EECS 213: Sample Final Exam

From the memory hierarchy to virtual memory.

Spring 2007

Name:

Major/Department/School:

Some words of advice:

- *Read all the questions first.*
- *Start from the easiest one and leave the harder ones for the end.*
- *Approximate results are almost always a valid answer; for sure I do not need 5-decimal precision answers!*
- *This is an* Open Book *exam; you may use any book or notes you like.*
- *Write clearly; if I can't read it I can't grade it.*

Good luck!

Question	Points	Credited
1	??	
2	??	
3	??	
4	??	
5	??	

Problems ...

1. (*?? problem*) In this problem, let `REF(x.i) --> DEV(x.k)` denote that the linker will associate an arbitrary reference to symbol x in module i to the definition of x in module k. For each example bellow, use this notation to indicate how the linker would resolve references to the multiply defined symbol in each module. If there is a link-time error (Rule 1), write "$ERROR''$". If the linker arbitrarily chooses one of the definitions (Rule 3), write "$UNKNOWN''$".

   Answers in italics after each problem.

   (a) .

```
/* Module 1 */
int main()
{
}
```

```
/* Module 2 */
static int main=1;
int p2()
{
}
```

   i. `REF(main.1) --> DEF(--.--)` **Answer:** $main.1$
   ii. `REF(main.2) --> DEF(--.--)` **Answer:** $main.2$

   (b) .

```
/* Module 1 */
int x;
int main()
{
}
```

```
/* Module 2 */
double x;
int p2()
{
}
```

   i. `REF(x.1) --> DEF(--.--)` **Answer:** $''UNKNOWN''$
   ii. `REF(x.2) --> DEF(--.--)` **Answer:** $''UNKNOWN''$

   (c) .

```
/* Module 1 */
int x=1;
int main()
{
}
```

```
/* Module 2 */
double x=1.0;
int p2()
{
}
```

i. `REF(x.1) --> DEF(--.--)` **Answer:** $''ERROR''$

ii. `REF(x.2) --> DEF(--.--)` **Answer:** $''ERROR''$

2. (*?? points*) The following problem concerns basic cache lookups.

   - The memory is byte addressable.
   - Memory accesses are to **1-byte words** (not 4-byte words).
   - Physical addresses are 13 bits wide.
   - The cache is 2-way set associative, with a 4 byte line size and 16 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0	99	04	03	48
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	2F	81	FD	09	0B	0	8F	E2	05	BD
3	06	0	3D	94	9B	F7	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	B7	26	2D	F0	0	0C	71	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

Part 1 The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO The block offset within the cache line
CI The cache index
CT The cache tag

12	11	10	9	8	7	6	5	4	3	2	1	0

Answer: *CT: [12-5] CI: [4-2] CO: [1-0]*

Part 2 For the given physical address, indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs.

If there is a cache miss, enter "-" for "Cache Byte returned".

Physical address: `0E34`

- Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

- Physical memory reference

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

Answer: *A.* 0 1110 0011 0100

Answer: *B. CO: 0x0; CI: 0x5; CT: 0x71; cache hit? Y; cache byte? 0x0B*

3. *?? problems* Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
main() {

  if (fork() == 0) {
    if (fork() == 0) {
      printf("3");
    }
    else {
      pid_t pid; int status;
      if ((pid = wait(&status)) > 0) {
        printf("4");
      }
    }
  }
  else {
    if (fork() == 0) {
      printf("1");
      exit(0);
    }
    printf("2");
  }

  printf("0");

  return 0;
}
```

Out of the 5 outputs listed below, circle only the valid outputs of this program. Assume that all processes run to normal completion.

A. 2030401 B. 1234000 C. 2300140

D. 2034012 E. 3200410

Answer: *A,C,E*

4. (*?? points*) The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to 4-byte words.
- Virtual addresses are 20 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way set associative with 16 total entries.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 32 pages are as follows:

TLB			
Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	0B	1	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	10	6	0
01	8	1	11	7	0
02	9	1	12	8	0
03	A	1	13	3	0
04	6	0	14	D	0
05	3	0	15	B	0
06	1	0	16	9	0
07	8	0	17	6	0
08	2	0	18	C	1
09	3	0	19	4	1
0A	1	1	1A	F	0
0B	6	1	1B	2	1
0C	A	1	1C	0	0
0D	D	0	1D	E	1
0E	E	0	1E	5	1
0F	D	1	1F	3	1

Part 1

(a) The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO The virtual page offset
VPN The virtual page number
TLBI The TLB index
TLBT The TLB tag

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

PPO The physical page offset
PPN The physical page number

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Answer: *A: VPN: [19-12] VPO: [11-0] TLBT: [19-14] TLBI: [13-12]*

Answer: *B: PPN: [15-12] PPO: [11-0]*

Part 2 For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a page fault, enter "-" for "PPN" and leave part C blank.

Virtual address: `7E37C`

(a) Virtual address format (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(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 (one bit per box)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Answer: *A:* `0111 1110 0011 1110 1100`

Answer: *B: VPN: 7E; TLBI: 2; TLBT: 1F; TLB hit? Y; page fault? N; PPN: 8*

Answer: *C:* `1000 0011 1110 1100`

Virtual address: `16A48`

(a) Virtual address format (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(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 (one bit per box)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Answer: *A:* `0001 1100 1010 0010 0100`

Answer: *B: VPN: 1C; TLBI: 0; TLBT: 1C; TLB hit? N; page fault? Y; PPN: -*

Answer: *C: blank*

5. (*?? points*) Determine the block sizes and header values that would result from the follwoing sequence of `malloc` requests. Assumptions: (1) The allocator maintains double-word alignment, and uses an implicit free list withthe block format from Figure 10.37 in your book. (2) Block sizes are rounded up to the nearest multiple of eight bytes.

Answers in italics.

Request	Block size (decimal bytes)	Block header (hex)
`malloc(3)`	*8*	*0x9*
`malloc(11)`	*16*	*0x11*
`malloc(20)`	*24*	*0x19*
`malloc(21)`	*32*	*0x21*