University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Science

Spring 2004 Instructor: Dan Garcia 2004-05-22

☺ CS61C Final ☺

Last Name	
First Name	
Student ID Number	
Login	`cs61c-`
The name of your TA (please circle)	Alex Chema Jeremy Paul Roy
Name of the person to your Left	
Name of the person to your Right	
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. ***(please sign)***	

Instructions

You are allowed to use two 8.5" x 11" double-sided handwritten pages of notes. No calculators are allowed. This booklet contains questions M0-M4 and F1-F5 on 10 numbered pages (including the cover page) and 3 duplicated pages from P&H and K&R. Write all your answers on this exam; **show all work and do not hand in other pieces of paper**. Question M0 (+1 points if correct) involves filling in the front of this page and putting your name & login on every *sheet* of paper. You have 3 hours to complete the exam. Good skill!

Problem	**M0**	**M1**	**M2**	**M3**	**M4**	**Total**
Minutes	0	15	15	15	15	60
Max Score	**1**	**11**	**11**	**11**	**11**	**45**
Your Score						

Problem	**F1**	**F2**	**F3**	**F4**	**F5**	**Total**
Minutes	24	24	24	24	24	120
Max Score	**18**	**18**	**18**	**18**	**18**	**90**
Your Score						

Overall Final Exam Score:	

Question M1: Numbers (11 Points – 15 minutes)

a) In *two's complement* addition, how many of the 16 possible combinations of inputs to a 2-bit adder result in *overflow*? Assume no carry in. (3 points)

b) We're going to use a nibble (4 bits) to represent a floating point number as follows: SEEM (1 bit for Sign, 2 for Exponent, 1 for Mantissa), with bias=1. We'll follow the IEEE standard to reserve the smallest exponent for 0 & *denorms* (assume implicit exponent = 0), and the largest for NANs and ± ∞. Fill in the table below: (8 points)

Description	**Number that the floating point notation on the right represents (in Decimal)**	**Binary representation of the floating point notation (in Hex)**
Most negative # (that is not -∞)		
Zero	0	`0x0`
Smallest positive #		
Next-smallest positive #		

Name: ____________________________ Login: `cs61c-`____

Question M2: Memory & C (11 Points – 15 minutes)

a) How many *bytes* are used in the *static area*, *stack* and *heap* as a result of line `8`? For the heap, we're asking for the *requested* amount. Assume the arguments are passed in registers, as usual. (4 points)

```
0   struct foo {
1      int  iarray[2];
2      char carray[4];
3      int* parray[2];
4      struct foo *next;
5    };
6
7   int main() {
8      struct foo *myFoo = (struct foo *) malloc (2 * sizeof(struct foo));
9   }
```

Static area	Stack	Heap

b) For a lab you wrote something similar to the following code: (`s1,s2` are non-empty strings)

```
char  *c1ptr;
char  *c2ptr;
c1ptr = (char *) malloc ( sizeof(char) * (strlen(s1) + 1) );
c2ptr = (char *) malloc ( sizeof(char) * (strlen(s2) + 1) );
```

Your friend wants to combine those `malloc` calls into one:

```
char  *c1ptr;
char  *c2ptr;
c1ptr = (char *) malloc ( sizeof(char) * (strlen(s1) + strlen(s2) + 2) );
c2ptr = (char *) c1ptr + (sizeof(char) * (strlen(s1) + 1));
```

Assume there is *no heap overhead* used for the memory chunk header. Aside from your friend's poor style, in one sentence each provide a single *advantage* and a single *disadvantage* to the approach. If there is none, write NONE. (2 points each)

Advantage of friend's approach	Disadvantage of friend's approach

c) What is the veracity of the following *memory management* statements? Circle **T** or **F**: (1 pt each)

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).

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

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*.

Question M3: C (11 Points – 15 minutes)

You are called upon to find the bugs and predict the output of the following program that was typed into a PC word processor, like `notepad` or `textedit`, so we can't trust the indenting. Specify actual or potential errors (not warnings) at *compile-time* (e.g., syntax) or *run-time* (e.g., out-of-bounds access) for lines `(a)-(d)`. Concisely state your reasons (and suggest a fix) for all errors you find. Finally, give the output of line `(e)`, assuming the compiler and the system *ignores all the errors it finds*. (2 pts each)

```
struct dlist {
   int value;
   struct dlist *next;
};

int main() {
   struct dlist **p;
   int i,j;
   p = (struct dlist **) malloc (10 * sizeof(float *));           /* line (a) */
   for (i=0; i<10; i++)
      *(p+i) = (struct dlist *) malloc (20*sizeof(struct dlist)); /* line (b) */

   for (i=0;i<10; i++) {
      for(j=0; j<20; j++) {
         if(j<19)
         p[i][j].next = *(p[i][j+1]);     /* line (c) */
         (*(p+i)+j+1)->value = i*j;       /* line (d) */
      }
   }

   printf("%d",p[5][15].value);           /* line (e) */

   return 0;
}
```

Line	Error? Circle "NO" or "CT" (for compile-time) or "RT" (for run-time)	If it *is* an error, briefly state the reason (and suggest a fix)
(a)	**NO CT RT**	
(b)	**NO CT RT**	
(c)	**NO CT RT**	
(d)	**NO CT RT**	

What is printed by line *(e)* (assume the compiler & system *ignores all the errors it finds*)? ______

Name: ___________________________ Login: cs61c-____

Question M4: MIPS (11 Points – 15 minutes)

The MIPS instruction set architecture (ISA) is going to be updated, and the Recording Industry Association of America (RIAA) has asked the designers to add an anti-piracy feature. The new instruction, `riaa`, will search for copyrighted material in your hard disk. In order to make space for the new instruction, they decided to remove variable logical shifts from the language. Thus, `sllv` will disappear from MIPS and its (R-type) opcode & function will be used for `riaa`. Recall the format: `sllv $rd, $rt, $rs` which shifts the value in register `rt` to the left by the # of bits specified by register `rs` and puts the result in destination register `rd`.

We wish to maintain MAL backward-compatibility, so we devise a clever solution in which we consider `sllv` to be a MAL *pseudoinstruction* & translate it into a *set* of TAL instructions that do the same thing.

The key idea is that we're going to use self-modifying code! We'll stuff the low-order few bits from the contents of the `rs` register into the `shamt` field of a vanilla `sll` instruction (#11 below). We're telling you *how* to do it, and your job is to figure out *where* to put the temporary values (and a few other details). Here are the `sll` and `sllv` instructions, and the field widths:

	6	5	5	5	5	6
sll	0	rs	rt	rd	shamt	0
sllv	0	rs	rt	rd	0	4

Thus, the single `sllv` instruction (`$dst`, `$src` and `$shamtreg` are *abstractions* for the actual registers)

```
sllv $dst, $src, $shamtreg
```

…would translate to the following *set* of TAL instructions: (fill in the blanks, we've shown comments)

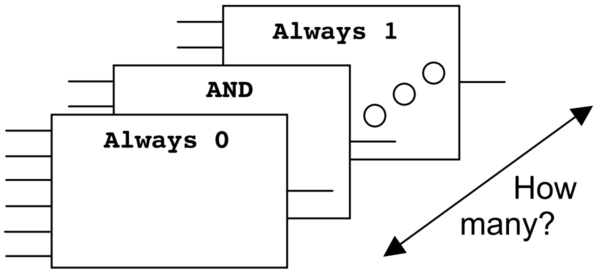
```
01            add  ____, $0, $shamtreg          # copy $shamtreg so we don't alter it
02            andi ____,____,_____              # The shamt has a maximum size!
03            ____________________              # shift the shamt to the right location
04            lui  $at, shiftLby0(upper)  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

This concluses the Midterm-level questions. You're 2/3rds of the way through!

Question F1: Verilog and Logic (18 Points [6 each] – 24 minutes)

a) We've seen 6-input, 1-output logic gates like ANDs, ORs, NORs, NANDs, XORs, XNORs, etc. Radio Shack needs to have one copy of *ALL* the possible 6-input, 1-output logic gates there are possible (even stupid degenerate ones, like the one that ignores all the inputs and always outputs 0), and each costs $1. How much does Radio Shack need to spend? Don't write your answer as an expression, work it out and write it in computer-ese, ala 1K$, 64M$, 2G$, etc.

____



b) Given the following sum-of-products expression for `Foo`, simplify this expression to a sum of products of (at most) 3 two-variable terms (e.g., `AB + CD + AD`): What's a good name for `Foo`?

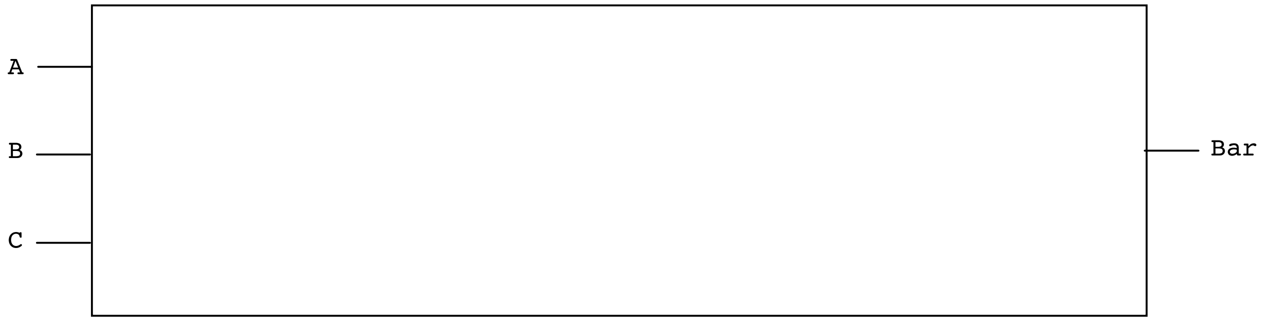
$\texttt{Foo} = A*\overline{B}*\overline{C} + \overline{A}*\overline{B}*\overline{C} + \overline{A}*\overline{B}*C + \overline{A}*B*\overline{C}$

`Foo =` ________________________, and is better named "_________________________________".

c) Given the following simplified sum-of-products expression for `Bar`, given `A`, `B` and `C`:

$\texttt{Bar} = A*\overline{B} + A*C + \overline{B}*C$

Draw the circuit diagram for the `Bar` function. You are required to instantiate *one 1-bit multiplexor*, and plug `C` into its select line (label its 0 and 1 inputs). You may only use basic gates AND, OR & NOT. Full credit will only be given to solutions with a mux and *3* basic gates.



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

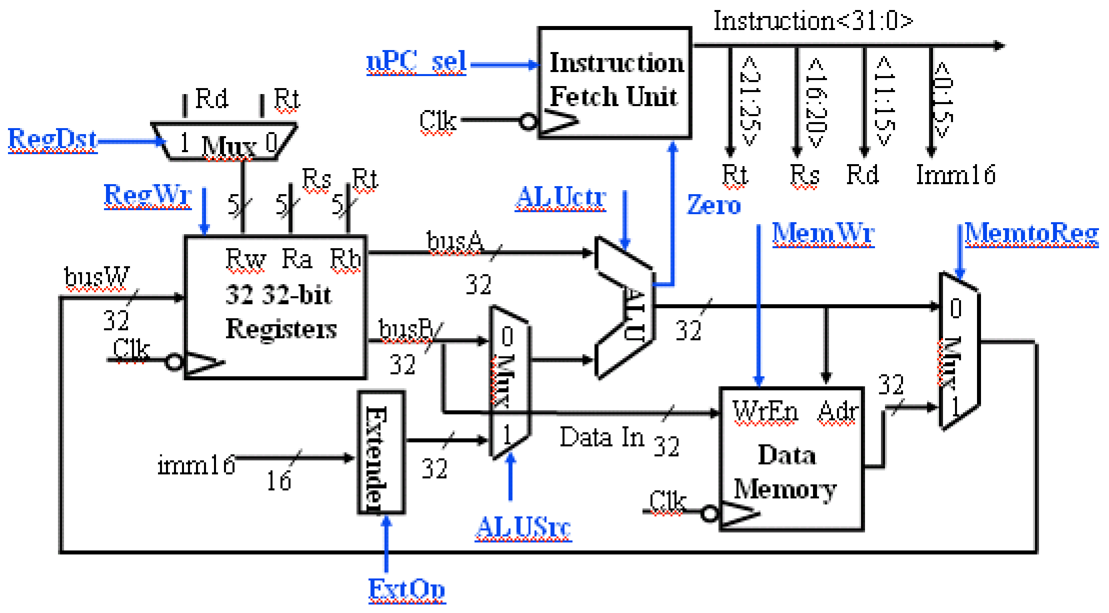
Scratch space

Name: ______________________________ Login: `cs61c-____`

Question F2: Control and Datapath (18 Points – 24 Minutes)

Modify the following single cycle MIPS datapath diagram to accomodate a new instruction `swai` (store word then auto-increment). The operation performs the regular `sw` operation, then post-increments the `rs` register by 1. Your modification may use simple adders, mux chips, wires, and new control signals. You may replace original labels where necessary. Recall the RTL for `sw` is:
`Mem[ R[rs] + SignExt[imm16] ] = R[rt]; PC=PC+4`, & that `sw` (and `swai`) has the following fields:

Opcode	Rs	Rt	Immediate



a) **Modify the picture above** and list your changes below. You may not need all the boxes. Please write them in "pipeline stage order" (i.e., changes affecting IF first, MEM next, etc)

(A)	
(B)	
(C)	
(D)	
(E)	
(F)	

b) We also wish to do the same thing with `lw`, namely create `lwai`. Will this work? Circle YES or NO and argue your point in one sentence. (3 points)

YES **NO** because

Question F3: Pipelining (18 points, 24 minutes)

Given the following MIPS code snippet (note that instruction #6 could be anything):

```
loop:
1     addi $t0, $t0, 4
2     lw   $v0,  0($t0)
3     sw   $v0, 20($t0)
4     lw   $s0, 60($t0)
5     bne  $s0, $0, loop
6                           ## ← The following instruction could be anything!
```

a) Detect hazards and insert `no-op`s to insure correct operation. *Assume no delayed branch, no forwarding units and no interlocked pipeline stages*. Your answer on the right should take the form of pair(s) of numbers: `num@location` – indicating `num` `no-op`s should be placed at `location`. E.g., if you wanted to place 6 `noop`s between lines 2 and 3 (i.e., `location=2.5`) and 8 `noop`s between lines 5 and 6 (i.e., `location=5.5`), you would write: "`6@2.5, 8@5.5`". (6 points)

Scratch space

b) Now, reorder/rewrite the program to maximize performance. *Assume delayed branch and forwarding units, but no interlocked pipeline stages*. For unknown reasons, the first instruction after the `loop` label *must* be the `addi`. Feel free to insert `no-op`s where needed. You should be able to do it using 6 instructions per loop (easier, half credit) or only 5 (hard, full credit). (12 pts)

```
______________________________  ## Extra instructions before the loop if necessary

______________________________  ## Extra instructions before the loop if necessary

loop:
1     addi $t0, $t0, 4

2 ______________________________

3 ______________________________

4 ______________________________

5 ______________________________

6 ______________________________
                                  ## ← The following instruction could be anything!
```

Name: ___________________________ Login: `cs61c-`____

Question F4: Caches & VM (18 points, 24 minutes)

This is for questions (a) and (b). 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 following? Show your work on the right. (3 points)

Tag	Index	Offset

b) For each of the following changes to the initial conditions above, indicate how these bits (i.e., the width of these fields) shift around. E.g., if a bit field stays the same, write "0", if a bit field *increases* by 5, write "+5", if a bit field *decreases* by 1, write "-1". (6 points)

Change	**Tag**	**Index**	**Offset**
Double the cache size (from 64 KB to 128 KB)			
Double the word size (from 32 bits to 64 bits)			
Change the associativity to fully associative			

c) A rich student who didn't take CS61C decides to splurge and buy 4 GB of rocket-fast RAM for their 32-bit MIPS system. They think to themselves: "Why should I turn on Virtual Memory?". What's the strongest argument (one sentence max) for turning it on? (3 pts)

d) Suppose a computer has a 32-bit virtual address space, 64 MB physical memory and a 4 KB page size. Based on this information, answer the following questions. Show your work. (6 pts)

How many virtual pages are there?	
How many physical pages are there?	
Assuming a one-level page-table design with each page table entry consuming 1 word, what is the size of the page table, in bytes?	