CS61C Summer 2013 Midterm

Your Name: __Peter Perfect_____ **SID:** _00000000_____

Your TA (Circle): Albert Kevin Justin Shaun Jeffrey Sagar

Name of person to your LEFT: __Sammy Student_____

Name of person to your RIGHT: __Larry Learner_____

This exam is worth 95 points and will count for 24% of your course grade.

The exam contains 7 questions on 14 numbered pages. Put all answers in the spaces provided. Some pages are intentionally left blank and will not be graded.

Question 0: You will receive 1 point for properly filling out this page as well your login on every page of the exam. (0 of 1 for forgetting any of these)

Question	Points (Minutes)	Score
0	1 (0)	
1	10 (16)	
2	27 (48)	
3	15 (30)	
4	8 (16)	
5	22 (45)	
6	12 (25)	
Total	**95 (180)**	

All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet.

Signature: _____

Question 1: *Reppin' Yo Numbas* (10 points, 16 minutes)

For this question, we are using 16-bit numerals. For Floating Point, use 1 sign bit, 5 exponent bits, and 10 mantissa bits. For Biased use a bias of $-2^{15}+1$.

Scoring: +1pt for every correctly filled blank, -0.5 for additional incorrect responses in part (a), minimum of 0.

a) Indicate in which representation(s) the numeral is **closest to zero**: Two's Complement (T), Floating Point (F), or Biased (B). The first one has been done for you. NaN is not valid in comparisons.

	Numeral:	**Closest to zero:**
1)	0x0000	TF
2)	0xFFFF	**T**
3)	0x0001	**F**
4)	0xFFFE	**T**
5)	0x8000	**F**
6)	0x7FFF	**B**

b) We now wish to **add** the numerals from **top to bottom (1 to 6)**. However, it is possible that we encounter an error when performing these addition operations. For each number representation, state the FIRST error that is encountered and which numeral causes it; if no error is encountered, answer "no error."

Possible arithmetic errors are: OVERFLOW, UNDERFLOW, NaN, and ROUNDING (assume we are rounding using a truncating scheme).

Representation:	**Arithmetic Error:**	**Numeral #:**
Two's Complement	**Overflow**	**5**
Floating Point	**NaN**	**2**
Biased	**No Error**	**x** **(this blank was worth zero points)**

Question 2: *This Problem is Like a Box of Chocolates* (27 points, 48 minutes)

a) 4 points. In C, `char` is in fact a *signed* variable type. Assume we have 4 `char`s != 0x00 loaded into a 32-bit `int`, one in each byte. Complete the function below that will negate the `char` in byte `i` of the `int` "in place", with byte 0 being the least significant and byte 3 being the most significant.

```
void negByte(int *data, char i) {

   *(((char *)data)+3-i) = -*(((char *)data)+3-i);

Also accepted: *data = *data^(0xFF << (8*i)) + (1 << (8*i))

}
```

Parts of the problem:
+1 point for dereferencing data and storing the value back
+1 point for manipulating the memory location of the i-th byte within data
+2 points for correctly negating the byte. This meant both flipping the bits and adding one if students took the masking approach.
Because of the wide range of responses, scoring was done holistically. Deductions of 0.5 points were given for small mistakes (eg. Issues with parentheses).

---

b) 4 points. You have been given access to a version of C that does not have the `sizeof` function implemented. However, this version of C still knows how large each type is internally (and a `char` is still one byte). Implement a constant time `sizeof` operation by filling in the blank below. Note that your `sizeof` operation will take in a *variable* of the type you wish to find the size of, rather than the type itself.

```
#define sizeof(type) ((char *)(&type+1) - (char *)(&type))
```

Answers to this problem were widely varied. Points were awarded based on how close it was to the correct answer. In particular, it was important to use pointer arithmetic on the address of the variable.
Answers that dealt with addresses received 1 point, and answers that showed some sort of tangential understanding regarding variable sizes received 0.5 points. Simply using strlen() or len() on type received no credit.

---

c) 5 points. For the following MIPS code, fill in the branch immediates (in decimal) AFTER pseudo-instruction replacement. Then fill out the relocation and symbol tables. Recall that the assembler will assign addresses starting at 0x0.

Write addresses in hexadecimal and do not show leading zeros. The tables are part of an object file and only recognize TAL.

For this problem, it was important to recognize that both li and la would get broken up into two instructions when translated into TAL.

Scoring:
+0.5 points for each decimal branch immediate, 1.0 point total. 0.5 points instead of 1 point was taken away from the whole section if both branches were off by one.
+1.5 points for correct symbol table, with points given at 0.5 increments based on the percentage of the table correct. Having all correct symbols but wrong addresses got 1 point, having most but not all symbols with wrong addresses got 0.5 points, and having wrong symbols and addresses got 0 points.
+2.5 points for correct relocation table, with points given at 0.5 increments based on the percentage of the table correct. Students who wrote la instead of lui and ori received a -0.5 point penalty, but otherwise were treated as having written both down. A -0.5 point penalty was given for writing the entire line inside the instruction column. Students who included the correct instructions as part of a long list of nearly all the instructions present received 0.5 points.

```
        .text
start:  li   $t0, 0x10000
        addi $t1, $t0, 1
        bne $a0, $t0, start → -4
        beq $a0, $t1, next  → 0
next:   la $a0, str
        jal printf
        .data
str:    .asciiz "C.A.L.L. me, maybe?"
```

Symbol Table

Symbol	Address
start next str	0x0 0x14 0x0 (data segment addresses are separate)

Relocation Table

Instruction	Address	Dependency
jal lui ori	0x1c 0x14 0x18	printf l.str r.str

d) 3 points. Ash Ketchum has six slots in his party, each of which can hold a single Pokémon. Additionally, Ash has access to a PC (personal computer) which holds the rest of the Pokémon he owns. Essentially, his party acts as a "cache" for accesses to the PC (the "memory").

i. Each slot in Ash's party can hold any Pokémon. What kind of cache is this analogous to? (Circle one)

Set-associative Write-back **Fully Associative** Direct Mapped Write-through

+1 point for circling the correct type.

ii. Ash's party exploits __**temporal**___ locality but not __**spatial**____ locality.
+0.5 points for correctly-filled "temporal locality" blank.
+0.5 points for correctly-filled "spatial locality" blank.

Explain in one sentence (the answer below is more than one sentence for clarity):

The party has "one-unit" slots and thus does not exhibit spatial locality; we don't pull any extra pokemon into the party when we make a request for a pokemon (effective block size of one). On the other hand, a fully associative cache will likely use some kind of LRU scheme, which takes advantage of temporal locality.

+0.5 points for identifying that fully-associative cache often holds recently-used elements.
+0.5 points for stating that the block size of one doesn't exploit spatial locality.

---

e) 3 points. Fill in the *recursive* definition of `strcmp()` below. `strcmp` returns 0 if the contents of the strings are identical, otherwise it returns the difference in the ASCII representations of the first non-matching characters (negative number if `s1 < s2` lexicographically). (**Clarification from when the exam was given: `strcmp` will never be called with `s1=NULL` or `s2=NULL`**)

```
int strcmp(char *s1, char *s2) {
      if ( *s1 == *s2 && *s1 != '\0' )
            return strcmp( s1+1, s2+1 );
      return (*s1 - *s2);
}
```

+1 point for correctly comparing *s1 and *s2 in condition (or any working variation).
+1 point for correctly checking for the end of the input string(s) (only one check is necessary).
+1 point for a correct recursive call in the return statement.
–0.5 points for using the pointers s1 and s2 where elements should have been used or vice-versa
NOTE: significant variations from the structure of this answer that were fully correct or close to correct in concept were individually graded. If fully correct, they received the full three points; otherwise, they were subject to the holistic grading standard set by this rubric.

---

f) 8 points total. MIPS already has a 64-bit architecture, so it's just a matter of time before MIPS128 is released. Let's help them out with their 128-bit instruction format design. Answer the following questions *independently* based on the MIPS32 design taught in class.
–1 point from the whole question if all three parts were answered for a 64-bit instruction format.

i. (2 points). If we doubled the opcode field size and quadruple the register field sizes, how many instructions *forward* can we reach with a single branch? Answer in IEC.

2^75 = 32 Zebi-instructions

Branch instructions are I-format. They have an opcode (now 12 bits), two register specifiers (now 20 bits each), and an immediate. Since the other fields take up 52 bits, the immediate must be 76 bits long. This immediate is a signed offset in terms of instructions, so it is possible to branch 275 − 1 instructions forward from PC+4, or 275 total. This is 32 zebi-instructions.

+0.5 points for properly calculating the length of the immediate.
+0.5 points for recognizing that the offset is in instructions, not bytes.
+0.5 points for recognizing that the offset is a signed number of instructions.
+0.5 points for proper IEC notation ("zetta" was accepted, as it was given on the Green Sheet).

ii. (3 points). If we quadruple the total number of I and J format instructions, what fraction of memory could we reach with a single jump instruction? Feel free to leave as a power of 2.

2^-4 or 1/16

Quadrupling the number of instructions extends the opcode field to 8 bits. Therefore, the jump target immediate can be 120 bits long. Since this value must be resolved to an instruction-aligned address by adding some implicit zeros at the less-significant end of the address, the immediate is shifted left 4 bits, as an instruction is now 24 bytes long. The resulting 124-bit specifier can cover 1/16 of the 128-bit address space.
+1 point for properly calculating the length of the immediate.

+1 point for recognizing that the target needs to be shifted at all – this was given for those who multiplied the 2120 possible immediate values by 4 and answered 2−6.
+1 point for getting the correct answer.
−0.5 points for answering in terms of the number of addresses reachable instead of a fraction.

iii. (3 points). If we want to keep the same number of R, I, and J instructions, what is the maximum number of registers out architecture can support? Answer in IEC.

64 Gibi-registers

This question implies that the opcode and funct fields will remain at 6-bit length. However, in order to retain proper shift functionality, the shamt field must be lengthened to 7 bits. This leaves 128 − 6 − 6 − 7 = 109 total bits for three register specifiers in an R-type instruction, yielding a max of 36 bits for each register specifier.

+1 point for correctly expressing how the R-type format constrains the register length.
+1 point for recognizing that the shamt size must change.
+1 point for getting the correct answer.
−0.5 points for lack of proper IEC notation.
−3 points for work that indicated the need for four fields (four registers?) that are the length of a register specifier in an R-type instruction.

Question 3: *It's a Bird… It's a Plane… It's Supermalloc!* (15 points, 30 minutes)

Suppose we are writing a program that will be dynamically allocating a LOT of different things (hw2, anyone?). We want to create an easy way to free everything we've ever dynamically allocated. Implement the following scheme:

`void **freeAr` – An array that holds a pointer to every space `malloc`'ed in the program.

`int freeArSize` – Keeps track of the length of `freeAr`.

`void* supermalloc()` – Wrapper function for `malloc()`. When called, allocate the requested space, add the pointer to `freeAr`, then return the pointer.

`void superfree()` – Wrapper function for `free()`. To be called once just before exiting. Frees ALL dynamically allocated memory used in program.

a) 7 points. Fill in the missing code below to correctly implement this scheme. You may find the following functions useful. For this problem, assume allocations always succeed.

```
void* malloc (size_t size);
void* realloc (void* ptr, size_t size);    // works when ptr = NULL
void  free (void* ptr);                    // does nothing when ptr = NULL
```

```
#include <stdio.h>
#include <stdlib.h>

void* supermalloc(size_t size, void ***toFree, int *toFreeSize) {

    *toFree = (void *)realloc(*toFree, (*toFreeSize+1)*sizeof(void*));//2.5pt
                              +1 point for using realloc
                              +1.5 points for second argument to realloc

    void *temp = _malloc(size)_____; // allocate - 0.5 points

    (*toFree)[*toFreeSize] = temp; // update toFree - 1.5 points
                              +0.5 points for temp on right of assign.
                              +1 point for correct deref/array use
                              -0.5 for minor errors in deref/array use

    *toFreeSize = *toFreeSize + 1; // update toFreeSize - 0.5 points
    return temp;
}

void superfree(void **toFree, int toFreeSize) {
    for(int i = 0; i < toFreeSize; i++)

        free(toFree[i]); // 0.5 points

    free(toFree);        // 0.5 points
}

int main() {

    void **freeAr = ______NULL_________;  // 0.5 points

    int freeArSize = _____0____________;  // 0.5 points
```

```
    // acts like malloc(4*sizeof(int)), but with additional tracking features
    int *test = supermalloc(4*sizeof(int), &freeAr, &freeArSize);

    superfree(freeAr, freeArSize);
}
```

b) 1 point. How many total Stack frames are created in the execution of this program? You may assume that all library functions are self-contained (do not call other functions). Note that `sizeof` is an *operator*, not a function.

_____**7**____

This question was generally all or nothing, unless there was an exceptional circumstance where substantial work was shown but a small error was made. Stack frames: main, supermalloc, realloc, malloc, superfree, free, free.

c) 1 point. What is the maximum Stack frame *depth* (in # of frames) during the execution of this program?

_____**3**_____

This question was generally all or nothing, unless there was an exceptional circumstance where substantial work was shown but a small error was made. Stack frame depth: (main, supermalloc, realloc) OR (main, supermalloc, malloc) OR (main, superfree, free) OR (main, superfree, free), for a max depth of three at any time.

Assume we are running this program on a 32-bit machine and that `sizeof(size_t)=4`. Consider the execution right before `supermalloc()` returns (the frame still exists).

d) 6 points. How many bytes are allocated in each section of memory? Assume all declared variables are stored in memory and that the space for `test` was allocated *before* the call to `supermalloc()`.

Think CAREFULLY! What needs to be saved on the Stack? Don't worry about Stack frame alignment (assume you allocate just as much space as you need).

For partial credit, list the names of the variables that are stored on the Stack in the box below.

Stack: ___**36 OR 40**____ **Heap:** ____**20**____ **Static Data:** ____**0**___

(4 points, see below) **(1 point, for correct) (0.5 points for 16)** **(1 point, all or nothing)**

For stack: Automatic +4 for answer of 36 or 40. Both were accepted since the problem statement was unclear with respect to $a2. Otherwise, +0.5 for (argc and argv), +0.5 for each of the other items on the stack, -0.5 for incorrect items on stack, minimum of zero, capped at +3.5.

Vars on Stack:

argc, argv, freeAr, freeArSize, test, size ($a0), toFree ($a1), toFreeSize ($a2), $ra, temp

Question 4: *U2 Can Write MIPS in Mysterious Ways* (8 points, 16 minutes)

Answer the questions below about the following MIPS function.

```
Mystery:
      addiu $t0, $0,  0
      addiu $t1, $0,  0
      addiu $t7, $0,  32
Lbl1: addu  $t2, $a0, $t0
      addu  $t3, $a0, $t1
      lb    $s0, 0($t2)
      beq   $s0, $0,  Lbl3
      beq   $s0, $t7, Lbl2
      sb    $s0, 0($t3)
      addiu $t1, $t1, 1
Lbl2: addiu $t0, $t0, 1
      j     Lbl1
Lbl3: sb    $0,  0($t3)
      addu  $v0, $a0, $0
      jr    $ra
```

a) 1 point. What variable type would `$a0` be in the corresponding C program? _**char ***_ `a0;`

(* char earned 0.5 points. Anything else was a 0)

b) 1 point. What does the number `32` represent in the 3rd instruction shown?

_**' '**__(the ascii character for a space)

("an ascii character" or an incorrect ascii character earned 0.5 points)

c) 1.5 points. Give the following labels more intuitive/functional names:

`Lbl1` __**loop**___ `Lbl2` _**skip; space; spacefound**_ `Lbl3` __**return**___

(0.5 points per blank)

d) 2.5 points. Describe in ONE sentence what the MIPS function *accomplishes*. You are not allowed to use any register names, so don't describe it instruction-by-instruction.

This function removes all spaces from a string. (Mentioning something about removing spaces earned 0.5 points)

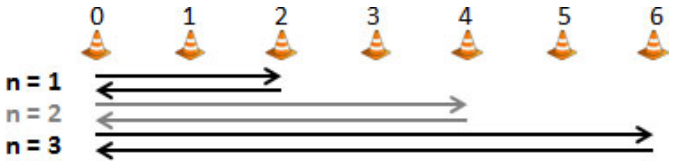
e) 2 points. Your friend tells you that using the function above broke the rest of his code! In one sentence EACH, describe two DIFFERENT ways to fix `Mystery`:

Method 1: **Mystery needs to store $s0 to the stack and restore it before returning, per register calling conventions. (1 point)**

Method 2: **Mystery could use an unused temporary register (e.g. $t6) instead of using $s0. (1 point)**

Question 5: *Dying For Some Cache* (22 points, 45 minutes)

In sports, the exercise known as *suicides* is where an athlete makes successively longer sprints to and from the same starting position. The following function counts the number of times the athlete passes evenly-spaced cones (including cone 0 but not including the end cone). Each array entry `cones[i]` is the `i`-th cone from the start and the athlete makes `n` runs, with each successive run being `stride` cones longer than the last. The example shown below is for `stride=2` and `n=3`:



Returned array:
`cones[] = {6,6,4,4,2,2}`

```
1     int *run_suicides(int stride, int n) {
2           int i, j, *cones = (int *) calloc(sizeof(int)*n*stride);
3           for (i = 1; i <= n; i++) {
4                 for (j = 0; j < i*stride; j++)      cones[j]++;
5                 for (j = i*stride-1; j >= 0; j--)   cones[j]++;
6           }
7           return cones;
8     }
```

Assume our system has the following parameters:

- 16MiB address space
- Block size of 16B
- Cache with 8 slots: write-back and write allocate

SHOW YOUR WORK FOR PARTIAL CREDIT!!!

a) 4 points. If the cache is fully associative with random replacement,

(2 points) TIO Breakdown: **_20_:_0_:_4_**

O = $\log_2(16)$ = 4. A = $\log_2(16\text{Mi})$ = 24. Fully associative = 1 set, so I = log2(1) = 0. T = 24-0-4.
-0.5 if wrong O, -0.5 if wrong A, -1.0 if I ≠ 0.

(2 points) Bits to implement cache: **___1200_____**

Random replacement means 0 replacement bits. 8×(8×16+20+1+1) = 8×150 = 1200.
Full credit for using your Tag calculated above. -0.5 for algebra errors, bits per slot, or no Dirty bit.

b) 4 points. If the cache is 2-way set associative with LRU replacement,

(2 points) How many blocks map to each set? Answer in IEC. **___256 Ki_____**

Now have 8/2 = 4 sets, so I = 2 and TIO breakdown is 18:2:4. Width of Tag gives answer so 2^{18} = 256 Ki-blocks. Alternatively, 16 MiB address space split into 16 MiB/16 B = 1 Mi-blocks, which are then split into 4 different sets, so 1 Mi-blocks/4 = 256 Ki-blocks. **+1.0 if solved for correct Tag or # of blocks in address space. -0.5 if not IEC.**

(2 points) Minimum LRU bits for whole cache? **_____4_____**

Each set has two slots, so need 1 bit/slot (to exactly indicate slot 0 or 1). 4 slots in cache, so 4 bits. **Pretty much all-or-nothing.**

For the following questions, assume that `calloc()` returns a *block-aligned* address and sets the allocated memory to zero in sequential order starting from `cones[0]`. Our cache is 2-way set associative with LRU replacement. Consider ONLY the hit and miss rates for the loops (lines 3-6).

Note: The acceptable answers to part (c)-(e) depended on the following facts:

- `As described above, calloc() causes memory accesses before the loops as it sets data to 0!`
- `cones[]` is an array of `int`s (4 B – 4 per block), not `char`s (1 B – 16 per block).
- The expression `cones[j]++` is TWO memory accesses – a read followed by a write.

If work was shown and we could determine that you fell into any of the following categories: (1) int data, RdWr; (2) int data, WrOnly; (3) char data, RdWr; (4) char data, WrOnly; then we assigned partial credit based on the different expected answers shown below. If no work was shown or we could not determine which category you fell under, then it was all-or-nothing based on the correct answer.

c) 2 points. If `n=1`, what is the minimum miss rate? ___**0%**_____

With the recognition that `calloc` writes to the array, then the minimum of 0% MR will be achieved as long as `sizeof(int)*n*stride` is less than your cache size because all necessary blocks will be loaded before the loops. **+2.0** for answer of 0%. Also **+2.0** for the case `stride=0`.

Ignoring `calloc`, then the minimum MR will be achieved at every multiple of a block size ≤ the cache size. Because we access sequential indices, we will get all hits after the compulsory miss in each block. Assuming we don't access more data than we can fit in the cache, then the 2nd loop will have only hits.

(1) int data is 4 per block, RdWr is 2 accesses per index per loop, so **+1.5** for 1/16.
(2) int data is 4 per block, WrOnly is 1 access per index per loop, so **+1.0** for 1/8.
(3) char data is 16 per block, RdWr is 2 accesses per index per loop, so **+1.0** for 1/64.
(4) char data is 16 per block, WrOnly is 1 access per index per loop, so **+0.5** for 1/32.

+0.0 for assuming `stride=2` (as shown in the graphic). The problem asked for *minimum* MR.

d) 2 points. If `n=1`, what is the maximum `stride` value before the hit rate drops below its maximum? ___**32**_____

Once the size of the `cones` array exceeds the size of the cache, then the first blocks of `cones` get replaced before the loops begin, leading to some compulsory misses and a HR < 100% (MR > 0%). Cache size is 8 slots of 16 B each, so 128 B = 32 ints.

Ignoring `calloc`, then the question becomes ill-posed. As noted above, the max HR is achieved at *every* multiple of a block size ≤ the cache size. **+2.0** for answer based on the cache size (32 for ints, 128 for chars). **+1.0** for answer based on the block size (4 for ints, 16 for chars).

+1.0 for 33/129, **+0.5** for 5/17.

e) 2 points. Explain in 1-2 sentences how switching to no-write allocate affects your answer to part (c):

Using no-write allocate, the call to `calloc`, which only writes, will not load entries into the cache; thus we will get MR > 0%.

(1)(3) RdWr to same address is read miss followed by write hit. **+2.0** if no change.

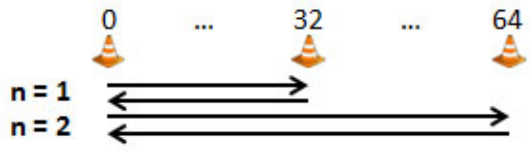
(2)(4) WrOnly is all write misses, so never loads data into cache. **+2.0** if 100% MR (0% HR).

+1.0 if correct definition of no-write allocate, **+0.5** if almost correct definition (no write misses).
+0.0 if answer given WITHOUT an explanation.

f) 4 points. If `n=2` and `stride=32`, what is the miss rate?

____1/16_____

Diagram of the access pattern of `run_suicides()` for `n=2`:



`cones[]` is of size `sizeof(int)*n*stride=4*2*32=256B`, or twice the cache size. This means `calloc` has no effect on this problem, as all blocks for the lower half of the array have been replaced.

Let's split the total accesses into 6 equally-sized segments: `n=1` up, `n=1` down, 2 for `n=2` up, 2 for `n=2` down. You get 1 compulsory miss per block if the block is not currently in the cache, or all hits if it is in the cache. Notice that the first half of `n=2` up should HIT, as it was loaded during `n=1` down. For int data, only 32 entries fit in the cache at once. For char data, all 64 entries can fit in the cache.

(1) int data, RdWr means 8 accesses per block. MR for segments are 1/8, 0/8, 0/8, 1/8, 0/8, 1/8 which leads to a total of 3/48 = 1/16.

(2) int data, WrOnly means 4 accesses per block. MR for segments are 1/4,0/4, 0/4, 1/4, 0/4, 1/4 which leads to a total of 3/24 = 1/8.

(3) char data, RdWr means 32 accesses per block. MR for segments are 1/32, 0/32, 0/32, 1/32, 0/32, 0/32 which leads to a total of 2/192 = 1/96.

(4) char data, WrOnly means 16 accesses per block. MR for segments are 1/16, 0/16, 0/16, 1/16, 0/16, 0/16 which leads to a total of 2/96 = 1/48.

+1.0 for decent work shown or demonstration of understanding of overall access pattern (e.g. picture).

+2.0 for some analysis of access pattern.

+3.0 for mostly correct analysis (maybe algebra errors or wrong MR for 1-2 segments).

g) 4 points. Consider each of the changes listed below independently. Circle the one(s) that would DECREASE the miss rate in part (f):

Halve `n`	Double `stride`	Decrease associativity
Halve cache size	**Double block size (same cache size)**	Write-through policy

Write-through policy has no effect on MR. Associativity has no effect for sequential accesses. Doubling `stride` or halving cache size generally increases block replacements for each iteration of `i`.

Doubling block size decreases MR per block. Halving `n` gets you back to the scenario in part (c), where `calloc` puts you at MR 0%.

+2.0 for each correct answer, **-1.0** for each incorrect answer. Minimum score of 0.

Question 6: *AMATter of Performance* (12 points, 25 minutes)

We wish to implement the following function, which returns the name of the last node in a linked list:

```
struct node {
   char *name;
   struct node *next;
};
```

```
// returns name of last node in linked list
// assume head != NULL
char *lastNodeName(struct node *head) {
   while (head->next)
     head = head->next;
   return head->name;
}
```

a) 5 points. We compile the function into True Assembly Language (TAL). Complete the implementation below. You are not allowed to introduce any additional labels.

```
lastNodeName:
      lw   ___$t0, 4($a0)_____
      beq  ___$t0, $zero, Ret___
      lw   ___$a0, 4($a0)_____
      j    ___lastNodeName_____
Ret:  lw   ___$v0, 0($a0)_____
      jr   $ra
```

1 pt per line:

If wrong offset, **-0.5** (per line)

Wrong register, or register + offset, **-1.0**

Clearly invalid instruction, **-1.0**

Special cases:

Didn't use $a0 for head, **-1.0**

Use other register instead of $t0, **OK if function works**

Suppose we are running code on a machine with the following cache parameters:

- **Unified** L1$ with a hit time of 3 cycles and a hit rate of 90%
- Miss Penalty to main memory of 100 cycles
- Base CPI of 5 (in the absence of cache misses)

b) 1 point. Calculate our machine's AMAT:

AMAT = Hit Time + Miss Rate * Miss Penalty = 3 cycles + 0.1*100 cycles = 13 cycles

Wrong values: **-1.0**, Wrong equation: **-0.5**, Bad math: **-0.5**

__13 cycles___

c) 2 points. We decide to add a L2$ to reduce our AMAT to 5. We know the global miss rate is 1%. What's the worst L2$ Hit Time that will still meet our AMAT goal?

AMAT = L1 Hit Time + L1 Miss Rate * (L2 Hit Time + L2 Miss Rate * Mem Miss Penalty)

AMAT = L1 Hit Time + L1 Miss Rate * L2 Hit Time + L1 Miss Rate * L2 Miss Rate * Mem Miss Penalty

Since L1 Miss Rate * L2 Miss Rate = Global Miss Rate:

AMAT = L1 Hit Time + L1 Miss Rate * L2 Hit Time + Global Miss Rate * Mem Miss Penalty

5 = 3 cycles + 0.1*L2 Hit Time + 0.01*100

1 cycle = 0.1 * L2 Hit Time, therefore, required L2 Hit Time = 10 cycles

Wrong values: **-1.0 each, including wrong Hit time or using global miss rate as local miss rate**, Bad math or wrong answer: **-1.0**

__10 cycles___

d) 3 points. Back to only L1$: what is the CPI_{stall} for `lastNodeName` if it is called on a linked list of length N?

$CPI_{stall} = CPI_{base}$ + Accesses/Instruction * Miss Rate * Miss Penalty

Here, regardless of the length of the Linked list, every other instruction in our program is a load, which requires 2 memory accesses and every other instruction in the program only requires a single memory access (for the instruction itself to be fetched from memory). Thus, our value for accesses/instruction is 1.5. All other quantities are given in the problem statement:

CPI_{stall} = 5 cycles + 1.5 accesses/instruction*0.1*100 cycles = 20 cycles

Wrong values: **-1.0 each, including wrong accesses/instruction and wrong MR/MP**, Including "N" incorrectly arbitrarily: **-1.0**, Other egregious error or bad math: **-1.0,** Writing correct equation worth **1 pt**

__20 cycles___

e) 1 point. In 1 sentence, describe a 1-line change to the code in part (a) that would decrease our CPI_{stall}:

We can replace the second load word with an instruction that moves the value in $t0 to $a0, for example: addi $a0, $t0, 0.

Adding a line to code that doesn't have any function or purpose except to reduce accesses/instruction: **-1 pt,** Altering more than one line of code: **-1 pt**,
Stated that lw should be modified but not which one: **-0.5 pt**,
Vague and no example given such as addu or addi: **-0.5 pt**