5.1

Cx.y is cache line y in core x.

a. C0: R AC20 → C0.0: (S, AC20, 0020), returns 0020

b. C0: W AC20 80 → C0.0: (M, AC20, 0080)
C3.0: (I, AC20, 0020)

c. C3: W AC20 80 → C3.0: (M, AC20, 0080)

d. C1: R AC10 → C0.2: (S, AC10, 0030)
M: AC10, 0030 (write-back to memory)
C1.2: (S, AC10, 0030)

e. C0: W AC08 48 → C0.1 (M, AC08, 0048)
C3.1: (I, AC08, 0008)

f. C0: W AC30 78 → C0.2: (M, AC30, 0078)
M: AC10 0030 (write-back to memory)

g. C3: W AC30 78 → C3.2 :(M, AC30, 0078)

5.2

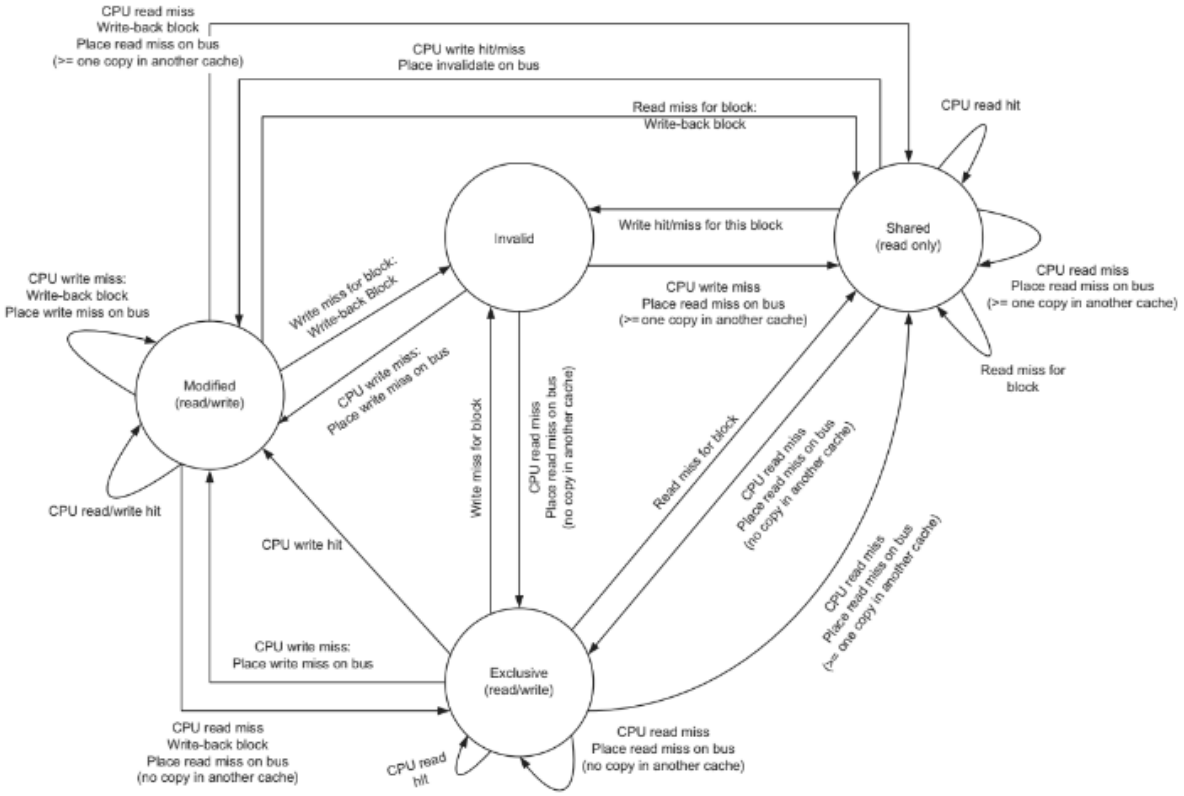
a. C0: R AC20 Read miss, satisfied by memory
C0: R AC28 Read miss, satisfied by C1's cache
C0: R AC20 Read miss, satisfied by memory, write-back 110
Implementation 1: 100 + 40+ 10 + 100 + 10 = 260 stall cycles
Implementation 2: 100 + 130 + 10 + 100 + 10 = 350 stall cycles

b. C0: R AC00 Read miss, satisfied by memory
C0: W AC08 ← 48 Write hit, sends invalidate
C0: W AC20 ← 78 Write miss, satisfied by memory, write back 110
Implementation 1: 100 + 15 + 10 + 100 = 225 stall cycles
Implementation 2: 100 + 15 + 10 + 100 = 225 stall cycles

c. C1: R AC20 Read miss, satisfied by memory
C1: R AC28 Read hit
C1: R AC20 Read miss, satisfied by memory
Implementation 1: 100 + 0 + 100 = 200 stall cycles
Implementation 2: 100 + 0 + 100 = 200 stall cycles

d. C1: R AC00 Read miss, satisfied by memory
C1: W AC08 ← 48 Write miss, satisfied by memory, write back AC28, sends invalidate
C1: W AC20 ← 78 Write miss, satisfied by memory
Implementation 1: 100 + 100 + 10 + 100 + 15 = 325 stall cycles
Implementation 2: 100 + 100 + 10 + 100 + 15 = 325 stall cycles

5.3



5.4

a.

C0: R AC00, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C0: W AC00 ← 40 MSI: send invalidate, MESI: silent transition from E to M
MSI: 100 + 15 = 115 stall cycles
MESI: 100 + 0 = 100 stall cycles

b.

C0: R AC20, Read miss, satisfied in memory, sharers both to S
C0: W AC20 ← 60 both send invalidates
Both: 100 + 15 = 115 stall cycles

c.

C0: R AC00, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C0: R AC20, Read miss, memory, silently replace 120 from S or E
Both: 100 + 100 = 200 stall cycles, silent replacement from E

d.

C0: R AC00, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C1: W AC00 ← 60, Write miss, satisfied in memory regardless of protocol
Both: 100 + 100 = 200 stall cycles, don't supply data in E state (some protocols do)

e.

C0: R AC00, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C0: W AC00 ← 60, MSI: send invalidate, MESI: silent transition from E to M
C1: W AC00 ← 40, Write miss, C0's cache, write-back data to memory
MSI: 100 + 15 + 40 + 10 = 165 stall cycles
MESI: 100 + 0 + 40 + 10 = 150 stall cycles

5.5

Loop 1

Repeat i: 1 .. n
    A[i] ← A[i-1] +B[i];

Loop2

Repeat i: 1 .. n
    A[i] ← A[i] +B[i];

If A, B, are larger than the cache and n is large enough, the hit/miss pattern (running on one CPU) for both loops for **large values of i** is shown in the table (hit times ignored).

Cache/ memory accesses	Loop1							Loop2						
	A[i]		A[i-1]		B[i]		Total	A[i]		A[i]		B[i]		Total
No coherence protocol	Write miss + writeback	110 cycles	Read hit	–	Read miss	100 cycles	210 cycles	Write hit	–	Read miss + writeback	110 cycles	Read miss	110 cycles	220 cycles
MESI	Write miss + writeback	110 cycles	Read hit	–	Read miss	100 cycles	210 cycles	Write hit	–	Read miss + writeback	110 cycles	Read miss	110 cycles	220 cycles
MSI	Write miss + writeback	110 cycles	Read hit	–	Read miss	100 cycles	210 cycles	Write hit + invalidate	15	Read miss + writeback	110 cycles	Read miss	110 cycles	235 cycles

When the cache line is large enough to contain multiple elements—M, the average cost of the memory accesses (ignoring cache hits) will be divided by M. When hits and non-memory accessing instructions are considered, the relative performance of Loop1 and Loop2 will get closer to 1.

5.9

a. i. C3:R, M4

Messages:

- Read miss request message from C3 to Dir4 (011 → 010 → 000 → 100)
- Read response (data) message from M4 to C3 (100 → 101 → 111 → 011)

C3 cache line 0: <I, x, x, …..> → <S, 4, 4, …..>
Dir4: <I, 00000000> → <S, 00001000>, M4 = 4444…..

---

ii. C3:R, M2

Messages:

- Read miss request message from C3 to Dir2 (011 → 010)
- Read response (data) message from M2 to C3 (010 → 011)

C3 cache line 0: <S, 4, 4, …..> → <S, 2, 2, …..>
C2 Dir: <I, 00000000> → <S, 00001000>, M4 = 4444…..
Note that Dir4 still assumes C3 is holding M4 because C3 did not notify it that it replaced line 0. C3 informing Dir4 of the replacement can be a useful upgrade to the protocol.

---

iii. C7: W, M4 ← 0xaaaa

Messages:

- Write miss request message from C7 to M4 (111 → 110 → 100)
- Invalidate message from Dir4 to (100 → 101 → 111 → 011)
- Acknowledge message from C3 to Dir4 (011 → 010 → 000 → 100)
- Acknowledge message from Dir4 to C7 (100 → 101 → 111)

C3 cache line 0: <S, 4, 4, …..> → <I, x, x, …..>
C7 cache line 0: <I, x, x, …..> → <M, aaaa, …..>
Dir4: <S, 00001000> → <M, 10000000>, M4 = 4444…..

---

iv. C1: W, M4 ← 0xbbbb

Messages:

- Write miss request message from C1 to M4 (001 → 000 → 100)
- Invalidate message from Dir4 to C7 (100 → 101 → 111)
- Acknowledge message (with data write-back) from C7 to Dir4 (111 → 110 → 100)
- Write Response (data) message from Dir4 to C1 (100 → 101 → 001)

C7 cache line 0: <M, aaaa, …..> → <I, x, x, …..>
C1 cache line 0: <I, x, x, …..> → <M, bbbb, …..>
Dir4: <M, 10000000> → <M, 00000001> M4 = aaaa…..

Example message formats:

No data message: <no data message flag, message type, destination (dir/cache, & number), block/line number>

Data message: <data message flag, message type, destination (dir/cache, & number), block/line number, data>

(b), (c) Same analysis like (a)

5.11

a. C1: W, M4 ← 0xbbbb C3: R, M4 C7: R, M2
C3: W, M4 ← 0xaaaa

It should be noted that since both C1 and C3 are accessing M4, either of them can get access first and the behavior is non-deterministic (only decided by implementation delays). In this question we will assume simple delays—based on the Hamming distance between source and destination addresses. Hence, C1 will win over C3!
So the ordering of the transactions on M4 is

C1: W, M4 ← 0xbbbb (wins) → C3: R, M4 → C3: W, M4 (serialization on C3)

The transaction on M2 is independent of the 3 above transactions.

M4 transactions

(a) C1: W, M4 ← 0xbbbb

- Write miss request message from C1 to M4 (001 → 000 → 100)
- Write Response (data) message from Dir4 to C1 (100 → 101 → 001)

C1 cache line 0: <I, x, x, …..> → <M, bbbb, …..>
Dir4: <I, 00000000> →<M, 00000001> M4 = 4444…..

(b) C3: R, M4

- Read miss request message from C3 to Dir4 (011 → 010 → 000 → 100)
- Request data message (and move to shared state) from M4 to C1 (100 → 101 → 001)
- Write response (data) message from C1 to Dir4 (001 → 101 → 100)
- Read response (data) message from Dir4 to C3 (100 → 101 → 111 → 011)

C1 cache line 0: <M, bbbb, …..> → <S, bbbb, …..>
C3 cache line 0: <I, x, x, …..> → <S, bbbb, …..>
Dir4: <M, 00000001> →<S, 00001010>, M4 = bbbb…..

(c) C3: W, M4 ← 0xaaaa

- Write hit request message from C3 to M4 (011 → 010 → 000 → 100)
- Invalidate message from Dir4 to C1 (100 → 101 → 001)
- Acknowledge message from C1 to Dir4 (001 → 000 → 100)
- Write hit response (message from Dir4 to C3 (100 → 101 → 111 → 011)

C1 cache line 0: <S, bbbb, …..> → <I, x, x, …..>
C3 cache line 0: <S, bbbb, …..> → <M, bbbb, …..>
Dir4: <S, 00001010>, →<M, 00001000>, M4 = bbbb…..

M2 transaction

C7: R, M2

- Read miss request message from C7 to Dir2 (111 → 110 → 010)
- Read response (data) message from M2 to C7 (010 → 011 → 111)

C7 cache line 0: <I, x, x, …..> → <S, 2222, …..>
Dir2: <I, 00000000> → <S, 1000000>, M4 = 222…..

(b), (c) Same analysis like (a)

5.16

P1:	P2:
A = 1;	B = 1;
A = 2;	While (A <> 1);
While (B == 0);	B = 2;

Without an optimizing compiler the threads, SC will allow different orderings. Depending on the relative speeds of P1 and P2, "While (A <> 1);" may be legitimately executed

a. Zero times:
   B = 1; → A = 1;→ While (A <> 1);→ B = 2; → A = 2; While (B == 0);
   B will be set to 2

b. Infinite number of times:
   B = 1; → A = 1;→ A = 2; →While (A <> 1); ……
   B will be set to 1

c. A few times (A is initially 0)
   B = 1; → While (A <> 1);→A = 1;→ B = 2;→ A = 2; ….
   B will be set to 2

An optimizing compiler might decide that the assignment "A = 1;" is extraneous (because A is not read between the two assignments writing to it) and remove it. In that case, "while A .." will loop forever.

5.22

i. 64 processors arranged as a ring→largest number of communication hops = 32
   100+10x32=420 ns
ii. 64 processors arranged as a 8x8 grid→largest number of communication hops = 14
   100+10x14=240 ns
iii. 64 processors arranged as a hypercube→largest number of communication hops = 6 (log64)
   100+10x6=160 ns

b.

i. Worst case CPI = 0.75+0.2/100x(420)x**2.0 = 2.43**
ii. Worst case CPI = 0.75+0.2/100x(240)x**2.0 = 1.71**
iii. Worst case CPI = 0.75+0.2/100x(160)x**2.0 = 1.39**