CS237 Final Exam

Practice Exam

This is a closed-book exam (no calculators, no phones, no outside materials). You have 2 hours and 30 minutes to complete the exam. All intended answers should fit in the space provided. You may use the backs of exam pages as scratch paper if you wish, but mark your final answers clearly on the exam. When you are done, submit your exam along with the signed honor code statement at the bottom of this page.

Be sure to give yourself enough time to answer each question. The point values should help you to manage your time.

Question	**Value**	**Score**
Multiple Choice	12	
Short answer	16	
Impact of Pipelining	8	
Cache organization	6	
VM Address Translation	12	
Memory allocation	10	
Total	**64**	

I have neither given nor received unauthorized aid on this examination.

Signature: ______________________________

Name: _________________________________

Useful notes and scratch space

Powers of 2:

2^{12}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
4096	1024	512	256	128	64	32	16	8	4	2	1

Powers of 16:

16^3	16^2	16^1	16^0
4096	256	16	1

Data units:

2^{40}	2^{30}	2^{20}	2^{10}	2^0
1 TiB	1 GiB	1 MiB	1 KiB	1 B

C Declaration	Intel data type	Assembly code suffix	Size (bytes)
char	byte	b	1
short	word	w	2
int	double word	l	4
long	quad word	q	8
char *	quad word	q	8
float	single precision	s	4
double	double precision	l	8

Registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are used to pass the first six integer or pointer parameters to procedures.

Problem 1. (12 points):

Answer the following multiple-choice questions by circling all answers that apply.

A. According to the struct alignment rules, each field must be aligned to an address that is a multiple of its size. Given the structure definition below, what value is yielded by the expression `sizeof(struct example)`?

```
struct example {
   char a;
   int b;
   char *d;
   int *c;
};
```

(a.) 10

(b.) 21

(c.) 24 — int b must start at an address that is a multiple of 4, so 3 bits of padding are added after char a to satisfy the alignment requirement.

(d.) 32

(e.) None of the above.

B. Consider a memory allocator that does not coalesce free blocks and a second implementation that is identical except it does coalesce, which of the following is true about their utilization scores on an arbitrary trace? (You may assume the first implementation stores enough information to make coalescing possible, so the only difference is that the second implementation actually performs the coalescing.)

(a.) The coalescing malloc will definitely get a better utilization score.

(b.) The coalescing malloc might get a better utilization score and might get the same utilization score, but it cannot get a worse utilization score.

(c.) The coalescing malloc might get a better utilization score, might get the same utilization score, and might get a worse utilization score.

(d.) The coalescing malloc will definitely get a worse utilization score.

(e.) None of the above.

It seems like coalescing should always do at least as good as an identical implementation that does not coalesce, but a carefully-chosen adversarial set of mallocs/frees with the first-fit allocation algorithm can produce a heap with worse utilization!

C. How many bits are needed for the Virtual Page Offset if the page size is 128 bytes?

(a.) 7

(b.) 8

(c.) 9

(d.) 10

(e.) None of the above.

We need enough bits to uniquely index all 128 bytes in the page, and 128 = 2^7

D. Consider a 4-way set associative cache (E = 4). Which one of the following statements is true?

(a.) The cache has 4 blocks per line.

(b.) The cache has 4 sets per line.

(c.) The cache has 4 lines per set.

(d.) The cache has 4 sets per block

(e.) None of the above.

E. In the following code, what order of loops exhibits the best locality?

```
// int a[X][Y][Z] is declared earlier
int i, j, k, sum = 0;
for (i = 0; i < Y; i++)
    for (j = 0; j < Z; j++)
        for (k = 0; k < X; k++)
            sum += a[k][i][j];
```

(a.) `i` on the outside, `j` in the middle, `k` on the inside (as is).

(b.) `j` on the outside, `k` in the middle, `i` on the inside.

(c.) `k` on the outside, `i` in the middle, `j` on the inside.

(d.) The order does not matter.

Problem 2. (16 points):

Answer the following questions. Your responses should fit in the space provided. Most questions can be answered in three sentences or less. Thoughtful answers will be given partial credit; the most succinct and correct answers will be awarded the most credit. Pictures/diagrams may be used when appropriate.

A. For a sequence of memory accesses, the *stride* refers to the gap between consecutive adresses that are read/written. In (approximately) three sentences or less, explain the impact of stride on cache efficiency.

Programs with better locality have better cache performance, and stride is a measurement of the quality of a program's spatial locality (the lower the stride, the better the locality). With a small stride, consecutive accesses should fall within the same cache line, so the program will extract the maximum benefit from from the cost of installing a given cache line into the cache.

B. In (approximately) three sentences or less, explain why a programmer might choose to use processes to implement parallelism, rather than threads.

All peer threads share the same address space, so if a program wants each of the parallel flows of execution to have an isolated address space, then they would use processes instead of threads.

C. What main advantage do multi-level page tables provide over single-level page tables. Based on this advantage, describe a hypothetical situation where using a multi-level page table would **not** be a good design decision.

Virtual address spaces are typically very large but very sparsely populated. Since a process's page table is stored in memory, a multi-level page table allows a process to store only the regions of the page table that are actively in use.

A multi-level page table would not be a good idea if the virtual address space was densely populated. In such a situation, there would be no savings, and the process would have to pay the additional cost of traversing through multiple levels of page table's tree structure during address translation.

D. x86 assembly contains instructions that perform conditional moves of data, copying a source value S to the destination R only if the move condition holds. Describe a scenario where a conditional move would result in dangerous or poorly performing code.

A conditional move can be unsafe if the computation is dangerous. For example:
x = p ? *p : 0;
If the pointer p is a NULL pointer, then evaluating *p would lead to a segmentation fault.

A conditional move can perform poorly if the computation is expensive, since both branches are executed. For example:
val = Test(x) ? Easy(x) : Hard(x);
If the cost of the test is inexpensive, then it would be preferable to execute Test(x) and Easy(x) and avoid Hard(x) entirely (if possible).

A conditional move can lead to incorrect behavior if the branches have side effects. For example:
val = x > 0 ? x*=7 : x+=3;
Executing exactly one of the branches (and the correct one) is the only way for the value of x to be correct.

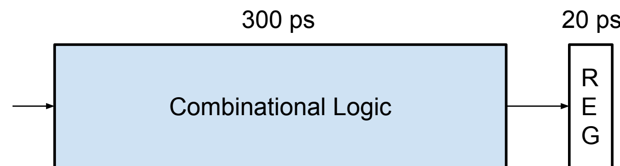
E. Fill in the following table. You will find it helpful during later problems.

Hex Digit	Binary Representation
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
A	
B	
C	
D	
E	
F	

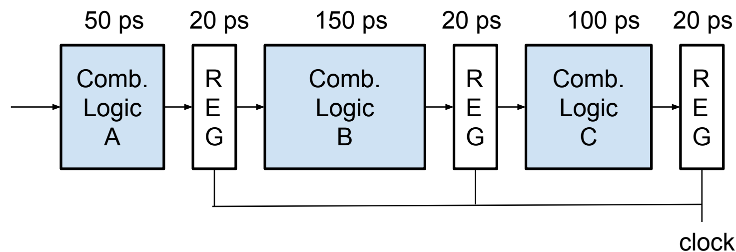
Problem 3. (8 points):

Pipelining is a powerful technique. This question will explore some of the tradeoffs and challenges of designing pipelined systems.

Suppose we have a system that spends 300 picoseconds (ps) to evaluate a combinational logic function, and 20 ps to store the results in a register. This unpipelined system, shown below, has a latency of 300ps + 20ps = 320ps, and a throughput of (1 instruction) / (300ps + 20ps).



Now suppose we can divide the combinational logic into a series of three stages, each separated by a pipeline register (shown below). After the specified delay, data propagates through a stage's combinational logic, but the registers' states remain unchanged until the clock rises. Thus, the clock rate is limited by the delay of the slowest stage.

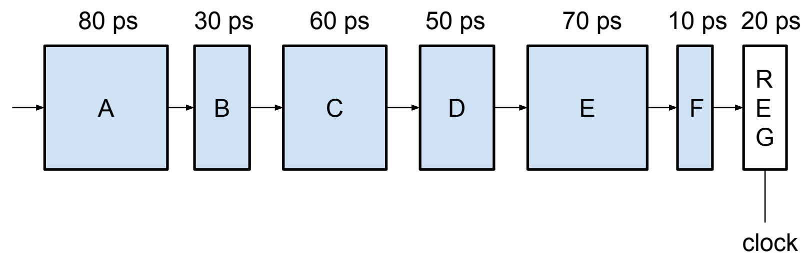


A. What is the latency and the throughput of the pipelined system shown above?

- Latency: The latency is 150+20 + 150+20 + 150+20= 510 ps (the slowest stage determines the clock speed, so the clock ticks every 150+20 ps)
- Throughput:

1 instruction /170 ps (which is 5.88 GIPS)

Now suppose we analyze the combinational logic again, and we realize that it can be separated into a sequence of six blocks, named A to F, having delays of 80, 30, 60, 50, 70, and 10 ps (shown below).



We can create a pipelined version of this design by inserting pipeline registers in between any pair of the combinational logic blocks. Assume that each pipeline register has a delay of 20ps.

B. Inserting a single register gives a two-stage pipeline. Where should the register be inserted to maximize the throughput? What would be the throughput and the latency?

- Location: Between C and D (ABC and DEF are the stages)
- Latency: The slowest stage is 170ps, so:170+20 + 170+20 = 380ps
- Throughput: 1 instruction/190ps (which is 5.26 GIPS)

C. Inserting two registers gives a three-stage pipeline. Where should the registers be inserted to maximize the throughput? What would be the throughput and the latency?

- Location: Between B and C as well as Between D and E (AB, CD, EF are the stages)
- Latency: The slowest stage is 110ps, so:110+20 + 110+20 + 110+20 = 390ps
- Throughput: 1 instruction/130ps (which is 7.69 GIPS)

D. (Challenge Question) What is the minimum number of stages that would yield a design with the maximum achievable throughput? How many registers would you insert, and where would they be placed?

Problem 4. (6 points):

(To solve this problem, think back to Lab 4. Drawing a picture of a cache and labeling the parts may be helpful.) The following table gives the parameters for a number of different caches, where m is the number of physical address bits, C is the cache size (number of data bytes, not including the overhead such as valid or tag bits. $C = B \times E \times S$), B is the block size in bytes, and E is the associativity (number of lines per set). For each cache, determine the total number of cache sets (S), number of tag bits (t), number of set index bits (s), and number of block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	1	256	22	8	2
2.	32	1024	8	4	32	24	5	3
3.	32	1024	32	32	1	27	0	5

1 End-to-end Address Translation

Consider a simplified system with a TLB, an L1 d-cache, and the following properties

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Virtual addresses are 14 bits wide ($n = 14$).
- Physical addresses are 12 bits wide ($m = 12$).
- The page size is 64 bytes = 2^6.
- The TLB is 4-way set associative with 16 total entries.
- The L1 d-cache is physically addressed and direct mapped, with a 4-byte line size and 16 total sets.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 16 page table entries (PTEs) are as follows:

TLB			
Index	Tag	PPN	Valid
0	03	–	0
	09	0D	1
	00	–	0
	07	02	1
1	03	2D	1
	02	–	0
	04	–	0
	0A	–	0
2	02	–	0
	08	–	0
	06	–	0
	03	–	0
3	07	–	0
	03	0D	1
	0A	34	1
	02	–	0

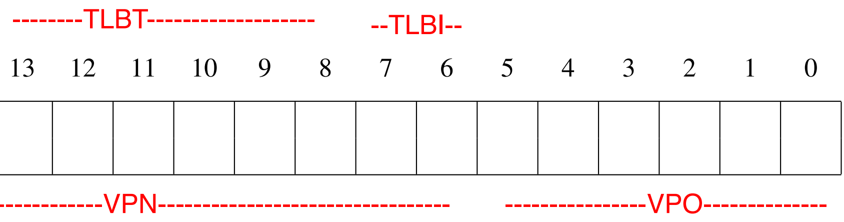
Page Table		
VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

Cache						
Index	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Answer the following questions using this information.

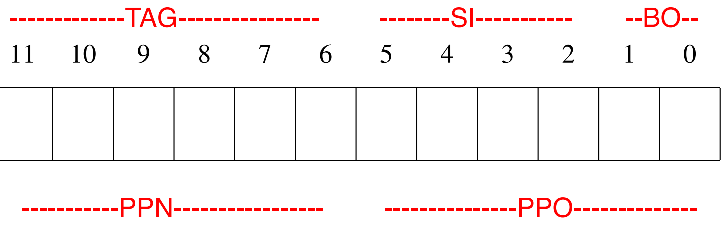
- The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

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



- 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
Block Offset	The cache's block offset
Set Index	The cache's set index
Tag	The cache's tag



For the given virtual addresses (continued on the next page), 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: `0x03D7`

- Virtual address format (one bit per box)

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

- Address translation

Parameter	Value
VPN	0x f
VPO	0x 17
TLB Index	0x 3
TLB Tag	0x 3
TLB Hit? (Y/N)	Y
Page Fault? (Y/N)	N
PPN	0x d
PP0	0x 17

- Physical address format (one bit per box)

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

- Physical memory reference

Parameter	Value
Byte offset	0x 3
Cache Index	0x 5
Cache Tag	0x d
Cache Hit? (Y/N)	Y
Cache byte returned	0x 1d

Problem 5. (10 points):

In the bitmap implementation from lab 4, we stored individually addressable bits inside an array of bytes (type `unsigned char`). For this question, consider a bitmap implementation that instead uses an array of 4-byte values (type `unsigned int`). Further:

- Within each `unsigned int`, bits are indexed from the least significant bit (LSB) to the most significant bit (MSB). So, for example, a bitmap that can store 32 bits requires one `unsigned int`, and if `set_bit(bitmap, 2)` was called, the value of that `unsigned int` would be `4`.

Four helper routines are given to facilitate the implementation of the bitmap structure defined below. The functionality of each routine is explained in the comment above the function definition. Fill in the missing functionality in each function. You should only need to write/finish a few lines of code for most functions. You may call other helper functions that you define. You may not, however, use any macro that is not shown below.

```
struct bitmap {
    uint64_t n_ints; // the number of ints in the bitmap (not bits!)
    uint64_t n_valid_bits; // number of meaningful bits
    unsigned int *bits;
};

/**
 * Allocates a struct bitmap and its array of unsigned ints such that
 * it is the smallest array large enough to hold num_bits valid bits.
 * allocate_bitmap initializes all struct fields appropriately.
 */
struct bitmap *allocate_bitmap(uint64_t num_bits) {
    struct bitmap *bmap = malloc(sizeof *bmap);
    if (bmap == NULL)
        return NULL;


    bmap->n_ints =   (num_bits + 31) / 32;     // Allocate in multiples of 32 bits, andround up to
                                               // the next int if needed

    bmap->n_valid_bits =   num_bits;           // we may be able to store more bits in our bitmap due to
                                               // padding, but only the first num_bits are valid

    bmap->bits = malloc(bmap->n_ints * sizeof(unsigned int));
    if (bmap->bits == NULL) {
        free(bmap);
        return NULL;
    }

    // initialize all bits to 0
    memset(bmap->bits, 0, bmap->n_ints * sizeof(unsigned int));

    return bmap;
}
```

```
/**
 * Yields 1 if the given bit is set, 0 otherwise.
 */
char get_bit(struct bitmap *map, uint64_t i) {

    size_t which_int = i / 32;
    size_t which_bit = i % 32;
    return !!(map->bits[which_int]  & (1 << which_bit));   // mask target bit, then use !! to return 0 or 1

}

/**
 * Sets the bit at position i to 1 (regardless of whether already set)
 */
void set_bit(struct bitmap *map, uint64_t i) {

    size_t which_int = i / 32;
    size_t which_bit = i % 32;
    map->bits[which_int]  |= (1 << which_bit));

}

/*
 * Returns the index of the first unset bit if one exists, or
 * bitmap->n_valid_bits if one does not exist
 */
uint64_t first_unset_bit(struct bitmap *map) {
    int i = 0;
    while (i < bmap->n_ints) {
        if (map->bits[i] != 0xFFFFFFFF) {
            // find the first free bit in the target int
            for (int bit = 0; bit < sizeof(unsigned int); bit++) {
                if (!(map->bits[i] & (1 << bit))) {

                    return i * 32 + bit;

                }
            }
        }
        i++;
    }
    // no unset bit was found
    return map->n_valid_bits;
}
```