University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2006 Instructor: Dan Garcia 2006-12-14

☺ CS61C FINAL EXAM ☺

Last Name	ANSWER
First Name	KEY
Student ID Number	
Login	cs61c-
Login First Letter (please circle)	a b c d e f g
Login Second Letter (please circle)	a b c d e f g h i j k l m n o p q r s t u v w x y z
*The name of your **LAB** TA (please circle)*	Scott Aaron David P. Sameer David J.
Name of the person to your Left	
Name of the person to your Right	
*All the work is my own. I have 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 (Read Me!)

- This booklet contains 9 numbered pages including the cover page. Put all answers on these pages (feel free to use the back of any page for scratch work); don't hand in any stray pieces of paper.
- Please **turn off** all pagers, cell phones & beepers. Remove all hats & headphones. Place your backpacks, laptops and jackets at the front. Sit in *every other* seat. Nothing may be placed in the "no fly zone" spare seat/desk between students.
- Fill in the front of this page and put your name & login on every sheet of paper.
- You have 180 minutes to complete this exam. The exam is closed book, no computers, PDAs or calculators. You may use two pages (US Letter, front and back) of notes, plus the green reference sheet from COD 3/e.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided. "IEC format" refers to the mebi, tebi, etc prefixes. You have 3 hours...relax.
- **You must complete ALL THE QUESTIONS, regardless of your score on the midterm.** Clobbering only works from the Final to the Midterm, not vice versa.

Problem	**M1**	**M2**	**M3**	**Ms**	**F1**	**F2**	**F3**	**F4**	**Fs**	**Total**
Minutes	20	20	20	60	30	30	30	30	120	180
Points	**10**	**10**	**10**	**30**	**22**	**22**	**22**	**24**	**90**	**120**
Score										

Name: ______________________________ Login: cs61c-____

Midterm Revisited

M1) "Son of a bits…" (10 pts, 20 min)

a) How many bits does it take to address N things? (hint: you may use the `floor` or `ceiling` function)

`ceiling(log2(N))`

Recall the `quarter` definition from midterm (skip this paragraph if you remember it):

> Early processors had no hardware support for floating point numbers. Suppose you are a game developer for the original 8-bit Nintendo Entertainment System (NES) and wish to represent fractional numbers. You and your engineering team decide to create a variant on IEEE floating point numbers you call a `quarter` (for quarter precision floats). It has all the properties of IEEE 754 (including denorms, NaNs and ± ∞) just with different ranges, precision & representations. A `quarter` is a single byte split into the following fields (1 sign, 3 exponent, 4 mantissa): `SEEEMMMM`. The bias of the exponent is 3, and the implicit exponent for denorms is -2

You're also familiar with the *Fixed-Point Representation*, where the binary point is always in the same place so there's no need to store the exponent. E.g., you could imagine splitting the Nintendo's byte into two nibbles with the left nibble representing the unsigned whole number component (W), and the right representing the fractional component (F): `WWWW.FFFF`. Thus the bit pattern `0xa8` would be interpreted as the unsigned fixed-point value `0xa.8` = `0b1010.1000` = `10.5`$_{10}$. As a systems designer, you could choose to interpret a byte any way you want, so you could change the point location (e.g., `WW.FFFFFF` or `WWWWWWW.F`) to suit your needs.

b) One of your games involves velocities that always fall in the range of [10, 15), i.e., $10 \le v < 15$. If you only have a *single NES byte*, you're asked to design **a novel representation** to encode a velocity (assume the hardware can handle whatever you do). It should be **better than** a `quarter`, fixed-point, and any 8-bit encoding we've discussed! What is "better"? You will be judged on four criteria (check the box to the left of the ones you think you satisfy, listed in decreasing priority order). Explain your **de**coding below on the left (how to go from a bit pattern `b` to a velocity `v`), and on the right show the bit patterns that would result from encoding numbers closest to 10, 12.5 and 15 as well as the velocity each bit pattern actually represents.

____√____ Most bit patterns encoding the most different numbers in [10, 15)
____√____ You have bit patterns that are as close as possible to 10, 12.5, and 15
____√____ Uniform spacing between numbers in [10, 15) is better than non-uniform
____√____ Simplicity

My scheme has ___256___ NES byte bit patterns representing the range [10, 15). Here's how I go from a bit pattern `b` to a velocity `v`: (if you'd like, you may write it mathematically… `v` as a function of `b`).	*Number closest to…*	*…and its bit pattern*	*…and the velocity it represents*
`10 + (unsigned char) b * (5.0/256)`	10	0b00000000	10
	12.5	0b10000000	12.5
	15	0b11111111	10 + 255 * (5/256)

Name: ____________________________ Login: cs61c-____

M2) "Those are some big numbers you got there…" (10 pts, 20 min)

A bignum is a data structure designed to represent large integers. It does so by abstractly considering all of the bits in the `num` array as belonging to one very large integer. This code is run on a standard 32-bit MIPS machine, where a `word` (defined below) is 32 bits wide and `halfword` is 16 bits wide.

```
typedef unsigned int   word;
typedef unsigned short halfword;
typedef struct bignum_struct {
    int length;     // number of words
    word *num;      // the actual data
} bignum;
```

This function shows how bignums are used:

```
void print_bignum(bignum *b) {
   printf("0x"); // Print hex prefix
   for (int i = b->length-1; i>=0; i--)
      printf("%08x", b->num[i]);
}
```

a) Is the ordering of `word`s in the `num` array BIG or √LITTLE endian? (circle one)

b) How many bytes would be used in the *static*, *stack* and *heap* areas as the result of lines `1`, `3` and `4` below? **Treat each line independently!** E.g., For line `3`, don't count the space allocated in line `1`.

```
1 bignum biggie;
2 int main(int argc, char *argv[]) {
3   bignum bigTriple[3], *bigArray[4];
4   bigArray[1] = (bignum *) malloc (sizeof(bignum) * 2);
```

	static	***stack***	***heap***
`Line 1`	8	0	0
`Line 3`	0	3*8 + 4*4 = 40	0
`Line 4`	0	0	2*8 = 16

b) Complete the `add` function for two bignums, which you may assume **are the same** `length`**.** Our C compiler translates `z = x + y` (where `x,y,z` are `word`s) to `add` (not `addu`, as is customary) and thus could generate a hardware (HW) overflow we don't want, as we're running on untrusted HW. Your code should be written so that `word`s never overflow in HW (so we do all adding in the `halfword`).

```
void add(bignum *a, bignum *b, bignum *sum, word carry_in, word *carry_out) {

   // reserve space for num array. Remember a and b are the SAME length...

   sum->num =  (word *) malloc (a->length * sizeof(word))

   for (int i=0; i < a->length; i++) {  // word-by-word do addition of lo, hi halfwords

      // add lo halfwords of a,b

      word lo     = (a->num[i]&0xffff) + (b->num[i]&0xffff) + carry_in;

      // add hi halfwords of a,b (but in the safe, low halfword area so no HW overflow)

      word hi     = (a->num[i] >> 16 ) + (b->num[i] >> 16 ) + (lo >> 16);

      // combine low and hi halfwords (put back in their places), like a lui-ori

      sum->num[i] = (hi << 16) | (halfword) lo;

      // what's the carry_in for the next word?

      carry_in    = hi >> 16;
   }
   sum->length = a->length;
   *carry_out  = carry_in;
}
```

Name: ______________________________ Login: cs61c-____

M3) "What's the MIPS is going on here?" (10 pts, 20 min)

a) Given the MIPS code below, write the equivalent (from a functional point of view) C function below in the structure we've provided. When you're writing the C code, you can assume that `Mystery` will be called fewer than 100 times. (Later questions ask what happens when it's called more times.) Feel free to add comments to help your disassembly. You may assume `la` will *always* be expanded into a `lui/ori` pair that fills up (clobbers) the `nop`.

```
Mystery:  la $t0, Mystery
          nop
          lw $t1, 20($t0)
          addiu $t1, $t1, 1
          sw $t1, 20($t0)
          addiu $v0, $0, 0
          jr $ra
```

```
// Mystery called < 100 times
short timesCalled = 0;

 _____________short_ Mystery() {

   return ++timesCalled;

}
```

b) In one sentence, explain what this MIPS code does.

It returns the number `Mystery` has been called (up to a point, see below).

c) What is the most times this function can be called so that it still does what you described in part (b)? (It can be left as an expression)

$2^{15}-1$

d) What will it return (exactly, but it may be left as an expression) if it is called one more time?

It will return the negative number (0xFFFF8000 = -2^{15}).

e) What will happen if it is called twice as many times as in (d)? Will it crash? Hang forever? What's returned, if anything? Describe the effect from the caller's standpoint; be explicit.

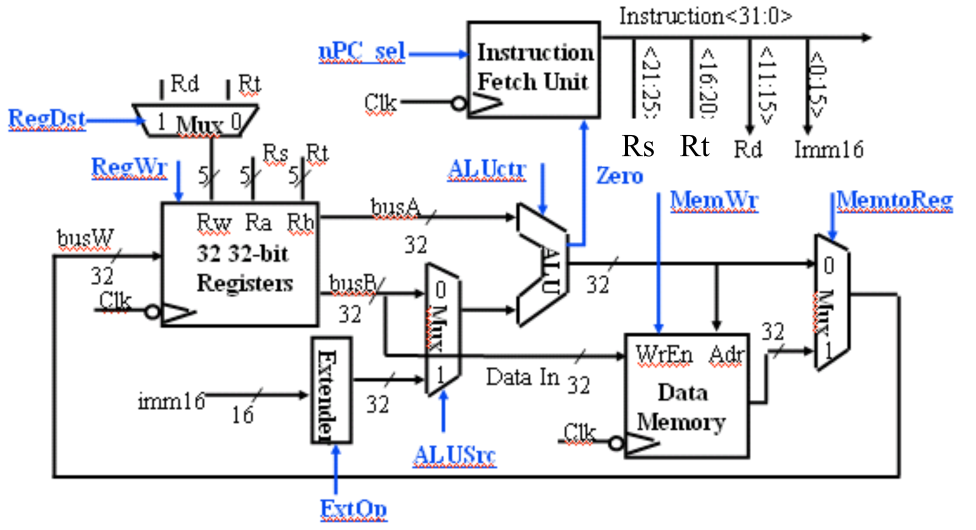
Since it now modifies `$v1` instead of `$v0` (and `$v0` doesn't get touched) the caller will get a "random" = "garbage" return value, probably whatever was left in `$v0` from the return value of the most recent function that was called.

Name: ______________________________ Login: cs61c-____

Post-Midterm Questions

F1) "The Datapath less traveled…" (22 pts, 30 min)

On the right is the single-cycle MIPS datapath presented during lecture. Your job is to modify the diagram to accommodate a new MIPS instruction. Your modification may use simple adders, shifters, mux chips, wires, and new control signals. If necessary, you may replace original labels.



We want to add a new MIPS instruction so that the following C statement (`p` is a pointer to an `int`, and `CONSTANT` is small and can be negative) could be performed in <u>one</u> I-type MAL MIPS instruction: `*p = CONSTANT;`

a) Make up the syntax for the I-type MAL MIPS instruction (call it `sc` for "store constant") that does it (show an example if the pointer lives in `$v0` and the `CONSTANT` is `42`). On the right, show the register transfer language (RTL) description of `sc`.

Syntax: ____`sc 42 ($v0)`__________ RTL: _`R[rs]<-signExtImm; PC<-PC+4`____________

b) For a larger `CONSTANT` (say `0xFAB5BEEF`), to what *exact* TAL instructions would the MAL above expand?

___`lui $at, 0xFAB5; ori $at, 0xBEEF; sw $at 0($v0)`________________________

c) **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)

(i)	Add a 2-input mux (dataIn={busB,Extender}) to select the memory Data In. ...OR... Take the ALUSrc Mux, and have another output (tied to DataIn) which is the **opposite** of what ALUSrc chooses. When ALUSrc is 1, Data In gets busB and the normal output gets the imm16 for the offset (as in sw). When ALUSrc is 0, Data In gets imm16 (as in **`*p=CONSTANT`**) and the normal input is fed to the ALU which is ignored by the following mux.. You can also have the control always set RT to 0 (regardless of what it is in the instruction).
(ii)	Add a 2-input mux (memAdr={ALU,busA}) to select the memory Adr with busA on input 1 ... OR ... Rewrite the ALU so that it can take an ALUSrc command to "pass A unchanged"
(iii)	
(iv)	

d) We now want to set all the control lines appropriately. List what each signal should be (an intuitive name or {0, 1, x = don't care}). Include any *new* control signals you added.

RegDst	RegWr	nPC_sel	ExtOp	ALUSrc	ALUctr	MemWr	MemtoReg	memAdr	memDataIn

x	0	+4	Sign	x	x	1	x	busA	Extender

Name: ____________________________ Login: cs61c-____

F2) "*Don't let me fault*" (22 pts, 30 min)

The specs for a MIPS machine's memory system that has *one* level of cache and virtual memory are:

- 1MiB of Physical Address Space
- 4GiB of Virtual Address Space
- 4KiB page size
- 16KiB 8-way set-associative write-through cache, LRU replacement
- 1KiB Cache Block Size
- 2-entry TLB, LRU replacement

The following code is run on the system, which has no other users and process switching turned off.

```
#define NUM_INTS 8192                                  // This many ints...
int *A = (int *)malloc(NUM_INTS * sizeof(int)); // malloc returns address 0x100000
int i, total = 0;
for(i = 0; i < NUM_INTS; i += 128) A[i]    = i;
for(i = 0; i < NUM_INTS; i += 128) total += A[i];  // SPECIAL
```

a) What is the `T:I:O` bit breakup for the cache (assuming byte addressing)? ___9_:__1__:__10_

b) What is the VPN : PO bit breakup for VM (assuming byte addressing)? ___20___:___12___

For the following questions, only consider the line marked "*SPECIAL*". Your answer can be a fraction.

c) Calculate the hit percentage for the cache ½ = 50%

d) Calculate the hit percentage for the TLB 7/8 = 87.5%

e) Calculate the page hit percentage for the page table 100%

Show all your work below...

Name: ______________________________ Login: `cs61c-`____

F3) "These Pipes are Clean…" (22 pts, 30 min)

Consider a processor with the following specification:

- Standard five (5) stage (F, D, E, M, W) pipeline.
- No forwarding.
- Stalls on all data and control hazards.
- Non-delayed branches
- Branch comparison occurs during the second stage.
- Instructions are not fetched until branch comparison is done.
- Memory CAN be read/written on same clock cycle.
- The same register CAN be read & written on the same clock cycle.
- No out-of-order execution
- "Dumb" control that does not optimize for "always-branch" conditional branches

a) Count how many cycles will be needed to execute the code below and write out each instruction's progress through the pipeline by filling in the table below with pipeline stages (F, D, E, M, W).

```
add $t1, $t2, $t3
xor $t1, $t4, $t5
lw  $t3, 0($t1)
beq $t3, $t3, 1
lw  $t5, 0($t3)
xor $t4, $t5, $t6
add $t5, $t5, $t4
```

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Inst 1	F	2 3 D	E	M	W 1																				
Inst 2		F	4 5 D	E	M	W 1																			
Inst 3			F	F	F	1 D	E	M	W 3																
Inst 4				F	F	F	F	F	3 D =	E	M	W													
Inst 5										F	56 D	E	M	W 4											
Inst 6											F	F	F	4 D	E	M	W 5								

b) Considering the following three *changes*, fill in the table again:

- Our processor now forwards values
- Interlocks on load hazards
- "Intelligent" control that optimizes for "always-branch" conditional branches

Cycle→	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Inst 1	F	2 3 D	E	M	W 1																				
Inst 2		F	4 5 D	E 1	M	W 1																			

Inst 3			F	D	1 E	M 3	W																		
Inst 4				F	3 D	E	M	W																	
Inst 5					F	5 6 D	E 4	M	W 4																
Inst 6						F	5 D	4 E	M	W 5															

Name: ______________________________ Login: cs61c-____

F4) The CS61C Variety Pack… (24 pts, 30 min)

The table on the right is only used for questions (a)-(c). Given the following instruction mix and CPI$_i$ for each instruction$_i$:

Instruction$_i$	Frequency$_i$	CPI$_i$	*Scratch space*
ALU	25%	1	0.25
Load	35%	3	1.05
Store	10%	5	0.50
Branch	30%	4	1.20

a) What is the *overall* CPI? 3.0

b) If Stores were free (CPI=0), how many times faster would the CPU be? 6/5 = 1.2

c) If you could make one instruction type twice as fast, what should it be? Branch

d) What problem prevents us from easily transitioning to quad-, 8-, or more-core processing? (The proverbial fly in the ointment ☺) Cache coherency

e) What RAID # should be used, if you want to maximize hard drive read speed, want the most space possible, and can use never-fail disks? 0

A large computing task is at hand, but thankfully, we've got a cluster of computers at our disposal. Assume that the `for` loop is fully parallelizable, but `serial()` is not. We run this code:

```
#define ITERATIONS 96
int s = serial(); // 40 cycles to complete
for (int n = 0; n < ITERATIONS; n++)
   parallel(s,n); // 10 cycles per loop
```

f) How many times faster are we if we parallelize the code over *many* machines? 1000/50 = 20

g) Match the following items. Some items on the right will not be used. or mav be used

G	Makes more efficient use of available disk area
J	The basis of network abstraction
Q	This guarantees delivery over a network
M	Work per unit time
F	Time to complete a single task
I	Bigger blocks take advantage of this
B	All caches take advantage of this
R	"It's getting harder to build a new chip fab plant!"

A) LRU
B) Temporal Locality
C) Synchronization
D) Write-back
E) Full-duplex
F) Latency
G) Constant Bit Density
H) Amdahl's Law
I) Spatial Locality
J) Encapsulation
K) Fragmentation
L) Synchronization
M) Throughput
N) Parallelization
O) AMAT
P) Constant angular velocity
Q) Ack
R) Rock's law
S) Superscalar
T) Pipelining
U) Superparamagnetism
V) Polling (not David)