Andrew login ID: _____

Full Name: _____

CS 15-213, Fall 2002

Exam 2

November 12, 2002

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 66 points.
- 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.
- This exam is OPEN BOOK. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (09):
2 (08):
3 (08):
4 (12):
5 (10):
6 (10):
7 (09):
TOTAL (66):

Problem 1. (9 points):

This problem tests your understanding of code optimization. Consider the following function for computing the product of an array of n integers. We have unrolled the loop by a factor of 4.

```
int aprod (int a[], int n)
{
    int i, w, x, y, z, r=1;

    for (i = 0; i < n-3; i += 4) {
        w = a[i]; x = a[i+1]; y = a[i+2]; z = a[i+3];
        r = r * w * x * y * z;  // Product computation
    }
    for (; i < n; i++)
        r *= a[i];
    return r;
}
```

For the line labeled `Product computation`, we can use parentheses to create 3 different associations of the computation, as follows:

```
        r = (((r * w) * x) * y) * z;   // A1
        r = (r * w) * ((x * y) * z);   // A2
        r = r * (w * (x * (y * z)));   // A3
```

Complete the following table with the theoretical CPE (cycles per element) of each of these associations. Assume that this machine has an infinite number of integer multipliers, all capable of operating in parallel with each other. Also, assume that integer multiplication on this machine has a latency of 4 cycles and an issue time of 1 cycle.

Version	Theoretical CPE
A1	
A2	
A3	

Here are some hints:

- Recall that the CPE measure assumes that the run time, measured in clock cycles, for an array of length n is a function of the form $Cn + K$, where C is the CPE.
- "Theoretical CPE" means the performance that would be achieved if the only limiting factors were the data dependences of computation and the latency and issue time of the integer multiplier.

Problem 2. (8 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 14 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 64 total lines.

In the following tables, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

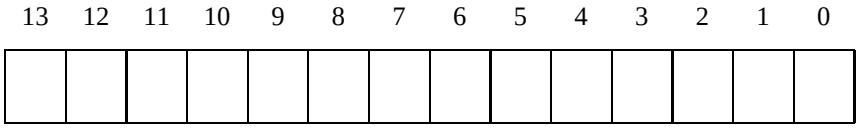
The contents of the cache are as follows:

4-way Set Associative Cache									
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	0C	0	03 3E CD 38	A0	0	16 7B ED 5A	40	0	8E 4C DF 18	58	0	FB B7 12 02
1	3A	1	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	84	1	48 16 81 0A
2	26	0	75 F7 3F C6	78	1	9E 3A 0F DA	26	1	00 4C B6 A8	5E	1	92 04 E5 2E
3	B8	1	E0 22 19 3A	D2	0	02 B3 8F B6	D4	1	25 31 E1 02	C2	0	18 09 73 02
4	54	1	86 B8 F0 C6	4C	1	AA 29 AE 16	56	1	76 46 80 6E	1C	1	13 EA A8 66
5	F6	0	04 2A 32 6A	9E	0	B1 86 56 0E	CC	0	96 30 47 F2	06	1	F8 1D 42 30
6	BE	0	2F 7E 3D A8	C0	0	27 95 A4 74	C4	1	07 11 6B D8	8A	1	C7 B7 AF C2
7	A0	0	D6 A4 89 92	10	0	FD FE D6 DA	76	0	DE D5 CD 4A	E2	0	7C 68 3A 1A
8	F0	1	ED 32 0A A2	E4	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	1	25 44 6F 1A
9	30	1	1E C2 AE 60	08	0	5C 3E DF F2	CA	0	25 CF 84 DA	5C	1	F1 6B DC DE
A	38	1	5D 4D F7 DA	82	1	69 C2 8C 74	9C	1	A8 CE 7F DA	3E	1	FA 93 EB 48
B	3A	1	61 C6 5E 74	64	0	03 97 BA 62	80	1	F8 11 72 12	E0	1	C5 EC 76 4E
C	D4	0	17 52 75 2C	AE	0	62 89 EF 18	8E	0	BB 7D 8C 7C	68	0	26 57 7F C2
D	DC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E
E	D6	0	14 9A 0D 4A	EA	1	C8 1D E6 6E	38	1	F3 38 F3 5C	64	0	6C 8F BD A8
F	7E	1	32 21 1C 2C	FA	1	22 C2 DC 34	BE	1	BA DD 37 D8	B8	0	E7 A2 39 BA

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



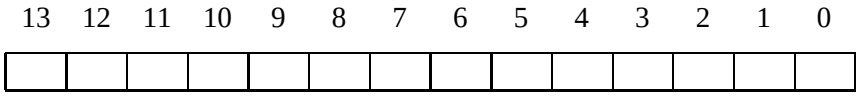
Part 2

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: `2BB2`

A. Physical address format (one bit per box)

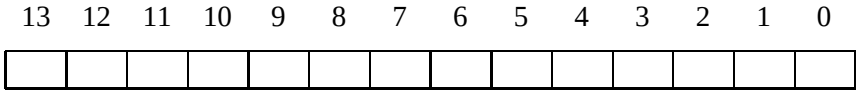


B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Physical address: `098B`

A. Physical address format (one bit per box)



B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Problem 3. (8 points):

This problem tests your understanding of cache conflict misses. Consider the following matrix transpose routine

```
typedef int array[2][2];

void transpose(array dst, array src) {
  int i, j;

  for (j = 0; j < 2; j++) {
    for (i = 0; i < 2; i++) {
      dst[i][j] = src[j][i];
    }
  }
}
```

running on a hypothetical machine with the following properties:

- `sizeof(int) == 4`.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 cache that is direct mapped and write-allocate, with a block size of 8 bytes.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

A. Suppose the cache has a total size of 16 data bytes (i.e., the block size times the number of sets is 16 bytes) and that the cache is initially empty. Then for each `row` and `col`, indicate whether each access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

`src` array		
	col 0	col 1
row 0	m	
row 1		

`dst` array		
	col 0	col 1
row 0	m	
row 1		

B. Repeat part A for a cache with a total size of 32 data bytes.

`src` array		
	col 0	col 1
row 0	m	
row 1		

`dst` array		
	col 0	col 1
row 0	m	
row 1		

The following problem concerns the cache performance of three functions that compute the sum of all elements in an $N \times N$ array for different values of N.

```
typedef int array_t[N][N];

int sumA(array_t a)
{
    int i, j;
    int sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            sum += a[i][j];
        }
    return sum;
}

int sumB(array_t a)
{
    int i, j;
    int sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++) {
            sum += a[i][j];
        }
    return sum;
}

int sumC(array_t a)
{
    int i, j;
    int sum = 0;
    for (j = 0; j < N; j+=2)
        for (i = 0; i < N; i+=2) {
            sum += (a[i][j] + a[i+1][j]
                    + a[i][j+1] + a[i+1][j+1]);
        }
    return sum;
}
```

Problem 4. (12 points):

This problem tests your ability to analyze the cache behavior of C code. Assume we execute the three summation functions shown on the previous page under the following conditions:

- `sizeof(int) == 4`.
- The machine has a 4KB direct-mapped cache with a 16-byte block size.
- Within the two loops, the code uses memory accesses only for the array data. The loop indices, and the value `sum` are held in registers.
- Array `a` is stored starting at memory address `0x08000000`.

Fill in the table for the approximate cache miss rate for the two cases: $N = 64$ and $N = 60$.

Function	`N = 64`	`N = 60`
`sumA`		
`sumB`		
`sumC`		

Problem 5. (10 points):

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 20 bits wide.
- Physical addresses are 18 bits wide.
- The page size is 1024 bytes.
- The TLB is 2-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal**.

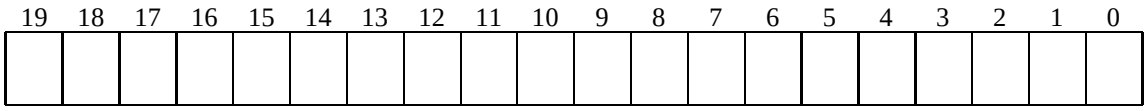
TLB			
Index	Tag	PPN	Valid
0	03	C3	1
	01	71	0
1	00	28	1
	01	35	1
2	02	68	1
	3A	F1	0
3	03	12	1
	02	30	1
4	7F	05	0
	01	A1	0
5	00	53	1
	03	4E	1
6	1B	34	0
	00	1F	1
7	03	38	1
	32	09	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
000	71	1	010	60	0
001	28	1	011	57	0
002	93	1	012	68	1
003	AB	0	013	30	1
004	D6	0	014	0D	0
005	53	1	015	2B	0
006	1F	1	016	9F	0
007	80	1	017	62	0
008	02	0	018	C3	1
009	35	1	019	04	0
00A	41	0	01A	F1	1
00B	86	1	01B	12	1
00C	A1	1	01C	30	0
00D	D5	1	01D	4E	1
00E	8E	0	01E	57	1
00F	D4	0	01F	38	1

Part 1

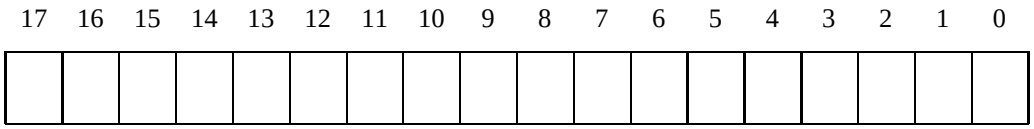
1. The diagram below shows the format of a virtual address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

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



2. The diagram below shows the format of a physical address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

   PPO The physical page offset
   PPN The physical page number

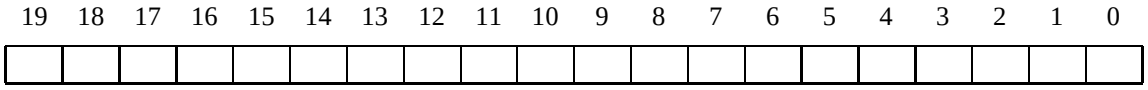


Part 2

For the given virtual addresses, please 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 the physical address blank.

Virtual address: `078E6`

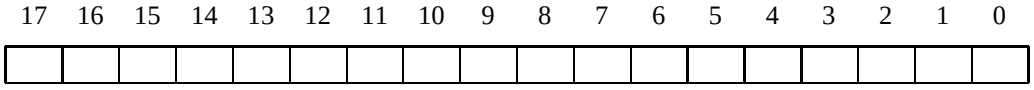
1. Virtual address (one bit per box)



2. Address translation

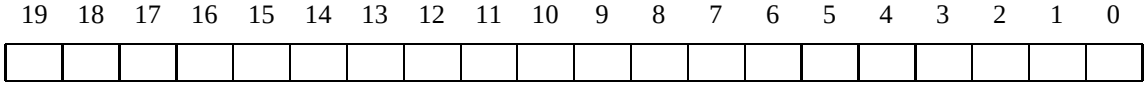
Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	PPN	0x

3. Physical address(one bit per box)



Virtual address: `04AA4`

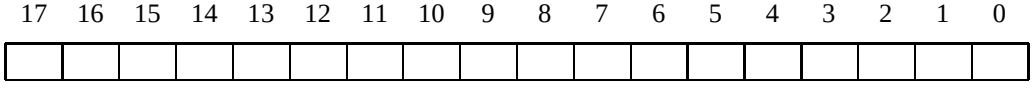
1. Virtual address (one bit per box)



2. Address translation

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

3. Physical address(one bit per box)



Problem 6. (10 points):

This problem tests your understanding of Unix process control.
Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main()
{
    int status;
    int counter = 1;

    if (fork() == 0) {
        counter++;
        printf("%d",counter);
    }
    else {
        if (fork() == 0) {
            printf("5");
            counter--;
            printf("%d",counter);
            exit(0);
        }
        else {
            if (wait(&status) > 0) {
                printf("6");
            }
        }
    }

    printf("3");
    exit(0);
}
```

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program.

A. `253063` Y N

B. `251633` Y N

C. `520633` Y N

D. `263503` Y N

E. `506323` Y N

Problem 7. (9 points):

This question tests your understanding of signals and signal handlers.

It presents 3 different snippets of C code. Assume that all functions and procedures return correctly and that all variables are declared and initialized properly. Also, assume that an arbitrary number of SIGINT signals, and only SIGINT signals, can be sent to the code snippets randomly from some external source.

For each code snippet, circle the value(s) of `i` that could possibly be printed by the `printf` command at the end of each program. *Careful: There may be more than one correct answer for each question. Circle all the answers that could be correct.*

Code Snippet 1:

```
int i = 0;

void handler(int sig) {
  i = 0;
}

int main() {
  int j;

  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  exit(0);
}
```

1. Circle possible values of `i` printed by snippet 1:

A. 0

B. 1

C. 50

D. 100

E. 101

F. None of the above

Code Snippet 2:

```
int i = 0;

void handler(int sig) {
  i = 0;
}

int main () {
  int j;
  sigset_t s;

  signal(SIGINT, handler);

  /* Assume that s has been
     initialized and declared
     properly for SIGINT */

  sigprocmask(SIG_BLOCK, &s, 0);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  sigprocmask(SIG_UNBLOCK, &s, 0);
  printf("i = %d\n", i);
  exit(0);
}
```

2. Circle possible values of `i` printed by snippet 2:

A. 0

B. 1

C. 50

D. 100

E. 101

F. None of the above

Code Snippet 3:

```
int i = 0;

void handler(int sig) {
  i = 0;
  sleep(1);
}


int main () {
  int j;
  sigset_t s;

  /* Assume that s has been
     initialized and declared
     properly for SIGINT */

  sigprocmask(SIG_BLOCK, &s, 0);
  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  sigprocmask(SIG_UNBLOCK, &s, 0);
  exit(0);
}
```

3. Circle possible values of `i` printed by snippet 3:

A. 0

B. 1

C. 50

D. 100

E. 101

F. None of the above