

Full Name: \_\_\_\_\_

## CS 351 Spring 2009

### Midterm Exam

March 13, 2009

**Instructions:**

- This exam is closed-book, closed-notes. Calculators are permitted.
- Write your full name on the front, and make sure that your exam is not missing any sheets.
- Show your answers in the space provided for each problem. If you make a mess, clearly indicate your final answer.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- Good luck!

1 (/10) :
2 (/12) :
3 (/10) :
4 (/12) :
TOTAL (/44) :

### Problem 1. (12 points):

Assume the following setup of virtual memory and cache:

- Memory is byte addressable
- Virtual addresses are 18 bits wide
- Physical addresses are 14 bits wide
- The page size is 1KB ( $2^{10}$  bytes)
- The TLB is 2-way set associative, with 8 total entries
- The cache is 4-way set associative, with 4-byte blocks, and 16 total lines

The TLB, page-table, and cache contents are shown below (only the first 20 PTEs are shown):

TLB			
Index	Tag	Valid	PPN
0	55	1	3
	02	0	9
1	29	1	1
	0C	0	2
2	7E	1	4
	33	1	6
3	15	1	7
	29	1	5

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	0A	4	1
01	A	1	0B	8	1
02	3	0	0C	2	0
03	6	1	0D	1	1
04	1	0	0E	B	1
05	0	0	0F	0	0
06	2	1	10	7	0
07	0	0	11	3	0
08	9	1	12	0	1
09	E	1	13	0	0

Cache						
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	05F	1	32	13	2E	00
	124	1	02	72	19	20
	16F	1	24	61	F5	D0
	219	0	77	40	0E	0D
1	147	1	11	21	37	F2
	002	0	02	62	87	20
	137	1	34	59	B2	E0
	174	0	15	31	4E	12
2	229	1	B2	1F	2E	90
	1C8	0	02	65	97	20
	0DA	1	04	C0	6C	A1
	183	1	04	07	66	70
3	052	1	09	01	46	78
	01A	1	04	59	22	71
	1B3	1	0D	62	87	20
	0F1	1	E6	48	0E	FD

**Part I (4 points)**

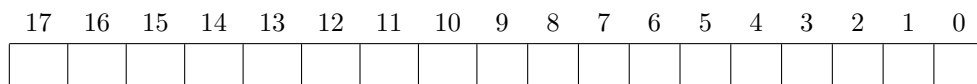
The box below shows the format of a virtual address. Label the diagram with the fields (indicate their positions) that would be used to determine the following:

**VPO** : The virtual page offset

**VPN** : The virtual page number

**TLBI** : The TLB index

**TLBT** : The TLB tag



The box below shows the format of a physical address. Label the diagram with the fields (indicate their positions) that would be used to determine the following:

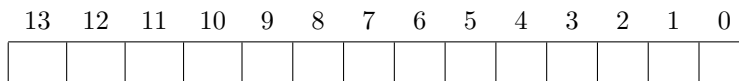
**PPO** : The physical page offset

**PPN** : The physical page number

**CO** : The cache block offset

**CI** : The cache set index

**CT** : The cache tag

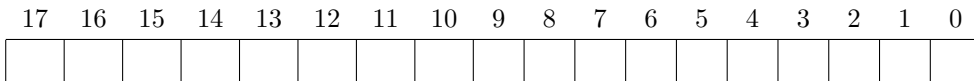


**Part II (8 points)**

For the virtual address on this and the following page, fill in the diagrams and tables in parts A-D using the address translation and cache structures described. If a page fault occurs, you should leave parts C and D blank.

Virtual Address: **0x047a0**

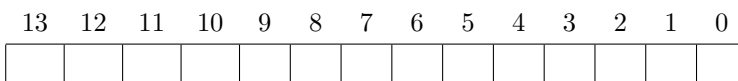
A. Virtual address format



B. Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

C. Physical address format

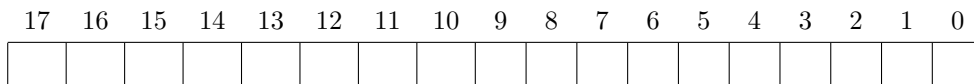


D. Physical memory reference

Parameter	Value
Byte offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Virtual Address: **0x29ef3**

A. Virtual address format



B. Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

C. Physical address format



D. Physical memory reference

Parameter	Value
Byte offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

**Problem 2. (5 points):**

In our discussion of caching, we simply assumed that the bottom  $\lg(N)$  (where  $N$  is the number of cache entries) bits of a physical address would suffice as a cache index. E.g., for the binary addresses 00101110100 and 00101110101, we could use the bit sequences 100 and 101, respectively, to probe for a hit in a direct mapped cache with 8 entries and 1-byte blocks.

The selection of the bottom  $\lg(N)$  bits as a cache index may seem arbitrary. Why, instead, do we not use the **top**  $\lg(N)$  bits of an address as the cache index, and the remaining lower bits as the tag?

### Problem 3. (5 points):

A student just starting to work on the shell lab comes up with the following code for the method `do_fg`, which is a helper function meant to put a currently stopped process into the foreground by sending it a `SIGCONT` signal. Also shown are the `waitfg` and `sigchld_handler` functions:

```
void do_fg(pid_t pid) {
    struct job_t *j = getjobpid(jobs, pid);
    kill(-pid, SIGCONT);
    j->state = FG;
    waitfg(pid);
}

void waitfg(pid_t pid) {
    struct job_t *j = getjobpid(jobs, pid);
    while (j->state == FG) {
        sleep(1);
    }
}

void sigchld_handler(int sig) {
    struct job_t *j;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        j = getjobpid(jobs, pid);
        j->state = UNDEF;
        j->pid = j->jid = 0;
    }
}
```

After testing his code, however, the student finds that sometimes things work as intended, but other times, after trying to put a stopped process into the foreground with the `fg` command, the shell hangs indefinitely. Can you identify where the problem might be? Can you suggest a fix?