Two’s Complement Representation:

Problem 1.

1. What is the 6-bit two’s complement representation of the decimal number -21?

   \[ -21 = 16 + 4 + 1 = 0b1010101, \text{(using 6 bit binary)} \]
   \[ -21 = 0b1010101 + 1 = 0b101011 \]

2. What is the hexadecimal representation for decimal -51 encoded as an 8-bit two’s complement number?

   \[ -51 = 32 + 16 + 2 + 1 = 0b0011_0011, \text{(using 8-bit binary)} \]
   \[ -51 = 0b1100_1101 = 0xCD \]

3. The hexadecimal representation for an 8-bit two’s complement number is 0xD6. What is its decimal representation?

   \[ 0xD6 = 0b1101_0110 = -128 + 64 + 16 + 4 + 2 = -42 \]

   Alternative (may be easier):
   \[ 0xD6 = 0b1101_0110 \text{ which is negative, so use } -0xD6 = 0b0010_1010 = 42, \text{ which gives } +0xD6 = -42 \]

4. Using a 5-bit two’s complement representation, what is the range of integers that can be represented with a single 5-bit quantity?

   \[-2^4 \text{ to } (2^4) - 1 \]
   \[-16 \text{ to } 15 \]

5. Can the value of the sum of two 2’s complement numbers 0xB3 + 0x47 be represented using an 8-bit 2’s complement representation? If so, what is the sum in hex? If not, write NO.

   Yes: negative + positive is always within range.
   \[ 0xB3 + 0x47 = \]
   \[ 0b1011_0011 + \]
   \[ 0b0100_0111 = \]
   \[ 0b1111_1010 = 0xFA \text{ (in decimal: -6, can you see why it’s a very small negative?)} \]
6. Can the value of the sum of two 2’s complement numbers 0xB3 + 0xB1 be represented using an 8-bit 2’s complement representation? If so, what is the sum in hex? If not, write NO.

No: negative + negative gave us positive, not okay.
0xB3 + 0xB1 =
0b1011_0011 +
0b1011_0001 =
0b0110_0100 :(( (“9th bit” dropped: we’re in 8-bit notation)

7. Please compute the value of the expression 0xBB – 8 using 8-bit two’s complement arithmetic and give the result in decimal (base 10).

0xBB - 8 = 0b1011_1011 + -0b0000_1000 =
0b1011_1011 +
0b1111_1000 =
0b1011_0011 =
-77 (negative, so positive is 0b0100_1101 = 77) =

This is okay: negative - positive is always okay. (Same as positive-negative)

8. Consider the following subtraction problem where the operands are 5-bit two’s complement numbers. Compute the result and give the answer as a decimal (base 10) number.

\[
\begin{array}{c}
10101 \\
- 00011 \\
\end{array}
\]

\[
\begin{array}{c}
0b1_0101 \\
- 0b0_0011 \Rightarrow + 0b1_1100 + 1 \Rightarrow + 0b1_1101 \\
= 0b1_0010 = -(0b0_1101 + 1) = -14
\end{array}
\]
Problem 2.

1. Given an unsigned N-bit binary integer $= b_{n-1} ... b_1 b_0$, prove that $v$ is a multiple of 4 if and only if $b_0 = 0$ and $b_1 = 0$.

   - Powers of 2 greater than or equal to 4 are multiples of 4 (for all $n\geq 2$, $2^n = 4 \times 2^n$ and $2^n$ is an integer)
   - The sum of numbers divisible by 4 is divisible by 4.
     Therefore any number of the form $b_{n-1} ... 00$ is a multiple of 4

     If a number ends in one of 01, 10, or 11 we are adding 1, 2, or 3 respectively to a multiple of 4. Therefore it a number is a multiple 4 only if it ends in 00.

1. Does the same relation hold for two’s complement encoding?

   Yes. The above proof is unmodified: the highest order bit is now $-2^n$ instead of $+2^n$. 

### Assembly Language:

#### MIT 6.004 ISA Reference Card: Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Description</th>
<th>Execution (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td>jal rd, buConstant</td>
<td>Load Upper Immediate</td>
<td>reg[rd] &lt;= buConstant &lt;&lt; 12</td>
</tr>
<tr>
<td>JAL</td>
<td>jal rd, label</td>
<td>Jump and Link</td>
<td>reg[rd] &lt;= pc + 4</td>
</tr>
<tr>
<td>JALR</td>
<td>jair rd, label</td>
<td>Jump and Link Register</td>
<td>reg[rd] &lt;= pc + 4</td>
</tr>
<tr>
<td>BEQ</td>
<td>b eq rs1, rs2, label</td>
<td>Branch if = reg[rs1] == reg[rs2]</td>
<td>: label : pc + 4</td>
</tr>
<tr>
<td>BNE</td>
<td>b neq rs1, rs2, label</td>
<td>Branch if != reg[rs1] != reg[rs2]</td>
<td>: label : pc + 4</td>
</tr>
<tr>
<td>BLT</td>
<td>b lt rs1, rs2, label</td>
<td>Branch if &lt; (Signed)</td>
<td>pc &lt;= (reg[rs1] &lt; reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BGE</td>
<td>b ge rs1, rs2, label</td>
<td>Branch if &gt;= (Signed)</td>
<td>pc &lt;= (reg[rs1] &gt;= reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BLTU</td>
<td>bltu rs1, rs2, label</td>
<td>Branch if &lt; (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &lt; reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BGEU</td>
<td>bgeu rs1, rs2, label</td>
<td>Branch if &gt;= (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &gt;= reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>LW</td>
<td>lw rd, offset(rs1)</td>
<td>Load Word</td>
<td>reg[rd] &lt;= mem[reg[rs1] + offset]</td>
</tr>
<tr>
<td>SW</td>
<td>sw rs2, offset(rs1)</td>
<td>Store Word</td>
<td>mem[reg[rs1] + offset] &lt;= reg[rs2]</td>
</tr>
<tr>
<td>ADD</td>
<td>addi rd, rs1, constant</td>
<td>Add Immediate</td>
<td>reg[rd] &lt;= reg[rs1] + constant</td>
</tr>
<tr>
<td>SUB</td>
<td>subi rd, rs1, constant</td>
<td>Subtract Immediate</td>
<td>reg[rd] &lt;= reg[rs1] - constant</td>
</tr>
<tr>
<td>SLT</td>
<td>sltiu rd, rs1, constant</td>
<td>Compare Immediate (Unsigned)</td>
<td>reg[rd] &lt;= reg[rs1] &lt; constant</td>
</tr>
<tr>
<td>SLTU</td>
<td>slti rd, rs1, constant</td>
<td>Compare Immediate (Signed)</td>
<td>reg[rd] &lt;= reg[rs1] &lt; constant</td>
</tr>
<tr>
<td>XOR</td>
<td>xor rd, rs1, rs2</td>
<td>Xor</td>
<td>reg[rd] &lt;= reg[rs1] ^ reg[rs2]</td>
</tr>
<tr>
<td>ORI</td>
<td>ori rd, rs1, rs2</td>
<td>Or Immediate</td>
<td>reg[rd] &lt;= reg[rs1]</td>
</tr>
<tr>
<td>ANDI</td>
<td>andi rd, rs1, constant</td>
<td>And Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &amp; constant</td>
</tr>
<tr>
<td>SLL</td>
<td>sll rd, rs1, constant</td>
<td>Shift Left Logical Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &lt;&lt; constant</td>
</tr>
<tr>
<td>SRL</td>
<td>srl rd, rs1, constant</td>
<td>Shift Right Logical Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &gt;&gt; constant</td>
</tr>
<tr>
<td>ADD</td>
<td>addi rd, rs1, rs2</td>
<td>Add</td>
<td>reg[rd] &lt;= reg[rs1] + reg[rs2]</td>
</tr>
<tr>
<td>SUB</td>
<td>subi rd, rs1, rs2</td>
<td>Subtract</td>
<td>reg[rd] &lt;= reg[rs1] - reg[rs2]</td>
</tr>
<tr>
<td>SLL</td>
<td>sll rd, rs1, rs2</td>
<td>Shift Left Logical</td>
<td>reg[rd] &lt;= reg[rs1] &lt;&lt; reg[rs2]</td>
</tr>
<tr>
<td>SRL</td>
<td>srl rd, rs1, rs2</td>
<td>Shift Right Logical</td>
<td>reg[rd] &lt;= reg[rs1] &gt;&gt; reg[rs2]</td>
</tr>
<tr>
<td>XOR</td>
<td>xor rd, rs1, rs2</td>
<td>Xor</td>
<td>reg[rd] &lt;= reg[rs1] ^ reg[rs2]</td>
</tr>
<tr>
<td>ORI</td>
<td>ori rd, rs1, rs2</td>
<td>Or Immediate</td>
<td>reg[rd] &lt;= reg[rs1] ^ constant</td>
</tr>
<tr>
<td>ANDI</td>
<td>andi rd, rs1, constant</td>
<td>And Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &amp; constant</td>
</tr>
</tbody>
</table>

(*) buConstant is a 20-bit value. offset and constant are signed 12-bit values that are sign-extended to 32-bit values.

#### MIT 6.004 ISA Reference Card: Pseudoinstructions

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>li rd, constant</td>
<td>Load Immediate</td>
<td>reg[rd] &lt;= constant</td>
</tr>
<tr>
<td>lw rd, rs1</td>
<td>Move</td>
<td>reg[rd] &lt;= reg[rs1] + 0</td>
</tr>
<tr>
<td>not rd, rs1</td>
<td>Logical Not</td>
<td>reg[rd] &lt;= reg[rs1]</td>
</tr>
<tr>
<td>neg rd, rs1</td>
<td>Arithmetic Negation</td>
<td>reg[rd] &lt;= 0 - reg[rs1]</td>
</tr>
<tr>
<td>jal label</td>
<td>Jump</td>
<td>pc &lt;= label</td>
</tr>
<tr>
<td>jal label</td>
<td>Jump and Link (with ra)</td>
<td>reg[ra] &lt;= pc + 4</td>
</tr>
<tr>
<td>jr ra</td>
<td>Jump Register</td>
<td>pc &lt;= reg[ra] &amp; 1</td>
</tr>
<tr>
<td>jair ra</td>
<td>Jump and Link Register (with ra)</td>
<td>reg[ra] &lt;= pc + 4</td>
</tr>
<tr>
<td>ret</td>
<td>Return from Subroutine</td>
<td>pc &lt;= reg[ra] &amp; 1</td>
</tr>
<tr>
<td>bgt rs1, rs2, label</td>
<td>Branch &gt; (Signed)</td>
<td>pc &lt;= (reg[rs1] &gt; reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>ble rs1, rs2, label</td>
<td>Branch &lt;= (Signed)</td>
<td>pc &lt;= (reg[rs1] &lt;= reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>bgtu rs1, rs2, label</td>
<td>Branch &gt; (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &gt; reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>bleu rs1, rs2, label</td>
<td>Branch &lt;= (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &lt;= reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>beq rs1, label</td>
<td>Branch == 0</td>
<td>pc &lt;= (reg[rs1] == 0) ? label : pc + 4</td>
</tr>
<tr>
<td>bnez rs1, label</td>
<td>Branch != 0</td>
<td>pc &lt;= (reg[rs1] != 0) ? label : pc + 4</td>
</tr>
<tr>
<td>bgt rs1, label</td>
<td>Branch &gt; (Signed)</td>
<td>pc &lt;= (reg[rs1] &gt; 0) ? label : pc + 4</td>
</tr>
<tr>
<td>ble rs1, label</td>
<td>Branch &lt;= 0</td>
<td>pc &lt;= (reg[rs1] &lt;= 0) ? label : pc + 4</td>
</tr>
<tr>
<td>bgtz rs1, label</td>
<td>Branch &gt; 0 (Signed)</td>
<td>pc &lt;= (reg[rs1] &gt; 0) ? label : pc + 4</td>
</tr>
<tr>
<td>blez rs1, label</td>
<td>Branch &lt;= 0 (Signed)</td>
<td>pc &lt;= (reg[rs1] &lt;= 0) ? label : pc + 4</td>
</tr>
</tbody>
</table>
Problem 1.

Compile the following expression assuming that a is stored at address 0x1100, and b is stored at 0x1200, and c is stored at 0x2000. Assume a, b, and c are arrays whose elements are stored in consecutive memory locations.

\[ \text{for } (i = 0; i < 10; i = i+1) \ c[i] = a[i] + b[i]; \]

\[
\begin{align*}
&\text{lui } x1, 0x1100 \quad // x1 = \text{address of } a[0] \quad \text{(lui } x1, 1; \text{ addi } x1, x1, 0x100) \\
&\text{lui } x2, 0x2000 \quad // x2 = \text{address of } c[0] \quad \text{lui } x2, 2 \\
&\text{li } x3, 0 \quad // x3 = 0 (i) \quad \text{addi } x3, x0, 0 \\
&\text{li } x9, 10 \\
&\text{loop:} \\
&\quad \text{sll } x4, x3, 2 \quad // x4 = 4 \times i \\
&\quad \text{add } x5, x1, x4 \quad // x5 = \text{address of } a[i] \\
&\quad \text{add } x6, x2, x4 \quad // x6 = \text{address of } c[i] \\
&\quad \text{lw } x7, 0(x5) \quad // x7 = a[i] \\
&\quad \text{lw } x8, 0x100(x5) \quad // x8 = b[i] \\
&\quad \text{add } x7, x7, x8 \quad // x7 = a[i] + b[i] \\
&\quad \text{sw } x7, 0(x6) \quad // c[i] = a[i] + b[i] \\
&\quad \text{addi } x3, x3, 1 \quad // i = i + 1 \\
&\quad \text{blt } x3, x9, \text{loop} \quad // \text{branch back to loop if } i < 10
\end{align*}
\]
Problem 2.

A) Assume that the registers are initialized to: x1=8, x2=10, x3=12, x4=0x1234, x5=24 before execution of each of the following assembly instructions. For each instruction, provide the value of the specified register or memory location. **If your answers are in hexadecimal, make sure to prepend them with the prefix 0x.**

1. **SLL x6, x4, x5**
   
   Value of x6: __0x34000000__________

2. **ADD x7, x3, x2**
   
   Value of x7: __22__________

3. **ADDI x8, x1, 2**
   
   Value of x8: __10__________

4. **SW x2, 4(x4)**
   
   Value stored: __10________ at address: ___0x1238_______

B) Assume X is at address 0x1CE8

```
li x1, 0x1CE8
lw x4, 0(x1)
blt x4, x0, L1
addi x2, x0, 17
beq x0, x0, L2
L1: srai x2, x4, 4
L2:
X: .word 0x87654321
```

Value left in x4? __0x87654321______________

Value left in x2? __0xF8765432______________
Problem 3.

Compile the following Fibonacci implementation to RISCV assembly.

# Reference Fibonacci implementation in Python
def fibonacci_iterative(n):
    if n == 0:
        return 0
    n -= 1
    x, y = 0, 1
    while n > 0:
        # Parallel assignment of x and y
        # The new values for x and y are computed at the same time, and then
        # the values of x and y are updated afterwards
        x, y = y, x + y
        n -= 1
    return y

// x1 = n
// x2 = final result
bne x1, x0, start
li x2, 0
j end   // (pseudo instruction for jal x0, end)
start:
addi x1, x1, -1 // n = n - 1
li x3, 0   // x = 0
li x2, 1   // y = 1 (you're returning y at the end, so use x2 to hold y)
loop:
bge x0, x1, end // stop loop if 0 >= n
addi x5, x3, x2 // tmp = x + y
mv x3, x2     // x = y   (pseudo instruction for addi x3, x2, 0)
mv x2, x5     // y = tmp  (pseudo instruction for addi x2, x5, 0)
addi x1, x1, -1 // n = n - 1
j loop        // pseudo instruction for
end: