Do not turn the page until 12:30.

Instructions

- This exam contains 14 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed two pages (US letter, double-sided) of handwritten notes.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, headphones, and watches.
- You have 110 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read all questions first and start where you feel the most confident.
- Relax. You are here to learn.

<table>
<thead>
<tr>
<th>Question</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>F9</th>
<th>F10</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible Points</td>
<td>16</td>
<td>4</td>
<td>16</td>
<td>23</td>
<td>12</td>
<td>10</td>
<td>19</td>
<td>18</td>
<td>14</td>
<td>18</td>
<td>150</td>
</tr>
</tbody>
</table>
Question M1: Numbers [16 pts]

(A) Take the 32-bit numeral (i.e. bit pattern) 0xFF800000. Circle the number representation below that has the smallest magnitude (i.e. closest to 0) for this numeral. [4 pt]

<table>
<thead>
<tr>
<th>Floating Point</th>
<th>Two’s Complement</th>
<th>Unsigned</th>
<th>Two’s AND Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>float: S = 1, E = 0b1111 1111, M = 0, so −∞.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned int: value is $2^{31} + 2^{30} + ... + 2^{23} = 2^{31} + x$.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int: value is $-2^{31} + 2^{30} + ... + 2^{23} = -2^{31} + x$, which is smaller in magnitude.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(B) What value will be read after we try to store $2^{-120} - 2^{-200}$ in a float? (Circle one) [4 pt]

$2^{-120}$ NaN 0 $2^{-120} - 2^{-200}$

$2^{-120}$ is a representable exponent with $E = -120 + $bias $= 0x07$. $2^{-200}$ is 80 exponents smaller, so way off the end of the mantissa, so subtracting has a negligible effect that gets rounded off.

(C) Complete the following C function that returns the signed value of the exponent (not the $E$ field) of a 32-bit floating point numeral (as an unsigned int argument as in Lab 1b). Ignore floating point special cases for this question. [4 pt]

```c
int getExp(unsigned int fp) {
    return ((fp >> 23) & 0xFF) - 127; // could mask before shifting
} // return value gets implicitly cast to int
```

(D) Dubs claims that the expression $(x != (float) x)$ will return True/1 if there was data loss during the cast of int $x$. Do you agree? Briefly explain why or why not. [4 pt]

Works? (circle one): Yes No

Explanation: During the comparison, $x$ will get implicitly cast to float, so this expression will always return False/0.

Question M2: Design Question [4 pts]

(A) Assume we decided to store/encode object files as text files instead of binary files. Name one advantage and one disadvantage of this design decision. [4 pt]

<table>
<thead>
<tr>
<th>Advantage: Some possible answers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Easier to read and interpret by humans</td>
</tr>
<tr>
<td>• Can be read by a human in a text editor (i.e. don’t need to use objdump)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disadvantage: Some possible answers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Consumes more space/memory because each hex digit of binary data now takes up 1 (ASCII) or 2 (Unicode) bytes</td>
</tr>
<tr>
<td>• More complicated process now needed by Linker (has to convert from text to binary) to build an executable</td>
</tr>
</tbody>
</table>
Question M3: Pointers & Memory  [16 pts]

Assume a 64-bit x86-64 machine (little endian). Below is the buggy `pcount_r` function disassembly from the midterm, showing where the code is stored in memory. Hint: read the questions before the assembly!

```
00000000004005d7 <pcount_r>:
4005d7:  b8 00 00 00 00  movl $0x0,%eax
4005dc:  48 85 ff  testq %rdi,%rdi
4005df:  75 02  jne 4005e3 <pcount_r+0xc>
4005e1:  f3 c3  repz retq
4005e3:  48 81 ef  shrq $1, %rdi
4005e6:  e8 ec ff ff ff  callq 4005d7 <pcount_r>
4005eb:  83 e7 01  andl $0x1,%edi
4005ee:  48 01 f8  addq %rdi,%rax
4005f1:  c3  retq
```

(A) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? Use the appropriate bit widths.  [8 pt]

<table>
<thead>
<tr>
<th>Register</th>
<th>Value (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x 0000 0000 0040 05e0</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x 0000 0000 0000 0007</td>
</tr>
<tr>
<td>%rax</td>
<td>0x 0000 0000 0040 05ee</td>
</tr>
<tr>
<td>%sil</td>
<td>0x ca</td>
</tr>
</tbody>
</table>

**leaq (%)%,%rsi,2, %rax**

**addb 2(%rdi), %sil**

`leaq` instruction calculates the address $0x4005e0 + 2*7 = 0x4005ee$.

`addb` instruction pulls the byte at memory address $0x4005e0 + 2 = 0x4005e2$, which is $0xc3$. Adding this with the lowest byte of %rsi yields $0xc3 & 0x07 = 0xca$.

(B) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let `short* shortP = 0x4005e2`. [8 pt]

```c
long* v1 = (long*)((_int/float_*)shortP + 4); // set v1 = 0x4005f2
short v2 = shortP[3]; // set v2 = -1
```

The difference between `v1` and `shortP` is $0x10 = 16$ bytes. Since by pointer arithmetic we are moving 4 “things” away, `shortP` must be cast to a pointer to a data type of size 4 bytes.

As two bytes (short), $-1 = 0x0FFFF$, which is found at addresses $0x4005e8$ and $0x4005e9$. $0x4005e8$ is 6 bytes = 3 shorts ahead of `shortP`.  

Question M4: Procedures & The Stack  [23 pts]

A Caesar cipher takes a string and shifts each character by the same amount (i.e. \(*\text{str} += \text{shift};\) while \(*\text{str} != '0'\)). For example, "justin" shifted by 2 becomes "lwuvkp". Below is the disassembly for an inefficient recursive implementation \texttt{caesar} that returns the length of the string:

```
0000000000400547 <caesar>:
   400547:   0f b6 07       movzbl (%rdi),%eax   # get *str
   40054a:   84 c0       test %al,%al
   40054c:   75 06       jne 400554 <caesar+0xd>
   40054e:   b8 00 00 00 00 mov $0x0,%eax   # base case
   400553:   c3       retq   # returns 0
   400554:   53       push %rbx
   400555:   8b 5c 24 10 mov 0x10(%rsp),%ebx   # get shift
   400559:   01 d8       add %ebx,%eax
   40055b:   88 07       mov %al,(%rdi)
   40055d:   48 83 c7 01 add $0x1,%rdi   # next char
   400561:   53       push %rbx
   400562:   e8 e0 ff ff ff callq 4000547 <caesar>
   400567:   83 c0 01 add $0x1,%eax   # length += 1
   40056a:   5b       pop %rbx
   40056b:   5b       pop %rbx
   40056c:   c3       retq
```

A) Which of the following generates the labels used in the disassembly above? Circle one. [2 pt]

- [Compiler]
- [Assembler]
- [Linker]
- [Loader]

B) What is the return address to \texttt{caesar} that gets stored on the stack during a recursive call? Answer in hex. [2 pt]

The address of the instruction after the \texttt{callq}.  

\[
0x \text{400567}
\]

C) Of the 16 instructions shown in the disassembly, how many of them access memory? [4 pt]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Access Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>movzbl, mov (0x400555), retq (x2), pop (x2)</td>
<td>READ 6</td>
</tr>
<tr>
<td>push (x2), mov (0x40055b), callq</td>
<td>WRITE 4</td>
</tr>
</tbody>
</table>

D) Briefly explain the purpose of the push at 0x400554. [2 pt]

To save the old value of the callee-saved register %rbx, which we are about to change.
(E) Briefly explain the purpose of the push at 0x400561. Hint: what value are we pushing and where did we get it from? Read the comments! [2 pt]

Push $rbx, which currently holds the value of shift (the 7th argument), for the recursive call to read.

(F) Assume main calls caesar on the string "cse" with a shift of 1. Fill in the stack snapshot below (in hex) as this call returns to main. For unknown words, write “0x unknown”. [8 pt]

```
0x7fffffffdcc8 <ret addr to main>
0x7fffffffdcc0 <original rbx>  caesar("cse",...,1)
0x7fffffffdcb8 0x 1  
0x7fffffffdcb0 0x 400567  
0x7fffffffdca8 0x 1  
0x7fffffffdca0 0x 1  
0x7fffffffdcb8 0x 400567  
0x7fffffffdcb9 0x 1  
0x7fffffffdcb8 0x 400567  
0x7fffffffdcb0 0x 1  
```

4 total stack frames of caesar created as shown above, each moving one character forward in the initial string. In the recursive case, first we push the old value in $rbx onto the stack before pushing the new value of $rbx (the value of shift read from the previous stack frame). The last stack frame hits the base condition and doesn’t push anything onto the stack.

(G) Name a way that we can reduce the memory usage of this function (either in amount of memory or number of memory accesses) while maintaining correct behavior and keeping it recursive and explain why the change helps. [3 pt]

Some acceptable responses:

- Pass shift in an unused argument register, which saves us from pushing it to the stack on every recursive call.
- Read shift into a caller-saved register instead of $rbx, so we don’t need to push the old value of the register at the beginning of every recursive call.
- Replace the first pop (0x40056a) with addq $8, %rsp, since we don’t need/use the restored value of shift.
Question M5: C & Assembly  [12 pts]

Answer the questions below about the following x86-64 assembly function, which uses a struct with two fields named `one` and `two`, declared in that order:

(A) %rdi contains a pointer to an instance of the struct. What variable type is field `one`?  [2 pt]

In Line 4, (%rdi) is used in a cmpb instruction.

___ char ___

(B) Based on Line 7, give a more intuitive name for the field `two` in the struct.  [1 pt]

Other variants accepted.

(C) This function fits into the following code skeleton. Fill in the corresponding parts below, using register names as variable names (e.g. al for the value in %al). None should be blank. Remember that the struct fields are named `one` and `two`.  [9 pt]

Grading notes:
- %rdi also accepted for loop Condition statement.
- *%rdi in place of %rdi->one in if-statement received partial credit.
Question F6: Structs [10 pts]

For this question, assume a 64-bit machine and the following C struct definition.

```c
typedef struct {
  char* title; 8 // title (e.g. "HW SW INTERFACE")
  char dept[3]; 1 // dept (e.g. "CSE")
  short num; 2 // course number (e.g. 351)
  int enrolled; 4 // students enrolled
} course;  Kmax = 8
```

(A) How much memory, in bytes, does an instance of `course` use? How many of those bytes are internal fragmentation and external fragmentation? [6 pt]

<table>
<thead>
<tr>
<th>sizeof(course)</th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 bytes</td>
<td>3 bytes</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Alignment requirements listed above in red next to the struct fields. A `course` instance:

```
<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>11</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>20</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>dept</td>
<td>num</td>
<td>enrolled</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The unused bytes around `num` count as internal fragmentation, the unused bytes after `enrolled` count as external fragmentation.

(B) Assume that an instance `course c` is allocated on the stack and an array `char ar[]` is allocated 40 bytes below `c` (i.e. `&ar + 0x28 == (char*)&c`). Fill in the blanks below with the new ASCII characters stored in `c.dept` after the following loop is executed. Hint: recall that the values 0x30 to 0x39 correspond to the ASCII characters '0' to '9'. [4 pt]

```c
for (int i = 0; i < 52; ++i) {
  ar[i] = i;
}
```

Starting from the beginning of `ar`, we store the values 0 to 39 before we reach the struct `c`. The values 40 to 47 overwrite the bytes of `c.title` (address 0x2f2e2d2c2b2a2928, assuming little-endian). `c.dept` then gets overwritten with the values 48 = 0x30 = '0', 49 = 0x31 = '1', and 50 = 0x32 = '2'.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>'1'</td>
<td>'2'</td>
</tr>
</tbody>
</table>
Question F7:  Caching  [19 pts]

We have 256 KiB of RAM and a 4-KiB L1 data cache that is 2-way set associative with 32-byte blocks and random replacement, write-back, and write allocate policies.

(A) Calculate the TIO address breakdown:  [3 pt]

<table>
<thead>
<tr>
<th>Tag bits</th>
<th>Index bits</th>
<th>Offset bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

18 address bits. $\log_2 32 = 5$ offset bits. $2^{12}$-B cache = 128 blocks. 2 blocks/set $\rightarrow 64 = 2^6$ sets.

(B) The code snippet below accesses two arrays of doubles. Assuming $i$ is stored in a register and the cache starts cold, give the memory access pattern (read or write to which elements/addresses) and compute the miss rate.  [6 pt]

```c
#define SIZE 128
double src[SIZE];  // &src = 0x08000 (physical addr)
double dst[SIZE];  // &dst = 0x0E000 (physical addr)
for (int i = 0; i < SIZE; i += 1) {
    dst[i] = src[i];
    src[i] = i;
}
```

Per Iteration: (circle) $\rightarrow$ Access 1: R / W to src[$i$] (fill in) $\rightarrow$ Access 2: R / W to dst[$i$] Access 3: R / W to src[$i$]

src[$i$] and dst[$i$] map into the same set because their index fields match. However, our cache is 2-way set associative, so they do not conflict. Each block holds 32 B = 4 doubles, so for the 4 iterations in the same cache block, we get MMH|HHH|HHH|HHH for a miss rate of $2/12 = 1/6$.

(C) For each of the proposed (independent) changes, draw ↑ for “increased”, — for “no change”, or ↓ for “decreased” to indicate the effect on the miss rate from Part B for the code above:  [8 pt]

Use float instead __↓__ Double the cache size __——
Half the associativity __↑__
No-write allocate __↑__

Using floats means we access each block twice as much (MR = 1/12). Doubling cache size doubles the number of sets, but src[$i$] and dst[$i$] still map to the same set. Direct-mapped would cause src[$i$] and dst[$i$] to generate conflict misses. No-write allocate means we don’t bring in the block for dst into the cache on access 2, so future access 2s continue to be Misses.

(D) Assume it takes 160 ns to get a block of data from main memory. If our L1 data cache has a hit time of 5 ns and a miss rate of 5%, what is our average memory access time (AMAT)?  [2 pt]

$$AMAT = HT + MR \times MP = 5 \text{ ns} + 0.05 \times 160 \text{ ns} = 5 + 8 \text{ ns} \quad 13 \text{ ns}$$
**Question F8: Processes [18 pts]**

(A) The following function prints out four numbers. In the following blanks, list three possible outcomes: [6 pt]

```c
void concurrent(void) {
    int n = 5;
    if (fork()) {
        n++;
        if (fork()) {
            n++;
            wait();
        }
        printf("%d, ", n);
        exit(0);
    } else {
        printf("%d, ", n);
    }
    printf("%d, ", n);
    exit(0);
}
```

The 7 possible outcomes:
1) 5, 5, 6, 7,
2) 5, 5, 7, 6,
3) 5, 6, 5, 7,
4) 5, 6, 7, 5,
5) 6, 5, 5, 7,
6) 6, 5, 7, 5,
7) 6, 7, 5, 5,

(B) For the following examples of exception causes, write “S” for synchronous or “A” for asynchronous from the perspective of the user process. [4 pt]

- System call  ___S___
- Divide by zero  ___S___
- Segmentation fault  ___S___
- Key pressed  ___A___

Everything but a key press is caused by an assembly instruction within your program.

(C) Fill in the following blanks with “A” for always, “S” for sometimes, and “N” for never if the following would be different when context switching to a different process? [4 pt]

- Process ID  ___A___
- Program  ___S___
- PTBR  ___A___
- Condition Codes  ___S___

Every process has a unique ID and its own page table, but could be running different instances of the same program. Each process has its own execution state (including the condition codes), but it is possible that the condition codes have the same values at the instance we switch.

(D) Is the following statement True or False? Provide a brief justification: a single process can execute multiple programs simultaneously. [4 pt]

Circle one: True / False

**Justification:** One process is dedicated to running one program at a time. The program defines the instructions, initial memory state, etc. of the process, so two programs can’t exist within the same process at once.
Question F9: Virtual Memory  [14 pts]

Our system has the following setup:
- 15-bit virtual addresses and 2 KiB of RAM with 256-byte pages
- A 4-entry fully-associative TLB with LRU replacement
- A PTE contains bits for valid (V), dirty (D), read (R), write (W), and execute (X)

(A) Compute the following values:  [8 pt]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th># of TLB sets</th>
<th>1 set</th>
</tr>
</thead>
<tbody>
<tr>
<td>page offset width</td>
<td>8 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td># of virtual pages</td>
<td>$2^7$ pages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minimum width of PTBR</td>
<td>11 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page offset is $\log_2 256 = 8$ bits wide. # of virtual pages is $2^{15} - p = 2^7$. The TLB is fully-associative, so only has 1 set. The page table lives in physical memory, so the PTBR must hold its physical address, which need to be at least 11 bits wide to address all 2 KiB of RAM.

(B) Assuming that the TLB is in the state shown (permission bits: 1 = allowed, 0 = disallowed), give example addresses that will fulfill the following scenarios:  [6 pt]

Find the desired entry in the TLB. Because the TLB is fully-associative, the TLB tag is exactly the virtual page number (VPN). Any page offset within this page will access that TLB entry.

<table>
<thead>
<tr>
<th>TLB</th>
<th>PPN</th>
<th>Valid</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>0xc</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x7f</td>
<td>0xa</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x7e</td>
<td>0xf</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x04</td>
<td>0xe</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A value in %rip that causes a TLB Hit and no exception:  
Want TLB entry with $V=1$, $X=1 \rightarrow$ VPN $0x04$.
A write address that causes a TLB Hit and segmentation fault:  
Want TLB entry with $V=1$, $W=0 \rightarrow$ VPN $0x2000$.

Grading notes:
- Answers without leading zeros accepted.
Question F10: Memory Allocation [18 pts]

(A) In the following code, briefly identify the TWO memory issues and their fixes. [6 pt]

```c
int N = 32;
long* func(long src[]) {
    long* p = (long*) malloc(N * sizeof(long));
    for (int i = 0; i < N; i++) {
        p[i] += src[i];
    }
}
```

Error 1: Using uninitialized memory in p[i].
Fix 1: Replace malloc with calloc or Change p[i] += src[i]; to p[i] = src[i];

Error 2: Memory leak – no way to access malloc’ed memory once func returns.
Fix 2: Add return p; at end of func.

(B) We are using a dynamic memory allocator on a 64-bit machine with an explicit free list, 16-byte boundary tags, and 8-byte alignment. Assume that a footer is always used. [6 pt]

<table>
<thead>
<tr>
<th>Request</th>
<th>block addr</th>
<th>return value</th>
<th>block size</th>
<th>internal fragmentation in this block</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = malloc(9);</td>
<td>0x628</td>
<td>0x_638_</td>
<td>48 bytes</td>
<td>39 bytes</td>
</tr>
</tbody>
</table>

Payload (returned addr) starts a header size after the block. Need at least 32 B for boundary tags and 9 B for payload = 41 B; padding for 8-B alignment gets us to 48 B (also the minimum block size in this explicit free list). Internal fragmentation is block size – payload = 48 – 9 = 39 B.

(C) Consider the C code shown here. Assume that the malloc call succeeds and that all variables are stored in memory (not registers). In the following groups of expressions, circle the one whose returned value (assume just before return 0) is largest. [6 pt]

```c
#include <stdlib.h>
long glob = 10;
char* str = "351";

int main() {
    short* sp = malloc(8);
    int ONE = 1;
    free(sp);
    return 0;
}
```

<table>
<thead>
<tr>
<th>Group 1:</th>
<th>&amp;sp</th>
<th>sp</th>
<th>&amp;str</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 2:</td>
<td>&amp;glob</td>
<td>main</td>
<td>str</td>
</tr>
<tr>
<td>Group 3:</td>
<td>glob</td>
<td>ONE</td>
<td>*str</td>
</tr>
</tbody>
</table>

8) &sp/&ONE (Stack)
7) sp (Heap)
6) &glob/&str (Static Data)
5) str (Literals)
4) main (Code)
3) *str ('3' = 0x33)
2) glob (10)
1) ONE (1)