Name:_____ Login:_____

University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

CS61c
Summer 2001

Aner Ben-Artzi
Jeremy Lin
Mark Marron
Songhwai Oh
Woojin Yu

Final Exam

This is a *closed-book* exam. No calculators please. You have 2 hours. Each question is marked with its number of points.

This exam booklet should have 15 pages. Check to make sure that you have all the pages. Put your name and login neatly on each page.

Show your answers in the space provided for them. Write neatly and be well organized. If you need extra space to work out your answers, you may use the back of previous questions. However, only the answers appearing in the proper answer space will be graded.

Good luck!

Problem	Maximum	Score
1	11	
2	15	
3	25	
4	15	
5	14	
6	20	
Total	100	

Name:_____ Login:_____

MIPS instructions

Important – please note: The MIPS instructions shown in this table are the ones that you must use on the entire exam. **Do not use any instructions that are not in this table. If you use any instructions not listed below, you will lose points.**

The columns under "format" show the bit fields of the instructions. The number in the parentheses following each name is the number of bits in that field.

In the table, PC refers to the *program counter*.

You may carefully tear this page from your exam booklet for easy reference.

name	Format						syntax	meaning
	Op(6)	rs(5)	rt(5)	rd(5)	shamt(5)	func(6)		
add	0				0	32	add rd,rs,rt	rd = rs + rt
sub	0				0	34	sub rd,rs,rt	rd = rs - rt
and	0				0	36	and rd,rs,rt	rd = rs AND rt
or	0				0	37	or rd,rs,rt	rd = rs OR rt
sll	0	0				0	sll rd,rt,shamt	rd = logical shift rt left shamt bits
srl	0	0				2	srl rd,rt,shamt	rd = logical shift rt right shamt bits
slt	0				0	42	slt rd,rs,rt	if rs<rt set rd=1 else rd=0
jr	0		0	0	0	8	jr rs	PC=rs
	Op(6)	rs(5)	rt(5)	immediate(16)				
addi	8						addi rt,rs,immed	rt = rs + immed
andi	12						andi rt,rs,immed	rt = rs AND immed
ori	13						ori rt,rs,immed	rt = rs OR immed
lw	35						lw rt,immed(rd)	rt = MEMORY[rd+immed]
sw	43						sw rt,immed(rd)	MEMORY[rd+immed] = rt
lui	15						lui rt,immed	rt = immed shifted left 16 bits
beq	4						beq rs,rt,label	branch if equal
bne	5						bne rs,rt,label	branch if not equal
	Op(6)	target address(26)						
j	2						j label	jump
jal	3						jal label	jump and link

Name:_____ Login:_____

Question 1(11 points)

Part (a) was worth 3 points, and the rest were worth 2 points each.

a. Give following float value in IEEE 754 single precision format (in hexadecimal).

118.125 (dec) = _____ (hex)

Answer. 1.110110001 x 2^6 = **42EC4000** (hex)

b. Give the truth table and the gate diagram for **NOT(P) OR Q**, where P and Q are inputs.

P	Q	Output
0	0	1
1	0	0
0	1	1
1	1	1

c. Describe how the relocation information in an object file is used and name a program that makes use of this information.

Answer. The linker uses the relocation information to help it figure out what parts of an object file to relocate and where to relocate them when it builds the final executable. Loader was also accepted as a program that uses the relocation information.

d. How does the cache block size affect performance?

Answer. Increasing the block size increases performance up to a point, but performance decreases if the block size is too large.

e. List one pro and one con of increasing the size of the TLB.

Answer. Increasing the TLB size tends to decrease TLB misses, but it also increases the complexity of the hardware, makes it harder to make it fast, etc.

Name:_____ Login:_____

Question 2(15 points)

a. In UNIX, directories are special files that contain information about the files in the directory. For example, say you are trying to read the file **final-sol.txt**, which is in a directory called **secret**. Then before the OS can read the file, it has to read the special directory file corresponding to **secret** and find out where **final-sol.txt** is stored on disk.

In order to make accessing the directory files fast, what general area of the disk would be ideal for storing the directory files? Briefly, but specifically, explain why. Assume that no caching of disk reads is involved.

This part was worth 2 points.

Answer. It should be stored on the middle tracks, since the disk heads are closest to these tracks on average. You got 1 point if you said that it should be placed closest to where the disk heads were or something similar.

b. (For this part, use the information and assumptions given in Part a.) On UNIX, your incoming mail is delivered by appending the message contents to a regular file called the *mail spool* (it basically serves as your inbox). So if your username was "**wooj**", new mail might typically be written to the file **/var/mail/wooj** (a file named "**wooj**" in the directory **/var/mail** – remember that directories can contain directories). Suppose that you have *PINE* (your email client) set up to display your inbox on startup (i.e., *PINE* will read the mail spool on startup). Assuming that all other non-disk-related work takes negligible time. You know these disk parameters:

Average seek time : 20 ms
Rotational speed : 3000 RPM
Block size : 4 KB
Transfer rate : 0.4 MB/sec
Overhead from controller, filesystem, etc. : negligible (0 seconds)

and that:

- Your disk is so fragmented, blocks belonging to a file are effectively spread out on disk randomly.
- The three directories accessed on the way to the mail spool are /, **var**, and **mail**. The size of **/** is 4 KB, **var** is 4 KB, and **mail** is 128 KB. Your *mail spool* is 4 MB large.

How long do you expect to wait for *PINE* to start up? Show your work.

This part was worth 6 points.

Answer. From the rotational speed, the average rotational latency is 10 ms. The average seek time is given as 20 ms. Transfer time takes about 10 ms per block. There are 1024 + 32 + 1 + 1 = 1058 blocks total. Since the blocks are distributed randomly on disk, rotational latency and seek time come into play on *every* block. Thus, 1058 · (10 ms + 20 ms + 10 ms) ≈ **42.32 seconds**. Because of slight variations in interpretation of megabytes and kilobytes and such, we also accepted close answers such as 40 or 41 sec, assuming that the general idea of the calculations was correct.

Name:_____ Login:_____

c. If you leave *PINE* open, it will tell you when you get new mail. It does this by periodically checking (polling) to see if your *mail spool* file has changed. You love getting mail, and you are considering switching to MailOS, a UNIX derivative that has the peculiar feature of being able to send a signal (sort of like an interrupt) to inform a program when the user's mail spool has changed. The MailOS version of *PINE* supports this feature. Assume the following:

- Your processor runs at 100,000 cycles/min.
- The faster you find out about new mail, the better, but you want to keep at least 50% of the CPU free for other processes. Aside from that, you don't care about CPU usage.
- You receive 10 emails per minute (you're just that popular). However, the e-mails are not necessarily evenly distributed (i.e., not always 1 message every 6 seconds).
- MailOS sends a signal for every new message received. Signaling takes 4500 cycles per signal.
- You can set *PINE* to poll as often as you want (up to hardware limits of course). Polling takes 2500 cycles per poll.

Which scheme fits your criteria best? Clearly explain why. (*This part was worth 4 points)*

This problem was actually subtler than we originally thought. Few people demonstrated good reasoning on this question. Many people seemed to have trouble understanding that the goal was to optimize how quickly you received mail, not CPU usage (as long as you kept to less than 50% usage). Your score depended solely on your explanation—there are cases to be made for both polling and interrupts. Interrupts take 2.7 seconds to go through, but polling can only be done 20 times a minute, or every 3 seconds. We accepted an answer that interrupts were better, with this explanation. However, if you assume a random email distribution, on average, polling may take only 1.5 seconds for notification, and we accepted this argument for polling. A couple people also noted that many signals might happen all at once, and use up over 50% CPU, so signaling would be unacceptable. Because whether this was possible was not addressed explicitly in the question, this was a valid concern. Basically, you got full credit if you made a clear argument for a particular scheme that was both valid and specific to the question (less than 10 people got full credit). Varying amounts of partial credit were also given.

d. A certain computer has a CPU running at 500 MHz. Your *PINE* session on this machine ends up executing 200 million instructions, of the following mix:

Type	CPI	Freq
A	4	20 %
B	3	30 %
C	1	40 %
D	2	10 %

i) What is the average CPI for this session? (*This part was worth 1 point)*

Answer. (20% · 4) + (30% · 3) + (40% · 1) + (10% · 2) = **2.3**.

ii) How much CPU time was used in this session? (*This part was worth 2 points)*

Answer. 500 MHz implies 2 ns/cycle. So,

(200 x 10^6 instructions) · (2 ns/cycle) · (2.3 cycles/instruction) = **0.92 seconds**.

Name:_____ Login:_____

Question 3(25 points)

a. This question gives you a C program and the corresponding MIPS code. The MIPS code is missing some of the lines, labels, and immediate values. Your job is to fill them in correctly. Each group of lines that is missing doesn't interact with any other missing lines of code, so you can work on them one at a time. The MIPS code follows the C code as closely as possible with no programming tricks or optimizations.

This program takes a list of integers as command line arguments, and prints them out in increasing order.

```
example:
% a.out 22 17 14 45
14 17 22 45
```

```
#include <stdio.h>
struct list {
    int value;
    struct list* next;
};
```

Name:_____ Login:_____

a. This function takes as arguments a pointer to a list, and a pointer to a node. It inserts the node into the correct location in the list, and returns the list with the node now in the correct location.

```
struct list* insert(struct list* the_list, struct list* item) {
    if ((the_list == NULL) || (item->value < the_list->value)) {
         item->next = the_list;
         the_list = item;
    } else {
         the_list->next = insert(the_list->next, item);
    }
    return the_list;
}
```

Fill in the 5 missing instructions and the one missing label.
Use `$s0` for `the_list`, `$s1` for `item`.

```
insert:    add   $sp, $sp, -12     # make room on the stack for 3 words
           sw    $ra, 0($sp)       # store the ra on the stack
           sw    $s0, 4($sp)       # store $s0 and $s1 on the stack
           sw    $s1, 8($s0)
1 pt       addi  $s0, $a0, 0       # move the_list to $s0
1 pt       addi  $s1, $a1, 0       # move item to $s1
           bne   $s0, $zero, else  # if the_list == NULL do the else
           lw    $t0, 0($s0)       # $t0 is the_list->value
           lw    $t1, 0($s1)       # $t1 is item->value
           slt   $t2, $t1, $t0     # $t2 = item->value < the_list->value
           beq   $t2, $zero, else  # if item->val < the_list->val do else
           sw    $s0, 4($s1)       # item->next = the_list
           add   $s0, $s1, $zero   # the_list = item
           j     end               # end of if section
else: 2pt  lw    $a0, 4($s0)       # set arg1 to the_list->next
2 pt       addi  $a1, $s1, 0       # set arg2 to item
           jal   insert            # call insert(the_list->next, item)
           sw    $v0, 4($s1)       # the_list->next = insert()
end: 1 pt  addi  $v0, $s0, 0       # put return value in v0
           lw    $ra, 0($sp)       # restore ra
           lw    $s0, 4($sp)       # restore save registers
           lw    $s1, 8($sp)
           add   $sp, $sp, 12      # restore stack pointer
           jr    $ra               # return
```

The original code had reversed logic for the branches, which made it confusing as to where the else label should go, so everyone got a free point for that.

Name:_____ Login:_____

b. Now we will use the insert function in our program. It looks like this:

```
void main(int argc, char* argv[]) {
    struct list* root = NULL;
    struct list* temp = NULL;
    int i;
    for (i = 1; i < argc; i++) {
          temp = (struct list*)malloc(sizeof(struct list));
          temp->value = atoi(argv[i]);
          root = insert(root, temp);
    }
    temp = root;
    while (temp != NULL) {
          printf("%d ", temp->value);
          temp = temp->next;
    }
}
```

Fill in the MIPS code on the following page to correspond to the main function printed above:

Name:_____ Login:_____

Use $s0 for argc, $s1 for argv, $s2 for root, $s3 for temp, and $s4 for i.
Fill in the 9 missing instructions, and 7 missing immediates.

```
main:       addi   $sp, $sp, -24        # make room on the stack for 6 words
            sw     $ra, 0($sp)          # store the ra on the stack
            sw     $s0, 4($sp)          # store s0-s4 on the stack
            sw     $s1, 8($sp)
            sw     $s2, 12($sp)
            sw     $s3, 16($sp)
            sw     $s4, 20($sp)
            move   $s0, $a0             # move argc into $s0
            move   $s1, $a1             # move argv into $s1
1 pt        addi   $s2, $0, 0           # root=NULL
1 pt        addi   $s3, $0, 0           # temp=NULL
            addi   $s4, $zero, 1        # i = 1
for_loop:   blt    $s4, $s0, end_for    # check of for loop
            addi   $a0, $zero, 8        # argument for malloc
            jal    malloc
            add    $s3, $v0, $zero      # temp = malloc()
            add    $t0, $s1, $s4        # t0 is argv+I
2 pt        lw     $a0, 0($t0)          # $a0 is argv[I]
            jal    atoi                 # atoi(argv[i])
2 pt        sw     $v0, 0($s3)          # temp->age=atoi()
            add    $a0, $s2, $zero      # arg1 is root
            add    $a1, $s3, $zero      # arg2 is temp
            jal    insert               # insert(root,temp)
            add    $s2, $v0, $zero      # root = insert()
            addi   $s4, $s4, 1          # i++ in for loop
            j      for_loop             # go to top of for loop
            add    $s3, $s2, $zero      # temp = root
while: 2pt  beq    $s3, 0, end_w        # check for while loop
            la     $t0, string1         # get pointer to format string
            lw     $t2, 0($s3)          # t2 is temp->value
1 pt        addi   $sp, $sp, -8         # make room on stack for args
1 pt        sw     $t0, 0($sp)          # arg1 is pointer to format
1 pt        sw     $t2, 4($sp)          # arg2 is temp->value
            jal    printf               # printf("%d ",temp->value)
            addi   $sp, $sp, 8          # restore stack back to normal
end_for:    lw     $s3, 8($s3)          # temp = temp->next
            j      while                # go to top of while loop
end_w:      addi   $sp, $sp, 24         # restore stack
            lw     $ra, -24($sp)        # restore $ra
1 pt        lw     $s0, -20($sp)        # restore $s0-$s4
for all     lw     $s1, -16($sp)
of these    lw     $s2, -12($sp)
            lw     $s3, -8($sp)
            lw     $s4, -4($sp)
            jr     $ra                  # return
```

Name:_____ Login:_____

c. Convert the following lines into binary machine code. Please use hexadecimal notation.

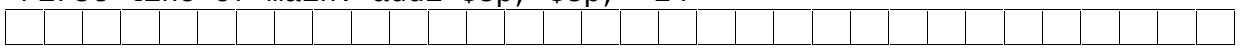
Each part here was worth 2 points.

Assume the address of

```
        add    $a0, $s2, $zero      # arg1 is root
```

is 0x1FFFFFF0

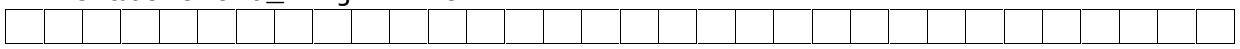
```
First line of main: addi $sp, $sp, -24
```



This part was worth 2 points.

final answer in hex: **23BDFFE8**

```
line above end_w: j while
```



This part was worth 2 points.

final answer in hex: **08000003**

d. Knowing that the address of

```
        add    $a0, $s2, $zero      # arg1 is root
```

is 0x1FFFFFF0, describe the problem that would occur with the instruction "j for_loop" in main?

This part was worth 1 point.

Answer. Everyone got a point for this part, because of the typo we originally had that gave the address as 0x1FFFFFE0, in which case there was no problem with the instruction. The problem that would have occurred is that a jump takes the highest four bits of the current PC to figure out the complete target address, but in this case, the top four bits of the target will become different from that of the current PC.

Name:_____ Login:_____

Question 4(15 points)

Our cache has:

32-byte cache, 4 byte blocks and is 2-way set associative.

The policies are:

Write Back, LRU replacement.

Assume physical memory is as follows:

Address in hex	Address in decimal	Value
0x4	4	w
0x5	5	x
0x6	6	y
0x7	7	z
…	...	…
0x14	20	a
0x15	21	b
0x16	22	c
0x17	23	d
…	...	…
0x24	36	p
0x25	37	q
0x26	38	r
0x27	39	s
…	...	…

a. For the each of the following instructions, indicate whether it is a hit or miss.

This part was worth 5 points.

`lb $t0, 21($0)`	**miss**
`lb $t0, 5($0)`	**miss**
`sb $t0, 22($0)`	**hit**
`lb $t0, 4($0)`	**hit**
`lb $t0, 37($0)`	**miss**

b. Fill in the following table to indicate what values are in the cache at the end of this sequence. Include values for data, tag, LRU bit, valid bit and the dirty bit.

This part was worth 8 points.

Cache Set (Index)	Valid Bit	Dirty Bit	LRU Bit	Tag	Data
0	0	0			
	0	0			wxyz
1	1	0	0	0	
	1	0	1	2	pqrs
2	0	0			
	0	0			
3	0	0			
	0	0			

c. Finally indicate any changes to the main memory.

This part was worth 2 points.

Answer. Memory location 22 holds x.

Name:_____ Login:_____

Question 5(14 points)
In what order do things happen when *emacs* is run? Listed below is a set of things that occour when *emacs* is run. Order the steps, the odd number steps are done for you, fill in the numbers for the even steps.

Assume:
-No part of the program has been loaded into memory.
-Page size is 4kbytes and there is only one cache.
-The page table entry loaded from the memory for page 0x00040 maps to physical page 0x19423
-TLB is between the CPU and the cache as in class (cache uses physical addresses).
-Block size is 32 bytes.

1. You type **emacs** at the command line.

3. The CPU attempts to load the first instruction, 0x00040000

5. The page table for this process is accessed to find the entry for virtual page 0x00040 which has the invalid bit set (not yet loaded from disk).

7. The TLB is updated with an entry mapping virtual page 0x00040 to physical page 0x19423

9. The cache misses for the block containing 0x194230000 and attempts to load the block from memory.

11. The instruction at virtual address 0x00040000 is loaded from the cache, completing the instruction fetch phase.

13. The CPU attempts to fetch the second instruction, 0x00040004.

15. The instruction at virtual address 0x00040004 is loaded from the cache, completing the instruction fetch phase.

Choices for the even steps: (Assign the even step numbers to the 7 instructions below)

_8_ The TLB hits for virtual page number 0x00040, the physical address 0x19423000 is sent to the cache.

_2_ A page table for the process is created by the operating system. Static memory area is created, space is allocated for the static parts of the program, heap and stack are initialized. All TLB entries are flushed.

14_ The TLB hits for virtual page number 0x00040, the physical address 0x19423004 is sent to the cache.

_4_ The TLB misses while attempting to find an entry for the virtual page number 0x00040.

12_ The instruction at virtual address 0x00040000 is successfully fetched, and on the next clock tick will move onto its decode stage.

_6_ Physical page number 0x19423 is loaded into memory from disk, and the page table is updated.

10_ The block containing 19423000 is loaded into the cache from memory.

Name:_____ Login:_____

Question 6(20 points)

Each part was worth 10 points.

a. Assume that you have a processor with the following specifications:

i) Five pipeline stages as seen in the lecture notes (IF, ID, AL, ME, WB)
ii) Branch comparing done in the third stage
iii) No forwarding implementation
iv) Register write and register read cannot happen in the same clock cycle
v) Multiple read/write possible with the memory in a clock cycle
vi) Memory stage takes one cycle
vii) Cannot fetch instruction until branch comparison is done
viii) No out-of-order execution

Fill in the rest of the pipeline execution.

I=Instruction Fetch
D=Instruction Decode
A=ALU
M=Memory Access
W=Write Back

add $3, $3, $5	I	D	A	M	W																		
sub $8, $3, $7		I				D	A	M	W														
bne $5, $5, exit						I	D	A	M	W													
lw $10,4($11)									I	D	A	M	W										
add $12, $10,$11										I				D	A	M	W						

How many clock cycles does it need to execute the above set of instructions?

17

Name:_____ Login:_____

b. Assume that you have a processor with the following specifications:

i) Five pipeline stages as seen in the lecture notes (IF, ID, AL, ME, WB)
ii) Branch comparing done in the third stage
iii) Forwarding units are implemented
iv) Registers are written in the first half of the clock cycle and read in the second half of the clock cycle
v) Multiple read/write possible with the memory in a clock cycle
vi) Memory stage takes one clock cycle
vii) Cannot fetch instruction until branch comparison is done
viii) No out-of-order execution

Fill in the rest of the pipeline execution.

I=Instruction Fetch
D=Instruction Decode
A=ALU
M=Memory Access
W=Write Back

add $3, $3, $5	I	D	A	M	W																		
sub $8, $3, $7		I	D	A	M	W																	
bne $5, $5, exit			I	D	A	M	W																
lw $10,4($11)						I	D	A	M	W													
add $12, $10,$11							I	D		A	M	W											

How many clock cycles does it need to execute the above set of instructions?

12