Name (print): __________________________________________

- **INSTRUCTIONS:**
  - Show your work to receive partial credit.
  - Keep your eyes on your own paper and do your best to prevent anyone else from seeing your work.
  - Do NOT communicate with anyone other than the professor/proctor for ANY reason in ANY language in ANY manner.
  - This exam is closed notes, closed books.
  - You may use your own calculator, but you may not share calculators.
  - Turn all mobile devices off and put them away now. You cannot have them on your desk.
  - Write neatly and clearly indicate your answers. What I cannot read, I will assume to be incorrect.
  - Stop writing when told to do so at the end of the exam. I will take 5 points off your exam if I have to tell you multiple times.
  - Academic misconduct will not be tolerated. Suspected academic misconduct will be immediately referred to the Emory Honor Council. Penalties for misconduct will be a zero on this exam, an F grade in the course, and/or other disciplinary action that may be applied by the Emory Honor Council.

- **TIME:** This exam has 6 questions on 12 pages including the title page. Please check to make sure all pages are included. You will have 75 minutes to complete this exam.

---

*I commit to uphold the ideals of honor and integrity by refusing to betray the trust bestowed upon me as a member of the Emory community. I have also read and understand the requirements and policies outlined above.*

Signature: ______________________________________________________________________________________

<table>
<thead>
<tr>
<th>Question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points:</td>
<td>10</td>
<td>6</td>
<td>5</td>
<td>20</td>
<td>13</td>
<td>21</td>
<td>75</td>
</tr>
<tr>
<td>Score:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1
1. Base Conversions: Convert the following numbers. You should use signed two's complement representation.

   (a) (2 points) \(-101_{10}\) to 8 bit two's complement binary (base-2)

   Solution:
   \[+64: \text{01100101}\]
   \[\text{fab: 10011010}\]
   \[+1: \text{1}\]
   \[\text{ans: 10011011}\]

   (b) (2 points) \(100_{10}\) to octal (base 8)

   Solution:
   \[100b: \text{001 100 100}\]
   \[\text{oct: 1 4 4}\]

   (c) (2 points) \(A2D_{16}\) to binary (base 2)

   Solution:
   \[\text{Hex: A 2 D}\]
   \[\text{Dec: 10 2 13}\]
   \[\text{Bin: 1010 0010 1101}\]

   (d) (2 points) 8-bit two's complement \(11001111_2\) to decimal (base 10)

   Solution:
   \[\text{fab: 00110000}\]
   \[+1: \text{1}\]
   \[\text{00110001 == 1+16+32 == 49}\]
   \[\text{original number began with 1 => negative, so final answer is -49}\]
   \[\text{Most common error: leaving off negative sign on final answer}\]

   (e) (2 points) \(13.875_{10}\) to IEEE Single Precision Floating Point format

   Solution:
   \[
   \begin{array}{ccc}
   \text{0} & \text{10000010} & \text{1011110...0} \\
   \text{sign} & \text{exponent} & \text{mantissa} \\
   \text{bit} & \text{8 bits} & \text{23 bits} \\
   \end{array}
   \]
   \[\text{Most common mistakes: exponent not in excess 127 code; mantissa not normalized with implied 1.}\]
2. Short Answer: answer each question briefly (1-2 sentences). You do not need to give a formal definition, just an explanation.

(a) (2 points) Assembly languages can hold certain advantages over machine or high level languages. List one advantage of assembly over machine language and one advantage of assembly over a high-level language.

**Solution:** over machine: assembly uses mnemonics making it easier to read or understand
over HLL: can be more efficient

(b) (2 points) You have 4 bytes of memory representing an integer as shown below. What is the value of this integer on a Big-Endian machine? On a Little Endian machine?

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Data (in binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00000001</td>
</tr>
<tr>
<td>02</td>
<td>00000000</td>
</tr>
<tr>
<td>03</td>
<td>00000000</td>
</tr>
<tr>
<td>04</td>
<td>00000010</td>
</tr>
</tbody>
</table>

**Solution:** BigEndian: (most significant byte at lowest address): 16777218 (or $2^{24} + 2^1$)
Little Endian (most significant byte at highest address): 33554433 (or $2^{25} + 2^0$)

(c) (2 points) Why don’t we use a signed magnitude representation of integers in computers? (In other words, what is one advantage of 2’s complement over sign magnitude)?

**Solution:** Signed magnitude has 2 representations of 0, positive 0 (eg 00000000) and negative 0 (eg 10000000). With 2’s complement notation, there is exactly 1 representation of 0, 00000000.

Two’s complement takes care of both addition and subtraction circuitry in one implementation. (No need for both add and sub. circuitry).
3. Consider the following class declaration in Java:

```java
public class Foo {
    Foo next;
    int i;
    short j;
    byte k;
}
```

Assuming that \( a \) is the starting address for a \( \text{Foo} \) object stored in memory...

(a) (1 point) what is the address of the class variable \texttt{next} for that object?

\( \text{(a) } a \)

(b) (1 point) what is the address of the class variable \texttt{i} for that object?

\( \text{(b) } a+4 \)

(c) (1 point) what is the address of the class variable \texttt{j} for that object?

\( \text{(c) } a+8 \)

(d) (1 point) what is the address of the class variable \texttt{k} for that object?

\( \text{(d) } a+10 \)

(e) (1 point) How many bytes of memory does a \( \text{Foo} \) object require?

\( \text{(e) } 11 \)

**Solution:** Most common errors:

- Not knowing size of address for variable \texttt{next}.
  See the Course Note on **Operands** for examples.

- not understanding that \texttt{next} is an address rather than another 11 byte object

- placing ints/shorts/addresses at odd memory addresses
  (see **Structure of Computer Main Memory** Course Note for discussion of variables in memory.)
4. (20 pts total) Consider the assembly code on the next page. It corresponds to some Java code (with some blanks for you to fill in). Reconstruct the Java code from the assembly code (part a). While working on this, you should also complete the table showing register usage (part b). (In fact, you may want to do this before filling in the Java code blanks.) This will help you understand the code.

(a) (14 points) Complete the following Java code:

```
int m = _____________;

int r = _____________;

int[] A = {___________________________};

int x = _____________;

while (______________) {
    if (_______________) {
        x = ___________;  
    }
}
```

**Solution:**

```
int m = 5;
int r = 0;
int[] A = {49, 21, 35, 4, 57};
int x = A[r];

while (++r < m) {
    if (A[r] <= x) {
        x = A[r];
    }
}
```

**Scoring:**

A, r, and m initializations: 1pt each
x initial value: 2pt
while condition: 3pt
if condition: 3pt
assignment stmt: 3pt
Note: So many people missed the pre-increment operator or used it incorrectly that I eliminated “incrementing r” as a scoring criteria.

(b) (5 points) Complete the register usage table. Use the Java variable names. If a register gets used to store multiple things, list them all. I have completed the first row in the table for you.

<table>
<thead>
<tr>
<th>Register</th>
<th>Java Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A</td>
</tr>
<tr>
<td>D0</td>
<td>r</td>
</tr>
<tr>
<td>D1</td>
<td>m</td>
</tr>
<tr>
<td>D2</td>
<td>A[r]</td>
</tr>
<tr>
<td>D3</td>
<td>x</td>
</tr>
<tr>
<td>D7</td>
<td>r, r+4, A[r] address, etc</td>
</tr>
</tbody>
</table>

**Solution:** Scoring: 1 pt. per register

Most common mistake: listing the values that were stored in the registers rather than just the variable names.

(c) (1 point) In plain English, what does this assembly/Java code do? (In other words, when the code completes, what value is stored in the variable \( x \)?)

**Solution:** Finds the minimum value in the array.
Listing 1: Assembly Code for Problem 4

Start:
move.l r, D0
move.l m, D1
move.l D0, D7
movea.l #A, A0
muls #4, D7
move.l 0(A0, D7.w), D2
move.l D2, x

add.l #1, D0
move.l D0, r

L1:
move.l r, D0
cmp.l D1, D0
bge Stop

move.l D0,D7
muls #4, D7
move.l 0(A0, D7.w), D2
move.l x, D3

cmp.l D3,D2
bgt L2

move.l D2, x

L2:
add.l #1, D0
move.l D0, r
bra L1

Stop:
nop

r: dc.l 0
m: dc.l 5
x: ds.l 1
A: dc.l 49, 21, 35, 4, 57

End: end
5. (13 pts total) Addressing Modes. In each part below, we want to add two pieces of integer data, one of which is in register D3. You need to get the other operand (i.e., a piece of integer data) from the location specified using the specified addressing mode. The result should be stored back into register D3.

Give the M68000 instruction (or sequence of instructions) to complete this operation. You must use the mode specified to access the data (although you can use other addressing modes in other instructions if you wish).

(a) (1 point) operand/data is equal to 6; immediate addressing

Solution: `add.l #6, D3`

(b) (2 points) operand/data in register D6; direct addressing

Solution: `add.l D6, D3`

(c) (2 points) address of operand/data in register A6; register indirect addressing

Solution: `add.l (A6), D3`

(d) (2 points) operand/data at memory location (address) 1000; register indirect addressing

Solution: `movea.l #1000, A0
add.l (A0), D3`
(e) (2 points) operand/data at memory location (address) 0x5A40; direct addressing

Solution: \texttt{add.l \$5A40, D3}

(f) (2 points) address of operand/data in memory location (address) 1000; register indirect addressing

Solution: \texttt{movea.l \#1000, A0}
\texttt{movea.l (A0), A1}
\texttt{add.l (A1), D3}

or
\texttt{movea.l 1000, A0}
\texttt{add.l (A0), D3}

Note: the ADDRESS of the data was contained at address 1000. Thus, you needed to get the data (address) at 1000 and then get the data (integer) from that address. Most common mistake was something like \texttt{add.l 1000, D3} which adds an address to the data/int currently in D3.

(g) (2 points) operand/data offset 12 bytes from base address in register A6; register indirect with displacement addressing

Solution: \texttt{add.l 12(A6), D3}
You are given the following Java variables. You do not have to initialize them, just use them. Translate the following statements into assembly code:

```java
int y;
short j;
int A[10];
```

**Solution:** Note: Most common mistake on entire problem: illegal addressing modes for various instructions. For example, trying to use an instruction like `cmp.l #6, y`. This is illegal. Looking at the addressing modes handout, we can see the 2nd operand MUST be a data register, as indicated by Dn. Likewise, with add instructions, one of the operands MUST be a data register. It is illegal to do something like `add.l j, y`.

(a) (2 points) `y++;`

```
Solution:
move.l y, D0
add.l #1, D0
move.l D0, y
```

(b) (2 points) `j++;`

```
Solution:
move.w j, D0
ext.l D0  *not necessary if you use an add.w inst. instead of add.l
add.l #1, D0
move.w D0, j
```

(c) (3 points) `j = y % j;`

```
Solution:
move.l y, D0
move.w j, D1
divs D1, D0  *common error: reversing operands
    *1st is short/16 bits, 2nd is int/32 bits
    *2nd operand is where result is always stored
swap D0
move.w D0, j
```

(d) (4 points) `A[y] = j*y;`
**Solution:**
move.l y, D0
move.w j, D1
ext.l D1  *not necessary because muls instr.*
          *only uses 16 bits of registers
muls D0, D1
movea.l #A, A0  *address calculation
muls #4, D0
move.l D1, 0(A0, D0.w)

(e) (4 points)  while (j < 4) {
    y++;
}

**Solution:**
Loop:  move.w j, D0
       ext.l D0  *could omit if you used cmp.w instead of cmp.l
       cmp.l #4, D0
       bge End
       move.l y, D0
       add.l #1, D0
       move.l D0, y
       bra Loop
End:   nop

Note: yes, this was an infinite loop. However, this makes no difference if you’re translating the code to assembly. We can make logical errors in assembly just as easily as in a HLL. However, if you incremented j, no penalty was applied.

Common mistakes: not storing value back to y and leaving intermediate results in data registers, not paying attention to operand sizes, incorrect branching condition (bgt for example).

(f) (6 points)  if (y < 6) {
    y -= 2;
}  
else {
    y += 4;
}
j = 1;

**Solution:**
move.l y, D0
cmp.l #6, D0
bge Else
If: sub.l #2, D0
move.l D0, y
bra After
Else: add.l #4, D0
move.l D0, y
After: move.l #1, j

Common mistakes: incorrect branching/labeling (e.g., no branch instruction at end of “If” section which allowed you to bypass the “Else” section; omitting the j = 1; instruction which occurs regardless if If/Else branching; not storing result back to y and leaving intermediate results in register.)