DIRO
IFT 1215

DÉMONSTRATION N° 8 & 9

– Correction –

Max Mignotte

DIRO, Département d'Informatique et de Recherche Opérationelle, local 2384.
http: //www.iro.umontreal.ca/∼mignotte/ift1215/
E-mail: mignotte@iro.umontreal.ca

CPU & Memory (2)

Exercice 8.7

a. (BL2+) The LOAD and STORE each take five steps (or four, if you assume that the PC may be incremented in parallel with other operations). The ADD and SUBTRACT also require five (or four) steps, IN and OUT require four (or three), SKIPs require four (or three), and JUMPs require three. Then a typical program mix requires

S = 0.25 (5 + 5) + 0.10 (5 + 5 + 4 + 4) + 0.05 (4 + 3) = 4.65 steps per instruction on average.

If the clock ticks at 10 MHz., the number of instructions executed in a second,

N = 10,000,000 / 4.65 = approximately 2.17 instructions per second.

If we assume that the PC is incremented in parallel with execution, then the result is

N = 10,000,000 / (0.25 (4 + 4) + 0.10 (4 + 4 + 3 + 3) + 0.05 (3 + 3))
= approx. 2.7 million IPS

b. (BL2+) With pipelining, each instruction is reduced by the two steps required for the fetch. (This, of course, assumes that bubbles in the pipeline can usually be avoided.) Then,

N = 10,000,000 / (0.25 (2 + 2) + 0.10 (2 + 2 + 1 +1) + 0.05 (2 + 1))
= approx. 5.7 million IPS

Although these calculations are only rough estimates of CPU performance, they illustrate clearly the substantial gains that have resulted from sophisticated modern design techniques.

b. (BL2+) The data is found in 1279 + 10 + 12 (in X) = 1301. This instruction leaves 2111 + 1322 = 3433 in A.

Exercice 8.8

(BL1) 2 billion instructions per second; approximately 6 billion instructions per second, assuming that instructions can be processed continuously, with no delays for memory access and data dependencies.

Exercice 8.9

(BL3) To see the interaction between the program and the cache memory, first suppose that the program processes the array one column at a time. Each cache line holds sixteen values. Since the cache line holds the next sixteen values in column order, all sixteen values will be used immediately, at which time, those values are completed "forever." This will be true for the entire array, so the total number of cache transfers is 160,000/16, or 10,000 transfers.

Now consider processing the array one row at a time. This situation is more difficult. As the program processes the first row, each value is found in a separate part of linear memory, and a new cache line must be loaded for each value in the row. When the 301st value is accessed, the cache is full, so it is necessary to replace a previous cache line. No matter which cache line is replaced, values in that line will be required for the next several rows, and the line will have to be reloaded. There are a number of different ways of selecting a cache line for replacement, but for simplicity, we can assume that the oldest line is replaced. If this is the case, the first 100 blocks will be replaced, in order to finish the first row. Processing the second row will then require replacing each of those lines. To do so, the *next* hundred will be eliminated. The result is a new cache line transfer for every value in the array. Although it would be possible to design a replacement algorithm that would reduce the number of transfers somewhat, the improved algorithm would be more complex, and would only apply to particular cases. This makes such an algorithm impractical and not worthwhile.

Exercice 8.11

(BL1+) The tag identifies the actual physical memory addresses corresponding to the data stored in a particular cache block. As such, the tags together act as a directory to identify the physical memory addresses for everything stored in cache memory.

Exercice 8.13

(BL1+) When a cache miss occurs, a new cache line must be moved from memory to cache before the memory access can occur. In the simplest case, there is space available in cache, and the only delay is the additional time required to access conventional memory to load the cache line. If it is necessary to write back another cache line to make space for the new line, however, there is an additional delay.for writing the old cache line to memory.

Input/Output

Exercice 9.4

(all BL1+)

a & b. The disk controller interrupts the CPU to notify it that the transfer is complete and the data ready for use. If there were no interrupt capability, the program that is using the data would have to wait long enough to assure that the data transfer is complete, in order to prevent data corruption. But how long is "long enough?"

c. When the interrupt occurs, it causes the CPU to suspend execution of the program being executed, then it saves the crucial parameters for later return to that program, and jumps to an interrupt handler program. The interrupt handler notifies the program that the data is available for use. Control is then returned to the program.

Exercice 9.14

(BL2-) When multiple interrupts occur, the first interrupt causes a suspension of the program executing at the time, storage of that program's critical parameters, and transfer of control to the program that handles the particular interrupt. When a second interrupt occurs, its priority is compared to that of the original interrupt. If its priority is higher, it takes precedence, and the original interrupt program is itself suspended. Otherwise, processing of the original interrupt continues, and the new interrupt is held until the original interrupt program is complete. When the higher priority interrupt process is completed, the lower interrupt is processed. If no further interrupts occur and if no interrupt results in suspension of all CPU processing, control eventually returns to the original program, which then resumes processing.

In general, multiple interrupts result in a queue of interrupt handler programs which will be executed in the order of the priorities associated with each interrupt. The top priority program in the queue executes unless it is replaced by an interrupt of even higher priority.