CMSI 284 Simple Machine Exercise
or, the Da Vinci Opcode

Submitted by: ______________________________

Instructions

Work out and answer the following questions based on the simple machine described in `http://cs.lmu.edu/~ray/notes/simplecomputer/`. You may submit this assignment in one of these ways:

- If you know LaTeX sufficiently, copy the *source file* of this exercise and add your solutions to this copy. Commit and push the file to your GitHub repository. *Advantage:* Drop-dead clear, sharp, unambiguous presentation. *Disadvantage:* Intermediate work such as the state of the machine at a given moment may be harder to write down.
- Alternatively, you may *print* the PDF version of this exercise and do your work on paper. Submit this printout with your *name* in the designated blank up top. *Advantage:* More convenient for showing your work. *Disadvantage:* Handwritten answers may be harder to read.

  If answering on paper, use separate blank sheets for your answers/code if necessary. Don't try to cram everything solely within the available space.

Mapping to Outcomes and Proficiencies

The overall assignment covers outcomes *1c*, *4d*, and *4f*. Each question specifically pertains to *1c* and will be given a score ranging from 0 to 4 based on the correctness of the answer. The average score for a given outcome, rounded, determines the final proficiency for the assignment.

Outcome *4d* will be determined by how well you use the information given in class to compute the requested answers, and how accurately you follow the instructions in this assignment.

Outcome *4f* will be determined by whether you submit the assignment on time.

1 Disassembly

Choose two (2) out of the following five machine language programs then: (a) disassemble them into an equivalent assembly language program and (b) state the computation that they perform. Some ground rules and tips:

- There might be "garbage" (i.e., unused) words in the memory dump, so just follow the instructions and trust in what they make the machine do.
- When disassembling the programs, choose meaningful labels, just as you would when writing programs in higher-level languages.
- If you recognize that the operations being performed result in something that is overall more meaningful (e.g., "the program squares the given number then multiplies it by π") then describe the program in those more meaningful terms rather than just spelling out the operations (e.g., "the program computes the area of a circle with the given radius").
- To help remember how you are interpreting the code, you may annotate your code with a comment (e.g., "`; current value of counter`").
- Many of these programs read or write values to and from ports. Make sure to recognize which ports are being used for what.

1.
```
0000000: C0000004
0000001: 0000000A
0000002: 00000001
0000003: 00000001
0000004: 00000002
0000005: 60000001
0000006: 10000002
0000007: 00000001
0000008: 50000003
0000009: D000000C
000000A: 10000001
000000B: C0000004
000000C: 00000002
000000D: 30000FAC
000000E: C000000E
000000F: 000000F1
0000010: 10000002
0000011: 30000004
0000012: 8000009A
```

```
0000013: 10000000
0000014: 20000AEF
0000015: FFFFFFFF
0000016: 00000000
```

dec:	1		;
factorial:	1		;
count:	10		;
	JUMP	start	;
start:	LOAD	factorial	;
	MUL	count	; multiply accumulator value by count
	STORE	factorial	;
	LOAD	count	; loads count to accum
	SUB	dec	; decrement count
	JZ	output	; jumps to endstage of program
	STORE	count	; stores count if necessary
	JUMP	start	;
output:	LOAD	factorial	;
	OUT	FAC	;
end:	JUMP	end	; infinite loop to end program

This program outputs (10!)

2. (*Hint:* Keep those bitwise operations in mind.)

```
0000000: 20000016
0000001: 10000008
0000002: 50000009
0000003: D0000003
0000004: 00000008
0000005: B0000009
0000006: 30000050
0000007: C0000000
0000008: 00000000
0000009: FFFFFFFF
000000A: 20000111
000000B: 00000003
000000C: 10000022
000000D: C0000005
000000E: 40000001
000000F: 30000001
0000010: 40000004
0000011: A000001D
```

```
0000012: 90000002
0000013: B000001E
0000014: 00000001
0000015: F0000002
0000016: 40000EBD
```

3. (*Hint:* You have probably written this program before.)

```
0000000: 200EA000
0000001: 10000014
0000002: 80000011
0000003: D0000005
0000004: C000000C
0000005: 00000014
0000006: 80000012
0000007: D0000009
0000008: C000000E
0000009: 00000014
000000A: 80000013
000000B: D000000E
000000C: 00000015
000000D: C000000F
000000E: 00000016
000000F: 300FF000
0000010: C0000010
0000011: 00000004
0000012: 00000064
0000013: 00000190
0000014: 00000000
0000015: 00000000
0000016: FFFFFFFF
```

4. (*Hint:* What is happening here relates to a well-known theorem.)

```
0000000: C0000003
0000001: 00FEA007
0000002: 9001A2B3
0000003: 20F80000
0000004: 10000001
0000005: 60000001
```

```
0000006: 10000001
0000007: 20F80000
0000008: 10000002
0000009: 60000002
000000A: 40000001
000000B: 30F90000
000000C: C000000C
000000D: 10000014
000000E: 80000011
000000F: D0000005
0000010: C000000C
0000011: 00000014
0000012: 80000012
0000013: D0000009
0000014: C000000E
0000015: 00000014
0000016: 80000013
```

a:	0		;
b:	0		;
	JUMP	start	;
start:	IN	F80000	;
	STORE	a	; stores input into memloc 1
	MUL	a	; squares input into accum
	STORE	a	; stores square of input
	IN	F80000	;
	STORE	b	;
	MUL	b	;
	ADD	a	;
	OUT	F90000	;
end:	JUMP	end	;

This program outputs the result of summing two squares from input.

5. (*Hint:* Remember the first 128 Unicode codepoints when reading through this one.)

```
0000000: C0000005
0000001: 00000061
0000002: 0000007A
0000003: 00000020
0000004: FA001F1F
0000005: 20000032
```

```
0000006: D0000006
0000007: 10000004
0000008: 50000001
0000009: E0000010
000000A: 00000004
000000B: 50000002
000000C: F0000010
000000D: 00000004
000000E: 50000003
000000F: 10000004
0000010: 00000004
0000011: 30000064
0000012: C0000005
0000013: 10000008
0000014: 50000009
0000015: D0000003
0000016: 00000008
```

2 Assembly

Choose three (3) out of the following five program descriptions then: (a) implement them in assembly language and (b) assemble your code into machine language. Start your assembled programs at memory location `0000000`. When reading/writing to/from ports is requested, you may choose the port numbers to use, but you should explicitly state which port is being used for what.

In all cases, assume that the programs work in infinite loops: after accepting the input and processing the output, they all start over and accept new input from the designated port.

1. *Two digit decimal-to-hex converter:* A program that reads two words from a given port that are meant to represent individual decimal digits. If the words are not in the range `0` to `9`, then the error result `FFFFFFFF` is sent to an output port. Otherwise, the integer represented by the two decimal digits is sent.

```
size_test:    9                     ;
raise_order:  A                     ;
error_out:    FFFFFFFF              ;
input_port:   444                   ;
output_port:  999                   ;
              JUMP     start        ;
start:        IN       input_port   ;
              SUB      size_test    ;
              JGZ      error        ;
              ADD      size_test    ;
              STORE    a            ; first digit
              IN       input_port   ;
              SUB      size_test    ;
              JGZ      error        ;
              ADD      size_test    ;
              STORE    b            ; second digit
              LOAD     a            ;
              MUL      raise_order  ;
              ADD      b            ;
              OUT      output_port  ;
              JUMP     start        ;
error:        LOAD     error_out    ;
              OUT      output_port  ;
              JUMP     start        ;
```

```
0000000: C0000006;
0000001: 00000000; a
0000002: 00000000; b
0000003: 00000009;
0000004: 0000000A;
0000005: FFFFFFFF;  error_out
0000006: 20000444;
0000007: 50000003;
0000008: F0000015;  jump if error
0000009: 40000003;
000000A: 10000001;
000000B: 20000444;
000000C: 50000003;
000000D: F0000015;
000000E: 40000003;
000000F: 10000002;
```

```
0000010: 00000001;
0000011: 60000004;
0000012: 40000002;
0000013: 30000999;
0000014: C0000006;
0000015: 00000005;
0000016: 30000999;
0000017: C0000006;
```

e.g., If the input port receives `00000003` then `00000008`, the output port gets `00000026` because 38 decimal is `26` hex. If the input port receives `00000001` then `0000000E`, the output port gets `FFFFFFFF` because `0000000E` is not a decimal digit.

2. *Overflow possibility checker:* A program that reads two words from a given port then checks whether adding them together *might* cause overflow. If overflow is possible, an output port receives `1`. If overflow cannot happen, the output port receives `0`.

   e.g., If the input port receives `FFFFAD05` then `6A004301`, the output port gets `0` because adding these numbers will not result in overflow. If the input port receives `0000001B` then `00000002`, the output port gets `1` because adding these numbers *might* result in overflow.

3. *Simulated bank account:* A program that reads two words then updates an in-memory word representing a current bank account balance. The first word may either be UTF-32 `D` or `W`. The second word is some integer amount. `D` stands for "deposit," and adds the given amount to the balance in memory. `W` stands for "withdrawal," and subtracts the given amount from the balance in memory. Any character other than `D` or `W` results in no action; both input words are ignored. You may assume that the amount is always a positive number.

   e.g., If the in-memory balance is $100 decimal and the input port receives `00000044` then `0000000A`, the balance becomes $110 decimal because the two words requested a deposit of $10. If the input port next receives `00000057` then `0000003C`, the balance becomes $50 decimal because the two words requested a withdrawal of $60. If the two words are `00000041` then `00000100`, nothing happens because `00000041` is neither Unicode `D` nor `W`.

```
          JUMP       start    ;
balance:  1000000000          ;
w:        00000057            ;
d:        00000044            ;
action:   00000000            ;
amt:      00000000            ;
start:    IN         0000444  ;
          STORE      action   ;
          IN         0000444  ;
          STORE      amt      ;
          LOAD       w        ;
          SUB        action   ;
          JZ         withdr   ;
          LOAD       d        ;
          SUB        action   ;
          JZ         depos    ;
          JUMP       start    ; jumps to start if action is not W or D
withdr:   LOAD       balance  ;
          SUB        amt      ;
          JLZ        start    ; jumps to start if withrawal is greater than balance
          OUT        0000999  ;
          JUMP       start    ;
depos:    LOAD       balance  ;
          ADD        amt      ;
          JLZ        start    ; jumps to start if somehow negative
          OUT        0000999  ;
          JUMP       start    ;
```

```
0000000: C0000006;
0000001: 7FFFFFFF; balance
0000002: 00000057; w
0000003: 00000044; d
0000004: 00000000; action
0000005: 00000000; amt
0000006: 20000444;
0000007: 10000004;
0000008: 20000444;
0000009: 10000005;
000000A: 00000002;
000000B: 50000004;
000000C: D0000011;
000000D: 00000003;
000000E: 50000004;
000000F: D0000016;
0000010: C0000006;
0000011: 00000001; withdraw
0000012: 50000005;
0000013: E0000006;
0000014: 30000999;
0000015: C0000006;
0000016: 00000001; depos
0000017: 40000005;
0000018: E0000006;
0000019: 30000999;
000001A: C0000006;
```

4. *Two-number sorter:* A program that reads two words then echoes them back through an output port in ascending order. e.g., If the input port receives `00AA0123` then `01FF9231`, the output port gets `00AA0123` then `01FF9231`. If the input port receives `008012AB` then `000039EF`, the output port gets `000039EF` then `008012AB`.

```
                JUMP      start           ;
a:              0                         ;
b:              0                         ;
out_port:       00000999                  ;
in_port         00000444                  ;
start:          IN        in_port         ;
                STORE     a               ;
                IN        in_port         ;
                STORE     b               ;
                SUB       a               ;
                JGZ       first_less      ;
                JLZ       first_greater   ;
first_less:     LOAD      a               ;
                OUT       out_port        ;
                LOAD      b               ;
                OUT       out_port        ;
                JUMP      start           ;
first_greater:  LOAD      b               ;
                OUT       out_port        ;
                LOAD      a               ;
                OUT       out_port        ;
                JUMP      start           ;
```

```
0000000: C0000003;
0000001: 00000000;
0000002: 00000000;
0000003: 20000444;
0000004: 10000001;
0000005: 20000444;
0000006: 10000002;
0000007: 50000001;
0000008: F000000A;
0000009: E000000F;
000000A: 00000001;
000000B: 30000999;
000000C: 00000002;
000000D: 30000999;
000000E: C0000003;
000000F: 00000002;
0000010: 30000999;
0000011: 00000001;
```

```
0000012: 30000999;
0000013: C0000003;
```

5. *Single-digit expression adder/subtractor:* A program that reads a single word meant to be a UTF-8 sum or difference of single digits, then evaluates that expression. The expression may be assumed to be valid.

   e.g., If the input port receives `00342B37`—i.e., `4+7`—the output port gets `0000000B` because that is the sum of those two numbers. If the input port receives `00312D39`—i.e., `1-9`—the output port gets `FFFFFFF8` because that is the difference of those two numbers.

3 Reflection

Which task do you think is harder—*following* instructions to determine what they compute (the first section), or *writing* instructions to perform a specified computation (the second section)? Why do you think so?

I found the first section more difficult because you are unsure of the function of the program until you've nearly completed executing it. It's like following instructions without knowing their purpose, which is significantly less intuitive than writing the instructions yourself.