

Exam 2

601.229 Computer Systems Fundamentals

November 3, 2025

Complete all questions.

Time: 50 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: _____ Solution

Print name: _____

Date: _____

Reference

Powers of 2 ($y = 2^x$):

x	0	1	2	3	4	5	6	7	8	9	10	11	12
y	1	2	4	8	16	32	64	128	256	512	1,024	2,048	4,096

x	13	14	15	16
y	8,192	16,384	32,768	65,536

Note that in all questions concerning C:

- `uint8_t` is an 8-bit unsigned integer type
- `uint16_t` is a 16-bit unsigned integer type
- `uint32_t` is a 32-bit unsigned integer type
- `int8_t` is an 8-bit signed two's complement integer type
- `int16_t` is a 16-bit signed two's complement integer type
- `int32_t` is a 32-bit signed two's complement integer type

x86-64 registers:

Callee-saved: `%rbx`, `%rbp`, `%r12`,
`%r13`, `%r14`, `%r15`

Caller-saved: `%r10`, `%r11`

Return value: `%rax`

Arguments: `%rdi`, `%rsi`, `%rdx`,
`%rcx`, `%r8`, `%r9`

Note that argument registers and
return value register are
effectively caller-saved.

Registers and sub-registers:

Register	Low 32 bits	Low 16 bits	Low 8 bits
<code>%rax</code>	<code>%eax</code>	<code>%ax</code>	<code>%al</code>
<code>%rbx</code>	<code>%ebx</code>	<code>%bx</code>	<code>%bl</code>
<code>%rcx</code>	<code>%ecx</code>	<code>%cx</code>	<code>%cl</code>
<code>%rdx</code>	<code>%edx</code>	<code>%dx</code>	<code>%dl</code>
<code>%rbp</code>	<code>%ebp</code>	<code>%bp</code>	<code>%bpl</code>
<code>%rsi</code>	<code>%esi</code>	<code>%si</code>	<code>%sil</code>
<code>%rdi</code>	<code>%edi</code>	<code>%di</code>	<code>%dil</code>
<code>%r8</code>	<code>%r8d</code>	<code>%r8w</code>	<code>%r8b</code>
<code>%r9</code>	<code>%r9d</code>	<code>%r9w</code>	<code>%r9b</code>
<code>%r10</code>	<code>%r10d</code>	<code>%r10w</code>	<code>%r10b</code>
<code>%r11</code>	<code>%r11d</code>	<code>%r11w</code>	<code>%r11b</code>
<code>%r12</code>	<code>%r12d</code>	<code>%r12w</code>	<code>%r12b</code>
<code>%r13</code>	<code>%r13d</code>	<code>%r13w</code>	<code>%r13b</code>
<code>%r14</code>	<code>%r14d</code>	<code>%r14w</code>	<code>%r14b</code>
<code>%r15</code>	<code>%r15d</code>	<code>%r15w</code>	<code>%r15b</code>

Stack alignment: `%rsp` must contain an address that is a multiple of 16 when any `call` instruction is executed.

Operand size suffixes: **b** = 1 byte, **w** = 2 bytes, **l** = 4 bytes, **q** = 8 bytes (Examples: `movb`, `movw`, `movl`, `movq`)

Question 1. [10 points] Consider the combine function, intended to combine strings together into a single string (unit test shown on right):

```
void combine(const char *s[],          const char *str[] = {
                int n, char *d) {      "CSF", "is", "FUN!",
    *d = '\0';                          };
    for (int i = 0; i < n; ++i)         char out[100];
        strcat(d, s[i]);               combine(str, 3, out);
}                                        assert(0 == strcmp(out, "CSFisFUN!"));
```

(a) What is the most significant inefficiency in the implementation of the combine function?

Each time strcat is called, it must spend time proportional to the current length of the string d points to in order to find the NUL terminator at the end of the string. If the array s has N strings, $O(N^2)$ time is needed to append all of them.

(b) Briefly explain how to fix the inefficiency you identified in (a).

The loop should keep track of where the end of the string is. E.g., rewrite the body of the function as follows:

```
for (int i = 0; i < n; ++i) {
    size_t len = strlen(s[i]);
    memcpy(d, s[i], len);
    d += len;
}
*d = '\0';
```

Question 2. [10 points] A memory cache has the following characteristics:

- Its data capacity is 1,024 KB (2^{20}) bytes
- Its block size is 32 (2^5) bytes
- It has 4,096 (2^{12}) sets

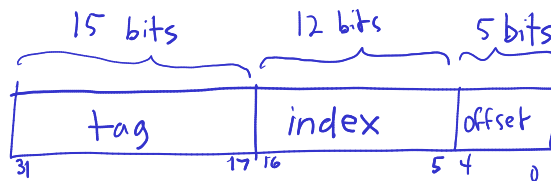
(a) What is the associativity factor of this cache? (I.e., how many blocks can be stored in each set?) Explain briefly.

$$\text{capacity} = \# \text{ sets} \cdot \text{block size} \cdot \text{assoc. factor}$$

$$2^{20} = 2^{12} \cdot 2^5 \cdot \frac{2^3}{1}$$

The associativity factor is $2^3 = 8$

(b) Assume that the system has 32-bit addresses. Sketch the address format for the cache, indicating which exact bits are the offset, index, and tag.

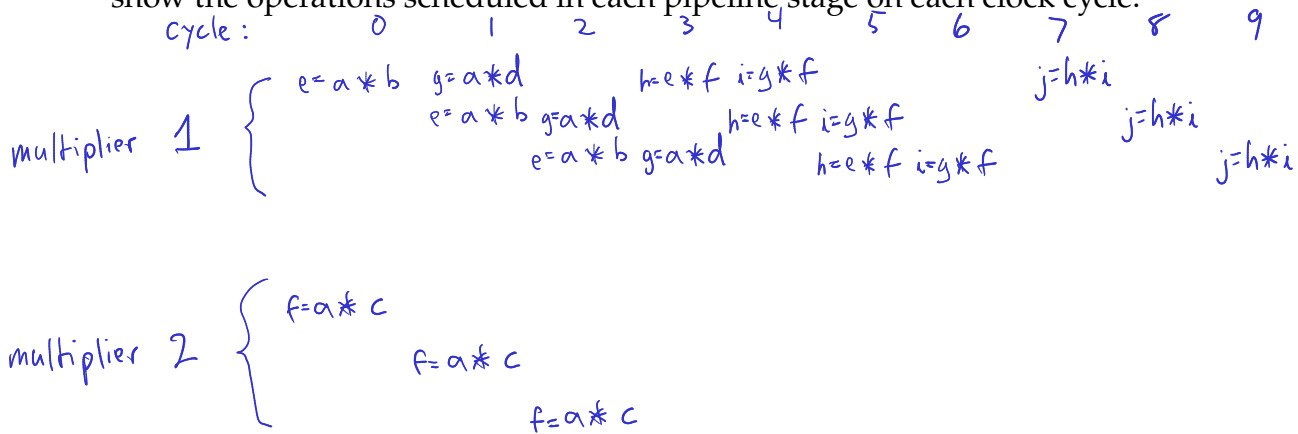


Question 3. [10 points] Assume that a superscalar CPU has two independent fully-pipelined integer multiplication units, each of which can complete an integer multiplication in 3 cycles.

Consider the following computation:

```
e = a * b
f = a * c
g = a * d
h = e * f
i = g * f
j = h * i
```

Assume that all variables (a through j) represent CPU registers. How many cycles are required to compute the value of to be stored in j? Explain briefly. Recommendation: show the operations scheduled in each pipeline stage on each clock cycle.



10 cycles are required

Question 4. [15 points] Consider the following code:

```
int arr[32];
for (int i = 0; i < 32; ++i)
    arr[i] = i;
```

Assume the following:

- There is a write-back memory cache with 32 byte blocks
- The address of `arr[0]` is an exact multiple of 32
- `sizeof(int) = 4`
- Before the above code executes, the cache is completely empty
- The variable `i` is a CPU register
- All data transfers between the cache and main memory (DRAM) are in units of 4 bytes

Also, assume that there is a separate instruction cache and that the instructions of the code above are resident in the instruction cache, so no cache misses due to instruction fetches occur.

(a) For which values of `i` is it likely that cache misses will occur? Explain briefly.

Cache misses are likely when `i=0`, `i=8`, `i=16`, and `i=24`. `arr[0]` is at the beginning of a cache block, and each 32-byte cache block consists of 8 array elements. The cache is initially empty, so each time a new block is accessed, a miss occurs.

(b) How many 4 byte reads from main memory (DRAM) occur when the code above is executed? Explain briefly.

The total number of 4 byte loads is 32. Each of the 4 misses requires loading 8 4-byte values (where $8 \times 4 = 32$), since each miss causes a 32-byte block of data to be read from DRAM to create a replica of the block in the cache.

(c) How many 4 byte writes to main memory (DRAM) are likely to occur when the code above is executed? Explain briefly.

Most likely, 0 writes to DRAM will occur, because in a write-back cache, data is only written to DRAM when a dirty block is evicted. Because the cache starts out as empty, the only way an eviction could occur is if the capacity of the cache allows it to store fewer than 4 blocks (i.e., fewer than 128 bytes), which is unlikely, since that would be a very small cache.

Question 5. [25 points] In a cache, addresses are 8 bits, blocks are 16 bytes, there are 4 sets, and the cache is 2-way set associative.

Complete the following table. For each address in the *Request* column, indicate the tags of cached blocks after handling the request. Addresses are specified in base-2. Assume each request is a load, and that they execute sequentially (top row is the first in the sequence.) All slots are initially empty. When a block is evicted, select the victim using the LRU (Least Recently Used) replacement policy.

blue = offset, magenta = index, light blue = tag

Request	Set 0		Set 1		Set 2		Set 3	
	Slot 0	Slot 1	Slot 0	Slot 1	Slot 0	Slot 1	Slot 0	Slot 1
	empty	empty	empty	empty	empty	empty	empty	empty
<u>00110110</u>							00	
<u>10010011</u>			10				↓	
<u>11010111</u>			↓	11			↓	
<u>01000001</u>	01		↓	↓			↓	
<u>10001011</u>	↓	10	*	↓			↓	
<u>10010110</u>	↓	↓	↓	↓			↓	
<u>11001011</u>	11		↓	↓			↓	
<u>10111001</u>	↓	*	↓	↓			↓	10
<u>10000000</u>		↓	↓	↓			↓	↓
<u>01011100</u>			↓	01			↓	↓
<u>00010000</u>			00	↓			↓	↓
<u>11001110</u>	↓	↓	↓	↓			* ↓	↓

Question 6. [10 points] Consider the following x86-64 instruction:

```
movl (%rax), %edi
```

State a reason why a page fault could occur when the CPU attempts to execute this instruction.

The instruction loads a value from the memory location that `%rax` points to. If this address is on a virtual page whose page table entry is marked as not valid, a page fault will occur. One reason this could occur is because the page has not yet been accessed by the process, so the OS kernel hasn't allocated a physical page yet, i.e., the page is in the "uncached" state. Another possibility is that `%rax` is an address not part of any valid virtual memory region, i.e., the page is in the "unallocated" state.

Question 7. [10 points] Assume that the following x86-64 instruction occurs in a program:

```
movl %esi, %r10d
```

Assume that a branch in the program jumps to this instruction. Is it possible that a page fault could occur when the CPU attempts to fetch and execute the above instruction? Answer "Yes" or "No," and then either state a reason a page fault could occur, or explain why a page fault definitely cannot occur.

Yes. A page fault could occur when the CPU tries to fetch the bytes of the machine instruction. For example, if this instruction is on a virtual page that the process has not accessed yet, e.g., this is the first time this code is being executed by the process, the OS kernel likely won't have allocated a physical page for it yet.

Question 8. [10 points] Consider the following program:

```
1:  #include <stdio.h>
2:
3:  int answer = 42;
4:
5:  int main(void) {
6:      printf("The answer is ");
7:      int x;
8:      x = answer;
9:      printf("%d\n", x);
10:     return 0;
11: }
```

Assume that the local variable `x` in the `main` function is allocated as a CPU register (e.g., `%esi`.)

(a) Explain why a page fault is likely to occur when line 8 is executed.

The memory reference to read the value of the global variable `answer` is likely the first reference to memory in the executable's `.data` section. A page fault will occur, since this might be the first reference to data on that page.

(b) Explain why in handling the page fault you explained in (a), the OS kernel's page fault handler will read data from a file. Explain which file the page fault handler will read from, and what data will be read from that file.

To handle the page fault, the OS kernel will allocate a page of physical memory, and then read one page of data from the `.data` section of the executable file, which contains the initial values for initialized global variables, and (in this case) the initial value of the `answer` variable. This data needs to be present in the allocated physical page before the OS kernel maps it into the address space of the process that experienced the page fault.

[Extra page for answers and/or scratch work.]

[Extra page for answers and/or scratch work.]