M1a.

Remember that overflow is defined as the result of the operation making no sense, which in 2's complement representation is equivalent to the mathematical result not fitting in the format.

- if any of the operands is zero, there is no overflow
- if one of the operands is positive, and the other is negative, there can be no overflow
- if the two operands have the same sign, we have the following cases:

10 (-2) + 10 (-2) = 00 (0 != -4)	1 case
11 (-1) + 10 (-2) = 01 (1 != -3)	1 case
10 (-2) + 11 (-1) = 01 (1 != -3)	1 case
01 (1) + 01 (1) = 10 (-2 != 2)	1 case

So there are 4 cases from the total 16.

M1b.

The easiest solution is to represent all possible values.

An 8-bit exponent covers [0–255], which with 127 bias means [-127–128].

Nibble for a float. SEEM. Exponent bias = 1, thus 00,01,10,11 = 0,1,2,3 -1,0,1,2

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

1 00 0 | - 0
1 00 1 | denorm: 0.1 x 2^0 = 0.1 = 1/2
1 01 0 | - 1.0 x 2^0 = -1
1 01 1 | - 1.1 x 2^0 = -1.5
1 10 0 | - 1.0 x 2^1 = 10 = -2
1 10 1 | - 1.1 x 2^1 = 10 = -3
1 11 0 | reserved for –∞
1 11 1 | reserved for NaN
…similarly for positive numbers

Ans: MostNeg= [-3 = 0b1101 = 0xD],
SmallestPos = [1/2 = 0b0001 =0x1]
NextSmallestPos = [1=0b0010=0x2]

M2a.

Static = 0 (no globals)
Stack = 4 (1 pointer)
Heap = 48 (2 * [8 (2 ints) + 4 (4 chars) + 8 (2 ptrs) + 4 (1 ptr)]) = 2*24 = 48

M2b.

Advantage: Saves a malloc call (could take a long time to search freelist) Alex: less internal fragmentation (?), freeing only requires one call (less programmer effort)
Disadvantage: Might not succeed where the other would have (if memory fragmented not one large chunk but two smaller chunks), we have to wait until both are unused before we can free!

M2c.

T **F** We discussed 3 schemes: *K&R*, *slab allocator*, and the *buddy system*. It's possible to write code to make any of these the best and any of these the worst performer (i.e., one will never always dominate another, performance-wise).

One of the truisms about these schemes is that there is no single best one.

T **F** Mark and sweep garbage collection *does not* work for circular data structures.

That's reference counting to which you're referring, my friend.

T **F** If you wrote code that had no calls to `free` and we only garbage collect when we have to, *reference counting* will start collecting before *copying*.

Copying would start collecting *before* RC, because it HAS to GC when the memory is half-full (RC can wait until it's virtually all full).

M3.

(a): NO. Pointer types (to float and to pointer to dlist) have the same size. Once this has been said, the author of line (a) should be turned into shark bait.

(b): NO. Pointer arithmetic is fine.

(c): CT. "p[i][j].next" requires a "struct dlist *" assignment. p[i][j] is a "struct dlist", so the right past should be preceded by an ampersand.

Right: p[i][j].next = &(p[i][j+1]);

(d): RT, the last iteration is out of bound. Note that this error occurs only when the out-of-bound memory is protected. Otherwise, we are just overwriting another position of memory. If we are unlucky, that "another position of memory" will correspond to the heap metadata, and we will realize this only when trying to free the following memory position.

(e): 70 because it's 5*14.

M4.

```
01              add  $dst, $0, $shamtreg          # copy $shamtreg so we don't alter it
02              andi dst, $dst, 0x1f              # The shamt has a maximum size!
03              sll  $dst, $dst, 6                # "slide" the shamt to the right location
04              lui  $at, shiftLby0(upper)        # This lui and the following ori serve to…
05              ori  $at, $at, shiftLby0(lower)   # "point" to the shiftLby0 instruction
06              lw   $at, 0($at)                  # reg now contains the shiftLby0 inst
07              or   $dst, $dst, $at              # "paste" shamt into instruction
08              lui  $at, shiftLby0(upper)        # Again, lui and the following oriserve to…
09              ori  $at, $at, shiftLby0(lower)   # "point" to the shiftLby0 instruction
10              sw   $dst, 0($at)                 # Self-modify our code!
11 shiftLby0: sll $dst, $src, 0                   # The shiftLby0 instruction
```

F1a.

There are $2^6 = 64$ lines in the truth table. That equates to a 64-bit number, which is 16E$.= 16Exa$

F1b.

A	B	C	Foo
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

$$Foo = \overline{X}\overline{Y} + \overline{X}Z + \overline{Y}Z$$

Foo is the *NotMajority* , or *AntiMajority* , or *Minority* circuit

F1c.

A	B	C	Bar
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

We are requested to use C as the selector line of a 1-bit multiplexor. We know a multiplexor has the form O= MUX(S,M,N) => O = S M + not(S) N

Bar = A not(B) + A C + not(B) C = A not(B) (C + not(C)) + A C + not(B) C =
C (A not(B) + A + not(B)) + not(C) (A not(B)) =
C (A + not(B)) + not(C) (A not(B))

We need a not gate for B, one OR for the C part, and one AND for the not(C) part.

notB = Not(B)
Bar = MUX(C, AND(A,notB), OR(A, notB))

1 Not
1 Mux
1 And
1 Or

F2a.

1. Add a small adder with one input tied to 1, the other tied to busA, & output tied to the MemToReg mux
2. Change the 2-input MemToReg mux to be a 3-input mux (adding output of adder)
3. Change the width of MemToReg from 1 to 2 (and the corresponding logic)
4. Change the RegDst-controlling 2-input Rd/Rt mux to be a 3-input Rd/Rt/Rs mux
5. Change the width of RedDst from 1 to 2 (and the corresponding logic)

F2b.

NO because we can't write two registers at once

F3a.

```
loop:                  ## [S=Stage, w=written, r=read]
1  addi $t0, $t0, 4    ## $t0 w S5           (need 2 no-ops for $t0)
2  lw   $v0,  0($t0)   ## $t0 r S2, $v0 w S5 (need 2 no-ops for $v0)
3  sw   $v0, 20($t0)   ## $v0, $t0 r S2
4  lw   $s0, 60($t0)   ## $t0 r S2, $s0 w S5 (need 2 no-ops for $s0)
5  bne  $s0, $0, loop  ## $s0 r S2, ALU S3
```

BEFORE

```
addi  I  D  A  M  Wt
lw       I  Dt A  M  Wv
sw          I tDv A  M  W
lw             I  Dt A  M  Ws
bne               I  Ds A  M  W
```

AFTER

```
addi  I  D  A  M  Wt
no-op              |
no-op              |
lw             I  Dt A  M  Wv
no-op                     |
no-op                     |
sw                     I tDv A  M  W
lw                        I  Dt A  M  Ws
no-op                                |
no-op                                |
bne                                  I  Ds A  M  W
```

2@1.5, 2@2.5, 2@4.5

F3b easy.

The addi lw hazard is handled with forwarding, now we only have to fill the load and branch delay slots. Thus, we can simplify the code to be:.

```
Loop:
1 addi $t0, $t0, 4    ##
2 lw   $v0,  0($t0)   ## (v0 read can't follow, otherwise must stall)
3 sw   $v0, 20($t0)   ##
4 lw   $s0, 60($t0)   ## (s0 read can't follow, otherwise must stall)
5 bne  $s0, $0, loop  ##
```

We can fill both load delay slots at once by swapping the sw and the lower lw and inserting a no-op in the branch-delay slot.

```
Loop:
1 addi $t0, $t0, 4    ##
2 lw   $v0,  0($t0)   ## (v0 read can't follow, otherwise must stall)
4 lw   $s0, 60($t0)   ## (s0 read can't follow, otherwise must stall)
3 sw   $v0, 20($t0)   ##
5 bne  $s0, $0, loop  ## (next instruction will always be executed)
6 no-op
```

Or we could move the sw to the branch-delay slot and insert a no-op in the lw s0 delay slot. But what if we swapped the two loads? Hmmm...

```
Loop:
1 addi $t0, $t0, 4    ##
2 lw   $v0,  0($t0)   ## (v0 read can't follow, otherwise must stall)
4 lw   $s0, 60($t0)   ## (s0 read can't follow, otherwise must stall)
6 no-op
5 bne  $s0, $0, loop  ## (next instruction will always be executed)
3 sw   $v0, 20($t0)   ##
```

We can't move one of the lws to the branch-delay slot because we don't know what instruction is after ours, and since there are no hardware interlocks (stalls on unprotected loads), we could have incorrect behavior.

F3b hard.

It seems like we're stuck! The two lw instructions "poison" $v0 and $s0 for one cycle, where nobody can read them immediately. But the two following instructions *read* $v0 and $s0! What to do, what to do... Well certainly the beq has to be at or near the end of the loop – ideally one away from the end. We'll need to fill its branch delay slot, and there can't be a lw $s0 right before it. So that constrains us to:

```
loop:
1     addi $t0, $t0, 4
_____________________
_____________________ ## Can't be lw $s0 (so can be sw $v0 or lw $v0)
5     bne  $s0, $0, loop
_____________________
```

So one constraint is that the lw $s0 can't be before it. And we've got to fill the branch-delay slot. We can't put a load there because we don't know what comes after our snippet. So the only movable instruction is the sw. But slot 3 can now only be lw $v0, which leaves lw $s0 for slot 2. This also comes from looking at the last "easy" solution and swapping the two loads.

```
loop:
1     addi $t0, $t0, 4
4     lw   $s0, 60($t0)
2     lw   $v0,  0($t0)
5     bne  $s0, $0, loop
3     sw   $v0, 20($t0)
```

F4a.

The first-level data cache for a certain processor can cache 64 KB of physical memory. Assume that the word size is 32 bits, the block size is 64 bytes, the size of the physical memory is 2 GB, and the cache is 4-way set associative.

a) How many bits are needed for the TIO?

Offset: 64-byte block, byte-addressing 2^6 bits to specify byte Offset=6 .

Index: 64 KB cache / 64 bytes = 1K blocks.
#sets = 1K blocks / 4-way set-associative = 256 sets = 2^8 sets Index=8 .

Tag = 32 bits – (8+6) = 18

F4b.

Double the Cache Size Offset = 0, Index +1, tag follows -1

Double the word size (from 32 bits to 64 bits) Offset = 0, Index=0, Tag+32

Change the associativity to fully associative Offset=0, Index-8, Tag+8 (no index!)

F4c.

Many reasons:

- Memory protection! (user-user and program-program)
- Without it, a program won't be able to "think" it can address all 32-bits. (remember, the OS, other apps & utilities are also running). E.g., if photoshop decides to open up a **huge** TIFF, nothing will be able to run!
- You can simultaneously run many large programs very efficiently because only a small subset will need to resident at any one time
- "Better programmer productivity since programmers don't have to manage memory overlays themselves" and
- "Enhanced portability since the program has no dependencies on the physical configuration of

the machine" (from http://www2.parc.com/csl/groups/sda/projects/oi/workshop-94/foil/note-vmem-advantages.html)

F4d.

32-bit virtual address space, 64 MB physical memory and a 4 KB page size.

How many virtual pages are there? 2^{32} bytes/VA-space / 2^{12} bytes/page = 2^{20} = 1M
How many physical pages are there? 2^{26} bytes/PA-space / 2^{12} bytes/page = 2^{14} = 16K
1-level page-table, each entry 4B, table size? 1M pages * 4 Bytes/page = 2^{22} = 4MB.

F5a.

CPU Time = InstructionCount * CPI * ClockCycleTime

Want:

$$\text{CPU Time}_A = \text{CPU Time}_B$$

which is

$$\text{InstructionCount}_A * \text{CPI}_A * \text{ClockCycleTime}_A = \text{InstructionCount}_B * \text{CPI}_B * \text{ClockCycleTime}_B$$

But

$$\text{InstructionCount}_A = \text{InstructionCount}_B$$ (same program run through both!)

Thus

$$\text{CPI}_A * \text{ClockCycleTime}_A = \text{CPI}_B * \text{ClockCycleTime}_B$$

or

$$\text{CPI}_A / \text{ClockCycleFrequency}_A = \text{CPI}_B / \text{ClockCycleFrequency}_B$$

or

$$\text{CPI}_A * \text{ClockCycleFrequency}_B = \text{CPI}_B * \text{ClockCycleFrequency}_A$$

which is

$$\text{CPI}_A * 1\text{GHz} = \text{CPI}_B * 3\text{GHz}$$

And dividing both sides by 1GHz gives

$$\text{CPI}_A = 3 * \text{CPI}_B$$

Calculating CPI (and plugging into the above equation) gives:

.2 * 3 + .3 * + .5 * 3 = 3 * [.2 * 1 + .3 * 1 + .5 * 2]

Multiplying both sides by 10 (to get those fractions out) gives

2 * 3 + 3 * + 5 * 3 = 3 * [2 * 1 + 3 * 1 + 5 * 2]

Dividing both sides by 3 (it's in every term) gives

2 + + 5 = 2 * 1 + 3 * 1 + 5 * 2 + 7 = 2 + 3 + 10 = 8

F5b.

Network transmission time: 1000[B]x8[b/B]/1000[Mb/s] = 8000b/(1000b/μs) = 8 μs

Effective bandwidth: 8000b/(X+8)s = 10 Mb/s X+8=800 X=792μs

F5c.

What should RAID 0 be called? AID, since there's NO redundancy!
RAID 6: TWO drive failures
Highest Data-rate I/O device? Gigabit Ethernet (@ $1x10^9$ b/s), RamDisk (even higher!)
New Benchmarks? Availability!