CS61C Spring 2013 Final Rubric

M1) Hacker's Delight (10 pts)

a) Regardless of the self-modifying part, if mystery is called with a non-null argument, the second-to-last instruction loads the data at address `$a0` into `$v0`. So in order to print 4 (also notice that the formatting character `%d` was used in the `printf`), you must call mystery with the *address* of a piece of memory that contains the *integer* data 4. Of the available variables declared in main, `A` holds garbage, `char4` holds 52, and `float4` holds `0x40800000`. So you need to pass the address of the 3rd entry of `pi`. Accepted answers were `$pi[2]` and `pi+2`. (2 pts)

   1 pt if you answered `pi+8` (remember pointer arithmetic!).

b) For a non-NULL argument, `mystery` self-modifies its first instruction (the `ori`) by adding 1 to it. This starts incrementing the immediate field, thus causing the `ori` to put the number of times mystery has been called with non-NULL arguments into `$v0` initially. However, a non-NULL call then overwrites `$v0` with the data at address `$a0` before returning. If called with a NULL argument, mystery never executes the self-modifying portion and returns the # of times it's been called with non-NULL addresses. (4 pts)

c) A NULL call to `mystery` changes nothing and simply returns the value of its internal counter, so it can be called as many times as you want. Correct answer was to put N/A in both blanks. Needed some of part (b) to be correct in order to get credit. (2 pts)

d) A non-NULL call to `mystery` adds 1 to the `ori` instruction. As an internal counter, this works fine until we overflow the immediate field into the `rd` field. Remember that `ori` is a bit-wise function so its immediate field is *zero-extended*, so mystery continues to work for the full width of the `ori`'s immediate field. So the answer is $2^{16}-1$. Calling it one more time causes the `ori` instruction to store into `$v1` instead of `$v0` (+1 to `rd` field), meaning a NULL call to `mystery` would return garbage in `$v0`. (2 pts)

M2) Cache Money, y'all (10 pts)

The key to this problem was analyzing the memory access pattern of `SwapLeft`. For every index `i`, `SwapLeft` performed (in order) a read from `A`, read from `B`, write to `A`, then write to `B`. So that's 4 memory accesses per byte (since data is type `uint8_t`) in strictly alternating fashion.

a) Best-case scenario for a direct-mapped cache is no conflict misses (i.e. `A[i]` and `B[i]` map to different rows). Since we access bytes in memory sequentially (`i++`), for every cache block for `A` or `B`, we get an initial miss then hit on the block boundary, followed by 2 hits for the rest of the bytes in the block (`a-1` bytes if block size is `a`). So in total this leaves us with a best ratio of `2*(a-1)+1:1` = `H:1`. Solving, we get `2a-1=H`, so `a=(H+1)/2`. (2 pts)

   1 pt if answered `H+1`.

b) Worst-case scenario is that `A` and `B` live at addresses that conflict (i.e. `A[i]` and `B[i]` *always* map to the same cache row). Because we alternate accesses between the arrays, we always conflict in the cache and we never get a hit, so the worst ratio is 0:<any non-zero number>. (1 pt)

c) For swapping, we *must* read and write from both A and B. To improve the worst case cache performance, we need to guarantee that we access one of the arrays consecutively. The two solutions are: read A, read B, write B, then write A and read B, read A, write A, then write B. Both of these involved using both temporary variables given to you (`tmpA`, `tmpB`). (1 pt)

d) Same worst case scenario as part (b) with conflicting addresses of `A[i]` and `B[i]`. But our new access pattern for `SwapRight` generates MMHM on the cache block boundary, followed by HMHM for the remaining `a-1` bytes of the block. Notice the compulsory miss on the first byte that actually becomes a hit in the remaining bytes because of the last write from the previous byte. This means our ratio for a full cache block is `1+2*(a-1):3+2(a-1)`, which simplified to `2a-1:2a+1`. (2 pts)

e) Moving to 2-way set associative, `SwapLeft` only accesses two arrays, so even if `A[i]` and `B[i]` map to the same set, they can both co-exist in the cache. With LRU, the read from `B` cannot kick out the block of `A`, regardless of whether the cache is empty or full, so we end up with the same cache performance as part (a), where we had `2*(a-1)+1:1` = `2a-1:1`. (2 pts)

   1 pt if answered `a-1:1`

   1 pt if answered `2a:1`

   MRU works essentially the same as direct-mapped once the cache is full (the LRU block in each set will remain there until the cache gets flushed). This leads us back to our worst-case scenario from part (b), which is 0:<any non-zero number>. (2 pts)

M3) What is that Funky Smell? Oh, it's just Potpourri (10 pts)

a) This question asked for *non-negative* floating point numbers < 2. This did NOT include -0. Some important things to remember are that all positive denorm numbers count and the floating point representation of +2 is `0x40000000` (exponent of `0x80`). So non-negative floating point numbers less than 2 are any combination where the 2 most significant bits are 0's. This leaves any combination of the lower 30 bits, so there are 2^{30} such numbers. (1 pt)

   +0.5 pt for value, +0.5 pt for work WITH correct value.

b) A jump instruction on a 32-bit MIPS system sets `PC={PC+4[31:28],target address,0b00}`. Biggest jump would happen when you are right *before* a boundary (e.g. `0x0FFFFFFC`). Then PC+4 is across the boundary at `0x10000000` and your farthest jump ends up at `0x1FFFFFFC` (assuming your target address field was all 1's). This distance is 2^{28} = 256 MiB. (2 pts)

   +1 pt for value, +1 pt for IEC format WITH correct value.

c) There were a number of necessary corrections, some of which could be combined with others to fit into the 5 given spaces. (7 pts)

   +1 pt, Line 1: CHANGE return type of `count_az()` to `int *`.

   +1 pt, Line 3: CHANGE `count` to a pointer to a malloc-ed array of 26 ints.

   +1 pt, Line 4: ADD line to initialize array from Line 3 to zeros.

      (the previous two could have been combined into a `calloc` call in Line 3)

   +1 pt, Line 7: CHANGE `&str` to `*str` (need to dereference instead of getting address).

   +1 pt, Line 7: CHANGE 0x97 constant to 97 (0x97 sits in extended ASCII codes).

      (the previous two *should* have been combined into a single CHANGE)

   +1 pt, Line 15: REMOVE `free(str)` because you need to return the array.

   +1 pt, Line 15: ADD `return count` to return the array.

      (the previous two *should* have been combined into a single CHANGE)

F1) Madonna revisited: "We Are Living in a Digital World" (22 pts)

a) The solution could be calculated as:

`out = ((a+~b) XNOR ~(a·~b))ab = ((a+~b) XNOR (a+~b))ab = 1ab = ab` (6 pts)

Alternatively, recognizing that out was the output of a 3-input AND gate that included `a` and `b`, you only needed to evaluate the upper part of the circuit for `a=1`, `b=1` to decide between `out=0` and `out=ab`.

A lot of students forgot NOT's when converting from the circuit to a Boolean expression, and each forgotten NOT was penalized by 1 pt. Other students misread the XNOR as an XOR, NOR or OR. In these cases 2 pts were deducted. Otherwise, assuming your solution was legible, 1 pt was deducted for each error made when manipulating the Boolean expression.

b) The solution (3^m)^(2^n) follows immediately from the first rule of counting. More explicitly, an n-to-m logic gate has 2^n rows and each output function is defined by one of 3 possible outputs (0, 1, high-Z) in m*2^n locations. (5 pts)

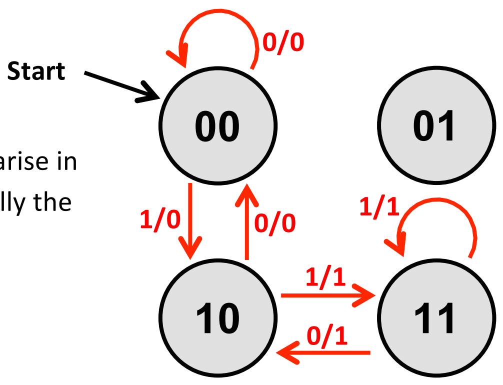
4 pts were awarded for solutions that mixed up m and n, and 2 pts were given for 2^n * 3^m.

c) Solution shown here. (6 pts)

Approximately half of the class used a four state solution to solve this problem. The trick to getting it down to 3 states is to merge the "seen 1 branch-taken" and the "seen 1 branch-not-taken" states that arise in the obvious 4 state solution into one, since they're really the same state.

-1 pt for each incorrect transition

-1 pt for using the fourth state



d) The critical path through the circuit is register -> multiplier -> adder -> mux -> register, so clk-to-q + 30 + 30 + 5 + 2 <= clock_period = 100 ns, which means that clk-to-q <= 33 ns. (5 pts)

-1 pt for arithmetic errors.

+1 pt for having the maximum delay formula somewhere in your work.

+3 pts if your solution was 29 ns or 35 ns, and your work clearly showed that this was the result of accidentally including the time to generate seed/reseed or forgetting set-up time, respectively.

F2) V(I/O)rtual Potpourri (23 pts)

a) Page table must contain an entry for every virtual page. Since we have 4 GiB of VM and 1 MiB pages, we have 4 Ki-entries = 2^{12} entries. (2 pts)

Common mistake: 512 (# of physical pages)

b) The page table base register holds the address of a page table, which we know sits in physical memory, so it must be at least as wide as a physical address. Since we have 512 MiB of physical memory, a physical address is $\log_2$(512 MiB) = 29 bits. (2 pts)

c) In worst case, the TLB is cold except for code page (we specified that ALL of code fit in one page, so the code page would be in physical memory because of the function call to `update_hist()`). Accessed data in `scores` fits in single page because those 40B of continuous memory start at a page boundary, so 1 page fault for `scores[0]`. Then values in `scores` point to data in `histogram` that live in 10 separate pages (yes, this implies `MAX_SCORE` is a ridiculously large #), leading to 11 page faults total. Because code statement was slightly ambiguous (whether we meant all of the program's code vs. all of the function's code), we accepted both 11 or 12 page faults. (6 pts)

Partial credit: +5 pts for 10 page faults, +3 pts for 1, 2, 3 page faults, +1 pt for >12 page faults

d) Best case means the TLB is already filled with the mappings for the pages holding the two arrays. Because the elements of `histogram` we access are determined solely by the entries of `scores`, the best case is we only access `histogram` elements in the same page (e.g. all entries in `scores` are the same). Because the index into `scores` always increments, that sets our maximum loop iterations. With 1 page for code and 1 page for `histogram`, we assume the remaining 30 TLB entries contain the `scores` data. 30 pages hold 30*(1MiB/4B) = $30*2^{18}$ ints. (7 pts)

Partial credit: 2^{20} (2 pts), 2^{18} (3 pts), 2^x where $x > 0$ (1pt)
32 (2 pts), 31 (3 pts), 30 (4 pts)

e) We want to improve the temporal locality of the data. Since our data set is mostly repeats of a small number of scores, we can generate more hits by sorting the scores. (6 pts)

Partial credit: warm up TLB (2 pts)
redefine histogram so that scores in clusters map to nearby indices (4 pts)

F3) Datapathology (22pts)

a) `R[rd] = Mem[R[rs]] + R[rt];   PC = PC+4;` (4 pts)

3 pts for main RTL, 1 pt for PC update. Full assembly code was automatic 0.

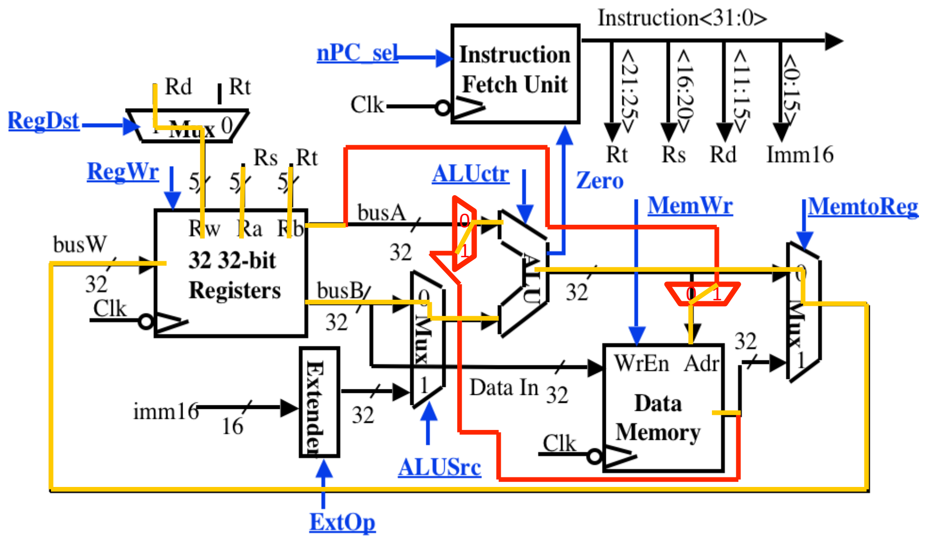
Common mistakes:	Forgetting R[rs] in Mem call -1 pt
	Forgetting all R[] -2 pts
	Forgetting Mem[] -2 pts
	Using MIPS notation just for registers -1 pt

b) Two datapath changes were needed (adding control signals did not count as a separate change – no deductions for this): muxing busA into Data Mem Adr ("mux1") and muxing DataMemOut into the upper input of the ALU ("mux2"). Each description was worth 2 pts and the corresponding drawing on the datapath was 1 pt each. (6 pts)

Common mistakes:	Mux1 on Data In instead of Data Adr -1 pt
	Mux2 on busB instead of busA -1 pt
	Mux1 on ALUOut (affects MemtoReg mux) -1 pt
	Pulling busW instead of DataMemOut into mux2 -1 pt
	Pulling busB instead of busA into mux1 -1 pt

Note: the instructions specified that you could NOT add adders! If you used an adder, your max score for this part (assuming your implementation actually worked) was 3 pts. Typically this involved muxing a 0 onto the lower input of the ALU and then muxing busW+busB onto busW.

c) Correct additions to datapath shown below in red. Active portions of datapath during a `madd` instruction are shown in orange (red portions are active as well). You should be able to determine the correct control signals based on this. (6 pts)



We were EXTREMELY generous with this part. Each blank was worth ½ pt, including naming your two new control signals. We accepted named values for your new control signals as well as 0 or 1 if correctly specified in part (b). Many students combined these into a single control signal and this was given full credit. We also graded correctness of the other signals based on your changes in part (b). We even gave credit to students who were clearly guessing.

Common mistakes: Putting 0 or 1 for `nPC_sel` or `ALUctr` -½ each
`MemtoReg` as 1 (need to pass ALU result to `busW` to store) -½

d) We were looking for anything that acknowledges the need to do an ALU operation *after* the memory read. Statements that alluded to needing to add an adder after the memory read were also given credit. Statements that did not specify which stages/datapath elements caused the problem were given half credit. (2 pts)

e) The key to this problem is remembering that a pipeline is run by a single clock, so every stage must be the same length. Let's examine the two scenarios: (4 pts)

In a 6-stage pipeline, the new adder stage fits within the current 100 ps clock period, so we extend it to 100 ps to fit, meaning the new latency for 1 instruction will be 6*100 = 600 ps. The throughput (how frequently an instruction completes) is the inverse of the clock period, so 1/(100 ps).

In an extended MEM pipeline, we are told that the memory access takes 100 ps and the adder adds 50 ps, meaning our new minimum clock period is 150 ps. So our instruction latency is 5*150 = 750 ps and the throughput is 1/(150 ps).

This problem was worth 1 pt for each square, so 4 pts total. Half credit was given if you put the inverse of the answer for throughput.

F4) What do you call two L's that go together? (22pts)

Page 1 – Each question was worth 4 points. Of this, the multiple-choice answer was worth 1 pt and the justification was worth 3 p.

a) +1 for correct MC selection (B)
   +1 for explanation of why the code does work correctly sometimes
   +2 for A not private / data race to increment A

b) +1 for correct MC selection (C)
   +1 for justifying correct result
   +2 for explaining duplication of work
   (D was also accepted as a correct answer, provided that the justification was sound)

c) +1 for correct MC choice (D)
   +2 for explanation of false sharing
   +1 for further explanation of mem scheme with multiple processors/threads

Page 2

Map: (5 pts)

-3 pts for not accounting for WIDTH
-5 pts for no for loop at all

Reduce: (6 pts)

-2 pts for not accounting for WIDTH
-6 pts for no for loop at all