EECS 470 Winter 2024
HW1 solutions

1a)

```
Loop:   LD      R1, 0(R2)
        DADDI   R1, R1, #1
        SD      0(R2), R1
        DADDI   R2, R2, #4
        DSUB    R4, R3, R2
        BNEZ    R4, Loop
```

Inst.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
LD	IF	ID	EX	M	WB													
DADDI		IF	ID*	ID*	ID	EX	M	WB										
SD			IF*	IF*	IF	ID*	ID*	ID	EX	M	WB							
DADDI						IF*	IF*	IF	ID	EX	M	WB						
DSUB									IF	ID*	ID*	ID	EX	M	WB			
BNEZ										IF*	IF*	IF	ID*	ID*	ID	EX	M	WB
X													IF*	IF*	IF			
LD																IF	ID	EX

15 cycles

1b)

Inst.	1	2	3	4	5	6	7	8	9	10	11	
LD	IF	ID	EX	MEM	WB							
DADDI		IF	ID*	ID	EX	MEM	WB					
SD			IF*	IF	ID	EX	MEM	WB				
DADDI					IF	ID	EX	MEM	WB			
DSUB						IF	ID	EX	MEM	WB		
BNEZ							IF	ID*	ID	EX	MEM	WB
X								IF*	IF	ID	EX	MEM
LD										IF	ID	EX

9 cycles

Note: we are stalling the BNEZ because it resolves in ID but needs data from the DSUB which doesn't exist until the end of DSUB being in EX.

1c)

```
Loop: LD R1, 0(R2)
      DADDI R2, R2, #4
      DSUB R4, R3, R2
      DADDI R1, R1, #1
      BNEZ R4, Loop
      SD -4(R2),R1
```

Inst.	1	2	3	4	5	6	7	8	9	10
LD	IF	ID	EX	MEM	WB					
DADDI		IF	ID	EX	MEM	WB				
DSUB			IF	ID	EX	MEM	WB			
DADDI				IF	ID	EX	MEM	WB		
BNEZ					IF	ID	EX	MEM	WB	
SD						IF	ID	EX	MEM	SB
LD							IF	ID	EX	MEM

6 cycles.

2.

a. Cache has 4MB/32bytes per line $=2^{22}$ bytes/(2^5 bytes/line)$=2^{17}$ lines. 8-way set associative so 2^{17} lines/2^3 lines/set $=2^{14}$ sets.
   - #bits of byte offset=lg(32)=5
   - #bits of set index=lg(2^{14})=14
   - #bits of tag=32-(5+14)=13 (or 32-lg($2^{22}/2^3$)=13)

b. Tag size * number of tags / bits per byte
   $13*2^{17}/8=$ 212992 bytes

c. Fully associative means tags are (32-5)=27 bits. $27*2^{17}/8=$ 442368 bytes
   Direct-mapped means tags are (32-5-17)=10 bits. $9*2^{17}/8=$163840 bytes

3. Offset is 4 bits. There are $2^{10}/2^4 =2^6$ lines. Note the cache grabs all 16 bytes on a miss.

a. Note there are $2^5 = 32$ sets. So offset is first 4 bits, index is next 5, tag is last 7.

0x4001	Set 0: Compulsory miss
0x400A	Set 0: Hit!
0x4017	Set 1: Compulsory miss
0x1000	Set 0, Compulsory miss
0x2000	Set 0, Compulsory miss
0x4000	Set 0, Conflict miss (a fully-associative cache would hit)
0x2005	Set 0, Hit!
0x4008	Set 0, Hit!

b. (note that index for DM is 6 bits…)

Address	DM	2-way SA
0x0000	Set 0	Set 0
0x0200	Set 32	Set 0
0x0600	Set 32	Set 0
0x0000	Set 0 (hit)	Set 0 (miss)

c.

Address	DM	2-way SS
0x0000	Set 0	Set 0
0x0400	Set 0	Set 0
0x0000	Set 0 (miss)	Set 0 (hit)

4.

a. characteristics that distinguish RISC from CISC:
   - Fixed-length instructions (RISC) vs. variable-length instructions (CISC)
   - RISC machines generally have dedicated memory instructions, CISC ALU instructions tend to be able to directly address memory.
   - RISC instructions tend to be simpler.
     - You'd not expect to find "solve quadratic equation" as an instruction in a RISC machine.
     - As such, it tends to mean you need more instructions in a RISC machine to do the same thing.
       - And in practice, few bits to encode those instructions (this isn't just because RISC instructions are simpler, it's because CISC machines can use small instruction encodings for commonly used instructions.)
   - CISC machines generally have more complex hardware to do more specialized instsructions

b. Terms:
   - **Register pressure** is when there aren't as many registers available as you could use.
   - **Register spills** are when you move a value to memory because there aren't enough registers.
   - **Register fills** (also called reloads) are when you retrieve that value from memory. *Register pressure results in spills and fills*.

c. The register encodings each use 5 bits and the 16-bit constant obviously takes 16 bits. Thus 26 bits are used, leaving 6. Thus, this uses $1/2^6$ of all the encodings, or about 1.5625% of the possible encodings.

d. Disadvantages:
   - Slow access time/more power/more die area (all more-or-less the same thing)
   - Instruction encoding gets harder as we need more bits to specify the register.

   Advantages

- Fewer spills and fills
- Fewer false (name) dependencies.

5. Pay careful attention to the definition of "speedup" in our text. Just to be careful we'll note it refers to how long an action takes. On a single core it took 1.0 sec to do 1000 transactions, or 1ms per transaction. If the speedup were 1.25 it would take 1/1.25 or 0.8ms. Or put differently we'd be doing 1,250 transactions per second.
    a. Quad-core: 0.7 speedup means we need 1.428571 ms for each transaction on a single core. With all four cores we are looking at 0.8*1.42/4+0.2*1.42 which is .57143ms. That is about 1750 transactions/sec.

       8-core: 0.8*2/8+.2*2 = .6 ms per transaction, or about about 1667 transactions/sec.

    b. The single core uses 100 mJ/transaction.
       The quad-core uses about 77 mJ/transaction.
       The 8-core uses 90mJ/transaction
       Quad core wins!

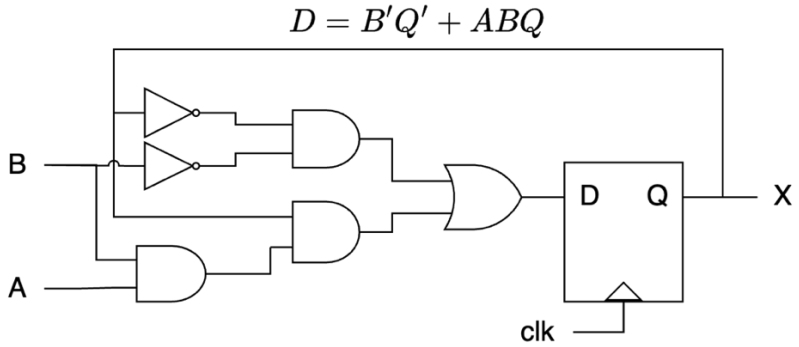
    c. Single core is still at 100mJ/transaction.
       Quad core uses ~48mJ/transaction.
       The 8-core uses 37.5mJ
       8-core wins!

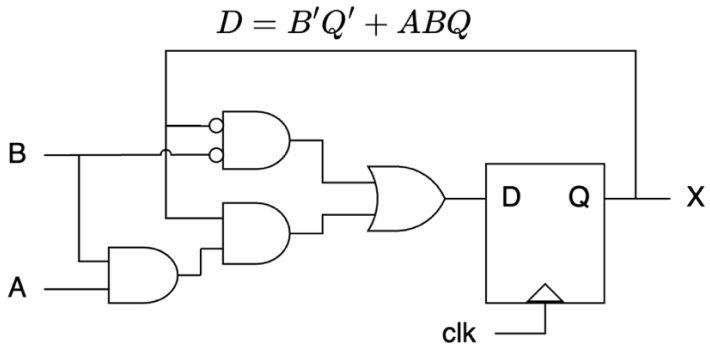
6. (use Q=X)

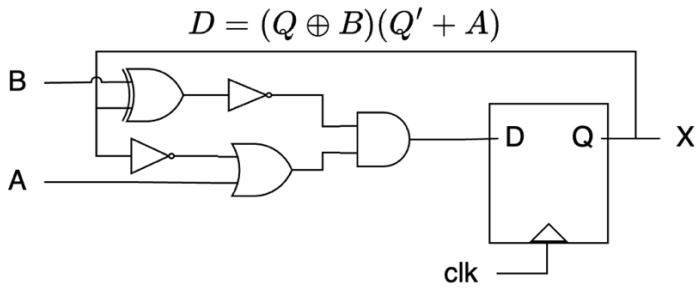
Q	A	B	D
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Simplest answer is `D=B'Q'+ABQ`
Another answer is `D=(Q^B)'(Q'+A)` (in words, Q==B and (Q=0 or A=1))
If X=Q', then `D=QB+Q'(AB)'=(Q^B)'+Q'A'=[(Q^B)(Q+A)]'`
Other equivalent answers acceptable

Possible drawings:







7.

a. Without the lock, you'd run the risk of two (or more) things incrementing at the same time and only having it increment once. If both read count before either of them changed it, they'd both end up reading the same original value then saving back the same incremented value instead of incrementing the value twice.

b. Two (or more) could enter the while loop before either sets lock to be 1. Thus, you end up with the same problem as before.

c. Note that lock=1 is now removed.

```
int increment()
{
   while(TAS(lock));  //This stays here until TAS returns 0
   count=count+1;
   lock=0;
}
```

Since the test and the set are atomic, it is no longer possible for multiple threads to be in the while loop at the same time and perform the increment simultaneously.

d.

- Both are intended to be changed by some other thread/process. You declare such variables volatile so the complier knows that the value of the variable may change at any time.
- None. Declaring something volatile has no impact on the caching of that value.
- It tells the complier not to optimize away the memory access—it can't assume, for example, that the value will be the same as it was the last time it was loaded. Among other things this means that the complier can't keep the value in a register and just keep grabbing the value from that register.

8.

a. period = 0.5 ns or 500ps. Distance = 0.5ns * $3x10^8$m/s = 15cm

b.

i) "R" (for resistance)

ii) Dynamic power is the energy consumed when transistors switch during logic computation. Static power is the energy consumed continuously due to the conduction that exists between Vcc and Gnd through transistors (which creates leakage curret)

iii) 1) Capacitance, Voltage, and Frequency. 2) Because performance is approximately proportional to frequency and frequency is more or less proportional to voltage for a given design and a small voltage range.

c. SPECint and SPECfp are benchmarks used to judge the speed of a microprocessor on standard integer and floating point workloads, respectively. They are important because they are the most cited benchmarks for processor performance and so computer architects pay close attention to them.

d.

- Resistance goes up by a factor of 2.
- The resistance goes down by a factor of 4 (4x the cross sectional area)
- The resistance goes down by a factor of 2 (2x the cross sectional area)
- Tin has a resistivity of about 1.09×10^{-7}. Copper has a resistivity of about 1.68×10^{-8} (These values will vary with temperature so you may have slightly different numbers). So resistance will drop by about 6.49x.

e. We'll take any answer that seems to be even mildly on-point. Some key points include:
   - It's a side channel attack that uses the data cache and measures the timing behavior of operations on private data.
   - Trains the branch predictor to misspeculate so it can do instructions on a misspeculated path that won't be committed. This avoids exceptions being thrown on accesses to private data.
   - Speculatively executed instruction(s) can bring private data into the cache because the permission error isn't broadcast until the instruction is retired.
   - Can observe timing behavior of operations relying on private data in the cache to learn about the private data.
   - No effective defense exists.