1. Propose a way to **modify or augment** a cache to **substantially reduce** the miss rate for each of the following kinds of cache misses:

   - Compulsory misses (misses from the first access to some data); and
   - Conflict misses (misses from replacing some data with other data that had to be stored in the same part of the cache as other data, even though there was plenty of capacity in the cache overall)

   Your proposals must not substantially increase the storage required for the cache. For example, if a cache stores around 10000 bytes, your proposal should not require adding more than 128 bytes of storage to the cache.

   (a) Compulsory misses.

   (b) Conflict misses.

2. For each of workloads listed below, is it likely to benefit substantially from simultaneous multithreading (also known as hyperthreading) on an out-of-order processor? Briefly explain your answer.

   (a) A multithreaded webserver that spends most of its time performing I/O or reading requests and responses to and from memory.

   (b) A multithreaded, compute-bound linear-algebra application.

   (c) A compute-bound linear algebra application which experiences a low cache miss rate running on the same core as a breadth-first search of a graph with a high cache miss rate.

3. For each of the following statements, identify whether it is true for VLIW (very long instruction word) processors, superscalar (out-of-order, multiple issue) processors, both, or neither:

   (a) These processors are likely to require compilers to know about instruction latencies to produce **correct** code.
   ◯ VLIW ◯ superscalar ◯ both ◯ neither

   (b) These processors can effectively used pipelined functional units.
   ◯ VLIW ◯ superscalar ◯ both ◯ neither

   (c) These processors can achieve a CPI (cycles per instruction) of less than one.
   ◯ VLIW ◯ superscalar ◯ both ◯ neither

   (d) A processor of this type can easily use an instruction set designed for an MIPS-like in-order pipelined processor.
   ◯ VLIW ◯ superscalar ◯ both ◯ neither

   (e) A processor of this type is likely to use a technique like register renaming to help identify inter-instruction dependencies.
   ◯ VLIW ◯ superscalar ◯ both ◯ neither

4. Suppose we have an in-order, pipelined processor with the following stages:

    1. Fetch (instruction cache)
    2. Decode and register read
    3. Execute, part 1
    4. Execute, part 2
    5. Memory (data cache), part 1
    6. Memory (data cache), part 2
    7. Writeback (register file write)

    Suppose we execute a program with the following mix of executed instructions:

    - 75% register-to-register arithmetic operations
    - 15% loads
    - 5% stores
    - 5% branches

    (a) Suppose the in-order processor does not do branch prediction, but instead stalls in the fetch stage after every branch until the branch instruction finishes both execute stages. **Assuming there are no other reasons for pipeline stalls**, what number of cycles per instruction will we observe? Show your work.

(b) Suppose we add simple branch prediction to the in-order processor and it correctly predicts 80% of branches. **Assuming there are no other reasons for pipeline stalls**, what number of cycles per instruction will we observe? Show your work.

(c) Suppose two-thirds of the load instructions in the program are immediately followed by a use of the resulting value as R1 in:

```
R1 ← M[R2]
R3 ← R1 + R8
```

Assuming there are no other reasons for pipeline stalls, what number of cycles per instruction will we observe? Show your work.

(d) Rewrite the following assembly snippet to execute in the minimum number of cycles on the processor. Assume **all possible forwarding paths** are implemented on the processor. M[x] represents an access to memory at address x. Each line represents a single instruction. Single letter variables represent registers.

```
X <– A * B
X <– X * C
Y <– D * E
Y <– Y * F
Z <– M[Q]
Z <– Z * X
Z <– Z * Y
M[Q] <– Z
```

5. Consider the following snippet of assembly for an out-of-order processor. `M[`x`]` represents accessing memory at address x.

```
R1 <— R2 + R3     # (A)
R5 <— M[R1]       # (B)
R6 <— R2 + R4     # (C)
R7 <— M[R4]       # (D)
R8 <— R3 * R4     # (E)
R9 <— R7 * R6     # (F)
M[R5] <— R10      # (G)
```

Suppose a page fault occurs during instruction D and the reorder buffer contains the following:

entry #	instr.	ready?	logical register	renamed register	prev. renamed	store?	exception?
0	A	yes	R1	X9	X1	no	no
1	B	yes	R5	X10	X5	no	no
2	C	no	R6	X11	X6	no	no
3	D	no	R7	X12	X7	no	page fault
4	E	yes	R8	X13	X8	no	no
5	F	yes	R9	X14	X9	no	no
6	G	yes	—	—	—	yes	no

(Similar to the MIPS R10000 and gem5's simulated processor, the processor uses a large physical register file and register renaming rather than storing results directly in the reorder buffer. Logical registers indicate the register specified in the instruction; renamed registers represent the physical register chosen by register renaming. "Prev. renamed" indicates the physical register assigned to the destination logical register before the instruction was renamed.)

(a) What, if any, instructions which are not ready *must* be executed before the page fault handler can be started?

(b) What will be done with the result computed by instruction E for logical register R8? Will the value of physical register X13 change because of this?

(c) What will be done with the result computed by instruction B for R5? Will the value of physical register X10 change because of this?

(d) Has memory been written by instruction G? Explain briefly.

6. (This question modified 2 December 2016 to add a third CPU.) Consider a three-processor system with bidirectional point-to-point links connecting the processors. Each processor has a **write-back** cache and has its own memory controller that talks to a local memory. Assume the following timings:

   - sending a message on processor-to-processor link takes 100 cycles (regardless of length), and contention for the link is negligible
   - writing a value to a local memory takes 50 cycles
   - writing a value to a remote memory takes 150 cycles (the time to send a message to the remote processor, plus the time to write the value)
   - processing a message to send a reply takes negligible time, even if this requires updating the local cache

   Assume directories are managed by the cache of the processor local to that memory.

   (a) Consider a write by processor A to a remote memory location if:

   - processor A does not have the value in its cache
   - processor B does have the value in its cache
   - processor C does have the value in its cache
   - the value in processor B's cache is the same as memory
   - the value in processor C's cache is the same as memory

   How long should this write take with a broadcast-based cache coherency protocol? Show your work.

   (b) How long should the write in the previous part take with a **directory-based** cache coherency protocol? Show your work.

(c) How long should the write in the previous part take with a **directory-based** cache coherency protocol if instead processor A, processor B, and processor C do **not** have the value in its cache? Show your work.

(This question added 2 Dec 2016; updated 39 to 37 in table on 5 December) Consider the following C function: (`char` is a one-byte type.)

```
char hugeArray[1024 * 1024 * 1024];
int testStride(int size, int stride) {
    for (int iteration = 0; iteration < 10; ++iteration) {
        for (int i = 0; i < size; i += stride) {
            hugeArray[i] += 1;
        }
    }
}
```

Suppose we execute this function with varying parameters and an initially empty single-level cache and TLB and observe the following results:

size	stride	TLB misses (`hugeArray` only)	cache misses (`hugeArray` only)
$2 \cdot 4096$	4096	2	2
$3 \cdot 4096$	4096	3	3
$4 \cdot 4096$	4096	4	4
$5 \cdot 4096$	4096	5	5
$5 \cdot 4096$	2048	5	10
$5 \cdot 4096$	1024	5	20
$6 \cdot 4096$	4096	6	60
$8 \cdot 4096$	4096	8	80
$9 \cdot 4096$	4096	37	90
$9 \cdot 4096$	$2 \cdot 4096$	32	5
$9 \cdot 4096$	$4 \cdot 4096$	30	3
$9 \cdot 4096$	$8 \cdot 4096$	2	2

Assume the following:

- the system uses 4 kilobyte (4096 byte) pages
- TLB and cache accesses other than to `hugeArray` have **no effect** on the number of misses for `hugeArray` accesses
- the system has a single-level page table
- the TLB and cache use a least-recently used replacement policy
- consecutive pages in virtual memory are mapped to consecutive pages in physical memory

(a) What is the associativity of the TLB? Show your work.

(b) How many page table entries can be stored in the TLB? Show your work.

(c) What is the associativity of the cache? Show your work.

(d) What is the size of the cache? (Do not include metadata like valid bits.) Show your work.

7. (This question added 2 December 2016) Consider the following snippet of assembly for an out-of-order processor. `M[`x`]` represents accessing memory at address x.

```
R1 <— R2 + R3     # (A)
R5 <— M[R1]       # (B)
R6 <— R2 + R4     # (C)
R7 <— M[R4]       # (D)
R8 <— R3 * R4     # (E)
R9 <— R7 * R6     # (F)
M[R7] <— R10      # (G)
```

Suppose the processor has the following functional units:

- an integer ALU that is can do additions and multiplies in **two cycles**, but which is fully pipelined to issue one operation per cycle
- a load/store unit that can accept one load or store per cycle

Given these functional units, how long will the processor take to execute the instructions above? Show your work.

8. (a) (This question added 2 December 2016) Briefly explain when spin-locks, as commonly implemented using a 'test-and-set' instruction, would likely be preferable to ticket locks or queue based locks implemented primarily using an atomic increment instruction.