1. [40 points] Each of the tables below shows a sequence of memory operations affecting a single cache block, where "Rx" means a read from processor x, and "Wx" means a write from processor x. There are three caches in the system: C0, C1, C2. Record the state of the block in all caches after the processor's request is completed. A state of "-" means that the block is not in the cache. (This is only used in the Dragon protocol; other protocols use "I" to indicate both invalidated as well as not present.)

   Each table specifies the **coherence protocol** to be used. For each table, use the following latencies for different types of transaction. Assume an atomic bus, in which only one transaction is active at a time, and each transaction completes before the next one begins. Even cache hits will be handled sequentially this problem. Add up the total latency for the sequence in each table.

Action	**Latency (cycles)**
Cache hit	0
Bus action with no data	20
Bus action with one word of data	20
Bus action with block transfer from another cache	50
Bus action with block transfer from memory	100

   For a bus transaction that requires both a request and a response, specify both (e.g., BusRd + MemData). If there is no bus transaction, write None. Use cache-to-cache transfers if protocol allows.

   Bus requests: BusRd, BusRdX, BusUpgr, BusUpd, BusWr (for write-through)

   Bus responses: Flush, FlushOpt, MemData

MESI Protocol

Operation	C0 state	C1 state	C2 state	Bus Transaction	Latency
-	I	I	I	-	-
R1	I	E	I	BusRd, MemData	100
W1	I	M	I	None	0
R0	S	S	I	BusRd, Flush	50
R2	S	S	S	BusRd, FlushOpt	50
W1	I	M	I	BusUpgr	20
W2	I	I	M	BusRdX, Flush	50
R0	S	I	S	BusRd, Flush	50
R1	S	S	S	BusRd, FlushOpt	50
R0	S	S	S	None	0
R2	S	S	S	None	0
W0	M	I	I	BusUpgr	20
W2	I	I	M	BusRdX, Flush	50
R2	I	I	M	None	0
W1	I	M	I	BusRdX, Flush	50
R0	S	S	I	BusRd, Flush	50
				Total Latency	540

Write-through protocol

Operation	C0 state	C1 state	C2 state	Bus Transaction	Latency
-	I	I	I	-	-
R1	I	V	I	BusRd, MemData	100
W1	I	V	I	BusWr	50
R0	V	V	I	BusRd, MemData	100
R2	V	V	V	BusRd, MemData	100
W1	I	V	I	BusWr	50
W2	I	I	I	BusWr	50
R0	V	I	I	BusRd, MemData	100
R1	V	V	I	BusRd, MemData	100
R0	V	V	I	None	0
R2	V	V	V	BusRd, MemData	100
W0	V	I	I	BusWr	50
W2	I	I	I	BusWr	50
R2	I	I	V	BusRd, MemData	100
W1	I	I	I	BusWr	50
R0	V	I	I	BusRd, MemData	100
				Total Latency	1100

Dragon protocol

Operation	C0 state	C1 state	C2 state	Bus Transaction	Latency
-	-	-	-	-	-
R1	-	E	-	BusRd(!C), MemData	100
W1	-	M	-	None	0
R0	Sc	Sm	-	BusRd(C), Flush	50
R2	Sc	Sm	Sc	BusRd(C), Flush	50
W1	Sc	Sm	Sc	BusUpd(C)	20
W2	Sc	Sc	Sm	BusUpd(C)	20
R0	Sc	Sc	Sm	None	0
R1	Sc	Sc	Sm	None	0
R0	Sc	Sc	Sm	None	0
R2	Sc	Sc	Sm	None	0
W0	Sm	Sc	Sc	BusUpd(C)	20
W2	Sc	Sc	Sm	BusUpd(C)	20
R2	Sc	Sc	Sm	None	0
W1	Sc	Sm	Sc	BusUpd(C)	20
R0	Sc	Sm	Sc	None	0
				Total Latency	300

2. [35 points] Given a single variable used to implement a mutual exclusion lock, compare the bus traffic required for a TT&S lock and an LL/SC lock in a four-processor system. (A value of 0 means that the lock is available, and 1 means the lock is taken.)

   For each action in the sequence below, write the bus transaction required (BusRd, BusRdX, BusUpgr, or None -- just the request, not the response), and the state of the block in each cache. The MESI protocol is used for coherence.

   TT&S lock

Action	**C0 state**	**C1 state**	**C2 state**	**C3 state**	**Bus Transaction**
Initially, P0 holds the lock	M	I	I	I	-
P1 reads the lock	S	S	I	I	BusRd,Flush
P2 reads the lock	S	S	S	I	BusRd,FlushOpt
P3 reads the lock	S	S	S	S	BusRd,FlushOpt
P0 releases the lock	M	I	I	I	BusUpgr
P3 reads the lock	S	I	I	S	BusRd,Flush
P3 acquires the lock	I	I	I	M	BusUpgr
P3 releases the lock	I	I	I	M	None
P2 reads the lock	I	I	S	S	BusRd,Flush
P1 reads the lock	I	S	S	S	BusRd,FlushOpt
P0 reads the lock	S	S	S	S	BusRd,FlushOpt
P0 acquires the lock	M	I	I	I	BusUpgr
P1 tries to acquire the lock (fails)	S	S	I	I	BusRd,Flush
P2 tries to acquire the lock (fails)	S	S	S	I	BusRd,FlushOpt
P3 reads the lock	S	S	S	S	BusRd,FlushOpt
P0 releases the lock	M	I	I	I	BusUpgr
P1 reads the lock	S	S	I	I	BusRd,Flush
P3 reads the lock	S	S	I	S	BusRd,FlushOpt
P1 acquires the lock	I	M	I	I	BusUpgr
P3 tries to acquire the lock (fails)	I	S	I	S	BusRd,Flush
				Total number of bus transactions	31

LL/SC lock

Action	**C0 state**	**C1 state**	**C2 state**	**C3 state**	**Bus Transaction**
Initially, P0 holds the lock	M	I	I	I	-
P1 reads the lock	S	S	I	I	BusRd,Flush
P2 reads the lock	S	S	S	I	BusRd,FlushOpt
P3 reads the lock	S	S	S	S	BusRd,FlushOpt
P0 releases the lock	M	I	I	I	BusUpgr
P3 reads the lock	S	I	I	S	BusRd,Flush
P3 acquires the lock	I	I	I	M	BusUpgr
P3 releases the lock	I	I	I	M	None
P2 reads the lock	I	I	S	S	BusRd, Flush
P1 reads the lock	I	S	S	S	BusRd,FlushOpt
P0 reads the lock	S	S	S	S	BusRd,FlushOpt
P0 acquires the lock	M	I	I	I	BusUpgr
P1 tries to acquire the lock (fails)	M	I	I	I	None
P2 tries to acquire the lock (fails)	M	I	I	I	None
P3 reads the lock	S	I	I	S	BusRd,Flush
P0 releases the lock	M	I	I	I	BusUpgr
P1 reads the lock	S	S	I	I	BusRd,Flush
P3 reads the lock	S	S	I	S	BusRd,FlushOpt
P1 acquires the lock	I	M	I	I	BusUpgr
P3 tries to acquire the lock (fails)	I	M	I	I	None
				Total number of bus transactions	25

3. [15 points] For the code example below, all variables are shared and **all values are initially zero**. The outcome is the value stored in each variable *after all of the code has completed execution*. For each possible outcome listed in the table, indicate whether that outcome is legal under sequential consistency. If legal, list an order of operations that produce the outcome. (There may be more than one such order -- just list one.)

   Note: SC specifies that memory operations occur in program order, but that does not mean all of the memory ops in a statement are atomic. When there's more than one operation in a statement use a format like *S1.Ra* or *S2.Wc* to indicate a read or write to a specific variable.

Thread 0:	Thread 1:

`S1: a = 1;`
`S2: c = d + b;`
	`S3: if (a > 0) {`
	`S4:    d = 4;`
	`     }`
	`S5: b = a + 1;`

Outcome				Legal under SC?	If legal, provide a possible ordering of operations.
a	**b**	**c**	**d**		
1	2	6	4	Y	S1, S3, S4, S5, S2
1	2	0	0	Y	S3, S4, S1, S2, S5
1	2	0	4	Y	S1, S2, S3, S4, S5
1	1	5	4	N	
1	2	4	4	Y	S1, S3, S4, S2, S5
1	1	1	0	Y	S3, S4, S5, S1, S2

4. [10 points] Use LL/SC instructions to construct an atomic compare and swap instruction "CAS R1, R2, L" which tests whether data in memory location L is equal to that in R1. If they are equal, write the value in R2 to L, and copy R1 to R2. Otherwise, nothing is done and CAS returns.

   NOTE: This is not the same as acquiring a lock; if the CAS does not succeed, do not re-execute it.

   For example, if initially R1=5, R2=10, L=5, after CAS we have R1=5, R2=5, and L=10. If initially R1=7, R2=10, L=5, nothing changes after CAS.

   Show your answer in assembly code, and annotate what each instruction does. (The format of the assembly code is not critical; use the code shown in the text as a pattern, but as long as you comment each line and the code is reasonable, it will be fine.) Keep as few instructions between LL and SC as possible.

```
cas:    LL  R3, &casvar    // read casvar to R3, LinkReg = &casvar    cmp R3, R1          // compare R3 and R1
```