load to the same address using the `ASI_PRIMARY_LITTLE`, `ldxa [%l1] ASI_PRIMARY_LITTLE, %l2` loads an integer using little-endian ordering. Table 12 in the architecture manual lists some of the address spaces.

Hint: Think about the destination register and the ASI.