Question 13–1: (Solution, p 4) Describe the inputs and outputs of a (1-way) 2×4 demultiplexer, and how they relate.

Question 13–2: (Solution, p 4) In implementing HYMN's control unit, the fetch cycle and the execute cycle each consisted of two clock phases: one phase in which the clock was at 0, and one phase in which the clock was at 1. Explain why having two parts to each cycle was important in designing the control unit. In other words, why should the control unit treat the two phases of each cycle differently?

Question 14.2–1: (Solution, p 4) In terms of their physical characteristics, what distinguishes an L1 cache from an L2 cache?

Question 14.2–2: (Solution, p 4) Suppose we have a system using six-bit addresses which uses a direct-mapped cache with two lines, where each line has two bytes. And suppose the following sequence of accesses of one-byte accesses: M[10], M[2], M[1], M[3], M[10], M[2].

a. Which of the accesses in the sequence hit the cache?

b. Draw a picture of the contents of the cache after completing this sequence.

Question 14.2–3: (Solution, p 4) Compare the virtues of a direct-mapped cache versus a fully associative cache.

Question 14.3–1: (Solution, p 4) Consider the following C code to add the numbers within an array.

```
int i;
int sum = 0;
for(i = 0; i < n; i++) {
    sum += a[i];
}
```

Note that this program never accesses the same array element twice. Explain how a direct-mapped memory cache helps this code run faster.

Question 14.3–2: (Solution, p 4) Consider the following C code to multiply two matrices.

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        for(int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Note that the innermost loop here iterates over `k`.
Suppose that our CPU has a direct-mapped cache with 128 lines, each of 64 bytes, and suppose that each array element takes 8 bytes. On average, how many times per iteration through the innermost loop will the CPU miss the cache in accessing each of the following?

a. `C[i][j]`

b. `A[i][k]`

c. `B[k][j]`

Question 14.3–3: (Solution, p 4) Consider the following C code fragment.

```
for(i = 0; i < n; i++) {
    j = a[i];
    sum += count[j];
}
```

Suppose that the CPU uses a direct-mapped cache in which each line holds four array entries. For each iteration of this loop, how many times, on average, will the CPU miss the cache? Explain your answer. (Assume that the array `a` holds random integers between 0 and 1,000,000.)

Question 9.1–1: (Solution, p 5) Define the *relocation problem* that arises in the context of segments.

Question 9.1–2: (Solution, p 5) Suppose we have an operating system that implements load-time relocation to place the program in available memory. Describe how the operating system starts a program in detail.

Question 9.1–3: (Solution, p 5) Explain how a CPU incorporating segment registers computes a memory address.

Question 9.1–4: (Solution, p 5) In class we saw three different ways of enabling a process to run in memory, including the "load-time technique," when the process's memory references are repaired at the time the process is loaded into memory, and the "run-time technique," where the process's memory references are repaired by the CPU at the time the memory is referenced. Name and explain one advantage of the run-time technique over the load-time technique.

Question 9.2–1: (Solution, p 6)

- **a.** Explain how a CPU supporting virtual memory would translate a virtual memory address into the actual memory address, assuming that the requested page is in memory.
- **b.** Explain what the CPU does when the requested page is not in memory.

Question 9.2–2: (Solution, p 6) Describe at least three elements of a typical page table entry.

Question 9.2–3: (Solution, p 6) Suppose we were using a virtual memory system with three page frames, using the FIFO algorithm to decide which frames to include in memory. At right, give the page located in each frame after each of the given page references.

page	frame		
ref	1	2	3
1	1		
2	1	2	
3	1	2	3
4			
2			
5			
4			
2			
1			

Question 9.2–4: (Solution, p 6) Suppose we were using a virtual memory system with three page frames, using the Clock algorithm to decide which frames to include in memory. Give the page located in each frame after each of the given page references.

page ref	frame 1	2	3
1	1		
2	1	2	
3	1	2	3
4			
2			
5			
4			
2			
6			

Question 9.2–5: (Solution, p 7) Explain the problem that leads to including the TLB in computing systems supporting virtual memory.

Question 9.2–6: (Solution, p 7) Explain the *concept* and *purpose* of page directories.

Question 9.2–7: (Solution, p 7) What advantages does a virtual memory system provide? Give at least two reasons.

Solution 13–1: (Question, p 1) A 2×4 demultiplexer has three inputs, including a data input and two select inputs representing the two bits of a number s. It has four outputs, number $00_{(2)}$ through $11_{(2)}$. The demultiplexer routes its data input to the output whose number is s; the other outputs' values will be 0.

Solution 13–2: (Question, p 1) The distinction between the two phases is important because of timing considerations. During the 0 phase of each cycle, the computer computes the values for the cycle; this is a matter of letting values propagate through the circuit to the input pins of the registers and memory where the values should be stored. During the 1 phase, these values continue to be propagated through to the registers and memory that should change, while the computer sends signals to the registers and memory that they should change.

Solution 14.2–1: (Question, p 1) The L1 cache is incorporated into the CPU chip itself, while the L2 cache is on a separate chip. (A secondary distinguishing factor is the L2 cache's larger size.)

Solution 14.2–2: (Question, p 1)

a. the fourth (M[3]) access only.

b.

line num	tag	line data
0	0000	m[0] m[1]
1	0000	m[2] m[3]

Solution 14.2–3: (Question, p 1) With a direct-mapped cache, in which every memory address can be cached in only one location within the cache, there is the possibility that an access to a distant memory address would eject a very recent access that will likely be used soon in the future. A fully associative cache doesn't share this occassional poor behavior, but it requires more logic per cache line, so that large and fast fully associative cache are unaffordable.

Solution 14.3–1: (Question, p 1) This code goes through the array in order, and several array elements will map to a single line in the cache. When the code misses the cache, the cache will load the entire line containing the desired array entry, including the next several adjacent memory locations. Thus, the next several iterations of the loop would hit the cache, saving a memory access each of these times.

Solution 14.3–2: (Question, p 1)

a. 0 (This is the same memory location each time through the innermost loop, so there will be no misses after the first iteration.)

b. 0.125 (As we go through the innermost loop, we are walking through successive memory locations. Each time we miss the cache, we will load eight successive locations in the row; thus this access misses the cache one time in eight.)

c. 1.0 (Each array access in in a different row, so they will be widely separated. Loading one element in the column will not help in reaching another. Thus each access to B will be a miss.)

Solution 14.3–3: (Question, p 2) 1.25. This loop goes through the `a` array in succession, and so when we load one element of `a` into the cache, the CPU will load the next three elements into the cache also. Thus, this code misses the cache when accessing `a[i]` an average of 0.25 each iteration. The pattern of accesses to `count`, however, is random, and it is highly unlikely that any access would be to an element already in the cache. Thus, the CPU is likely to miss the cache in accessing `count[j]` each time through the loop. We add these two numbers together (0.25 + 1.00) to get the total number of misses.

Solution 9.1–1: (Question, p 2) In an operating system supporting segments, the operating system chooses which part of memory the program will occupy when the program starts. When a program is loaded into memory, the program must contain references to specific memory addresses within the program's region of memory, but the executable file cannot specify these addresses directly because the compiler generates it without knowing where the operating system may choose to place the program. (Indeed, the operating system may be executing the same program in different regions of memory at the same time.) The *relocation problem* refers to the problem of adapting these memory addresses to the region of memory chosen by the operating system for the loaded program.

Solution 9.1–2: (Question, p 2) First the OS allocates a segment of memory to the program, sufficient to hold all the code and data of the program. Then it reads the memory image from the executable file, loading it into memory. This memory image, however, has all memory references placed assuming that the segment begins at address 0; thus, the OS must fix this by reading the offset of each memory address reference from the executable file and adding the segment's beginning address into the number at that offset in the loaded memory image. This repairs all the memory references to refer to actual memory address within the process's segment.

Solution 9.1–3: (Question, p 2) For each memory reference, the CPU adds the contents of the segment register to the requested address to get the actual address of the data. This actual address is what is sent to memory to fetch data. (The idea is that memory references will be offset by x as the instruction is executing, instead of doing it at the time that the instruction is placed in memory.)

Solution 9.1–4: (Question, p 2) There are several answers to this question.

- The run-time technique meshes nicely with restricting a process's memory accesses to that process's segment. This memory protection must be accomplished in hardware, because the performance hit if the OS had to check each memory access would be prohibitive. Essentially this circuitry already involves a segment register telling where the current process's memory begins. Thus, given that memory protection is a necessity in a modern computer, the run-time technique is not as complicated as the load-time technique.
- In the load-time technique, once the program is loaded, it cannot be moved to another location in memory, because the program may create references to other memory locations within the process segment, and these references are indistinguishible from other numbers. With the run-time technique, such movement is possible, because the only thing that would need to change with the program is the contents of the segment register.
- The load-time technique can significantly slow the time it takes to start a program, because the OS must "repair" all the links after it has loaded the image into memory. In contrast, the run-time technique can be efficiently implemented in hardware, so that there is no penalty during run-time for that technique.
- The load-time technique can make the executable file slightly larger, because it requires an additional section of the executable file enumerating the address references within the memory image. (This is a very minor sadvantage.)

Solution 9.2–1: (Question, p 2)

a. It splits the virtual memory address into two parts, a page index and an offset. The CPU looks into the page table, stored in RAM, to get the page table entry for the specified page index. This page table entry tells which page frame contains the page, and the CPU looks into this page frame at the offset to access the requested data.

b. It raises a page fault, which transfers control to the operating system's exception handler. The OS will then presumably load the requested page into some page frame and return from the exception handler. On return, the CPU will try the memory access again.

Solution 9.2–2: (Question, p 2) Here are four:

- The valid bit, saying whether the page is currently in memory.
- The location of the page in memory.
- The dirty bit, saying whether the page in memory has been changed.
- The referenced bit, saying whether the page in memory has been accessed recently.

Solution 9.2–3: (Question, p 2)

page ref	frame 1	2	3
1	1		
2	1	2	
3	1	2	3
4	4	2	3
2	4	2	3
5	4	5	3
4	4	5	3
2	4	5	2
1	1	5	2

Solution 9.2–4: (Question, p 3)

page ref	frame 1	2	3
1	1		
2	1	2	
3	1	2	3
4	4	2	3
2	4	2	3
5	4	2	5
4	4	2	5
2	4	2	5
6	6	2	5

Solution 9.2–5: (Question, p 3) A straightforward implementation of virtual memory places the page table in memory. Unfortunately, this means that each attempt to access a memory location actually requires two memory accesses — one to look up the page table entry in virtual memory, and one to access the requested memory within the page frame given by that entry. This effectively halves the time to access each page, when compared to a system that does not use virtual memory.
Designers reduce this problem dramatically by caching a small number of frequently-used page table entries on the CPU chip, in a portion of the chip called the *translation lookaside buffer*, or TLB. Since accessing data stored on the CPU chip is much faster than accessing data in memory, the TLB can dramatically reduce the time for looking up page table entries, restoring a access speed more comparable to a system without virtual memory.

Solution 9.2–6: (Question, p 3) A page directory is a table of pointers to page tables for regions of virtual memory. A CPU uses the page directory instead of a single page table, because a single page table can become very large and it must be in memory in order to allow for address translation. With a page directory, the partial page tables can be paged in virtual memory. (The page directory must occupy RAM for address translation, but the page directory would be much smaller than a single exhaustive page table.) Moreover, page tables for unused regions of memory need not occupy even virtual memory.

Solution 9.2–7: (Question, p 3)

- It provides a larger range of memory address beyond what actual memory the computer holds.
- It allows each process to have its own address space.
- It enables two processes to share the same memory easily.
- It enables memory to be moved between addresses rapidly, since altering the paging table moves the page without any actual memory copying going on.