Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2001

Exam 2

November 13, 2001

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 64 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 (13):	
2 (06):	
3 (05):	
4 (08):	
5 (08):	
6 (10):	
7 (04):	
8 (10):	
TOTAL (64):	

The following two problems concern the performance of two procedures that generate a data structure describing the *population statistics* for a set of data values. That is, suppose we had a set of integer data values $x_1, x_2, \ldots x_n$, having minimum value x_{min} and maximum value x_{max}. For each possible value of x between x_{min} and x_{max}, we want to record P_x, defined to be the number of values of i such that $x = x_i$.

We will record this information in a data structure of type `pop_ele` declared as follows:

```
typedef struct {
    int m;
    int xmin;
    int *p;
} pop_ele, *pop_ptr;
```

Since C arrays have minimum index 0, and the value of x_{min} can be an arbitrary integer, we explicitly store the value of x_{min} in this structure, and use an array `p` of size $m = x_{max} - x_{min} + 1$. Population value P_x is then stored as array value `p[x-xmin]`. For example, for data set $[-1, 1, 2, -1, -1]$, we would have $x_{min} = -1$, and $m = 2 - -1 + 1 = 4$. The array `p`, would represent this population as follows:

Population values	P_{-1}	P_0	P_1	P_2
Array Elements	`p[0]`	`p[1]`	`p[2]`	`p[3]`
Values	3	0	1	1

Here is some utility code for finding the values of x_{min} and x_{max}. We have instrumented the code with a counter to determine the total number of comparisons peformed between data values (`comp_cnt`).

```
/* Find minimum value in array a */
int min_val(int a[], int n)
{
    int i;
    int result = a[0];
    for (i = 1; i < n; i++) {
        result = result > a[i] ? a[i] : result;
        comp_cnt++;
    }
    return result;
}

/* Find maximum value in array a */
int max_val(int a[], int n)
{
    int i;
    int result = a[0];
    for (i = 1; i < n; i++) {
        result = result < a[i] ? a[i] : result;
        comp_cnt++;
    }
    return result;
}
```

Here are two versions of a function to generate population statistics. The first calls functions `min_val` and `max_val` repeatedly.

```
/* First version of population counting routine */
pop_ptr build_pop1(int a[], int n)
{
    int i;
    pop_ptr result = (pop_ptr) malloc(sizeof(pop_ele));
    result->xmin = min_val(a,n);          // MIN1
    result->m = (max_val(a, n)            // MAX1
                 - min_val(a, n) + 1);    // MIN2
    result->p = (int *) malloc(result->m * sizeof(int));
    /* Set population entries to zero */
    for (i = min_val(a,n);                // MIN3
         i <= max_val(a,n); i++)          // MAX2
        result->p[i-min_val(a,n)] = 0;    // MIN4
    /* Now update the population entries */
    for (i = 0; i < n; i++)
        result->p[a[i]-min_val(a,n)]++;  // MIN5
    return result;
}
```

The second only calls each of these functions once, storing the result in a termporary.

```
pop_ptr build_pop2(int a[], int n)
{
    int i;
    pop_ptr result = (pop_ptr) malloc(sizeof(pop_ele));
    int min = min_val(a,n);                // MIN1
    int max = max_val(a,n);                // MAX1
    result->xmin = min;
    result->m = (max - min + 1);
    result->p = (int *) malloc(result->m * sizeof(int));
    /* Set population entries to zero */
    for (i = min; i <= max; i++)
        result->p[i-min] = 0;
    /* Now update the population entries */
    for (i = 0; i < n; i++)
        result->p[a[i]-min]++;
    return result;
}
```

Problem 1. (13 points):

The comments on the right of the code for `build_pop1` indicate the places where functions `min_val` and `max_val` are called.

Suppose that m is defined to be the number of entries in the population array. That is, $m = x_{max} - x_{min} + 1$. As before n denotes the size of the data set.

A. Fill in the following table to indicate the total number of calls to `min_val` and `max_val` made at these different points in the program. Express your entries as formulas in terms of m and n. The final entry should show the total number of calls to `min_val` and `max_val`.

Call Point	Times Called
MIN1	
MAX1	
MIN2	
MIN3	
MAX2	
MIN4	
MIN5	
Total	

B. The counter `comp_cnt` counts the total number of comparisons made between data values. How many comparisons are made during a single call to `min_val` or `max_val`? Express your answer as a formula in terms of m and n.

C. If we start with `comp_cnt` equal to 0 and call function `build_pop1`, what will be the final value of `comp_cnt`? Express your answer as a formula in terms of m and n.

Problem 2. (6 points):

Now let us compare the overall performance of `build_pop1` to that of `build_pop2`, which avoids repeated calls to `min_val` and `max_val`.
Consider the following scenarios for the relation between m and n:

Dense: $m = \sqrt{n}$, i.e., there are many repeated values.

Matched: $m = n$, i.e., the range of values is about the same as the number of values.

Sparse: $m = n^2$, i.e., the range is much larger than the number of values.

Fill in the following table giving the *asymptotic complexities* of the two functions. Your answers should be formulas in big-O notation in terms of n, e.g., $O(n^3)$. Your answer will be marked incorrect if you do not simplify the formula. For example, you should write $O(n^2)$ instead of $O(2n^2 + 3n + 1)$.
Your analysis should consider not just the effort expended in calling `min_val` and `max_val`, but all of the operations performed by the two functions, as well.

Scenario	`build_pop1`	`build_pop2`
Dense		
Matched		
Sparse		

Problem 3. (10 points):

Consider the following function for computing the product of an array of n integers. We have unrolled the loop by a factor of 3.

```
int aprod(int a[], int n)
{
    int i, x, y, z;
    int r = 1;
    for (i = 0; i < n-2; i+= 3) {
        x = a[i]; y = a[i+1]; z = a[i+2];
        r = r * 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 5 different associations of the computation, as follows:

```
r = ((r * x) * y) * z; // A1
r = (r * (x * y)) * z; // A2
r = r * ((x * y) * z); // A3
r = r * (x * (y * z)); // A4
r = (r * x) * (y * z); // A5
```

We express the performance of the function in terms of the number of cycles per element (CPE). As described in the book, this measure assumes 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.

We measured the 5 versions of the function on an Intel Pentium III. Recall from Figure 5.12 of the book that the integer multiplication operation on this machine has a latency of 4 cycles and an issue time of 1 cycle.

The following table shows some values of the CPE, and other values missing. The measured CPE values are those that were actually observed. "Theoretical CPE" means that performance that would be achieved if the only limiting factor were the latency and issue time of the integer multiplier.

Version	Measured CPE	Theoretical CPE
A1	4.00	
A2	2.67	
A3		
A4	1.67	
A5		

Fill in the missing entries. For the missing values of the measured CPE, you can use the values from other versions that would have the same computational behavior. For the values of the theoretical CPE, you can determine the number of cycles that would be required for an iteration considering only the latency and issue time of the multiplier, and then divide by 3.

Problem 4. (5 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 12 bits wide.
- The cache is 4-way set associative, with a 2-byte block size and 32 total lines.

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

4-way Set Associative Cache																
Index	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1
0	29	0	34	29	87	0	39	AE	7D	1	68	F2	8B	1	64	38
1	F3	1	0D	8F	3D	1	0C	3A	4A	1	A4	DB	D9	1	A5	3C
2	A7	1	E2	04	AB	1	D2	04	E3	0	3C	A4	01	0	EE	05
3	3B	0	AC	1F	E0	0	B5	70	3B	1	66	95	37	1	49	F3
4	80	1	60	35	2B	0	19	57	49	1	8D	0E	00	0	70	AB
5	EA	1	B4	17	CC	1	67	DB	8A	0	DE	AA	18	1	2C	D3
6	1C	0	3F	A4	01	0	3A	C1	F0	0	20	13	7F	1	DF	05
7	0F	0	00	FF	AF	1	B1	5F	99	0	AC	96	3A	1	22	79

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

11	10	9	8	7	6	5	4	3	2	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: `3B6`

A. Physical address format (one bit per box)

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

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 5. (8 points):

A bitmap image is composed of pixels. Each pixel in the image is represented as four values: three for the primary colors(red, green and blue - RGB) and one for the transparency information defined as an alpha channel.

In this problem, you will compare the performance of direct mapped and 4-way associative caches for a square bitmap image initialization. Both caches have a size of 128 bytes. The direct mapped cache has 8-byte blocks while the 4-way associative cache has 4-byte blocks.

You are given the following definitions

```
typedef struct{
    unsigned char r;
    unsigned char g;
    unsigned char b;
    unsigned char a;
}pixel_t;

pixel_t  pixel[16][16];
register int i, j;
```

Also assume that

- `sizeof(unsigned char) = 1`
- `pixel` begins at memory address 0
- Both caches are initially empty
- The array is stored in row-major order
- Variables `i,j` are stored in registers and any access to these variables does not cause a cache miss

A. What fraction of the writes in the following code will result in a miss in the direct mapped cache?

```
for (i = 0; i < 16; i ++){
    for (j = 0; j < 16; j ++){
        pixel[i][j].r = 0;
        pixel[i][j].g = 0;
        pixel[i][j].b = 0;
        pixel[i][j].a = 0;
    }
}
```

Miss rate for writes to pixel:______________%

B. Using code in part A, what fraction of the writes will result in a miss in the 4-way associative cache?

Miss rate for writes to pixel: _______________ %

C. What fraction of the writes in the following code will result in a miss in the direct mapped cache?

```
for (i = 0; i < 16; i ++){
    for (j = 0; j < 16; j ++){
        pixel[j][i].r = 0;
        pixel[j][i].g = 0;
        pixel[j][i].b = 0;
        pixel[j][i].a = 0;
    }
}
```

Miss rate for writes to pixel:_______________%

D. Using code in part C, what fraction of the writes will result in a miss in the 4-way associative cache?

Miss rate for writes to pixel:_______________%

Problem 6. (8 points):

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

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

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

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      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.

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

`src` 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.

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

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

Problem 7. (10 points):

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

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

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

A. 32040 Y N

B. 34002 Y N

C. 30402 Y N

D. 23040 Y N

E. 40302 Y N

Problem 8. (4 points):

Consider the following C program. (For space reasons, we are not checking error return codes. You can assume that all functions return normally.)

```
int val = 10;

void handler(sig)
{
    val += 5;
    return;
}

int main()
{
    int pid;

    signal(SIGCHLD, handler);
    if ((pid = fork()) == 0) {
        val -= 3;
        exit(0);
    }
    waitpid(pid, NULL, 0);
    printf("val = %d\n", val);
    exit(0);
}
```

What is the output of this program? `val =` ______________

Problem 9. (10 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

A. 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

B. 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

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