CS61C Summer 2012 Midterm

Your Name: _Peter Perfect______________________ **SID:** _00000000__________

Your TA (Circle): Raphael Paul Brandon Sung Roa

Name of person to your LEFT: _Sammy Student_______________________

Name of person to your RIGHT: _Larry Learner__________________________

This exam is worth 110 points and will count for 20% of your course grade.

The exam contains 9 questions on 13 numbered pages, including the cover page. Put all answers on these pages; don't hand in stray pieces of paper.

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

Question	Points (Minutes)	Score
0	1 (0)	
1	15 (26)	
2	10 (14)	
3	10 (18)	
4	9 (16)	
5	10 (18)	
6	17 (26)	
7	12 (22)	
8	26 (40)	
Total	110 (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: *Potpourri – Hard to spell, nice to smell…* (15 points, 26 minutes)

a) **True/False:** **(1 pt each)**

T [✓ F] We prefer two's complement over the unsigned representation because two's complement can represent more values. Same number (e.g. 2^{32})

T [✓ F] The assembler uses symbol tables to resolve absolute addresses. Linker's job

T [✓ F] A program will always execute faster in a RISC architecture than a CISC architecture.
Mix of # instructions, CPI, and clock rate for a particular program can go either way

T [✓ F] The greater the number of memory accesses in a program, the greater the AMAT.
AMAT calculated *per* memory access

[✓ T] F Pseudo-instructions do not always use `$at`.

[✓ T] F Since `$s0` is a "saved register," it does not need to be saved before any function calls.

---

b) Fill in the function below, which returns a new copy of the argument (struct definition not shown):

```
struct something *cpySomething(struct something *old) {
    struct something *new = malloc(sizeof(struct something));
    *new = *old;
    return new;          (1 pt each line)
}
```

---

c) Here is what is currently on the heap:

A	**A**	**A**			**C**	**D**	**D**		

The order of allocation/frees: A allocated, B allocated, C allocated, B freed, D allocated

What allocation strategy was used? ________**next-fit** **(2 pts)**________

---

d) What is the average CPI of the program described in the table to the right? Which is better: halving memory access cycles or arithmetic cycles and why?

Instruction Category	**Cycles**	**Frequency**
Memory Access	10	0.1
Arithmetic	2	0.4
Branch	3	0.2
Comparison	1	0.3

CPI = 10*0.1 + 2*0.4 + 3*0.2 + 1*0.3 = **2.7** **(1pt)**

Half mem access: 5*0.1 + 2*0.4 + 3*0.2 + 1*0.3 = 2.2

Half arithmetic: 10*0.1 + 1*0.4 + 3*0.2 + 1*0.3 = 2.3, so **halving mem access is better**

(1 pt w/explanation)

e) I'm getting the message "`cannot execute binary file`". In one or two sentences, explain what the problem is and how to fix it. **(1pt w/explanation)**

Assuming not a corrupted file, then the executable was compiled on another machine with a different architecture and can't be read. **Re-compile on the current machine.**

---

f) In our 32-bit single-precision floating point representation, we decide to convert one significand bit to an exponent bit. How many **denormalized numbers** do we have relative to before? (Circle one)

More [✓ Fewer] Half as many because lost a significand bit **(1 pt)**

Rounded to the nearest power of 2, how many denorm numbers are there in our new format? (Answer in IEC format) **(1 pt)**

22 significand bits + sign bit but not counting ±0, so exactly 2^{23}-2 denorms __**8 Mebi #s**___

<u>Question 2</u>: *Flippin' Fo' Fun* (10 points, 14 minutes)

Assume that the most significant bit (MSB) of x is a 0. We store the result of flipping `x`'s bits into `y`. Interpreted in the following number representations, how large is the <u>magnitude</u> of `y` relative to the <u>magnitude</u> of `x`? Circle ONE choice per row. **(2 pts each)**

Unsigned	|y| < |x|	|y| = |x|	[✓ |y| > |x|]	Can't Tell
One's Complement	|y| < |x|	[✓ |y| = |x|]	|y| > |x|	Can't Tell
Two's Complement	|y| < |x|	|y| = |x|	[✓ |y| > |x|]	Can't Tell
Sign and Magnitude	|y| < |x|	|y| = |x|	|y| > |x|	[✓ Can't Tell]
Biased Notation (e.g. FP exponent)	|y| < |x|	|y| = |x|	[✓ |y| > |x|]	Can't Tell

- In unsigned, a number with the MSB of 1 is always greater than one with a MSB of 0.
- In one's complement, flipping all of the bits is the negation procedure, so the magnitude will be the same.
- In two's complement, `y` is a negative number. Its magnitude can be found by applying the negation procedure, which is flipping the bits and then adding 1, resulting in a larger magnitude than `x`.
- In sign and magnitude, the 2nd MSB bit will determine the relative magnitudes of `x` and `y`, so you can't tell for certain.
- In biased notation, you read the number the same as unsigned but apply a constant bias to BOTH numbers, so the relation is the same as in unsigned numbers.

<u>Question 3</u>: *Doctor Who?!?* (10 Points, 18 Minutes)

The Daleks are invading the Earth again, and we need the help of the Doctor! Find the errors in this code and fix them so that the code correctly prints **"The 10th Doctor and the Blue Police Box"**. There is exactly one coding error for each function and function call pair and can be fixed by changing 5 or 6 lines total. Fill in the corrections in the blanks on the opposite page.

```
1     void whichDoctor(int* input) {
2         input = 10;
3     }

4     void doctorChanger(char** input1, char** input2) {
5         char* temp = *input1;
6         *input1 = *input2;
7         *input2 = temp;
8     }

9     char* policeBoxGiver(char* input) {
10        *input = "The Master";
11        return "Police Box";
12    }

13    char* colorMaker(void) {
14        char* color = malloc(sizeof(char) * 4);
15        color[0] = 'B';
16        color[1] = 'l';
17        color[3] = 'u';
18        color[2] = 'e';
19        color[4] = 0;
20        return color;
21    }

22    char* colorFixer(char* input) {
23        char temp = *(input+2);
24        *(input+2) = *(input+1);
25        *(input+1) = temp;
26    }

27    int main(void) {
28        int * ith = malloc(sizeof(int));
29        whichDoctor(ith);
30        char* doctor = "Master";
31        char* master = "Doctor";
32        char* details = "and the";
33        char* color = colorMaker();
34        colorFixer(color);
35        char* box = "David Tennant";
36        doctorChanger(doctor, master);
37        policeBoxGiver(box);
38        printf("The %dth %s %s %s %s",*ith,doctor,details,color,box);
39    }
```

	Line #	**Corrected Code**
2 pts	__2___	`_*input = 10;_______________________________________`
2 pts	__14__	`_char* color = malloc(sizeof(char) * 5);_________________`
1 pt	__24__	`_*(input+2) = *(input+3);________________________________`
1 pt	__25__	`_*(input+3) = temp;______________________________________`
2 pts	__36__	`_doctorChanger(&doctor, &master);____________________________`
2 pts	__37__	`_box = policeBoxGiver(box);___________________________________`

For each answer you did NOT get correct, you –2 pts.

If you added a sixth error, –1 pt. (That means if you answered five correct lines and one incorrect line, you received 9 pts).

There were multiple accepted answers for the `colorFixer` function, which are:

```
23        char temp = *(++input + 2);
(between 22/23)    input++;
34        colorFixer(color + 1);
```

There were also **multiple accepted answers** for the `policeBoxGiver` function, which are:

```
11        return strcpy(input, "Police Box");
10        strcpy(input, "Police Box");
```

For line 14, –0.5 pts if you did:

```
char* color = malloc(sizeof(char) * 4 + SOME_VALUE)
```

Question 4: *Let Me Float This Idea By You* (9 Points, 16 Minutes)

(−1pt if 32 bits used)

For a very simple household appliance like a thermostat, a more minimalistic microprocessor is desired to reduce power consumption and hardware costs. We have selected a **16-bit** microprocessor that does not have a floating-point unit, so there is no native support for floating point operations (no `float`/`double`). However, we'd still like to represent decimals for our temperature reading so we're going to implement floating point operations in software (in C).

a) Define a new variable type called `fp`: **(1 pt)**

(also accepted: **`unsigned int`**)

_**`typedef int fp;`**______________________

Many people were not sure what to do here. 1 pt was given mainly to those who wrote a valid statement using typedef or the #define directive, or were close. Struct definitions were also accepted.

We have decided to use a representation with a **5-bit exponent field** while following all of the representation conventions from the MIPS 32-bit floating point numbers **except denorms**.

Fill in the following functions. Not all blanks need to be used. <u>You can call these functions and assume proper behavior regardless of your implementation</u>. Assume our hardware implements the C operator ">>" as *shift right arithmetic*.

b) **(1 pt)**

```
/* returns -num */
fp negateFP(fp num) {
      return _num ^ 0x8000__________________________________________;
}
```

If you assumed 32-bit type, then using 0x80000000 was okay.

c) **(1 pt mask/shift, 1 pt bias)**

```
/* returns the signed value of the exponent */
int getExp(fp num) {
      ______________________________________________________________
      return _((num & 0x7C00) >> 10) - 15_______________________________;
}
```

0x7c00 to zero out everything but the exponent field, shift right by 10 to get the unsigned value, then subtract bias of $2^4 - 1 = 15$ to get the actual signed value.

d) **(1 pt per line)**

```
/* multiplies floating point num by 2^n, while detecting over/underflow */
/* remember, there are no denorms */
fp multPow2(fp num,int n) {

      _int exp = getExp(num) + n; /* get exponent or exponent + n */_______

      if(_exp > 15___________________________________) exit(1);   #overflow

      if(_exp < -15__________________________________) exit(-1); #underflow

      _num &= 0x83FF; /* zero old exponent */______________________________

      return _num | ((exp + 15) << 10); /* set new exponent */_____________
}
```

5 pts total:

First line: 1pt for trying to get the exponent by means of getExp(num) or manually retrieving it.

Second and third line: **–0.5 pt each line** if the numeric value on the right was close, but not correct.

Fourth and fifth: needed to correctly zero out the exponent field of num, and OR or add the modified exponent back into that field. 1pt for not forgetting to re-add the bias, and 1pt for getting the masking/shifting right.

Other:

–1 pt for left shifting the exponent by n instead of adding.

If you didn't add the 15 bias because in getExp() you didn't subtract the 15 bias, then I didn't mark you off for that.

Question 5: *Who Says Less is Better?* (10 points, 18 Minutes)

We're going to take a page out of the ARM book and design a new instruction set architecture with just **16 32-bit registers**. This means that we only need 4-bit register fields in our instructions.

a) How many extra bits do we have now for other fields in the following formats? **(1 pt each)**

Just the # of register fields per format: R: ____3____ J: ____0____

b) For R-format instructions, would you give the extra bits to `opcode`, `shamt`, or `funct`? __**`funct`**___
Explain your choice in a sentence or two (no credit without explanation): **(2 pts w/explanation)**

Can represent more R-format instructions while maintaining consistent `opcode`. `shamt` only needs to be 5 bits for 32-bit registers. **–1 pt** if choice was `funct` but explanation did not include the reason for not picking `opcode` and `shamt`.

For I-format instructions, we naturally give the extra bits to the `immediate` field, resulting in the following format:

```
[ opcode (6) | rs (4) | rt (4) | immediate (18) ]
```

c) What fraction of our address space can we now reach with a branch instruction? **(2 pts)**

we get I think 2^20 B since we only need 4-bit reg. fields for instr. look up at underline _**1/4096 = 2^{-12}**_

Address space = 2^{32} B, branch immediate represents 2^{18} instr = 2^{20} B.

People forgot to put it in fraction, but if I saw 2^{32} B for address space and 2^{18} instr = 2^{20} B for branch reach, then I gave 1 pt.

d) Assume our PC currently contains the address `0x08000000`.
What is the LOWEST address (in hex) we can reach with a branch? ___**`0x07F80004`**_____

On a branch, `new_PC = PC+4 + 4*imm`. Most negative immediate is `0x20000`, so `imm*4 = 0x80000`. Sign-extended, this becomes `0xFFF80000`. Then `0x08000004 + 0xFFF80000 = 0x07F80004`.

(1 pt for seeing PC+4, 1 pt for seeing correct lowest branch distance, 1 pt for correct hex math)

e) Write out the Verilog pseudocode (as in the OPERATION column on the MIPS Green Sheet) for `beq`. Make sure you specify what `BranchAddr` is. **(1 pt)**

```
if( R[rs] == R[rt] )
    PC = PC+4 + { 12{immediate[17]}, immediate, 2'b0 }
```

Many, many people did not specify BranchAddr in the form above.
Some people didn't put **12** as the number of times to extend the immediate's sign bit.

Question 6: *Cache in While You Can* (17 points, 26 Minutes)

Consider a single 4KiB cache with 512B blocks and a write-back policy. Assume a 32-bit address space.

a) If the cache were direct-mapped, **(1 pt each)**

# of rows? ____**8**_____ # of offset bits? ____**9**_____

$2^{12}/2^{9}$ = 8. $\log_2(512)$ = 9.

b) If the cache were 4-way set associative,

# of tag bits? ____**22**____ **(1 pt)** # of index bits? ____**1**_____ **(1 pt)** # of bits per cache slot? ___**4120**____ **(2 pts)**

8/4 = 2, $\log_2(2)$ = 1. 32 – 1 – 9 = 22. 512*8 + 22 + 2 = 4120. If tag wrong, 512*8 + wrong_tag + 2 accepted (if obvious algebraic mistake, **–1 pt**).

Consider an array of the following `location` structs:

```
typedef struct {
      ... // some undefined number of other struct members
      int visited;
      int danger;
} location;

location locs[NUM_LOCS];
```

Here's a piece of code that counts the number of places we've visited. Assume this gets executed somewhere in the middle of our program, that `count` is held in a register, and the size of the array is greater than 4 KiB.

```
for(int i = 0; i < NUM_LOCS; i++)
      if(locs[i].visited) count++;
```

c) What's the fewest possible number of bytes written to main memory? **(1 pt)** _____**0 B**_____

d) What's the greatest possible number of bytes written to main memory? **(1 pt)** ____**4 KiB**____

We're reading, not writing. What will be written back are dirty blocks already in the cache.

Now consider if we store the `visited` and `danger` information in individual arrays instead:

```
int visited[NUM_LOCS];
int danger[NUM_LOCS];
```

–0.5 pt if missing the word "locality."

e) This way, the cache can exploit better ___**spatial locality**___ for the above task. **(1 pt)**

We can expect a ______**lower**______ (higher or lower) miss rate **(1 pt)**

because of the change in the number of _____**compulsory**____(type of cache miss) misses. **(1 pt)**

(also accepted: **read**)

Consider the following code with `NUM_LOCS > 2^10`.

```
for(int i = 0; i < NUM_LOCS; i++)
        if(visited[i] && danger[i] > 5) count++;
```

Two memory accesses are made per iteration: one into `visited`, the other into `danger`. Assume that the cache has no valid blocks initially. **You are told that in the worst case, the cache has a miss rate of 100%.** Consider each of the following possible changes to the cache individually.

f) Mark each as **E**, if it eliminates the chances of this worst-case scenario miss rate, **R** if it reduces the chances, or **N** if it's not helpful. **(2 pts each)**

- More sets, same block size, same associativity __R__
- Double associativity, half block size, same total cache size __E__
- Everything stays the same but use a write-through policy instead __N__

Given that the worst case miss rate is 100% for blocks that hold more than 1 piece of array data, our cache *must* be direct-mapped. In addition, the worst case happens when the `visited` and `danger` arrays start in blocks that map to the same row AND have the same offset.

- With more sets/rows, we are increasing the size of the cache. If the cache size changes such that addresses of `visited[i]` and `danger[i]` no longer map to the same row, then we no longer have the worst case scenario. This is not guaranteed to happen, so the chances are reduced.
- Increasing associativity completely removes the ping-pong effect.
- A write-through policy does not change the behavior of the cache at all.

Question 7: *Can't Make Copies Fast Enough* (12 points, 22 Minutes)

We are revisiting our friend the Fast String Copy from lecture! Recall that the function prototype in C is as follows:

```
        char *strcpy(char *dst, char *src);
```

Consider the following MIPS implementation of this function:

```
      jal   strcpy  # begin function call
      ...
strcpy:
      addi  $v0,$a0,0
loop: lb    $t0,0($a1)
      sb    $t0,0($a0)
      addiu $a0,$a0,1
      addiu $a1,$a1,1
      beq   $t0,$zero,exit
      j     loop
exit: jr    $ra
```

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

- **Unified** L1$ with a hit time of 2 cycles and a hit rate of 95%
- Miss Penalty to main memory of 200 cycles
- Base CPI of 1.5 (in the absence of cache misses)

a) Calculate our machine's AMAT: **(1 pt)**

AMAT = HT + MR × MP = 2 + 0.05×200 = **12 cycles** (–0.5 pt for algebraic mistake)

b) What is the CPI of a single call to `strcpy` with `src = ""` (the function call includes the `jal`)?

There are 8 instructions executed, 2 of which are lb/sb. Unified $ also handles instruction fetches.

CPI = CPI_{base} + Accesses/Instr × MR × MP = 1.5 + 1.25×0.05×200 = **14 cycles**

(1 pt data accesses, 1 pt instruction accesses (> 1 accesses/instr), 1 pt CPI calculation)

(–0.5 pt for algebraic mistake)

c) We decide to add a L2$ to reduce our AMAT to 6. Our L2$ has a hit time of 20 cycles. What's the worst Local Hit Rate that will still meet our AMAT goal? **(1 pt MR, 1 pt HR)**

AMAT = $HT_1 + MR_1(HT_2 + MR_2 \times MP)$. Solving with AMAT = 6, we get $MR_2 = 0.3$, so **worst HR_2 of 70%.**

(–0.5 pt for algebraic mistake, –1 pt for not substituting or substituting incorrect values)

d) In addition to speeding up our architecture, we want to speed up our code, so we decide to eliminate the return value (presumably the caller retains a copy of the destination pointer). In this case, the `strcpy` function above can be rewritten in just 6 instructions. Write out this implementation in the blanks below, introducing any necessary labels (don't worry about any label name clashes with `strcpy`).

```
strcpy2:

        _lb     $t0,0($a1)_____________ (1 pt removal of $v0 instr with 6 instructions here)

        _sb     $t0,0($a0)_____________

        _addiu  $a0,$a0,1______________

        _addiu  $a1,$a1,1______________

        _bne    $t0,$zero,strcpy2______ (1 pt bne and jr)

        _jr     $ra____________________
```

e) If we call `strcpy` and `strcpy2` on the same `src` string of length `n+1=N` (`N` *includes* `'\0'`), what is the ratio of instructions executed in `strcpy` to instructions executed in `strcpy2` (including the `jal`)? Leave your answer in terms of `N`. **(3 pts)**

`strcpy` executes `6*(N-1)+5+3` instructions (6 instr per full loop, 5 instructions on last loop, plus `jal/$v0/jr`) = `6N+2`. `strcpy2` executes `5*N+2` instructions (5 instr per loop plus `jal/jr`).

So ratio is **(6N+2)/(5N+2)**.

1 pt for numerator and denominator (**0.5 pt** each for counting the instructions in the loop, and **0.5 pt** each for counting the rest). **1 pt** for `strcpy/strcpy2` rather than `strcpy2/strcpy`.

f) Is the ratio in part (e) the same as the relative performance between these two functions? In a sentence or two, explain why or why not. **(1 pt w/explanation)**

No, performance (latency) is measured in execution time, not instruction count. The ratio in (e) does not take memory accesses into account.

Question 8: *Putting the Science in Computer Science* (26 points, 40 minutes)

DNA can be called the "alphabet of life." From a *very* simplified view, DNA within a cell produces amino acids, which in turn produce proteins, which are the building blocks for most of your body. Here we'd like to write some code for examining a strand of DNA.

a) DNA is made up of *nucleotides*, which we write shorthand as A, C, G, and T. DNA is in base 4 (quaternary)! Fill in the table below, using the DNA nucleotide symbols in alphabetical order (A < C < G < T). **(2 pts each)**

0 1 2 3

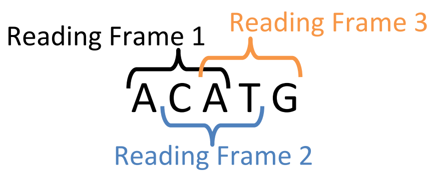
(1 pt given for visible algebraic mistakes)

Decimal	DNA
19	CAT
50	**TAG**

$1\times4^2 + 0\times4^1 + 3\times4^0 = 19$

$50 = 3\times4^2 + 2\times4^0$

An amino acid is encoded by three nucleotides. Because DNA is found in long strands, the following 5 nucleotides can be read 3 different ways:



The sequence **ATG** (as seen in the 3rd reading frame) signals the beginning of a protein ("start codon").

b) Fill in the blanks on the opposite page for the **recursive** function `find_start` in MIPS that returns the position of the first start codon found in the given strand of DNA. Assume each nucleotide is stored as a `char` in memory. *Blanks do not necessarily need to be filled*. Maximum points awarded for using the *fewest* amount of registers and memory. **(code on pg. 13, explanation on pg. 14)**

[Answer the following AFTER looking at the code]

Assume we call `find_start` from main with `char dna[] = "GCATGC";`.

c) How many total frames are created on the Stack (not including `main`)? ___**6 (2 pts)**___
3 frames each for `find_start` and `strlen`. **–1 pt** for 3.

d) What is the maximum depth of the Stack (in # of frames, not including `main`)? ___**4 (2 pts)**___
3rd call of `find_start` still calls `strlen`. **–1 pt** for 3.

e) What will the line `j ret` look like once this file is run through the assembler? _0x08000000 **(1 pt)**_
Object file is in machine language. Absolute address not known, so filled with zeros.

f) Where will the label `ret` show up? (Circle one) **(2 pts)**

Symbol Table Relocation Table [✓ Both] Neither

`ret` is a label in file, so in symbol table. Also needed later for `j ret`, so in relocation table.
All or nothing for grading.

C function prototype:
```
/* dna: start address of DNA strand */
/* pos: search position from start of strand */
int find_start(char *dna, int pos);
```

```
find_start:                                       find_start:
      addiu $sp,$sp,-4_  # PROLOGUE                     addiu $sp,$sp,-4_
      sw    $ra,0($sp)_       (2 pts)                   sw    $ra,0($sp)_
      _________________                                 _________________

      jal   strlen       # call strlen(dna);            jal   strlen

      _________________  # check end                    _________________
      _________________                                 addi  $t0,$v0,-3_
      slti  $t0,$v0,3__       (4 pts)                   slt   $t0,$t0,$a1
      beq   $t0,$0,chk   # check amino acid             beq   $t0,$0,chk
      addi  $v0,$0,-1    # return -1                    addi  $v0,$0,-1
      j     ret          # 'ret' for return             j     ret

chk:  move  $t0,$a0____       (2 pts)             chk:  add   $t0,$a0,$a1
      lb    $t1,0($t0)                                  lb    $t1,0($t0)
      addi  $t2,$0,65                                   addi  $t2,$0,65
      bne   $t1,$t2,rec                                 bne   $t1,$t2,rec
      lb    $t1,1($t0)                                  lb    $t1,1($t0)
      addi  $t2,$0,84__        (1 pt)                   addi  $t2,$0,84__
      bne   $t1,$t2,rec                                 bne   $t1,$t2,rec
      lb    $t1,2($t0)                                  lb    $t1,2($t0)
      addi  $t2,$0,71__        (1 pt)                   addi  $t2,$0,71__
      bne   $t1,$t2,rec                                 bne   $t1,$t2,rec
      move  $v0,$a1____        (1 pt)                   move  $v0,$a1____
      j     ret                                         j     ret

rec:  addiu $a0,$a0,1__       (2 pts)             rec:  addiu $a1,$a1,1__
      addiu $a1,$a1,1__                                 _________________
      _________________                                 _________________
      jal   find_start                                  jal   find_start
      _________________                                 _________________
      _________________                                 _________________

ret:  _________________                           ret:  _________________
      lw    $ra,0($sp)_       (2 pts)                   lw    $ra,0($sp)_
      addiu $sp,$sp,_4_                                 addiu $sp,$sp,_4_
      jr    $ra                                         jr    $ra
```

There were multiple ways to correctly fill in the code. Any equivalent lines (MAL and TAL both accepted) were counted provided proper behavior. Two different solutions are shown on the previous page.

In the left column, the recursion is done by incrementing both the pointer and position counter. In this case, the value returned by `strlen` in `$v0` decreases with each recursion and you need to check if `$v0<3` (checking for `$v0<0` also works). The pointer moves, so you just load characters off of `$t0=$a0`.

In the right column, the recursion is done by incrementing just the position counter. In this case, the value returned by `strlen` in `$v0` is constant (`n`) and you need to check if `pos>n-3`. The pointer doesn't move, so you load characters off of `$t0=$a0+$a1`.

No additional registers were needed beyond what was given (`$t0-$t2, $a0, $a1, $v0`).
–1 pt for each additional register used.

No saved registers are needed and no temporary registers are needed once the recursive call returns, so `$ra` is the only register that needed to be saved. **–1 pt** for each register used that did not follow MIPS register conventions. For example, if you used `$s0`, you needed to have saved/loaded it in the prologue/epilogue to avoid losing 2 points instead of 1.

Grading Procedure:

1) Checked prologue and epilogue. **–1 pt** for not saving `$ra`. As noted, **–1 pt** for each additional register saved (same for around the recursive call).
2) Checked choice of recursion implementation. **–1 pt** for not incrementing `$a1` (function prototype returns an `int`, not a `char *`).
3) Checked correct setting of `$t0` at `chk` label based on recursion choice.
4) Checked characters (84 for 'T' and 71 for 'G'). These were on the MIPS Green Sheet or could have been counted from 65 = 'A'. **–1 pt** if swapped (check for 'AGT' instead of 'ATG').
5) Checked return value if start codon found. Needed to be `$a1` (return `int`, not `char *`).
6) <u>Bounds checking</u>:

   –1 pt if jumped to `ret` without setting `$v0` to -1 (-1 indicates start codon not found, as noted in comments).

   –2 pts if logic was inverted (returned -1 when should have jumped to `chk` and vice-versa), `chk` was unreachable, or used check for other recursion method.
7) **–1 pt** for each additional register used and register use that didn't followed register conventions.