CSE 410 Final Review Solutions

Question 1: Number Representation

(A) What is the value of the char 0b 1101 1101 in decimal?
   If x = 0xDD, -x = 0x23 = 2^5+3 = 35
   Also accepted unsigned: 0xDD = (16+1)*13 = 221
   -35 or 221

(B) What is the value of char z = (0xB << 7) in decimal?
   0xB << 7 = 0b 1000 0000 = TMinchar = -128
   Also accepted unsigned: 0x80 = 128
   -128 or 128

(C) Let char x = 0xC0. Give one value (in hex) for char y that results in both signed and unsigned overflow for x+y.
   x<0, so need large enough (in magnitude) neg num for signed overflow. Unsigned overflow comes naturally along with this.
   0x80 ≤ y ≤ 0xBF

For the rest of this problem we are working with a floating point representation that follows the same conventions as IEEE 754 except using 8 bits split into the following vector widths:

<table>
<thead>
<tr>
<th>Sign (1)</th>
<th>Exponent (4)</th>
<th>Mantissa (3)</th>
</tr>
</thead>
</table>

(D) What is the magnitude of the bias of this new representation?
   Bias = 2^{4-1} - 1 = 7
   7

(E) Translate the floating point number 0b 1100 1110 into decimal.
   Notice that E indicates this is not a special case.
   Exp = 9 - 7 = 2, Man = 1.1102.
   \((-1)^1 \times 1.1102 \times 2^2 = -111_2 = -7\).
**Question 2: Pointers & Memory**

For this problem we are using a 64-bit x86-64 machine (little endian). Below is the factorial function disassembly, showing where the code is stored in memory.

![Disassembly](image)

(A) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? Remember to use the appropriate bit widths.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x0000 0000 0040 052D</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x0000 0000 0000 0003</td>
</tr>
<tr>
<td>%eax</td>
<td>0x0040 0530</td>
</tr>
<tr>
<td>%bl</td>
<td>0x07</td>
</tr>
</tbody>
</table>

The **leal** stores the *address* calculated by adding the contents of %rdi and %rsi together. The address calculation for movb equates to 0x40052D + 9. Nine bytes past the start of fact is the byte 0x07.

(B) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let `char* cp = 0x40052D`.

```c
char v1 = *(cp + 8); // set v1 = 0x75
int* v2 = (int*)((long/ double*)cp + 2); // set v2 = 0x40053D
```

The only 0x75 byte in fact is found at address 0x400535, 8 bytes beyond cp. The difference between v2 and cp is 16 bytes. Since by pointer arithmetic we are moving 2 “things” away, cp must be cast to a data type of size 8 bytes.
**Question 3: The Stack**

The recursive Fibonacci sequence function `fib()` and its x86-64 disassembly are shown below:

```c
int fib (int n) {
    if (n<2)
        return 1;
    else
        return fib(n-2) + fib(n-1);
}
```

000000000040055d <fib>:

- **000000000040055d:**
  - `55`: push %rbp
  - `53`: push %rbx
  - `89 fb`: mov %edi,%ebx
  - `83 ff 01`: cmp $0x1,%edi
  - `7e 16`: jle 40057c <fib+0x1f>
  - `8d 7f fe`: lea -0x2(%rdi),%edi
  - `e8 ef ff ff ff`: callq 40055d <fib>
  - `89 c5`: mov %eax,%ebp
  - `8d 7b ff`: lea -0x1(%rbx),%edi
  - `e8 e5 ff ff ff`: callq 40055d <fib>
  - `01 e8`: add %ebp,%eax
  - `eb 05`: jmp 400581 <fib+0x24>
  - `b8 01 00 00 00`: mov $0x1,%eax
  - `5b`: pop %rbx
  - `5d`: pop %rbp
  - `c3`: retq

(A) In no more than a sentence, explain what the instruction at address 0x40055f does (in terms of the function – don’t be too literal) and why it is necessary.

It is saving the current value of `n` into a callee-saved register (%rbx) so that it doesn’t get overwritten by the first recursive call.
(B) How much space (**in bytes**) does this function take up in our final executable?

Count all bytes (middle columns) or subtract 0x40055d from the address of the next instruction after `fib` (0x400584).

**39 B**

(C) Calling `fib(4)`: How many total `fib` stack frames are created?

```
fib(4)  →  fib(2)  →  fib(0)
         →  fib(1)
         →  fib(3)  →  fib(1)
         →  fib(2)  →  fib(0)
         →  fib(1)
```

**9**

(D) Calling `fib(4)`: What is the *maximum* amount of memory on the stack (**in bytes**) used for `fib` stack frames at any given time?

The maximum depth is 4 stack frames. From the assembly code, we know that `%rbp` and `%rbx` get pushed onto the stack *every* time `fib` is called. The return address to `fib` is also pushed whenever we make a recursive call, so that makes 3 words for the first 3 levels and only 2 words for the 4th level – a total of 11 words = 88 bytes. (96 bytes if counting return address as part of Callee’s stack frame – which is valid according to the x86-64 Application Binary Interface)

**88 or 96 bytes**

(E) Below is an incomplete snapshot of the stack during the call to `fib(4)`. Fill in the values of the four missing intermediate words in hex:

<table>
<thead>
<tr>
<th>%rbp and %rbx onto the stack</th>
<th>%rbx holds the old value of n (i.e. the node above this one in the “tree”). %rbp is used to hold the return value of the first recursive call to fib.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffc39b72e0</td>
<td>&lt;ret addr to main&gt;</td>
</tr>
<tr>
<td>0x7fffc39b72d8</td>
<td>&lt;original rbp&gt;</td>
</tr>
<tr>
<td>0x7fffc39b72c0</td>
<td>&lt;original rbx&gt;</td>
</tr>
<tr>
<td>0x7fffc39b72d0</td>
<td>0x400578</td>
</tr>
<tr>
<td>0x7fffc39b72c8</td>
<td>0x2</td>
</tr>
<tr>
<td>0x7fffc39b72c0</td>
<td>0x4</td>
</tr>
<tr>
<td>0x7fffc39b72b8</td>
<td>0x400578</td>
</tr>
<tr>
<td>0x7fffc39b72b0</td>
<td>0x1</td>
</tr>
<tr>
<td>0x7fffc39b72a8</td>
<td>0x3</td>
</tr>
</tbody>
</table>

In `fib(3)`’s stack frame, it will have stored 0x2 (the return value from the first recursive call of `fib(4)`) and 0x4 (`n` of the function that called `fib(3)`).

`fib(2)` is the second recursive call of `fib(3)` and `fib(3)` is the second recursive call of `fib(4)`, so the return address 0x400578 (not 0x40056e) is pushed onto the stack in both cases.
Question 4: C & Assembly

We are writing the recursive function `search`, which takes a `char` pointer and returns the address of the first instance in the string of a specified `char` `c`, or the null pointer if not found.

Example: `char* p = "TeST oNe"`, then `search(p, 'N')` will return the address `p+6`.

```
char *search (char *p, char c) {
    if (!*p)
        return 0;
    else if (*p==c)
        return p;
    return search(p+1,c);
}
```

Fill in the blanks in the x86-64 code below with the correct instructions and operands. Remember to use the proper size suffixes and correctly-sized register names!

```
search(char*, char):

1       movzbl (%rdi), %eax     # get *p
2       testb %al, %al        # conditional
3       je   .NotFound       # conditional jump
4       cmpb %sil, %al        # conditional
5       je   .Found          # conditional jump
6       addq $1, %rdi        # argument setup
7       call search         # recurse
8       ret

  .NotFound:
9       movl $0, %eax        # return value
10      ret

  .Found:
11      movq %rdi, %rax      # return value
12      ret
```

Grading Notes:

**Line 2:** `cmpb $0, %al` also accepted.

**Line 6:** Given that argument `p` is a pointer, needed to use the full `addq` and `%rdi`.

**Line 7:** `callq` also accepted
**Question 5: Caching**

We have 16 KiB of RAM and two options for our cache. Both are two-way set associative with 256 B blocks, LRU replacement, and write-back policies. **Cache A** is size 1 KiB and **Cache B** is size 2 KiB.

(A) Calculate the TIO address breakdown for **Cache B**: [1.5 pt]

<table>
<thead>
<tr>
<th>Tag bits</th>
<th>Index bits</th>
<th>Offset bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

14 address bits. \( \log_2 256 = 8 \) offset bits. 2 KiB cache = 8 blocks. 2 blocks/set \( \rightarrow \) 4 sets.

(B) The code snippet below accesses an integer array. Calculate the **Miss Rate** for **Cache A** if it starts **cold**. [3 pt]

```c
#define LEAP 4
#define ARRAY_SIZE 512

int nums[ARRAY_SIZE];       // &nums = 0x0100 (physical addr)
for (i = 0; i < ARRAY_SIZE; i+=LEAP)
    nums[i] = i*i;
```

Access pattern is a single write to \( \text{nums}[i] \). Stride = LEAP = 4 ints = 16 bytes. 256/16 = 16 strides per block. First access is a compulsory miss and the next 15 are hits. Since we never revisit indices, this pattern continues for all cache blocks. You can also verify that the offset of \&nums is 0x00, so we start at the beginning of a cache block.

(C) For each of the proposed (independent) changes, write **MM** for “higher miss rate”, **NC** for “no change”, or **MH** for “higher hit rate” to indicate the effect on **Cache A** for the code above:[3.5 pt]

- Direct-mapped: _NC_
- Increase block size: _MH_
- Double LEAP: _MM_
- Write-through policy: _NC_

Since we never revisit blocks, associativity doesn’t matter. Larger block size means more strides/block. Doubling LEAP means fewer strides/block. Write hit policy has no effect.

(D) Assume it takes 200 ns to get a block of data from main memory. Assume **Cache A** has a hit time of 4 ns and a miss rate of 4% while **Cache B**, being larger, has a hit time of 6 ns. What is the worst miss rate **Cache B** can have in order to perform as well as **Cache A**? [2 pt]

\[
\text{AMAT}_A = \text{HT}_A + \text{MR}_A \times \text{MP} = 4 + 0.04 \times 200 = 12 \text{ ns.}
\]

\[
\text{AMAT}_B = \text{HT}_B + \text{MR}_B \times \text{MP} \leq 12 \rightarrow 200 \times \text{MR}_B \leq 6 \rightarrow \text{MR}_B \leq 0.03
\]

0.03 or 3%
Question 6: Programs, processes, and processors (oh my!)

(a) Consider the following C code on the left (running on Linux), then give one possible output of running it. Assume that printf flushes its output immediately.

```c
void oz() {
    char * name = "toto\n";
    printf("dorothy\n");
    if (fork() == 0) {
        name = "wizard\n";
        printf("scarecrow\n");
        fork();
        printf("tinman\n");
        exit(0);
        printf("witch\n");
    } else {
        printf("lion\n");
    }
    printf(name);
}
```

Possible output:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dorothy</td>
<td>dorothy</td>
</tr>
<tr>
<td>scarecrow</td>
<td>lion</td>
</tr>
<tr>
<td>tinman</td>
<td>toto</td>
</tr>
<tr>
<td>tinman</td>
<td>scarecrow</td>
</tr>
<tr>
<td>lion</td>
<td>tinman</td>
</tr>
<tr>
<td>toto</td>
<td>tinman</td>
</tr>
</tbody>
</table>

(b) "Pay no attention to the man behind the curtain." We have seen several different mechanisms used to create illusions or abstractions for running programs:

A. Context switch
B. Virtual memory
C. Virtual method tables (vtables)
D. Caches
E. Timer interrupt
F. Stack discipline
G. None of the above, or impossible.

For each of the following, indicate which mechanism above (A-F) enables the behavior, or G if the behavior is impossible or untrue.

(i) _____ Allows operating system kernel to run to make scheduling decisions.

(ii) _____ Prevents buffer overflow exploits.

(iii) _____ Allows multiple instances of the same program to run concurrently.

(iv) _____ Lets programs use more memory than the machine has.

(v) _____ Makes recently accessed memory faster.

(vi) _____ Multiple processes appear to run concurrently on a single processor.

(vii) _____ Enables programs to run different code depending on an object’s type.

(viii) _____ Allows an x86-64 machine to execute code for a different ISA.
(c) Give an example of a synchronous exception, what could trigger it, and where the exception handler would return control to in the original program.

**Page fault**: triggered by access to virtual address not in memory, returns to the instruction that caused the fault.

**Trap**: used to for syscalls to do something protected by the kernel, returns to after the calling instruction.

(d) In what way does address translation (virtual memory) help make `exec` fast? Explain in less than 2 sentences. *Hint*: it may help to write down what happens during `exec`.

*Address translation is a form of indirection, it allows us to implement fork without copying the whole process’s memory, and exec without loading the whole program into memory at once.*

(e) Which of the following can a running process determine, assuming it does not have access to a timer? *(check all that apply)*

- [x] Its own process ID
- [ ] Size of physical memory
- [x] Size of the virtual address space
- [ ] L1 cache associativity
- [ ] When context switches happen

(f) For each of the following, fill in what is responsible for making the decision: hardware ("HW"), operating system ("OS"), or program ("P").

(i) __________ Which physical page a virtual page is mapped to.

(ii) **HW** Which cache line is evicted for a conflict in a set-associative cache.

(iii) **OS** Which page is evicted from physical memory during a page fault.

(iv) **HW** Translation from virtual address to physical address.

(v) **P** Whether data is stored in the stack or the heap.

(vi) **P** Data layout optimized for spatial locality
**Question F7: Virtual Memory**

Our system has the following setup:

- 24-bit virtual addresses and 512 KiB of RAM with 4 KiB pages
- A 4-entry TLB that is fully associative with LRU replacement
- A page table entry contains a valid bit and protection bits for read (R), write (W), execute (X)

(A) Compute the following values:

<table>
<thead>
<tr>
<th></th>
<th><strong>12</strong></th>
<th><strong>7</strong></th>
<th><strong>12</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Page offset width</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPN width</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entries in a page table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLBT width</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because TLB is fully associative, TLBT width matches VPN. There are $2^{VPN\text{ width}}$ entries in PT.

(B) Briefly explain why we make the page size so much larger than a cache block size.

Take advantage of spatial locality and try to avoid page faults as much as possible.
Disk access is also super slow, so we want to pull a lot of data when we do access it.

(C) Fill in the following blanks with “A” for always, “S” for sometimes, and “N” for never if the following get updated during a page fault.

Page table _A_ Swap space _S_ TLB _A/N_ Cache _S_ When the page is placed in physical memory, the new PPN is written into the page table entry. Swap space will get updated if a dirty page is kicked out of physical memory.

For this class, we say that the page fault handler updates the TLB because it is more efficient.

In reality not all do (OS does not have access to hardware-only TLB; instr gets restarted). To update a PTE (in physical mem), you check the cache, so it gets updated on a cache miss.

(D) The TLB is in the state shown when the following code is executed. Which iteration (value of i) will cause the protection fault (segfault)? Assume sum is stored in a register.

Recall: the hex representations for TLBT/PPN are padded as necessary.

```c
long *p = 0x7F0000, sum = 0;
for (int i = 0; i < 513; i++) {
    if (i%2)
        *p = 0;
    else
        sum += *p;
    p++;
}
```

<table>
<thead>
<tr>
<th>TLBT</th>
<th>PPN</th>
<th>Valid</th>
<th>R</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7F0</td>
<td>0x31</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x7F2</td>
<td>0x15</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x004</td>
<td>0x1D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x7F1</td>
<td>0x2D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ i = 513 \]

Only the current page (VPN = TLBT = 0x7F0) has write access. Once we hit the next page (TLBT = 0x7F1), we will encounter a segfault once we try to write to the page. We are using pointer arithmetic to increment our pointer by 8 bytes at a time. One page holds $2^{12}/2^3 = 512$ longs, so we first access TLBT 0x7F1 when $i = 512$. However, the code is set up so that we only write on odd values of $i$, so the answer is $i = 513$. 

9