Fall 2011 CS61C Midterm Answers

Question 1 :

a. (2 points - partial credit if the answer was not written in IEC format) - 1 Register = 64 bits = 8 B = 2^3 B. Total Storage in Registers = 64 * 8 B = 2^6 * 2^3 B = 2^9 B = 512 B
b. (2 points - partial credit if answer was not written in IEC format) - Number of valid opcodes = 2^6 - 1 (for the 0) = 63. The number of function bits = 64 - 3*6 (for the 3 Rs) - 6 (for shamt) - 6 (for opcode) = 64 - 5*6 = 34. Therefore, when it's in the R format, it's an additional 2^{34} instructions). Final answer: 2^6-1 + 2^{34} ~ 16 Gibi instructions.
c. (2 points - all or nothing) - They would shun them because the RISC philosophy is a minimalist one, with the fewer operations the better (with a few exceptions, like jumps instead of **`beq $0 $0 label`**).
d. (2 points - 1 point for the correct answer and 1 point for the explanation) - One fewer S means the number of NaNs shrinks from $2(2^S - 1)$ (sign bit * all the bit patterns of S bits minus 1 for inf) to $2(2^{S-1} - 1) = 2^{S+1} - 2 - 2^S + 2 = 2^S$, so the number of real numbers increases by 2^S.
e. (2 points - 1 point for the correct answer and 1 point for the explanation) - With more exponent bits, we could get to smaller numbers so the number of cases of underflow would decrease, since these are the numbers closest to zero we can't represent.
f. `(3 points - partial credit if someone realized that you needed to subtract the address of the top of the heap from the bottom of the stack) -` **`temp`** lives at the bottom of the stack, and **`PI`** lives at the top of the static region, just below the heap, so **`(unsigned int) &temp - (unsigned int) &PI`**
g. (2 points - 1 point for the correct answer and 1 point for the explanation) - We can't. First of all, the number of instructions can vary significantly. More than that, we'd need to know the memory speed, cache (number, design), compiler optimizations, relative machine loads, numbers of cores, parallelization of the code, disk speed (for loading, VM -- you'll learn later), I/O speeds and frequency of I/O requests. Also pipeline scheme, branch tax, etc you'll learn later.
h. (2 points - 1 point for the correct answer and 1 point for the explanation) - False: PC-relative addresses are handled before linking once MAL is converted to TAL by the assembler, and since the linker pieces the object files into the executable, it knows where everything in the text and data segment belongs.
i. **`0b1101 -`**
    1. (1 point - all or nothing) - S/M = -5
    2. (1 points - all or nothing) - 2s Compl = -3
    3. (1 point - all or nothing) - Unsigned = 13
    4. (2 points - partial credit if some steps were correct) - Float = $- 1.1_2 \times 2^{(\text{Exp-Bias})} = -1.1_2 \times 2^{(2-1)} = -1.1_2 \times 2 = -11_2 = -3_{10}$.
    5. (2 points - partial credit if some steps were correct) - ori \$0 \$0 \$0
    6. **`(2 points - partial credit if some steps were correct) - ((0b1101 & 0x6) | 0x9) >> 1 = (0x4 | 0x9) >> 1 = 0xD >> 1 = 0x6`**
j. Answers below (4 x 1 = 4 points - lost half a point for each answer if someone used the PC's address from the previous part)
    1. 000010 11111 11100 00000 00000 000011
       opcode = 2, so this is a j instruction; (PC+4)[31:28] = 0111 So, next instruction is at address 011111111 11100 00000 00000 00001100 = 0x7FF0000C
    2. 000000 01000 00000 00000 00000 001000
       This is jr \$t0, so the next instruction is at 0xDEADBEEF
    3. 000101 01000 00000 11111 11111 111110

This is bne $t0, $zero, offset = -0xFFFE = -2_{10}; So the next instruction is at PC+4-2^4 = 0x70000008+4-8 = 0x70000004

4. 000000 00000 01000 01000 00000 100011
   This is subu $t0, $zero, $t0. So the next instruction will be PC + 4= 0x7000000C

Question 2 :

a. **`GetNibble(nibble_t A[], index_t i){ return( (A[i/2] >> 4*(i%2)) & F )}`**
   i. 6pt total for any working solution. Partial credit for each of the following:
   ii. 1pt for selecting **A[i/2]**
   iii. 1pt for correctly shifting the bits using **>>** or **/**
   iv. 1pt for correctly implementing **i%2** or an equivalent expression
   v. 1pt for multiplying by **4** in order to shift the entire nibble instead of just 1 bit
   vi. 2pts for using **&0x0F** or equivalent to mask out the superfluous nibble

b. **`NewNibbleArray`** problems
   1. It moves the stack pointer **`$sp`** without restoring, violating register conventions
      i. 3pt for recognizing that a calling convention was violated
      ii. 2pt partial credit for recognizing other (less immediately catastrophic) memory errors
      iii. 1pt partial credit for describing a bug that only causes stack corruption in certain cases
   2. **`main`** works because it never stored anything *before* the call that it needed to retrieve *after* the call (which would have gone to the wrong place in memory, probably resulting in a bus error if the number of nibbles was not a multiple of 8, and/or a stack error if the new memory location was out of range)
      i. 3pt for correctly describing the general conditions under which main does not fail
      ii. 2pt partial credit for mentioning a specific thing that would have caused main to fail (such as calling a subroutine)
   3. One $a0 reflects the number of bytes we need (it's correct as written), move the stack down by a word to store $ra, call malloc, restore $ra, $sp (put it back up by a word), return via jr $ra
      i. 3pt for a complete bug fix, which involves using malloc or storing the array on the heap
      ii. 1pt partial credit for a solution that fixes the stack pointer inconsistency, but fails to address the larger problem of returning a pointer to memory in an invalid stack frame

Question 3 :

```
sum:  li $v0 0          # $v0 = 0
      beq $a0 $0 done   # if head == NULL goto done
      addiu $sp $sp -8  # "allocate" 2 words on stack
      sw $ra 4($sp)     # push $ra
      sw $a0 0($sp)     # push $a0
      lw $a0 4($a0)     # head = head->cdr
      jal sum           # Call sum recursively
      lw $a0 0($sp)     # pop $a0
      lw $ra 4($sp)     # pop $ra
      addiu $sp $sp 8   # "free" 2 words on stack
      lw $t0 0($a0)     # head->n
      addu $v0 $v0 $t0  # head->n + length(head->cdr)
done: jr $ra            # go home
```

1 pt for loading 0 into $v0.
1 pt for a beq to done, comparing head to $0. .5 pt for comparing something to $0.
1 pt or allocating some space on stack. .5 pt for just showing that we need to move $sp.
1 pt for saving $a0.
1 pt for saving $ra.
2 pts for loading head->next into $a0. 1 pt for loading into a register that's not $a0. 1.5 pt for wrong offset.
2 pts for jal sum. 1 pt for j.

1 pt for loading back \$a0 from stack.
1 pt for loading back \$ra from stack.
1 pt for moving \$sp back. .5 for reasonably showing necessity.
1 pt for loading the value of node. .5 if close.
1 pt for summing the values. .5 pt for just adding into \$v0. .5 for just adding the right value.
1 pt for jr \$ra.
-0.5pt for using saved registers without saving on stack first.

Question 4 :

a. 7 bit offset specifies 2^7 bytes. That's 2^5 = 32 words.

3 pts total for 2^5 or 32
Partial Credit Responses:
2 pts for a clear mention of 2^7 bytes per block
0 pts for other responses

b. $2^{(13+7)}$ Bytes = 2^{20} Bytes = 1 MebiByte = 1 MiB

3 pts total for 1 MebiByte or 1 MiB
Partial Credit Responses:
2 pts for 2 ^ 20 - not using IEC or incorrectly translating to IEC
1 pts for clear work showing proper units
0 pts for the wrong answer with no clear, correct work

c. Lowest hit rate happens when, on every node access, it's NOT in cache, we bring it in (and the entire block) when we reference head->n (MISS), it's there when we reference head->next (HIT), and repeat. 1:1 or 50% (not a function of A or B)

3 pts for 50% hit rate, 1 pt for 0%
Highest is that it's laid out linearly, like an array. With linear array accesses and one query per memory address, we always have 1 (miss) : n-1 (hits), where n is the number of remaining elements we're accessing in that block. Here we have A words. Therefore, it's A-1 hits:1 miss,or (A-1)/A % hits.
3 pts for correct expression, 1.5 pts for expression with wrong constant multipliers

d. The nodes are laid out in memory in such a way that whenever a block is kicked out, all of the other 15 nodes in that block will have been accessed (so a block is never kicked out without having a perfect A-1:1 hit rate). Said another way, if you draw memory block-size wide, then all the "memory blocks" are either completely fill with Nodes or completely empty, and the "linking" of them is such that all the Nodes on a particular block will have been visited before another Node is visited that shares its tag (so when that block is kicked out, all nodes have been visited).

3 pts for any correct explanation, 1.5 pts for explanation of access order permutation within blocks, 0 pts for any incorrect justifications

<-------- Width of one Block -------->
Node Node Node Node Node Node Node Node
Node Node Node Node Node Node Node Node
Node Node Node Node Node Node Node Node
Node Node Node Node Node Node Node Node