**Assignment 2**

Due: 11:59 PM, February 27

Solve Problem 2.1, 2.3, and 2.4 in the textbook (1st edition).

**2.1.** When we were discussing floating point addition, we made the simplifying assumption that each of the functional units took the same amount of time. Suppose that fetch and store each take 2 nanoseconds and the remaining operations each take 1 nanosecond.

**a.** How long does a floating point addition take with these assumptions?

**b.** How long will an unpipelined addition of 1000 pairs of floats take with these assumptions?

**c.** How long will a pipelined addition of 1000 pairs of floats take with these assumptions?

**d.** The time required for fetch and store may vary considerably if the operands/results are stored in different levels of the memory hierarchy. Suppose that a fetch from a level 1 cache takes two nanoseconds, while a fetch from a level 2 cache takes five nanoseconds, and a fetch from main memory takes fifty nanoseconds. What happens to the pipeline when there is a level 1 cache miss on a fetch of one of the operands? What happens when there is a level 2 miss?

| Time | Operation | Operand 1 | Operand 2 | Result |
|---|---|---|---|---|
| 0 | Fetch operands | $9.87 \times 10^4$ | $6.54 \times 10^3$ | |
| 1 | Compare exponents | $9.87 \times 10^4$ | $6.54 \times 10^3$ | |
| 2 | Shift one operand | $9.87 \times 10^4$ | $0.654 \times 10^4$ | |
| 3 | Add | $9.87 \times 10^4$ | $0.654 \times 10^4$ | $10.524 \times 10^4$ |
| 4 | Normalize result | $9.87 \times 10^4$ | $0.654 \times 10^4$ | $1.0524 \times 10^5$ |
| 5 | Round result | $9.87 \times 10^4$ | $0.654 \times 10^4$ | $1.05 \times 10^5$ |
| 6 | Store result | $9.87 \times 10^4$ | $0.654 \times 10^4$ | $1.05 \times 10^5$ |

**Table 2.3** Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

| Time | Fetch | Compare | Shift | Add | Normalize | Round | Store |
|---|---|---|---|---|---|---|---|
| 0 | 0 | | | | | | |
| 1 | 1 | 0 | | | | | |
| 2 | 2 | 1 | 0 | | | | |
| 3 | 3 | 2 | 1 | 0 | | | |
| 4 | 4 | 3 | 2 | 1 | 0 | | |
| 5 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 6 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 999 | 999 | 998 | 997 | 996 | 995 | 994 | 993 |
| 1000 | | 999 | 998 | 997 | 996 | 995 | 994 |
| 1001 | | | 999 | 998 | 997 | 996 | 995 |
| 1002 | | | | 999 | 998 | 997 | 996 |
| 1003 | | | | | 999 | 998 | 997 |
| 1004 | | | | | | 999 | 998 |
| 1005 | | | | | | | 999 |

Solution:
(a) 2 (fetch) + 1 (Compare) + 1 (Shift) + 1 (Add) + 1 (Normalize) + 1 (Round) + 2 (Store) = 9 nanoseconds
(b) 9 * 1000 = 9000 nanoseconds
(c) 2 * 999 + 9 = 2007 nanoseconds

| Time | F | C | Sh | A | N | R | St |
|---|---|---|---|---|---|---|---|
| 0 | 1 | | | | | | |
| 1 | 1 | | | | | | |
| 2 | 2 | 1 | | | | | |
| 3 | 2 | | 1 | | | | |
| 4 | 3 | 2 | | 1 | | | |
| 5 | 3 | | 2 | | 1 | | |
| 6 | 4 | 3 | | 2 | | 1 | |
| 7 | 4 | | 3 | | 2 | | 1 |
| 8 | 5 | 4 | | 3 | | 2 | 1 |
| 9 | 5 | | 4 | | 3 | | 2 |
| 10 | 6 | 5 | | 4 | | 3 | 2 |
| 11 | 6 | | 5 | | 4 | | 3 |

(d) From the previous part, we see that once the pipeline is full, when a fetch begins, it looks something like this:

| Time | F | C | Sh | A | N | R | St |
|---|---|---|---|---|---|---|---|
| $t$ | $n$ | $n-1$ | | $n-2$ | | $n-3$ | $n-4$ |

So if operand $n$ wasn't in Level 1 cache, but it was in Level 2 cache, then the fetch will take 5 nanoseconds. So our pipeline might look something like this:

| Time | F | C | Sh | A | N | R | St |
|---|---|---|---|---|---|---|---|
| $t$ | $n$ | $n-1$ | | $n-2$ | | $n-3$ | $n-4$ |
| $t+1$ | $n$ | | $n-1$ | | $n-2$ | | $n-3$ |
| $t+2$ | $n$ | | | $n-1$ | | $n-2$ | $n-3$ |
| $t+3$ | $n$ | | | | $n-1$ | | $n-2$ |
| $t+4$ | $n$ | | | | | $n-1$ | $n-2$ |
| $t+5$ | $n+1$ | $n$ | | | | | $n-1$ |

Of course a Level 2 miss will be even worse: the data will stay in the fetch stage from $t$ to $t+50$. Clearly, there is an opportunity to improve overall performance if there is some useful work for the processor to do while the fetch from main memory is taking place.

**2.3.** Recall the example involving cache reads of a two-dimensional array (page 22). How does a larger matrix and a larger cache affect the performance of the two pairs of nested loops? What happens if `MAX = 8` and the cache can store four lines? How many misses occur in the reads of `A` in the first pair of nested loops? How many misses occur in the second pair?

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

To better understand this, suppose `MAX` is four, and the elements of `A` are stored in memory as follows:

| Cache Line | Elements of `A` | | | |
|---|---|---|---|---|
| 0 | `A[0][0]` | `A[0][1]` | `A[0][2]` | `A[0][3]` |
| 1 | `A[1][0]` | `A[1][1]` | `A[1][2]` | `A[1][3]` |
| 2 | `A[2][0]` | `A[2][1]` | `A[2][2]` | `A[2][3]` |
| 3 | `A[3][0]` | `A[3][1]` | `A[3][2]` | `A[3][3]` |

Solution:

Suppose the matrix has order 8 or 64 elements; the cache line size is still 4; the cache can store 4 lines; and the cache is direct-mapped. Then the following table shows how $A$ is stored in cache lines.

| Cache Line | Elements of $A$ | | | |
|---|---|---|---|---|
| 0 | A[0][0] | A[0][1] | A[0][2] | A[0][3] |
| 1 | A[0][4] | A[0][5] | A[0][6] | A[0][7] |
| 2 | A[1][0] | A[1][1] | A[1][2] | A[1][3] |
| 3 | A[1][4] | A[1][5] | A[1][6] | A[1][7] |
| 4 | A[2][0] | A[2][1] | A[2][2] | A[2][3] |
| 5 | A[2][4] | A[2][5] | A[2][6] | A[2][7] |
| 6 | A[3][0] | A[3][1] | A[3][2] | A[3][3] |
| 7 | A[3][4] | A[3][5] | A[3][6] | A[3][7] |
| 8 | A[4][0] | A[4][1] | A[4][2] | A[4][3] |
| 9 | A[4][4] | A[4][5] | A[4][6] | A[4][7] |
| 10 | A[5][0] | A[5][1] | A[5][2] | A[5][3] |
| 11 | A[5][4] | A[5][5] | A[5][6] | A[5][7] |
| 12 | A[6][0] | A[6][1] | A[6][2] | A[6][3] |
| 13 | A[6][4] | A[6][5] | A[6][6] | A[6][7] |
| 14 | A[7][0] | A[7][1] | A[7][2] | A[7][3] |
| 15 | A[7][4] | A[7][5] | A[7][6] | A[7][7] |

Assuming that no lines of $A$ are in the cache when the first pair of loops begins, we see that there will be two misses for each row of $A$. Also, after the first two rows have been read, the cache will be full, and each miss will evict a line. As in the example in the text, an evicted line won't need to be read again. So we see that the total number of misses for the first pair of loops is 16, or

$$\frac{\text{number of elements in A}}{\text{cache line size}}.$$

More generally, then, for the first pair of loops, the number of misses is only affected by the size of $A$, not the size of the cache.

For the second pair of nested loops, let's also assume that no lines of $A$ are in the cache when the loops begin. When we're working with column 0 ($j = 0$), each time we multiply `A[i][0]*x[0]` we'll need to first load a new cache line, since the previously read lines will only contain elements from rows $< i$. So executing the loop with $j = 0$, will result in 8 misses. After reading in the four lines containing `A[0][0]`, `A[1][0]`, `A[2][0]`, and `A[3][0]`, respectively, subsequent reads of elements of column 0 will evict these four lines. So after completing the multiplications involving column 0 of $A$, no elements in the first 4 rows of $A$ will be stored in the cache. So every read of an element of $A$ in rows 0–3 of column 1 will result in a miss. In fact, we see that every multiplication will result in a miss and there will be 64 misses.

Observe, however, in the second pair of loops if we have an 8 line cache, then the multiplications involving columns 1–3 of $A$ won't result in misses, and we won't have additional misses until we start the multiplications by elements in column 4. Once the lines containing elements of column 4 are loaded, there won't be any additional misses, and we see that with an 8 line cache, the total number of misses is reduced to 16.

**2.4.** In Table 2.2, virtual addresses consist of a byte offset of 12 bits and a virtual page number of 20 bits. How many pages can a program have if it's run on a system with this page size and this virtual address size?

**Table 2.2** Virtual Address Divided into Virtual Page Number and Byte Offset

| Virtual Address | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Virtual Page Number | | | | | Byte Offset | | | | |
| 31 | 30 | ⋯ | 13 | 12 | 11 | 10 | ⋯ | 1 | 0 |
| 1 | 0 | ⋯ | 1 | 1 | 0 | 0 | ⋯ | 1 | 1 |

Solution: $2^{20}$ = 1,048,576 pages