Solution to Assignment 5

1(a)(i) number of cache blocks (slots) = cache size / block size = 1M / 256 = 4K

(ii) cache address = <slot#, byte#>
number of bits in slot# = $\log_2$ (number of slots) = $\log_2$ 4K = 12
number of bits in byte# = $\log_2$ (number of bytes/slot) = $\log_2$ 256 = 8

(iii) memory address = <tag, slot#, byte#> = <block#, byte#>
number of memory blocks = memory size / block size = 2G / 256 = 8M
number of bits in block# = $\log_2$ (number of blocks) = log 8M = 23
number of bits in tag = number of bits in block# - number of bits in slot# = 11

(b)(i) 1001100_{16} = 00010000 $\underline{000000010001}$ 00000000_2
Cache line (slot)# = 11_{16}

(ii) Similarly, for 2001000_{16} , it can be buffered in cache line# 10_{16}.

(iii) S1 accesses memory locations (ebx + esi*4), starting with ebx = 1001100_{16} and esi = 0 initially and repeats 1000_{16} times, each time esi is incremented by 1. Hence it will access memory locations: 1001100, 1001104, ….., 10050FC. These are found in cache lines (whose numbers are) 11, 12, …. 50 (all in hex). Similarly, S2 accesses memory locations (edx + esi*4), starting with edx = 101000_{16} and esi = 0 initially, and repeats 1000_{16} times. Hence it will access cache lines 10, 11, …, 4F (all in hex).

(iv) The instructions are stored starting from 2001000_{16}. Assuming all four instructions can be stored in a single block (and hence these instructions do not exceed 256 bytes in length), the cache line needed to buffer this block is 10_{16}.

(c)

Instruction	Memory Access	Cache Miss
S1	fetch instruction S1 into cache slot 10	Yes*
	fetch data from [1001100] into slot 11	Yes
S2	fetch instruction S2 from cache slot 10	No
	fetch data from [101000] into cache slot 10	Yes
	write data to [101000] which is in slot 10	No
S3	fetch instruction S3 into cache slot 10	Yes*
S4	fetch instruction S4 from slot 10	No
S1 (2^{nd} iteration)	fetch instruction S1 from slot 10	No
	fetch data from [1001104] from slot 11	No
S2	fetch instruction S2 from slot 10	No
	fetch data from [101004] into cache slot 10	Yes
	write data to [101004] which is in slot 10	No
S3	fetch instruction S3 into cache slot 10	Yes*
S4	fetch instruction S4 from slot 10	No

Total number of cache misses during the first two iterations = 6

Note: there is continual competition between operand fetch from [edx + esi*4] and instruction fetch during the first 256/4 = 64 iterations of this program loop, caused by the fact that the first data block of the former is buffered in the same cache slot as the instruction block.

The instruction misses are identified with a * in the above. There are two instruction misses in the first iteration, and one instruction miss in each of the

subsequent iterations until iteration 256/4 = 64. Afterwards, the instructions will reside in cache block 10 without conflicting with other data accesses, since [edx + esi*4] afterwards will move to slot 11, 12, …etc.

(d)(i) The total number of instructions executed = $4 * 1000_{16} = 8K$

(ii) The total number of data operand accesses = $3 * 1000_{16} = 6K$

(iii) The total number of cache misses for instruction fetches = 2 + 63 = 65

(iv) In the first 256/4 = 64 iterations, operand accesses to block 10 conflicts with instruction accesses. Hence there are 63 extra data misses (excluding the first iteration that brings in the block for the first time). The total number of blocks that are brought into the cache is $40_{16}*2 = 80_{16} = 128$. Hence total number of data misses = 128+63 = 191.

(v) Hit ratio = (number of accesses – number of cache misses)/ number of accesses

r = (8K + 6K – 191) / 14K

(vi) Memory speedup
= memory access time / [cache hit time + miss ratio * miss penalty]
= 4 / [1 + (1-r)*28].

(e) Doubling the cache size without changing the other design parameters will simply increase the number of cache slots. This does not change the actual slots to buffer the program code and data in this case (of program execution), as the same subset of cache slots will be used. So the hit ratio will not be affected.

(f) Doubling the block size also doubles the number of bytes brought into a slot. As a result the number of data slots used during the execution will be halved (from 80_{16} to 40_{16}). This affects the number of data misses and hence the hit ratio.

(g) With set associative mapping with a set size of 2, there will be two slots for each set. Hence the conflicts between the instruction block and the data block from [edx + esi*4] can be avoided. This will improve the hit ratio.

(h) Registers are directly connected to the internal bus of the CPU and controlled by the control unit in a datapath operation. Cache is accessed through a cache manager that performs the address translation and subsequent access (for read/write). Logically, registers are under user program control (at program design time). So a large number of registers will place a heavy responsibility on the programmer or compiler in ‘using’ these for buffering program variables, whereas the cache is under system control (at runtime by the cache manager in using temporal/spatial locality effectively). The runtime optimization can manage a larger buffer space than the program time optimization performed by a user.

(i) Instruction fetch exhibits spatial locality: it is likely that instruction control flow is sequential, and hence instructions within a vicinity (in code space) will likely be used. Instruction fetch also exhibits temporal locality: often instructions form program loops that instructions in a loop will likely be reused in the near future.

2(a)(i) A mouse has a single ‘bit’ input (when the mouse is clicked), and is infrequent and random. A printer involves line buffer output and has a very low bandwidth (date rate) requirement, compared with the rest of peripherals (except keyboard). Typically, the line buffer is filled by the CPU under interrupt management. A disk has a high data rate and its input/output data is often buffered and transferred directly to the memory under DMA, without involving the CPU.

(ii) Programmed I/O involves pure software polling of device interfaces. As a result, the busy wait principle renders the system non-responsive to spontaneous needs arising from other devices. In a personal computer environment, the user expects spontaneous service (interactive use). As a result, it is a less appealing solution to manage interactive applications, for example, while the printer is printing, the mouse click may not respond and so on.

(iii) Certainly interrupt I/O is most suitable for managing a mouse and to some extent a printer, but not for a disk. Transfer a few bytes under CPU management at a high data rate will consume too much CPU time unnecessarily.

(iv) Similarly, DMA has no place in managing a mouse, which does not have much data at all (besides being infrequent). DMA may not be justified for printer, which is also usually slow.

(b)(i) A DMA interface competes with other potential bus masters (CPU's, other DMA interfaces) to become the next bus master when the bus becomes free, often by going through the DMAR and DMAA handshakes. If the CPU is not using the bus, then the DMA interface can use the bus without affecting/stalling the operation of the CPU. Otherwise, the CPU may be stalled from using the bus to access the memory. This effectively allows the DMA interface to 'steal' bus cycles away from the CPU.

(ii) As mentioned above, if the CPU is doing internal operations, it will not be affected by the DMA transaction on the system bus (between a DMA interface and the memory).

(iii) If a DMA interface fails to get the system bus (being delayed for too long), the buffer in the DMA will not be emptied soon enough for the DMA device to refill it. For example, a disk drive that continues to rotate may be unable to fill the buffer in time before the read/write head move away from the current byte(s). As a result, the disk transfer is affected: the drive has to wait for a complete revolution so that the missed data returns to the same position to continue the read/write transfer.

(c)(i) This problem does a simple cost-performance tradeoff analysis.
Consider 10 hours of work consisting 6 hours CPU activity and 4 hours of disk activity.
$8,000 can reduce the disk time to 4/2.5 = 1.6 hours (hence total time by 24%)
$5,000 can reduce the CPU time to 6/1.4 = 4.3 hours (hence total time by 17%)
So 60% additional cost (from CPU to disk) brings (24-17)/17 < 50% additional performance return. As a result, the more effective improvement (cost-performance together) choice is the CPU.

(ii) Without considering money, the disk improvement will bring the best performance return.

(iii) $x/5000 = 24/17 \rightarrow x = 5000*24/17$ for equal cost-effectiveness.

(d)(i) If the second interrupt is not masked, then the interrupt handler of the first interrupt (from the disk) will be interrupted in order that the second interrupt can be served.

(ii) This may be a problem if the second interrupt handler performs a function that interferes with the integrity of the disk transfer. For example, the second interrupt

handler modifies the memory content involved with the disk transfer. As a result, the disk transfer may not be 'atomically' performed.

(iii) To avoid the issues mentioned in (ii), the interrupt handler (for the disk transfer) should have disabled (masked) those interrupts that could potentially affect the correctness (atomicity) of the disk transfer.

(e)(i) Capacity of the disk drive =
#surfaces*#tracks/surface*#sectors/track*#bytes/sector = 5*1024*256*512 bytes

(ii) Average access time = average seek time + ½ revolution time
= 8 ms + ½ * 60 * 1000/7500 ms
= 8 ms + 4 ms

[Here I assume the problem indicates the average seek time is 8 ms, not the movement time from one track to the next track on a surface, otherwise, the seek time has to be multiplied by the average number of tracks traversed and this will become an unreasonably large number for the given problem specification.]

(iii) No, it is slower, because its seek time is larger, and not compensated enough by the faster rotation speed.