Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below. Show your work for partial credit. You can use the extra white space and the backs of the pages for scratch work.

Problem 1. Binary Arithmetic (10 points)

(A) (4 points) What is 0x79 << ~(0xFC) where ~ is bitwise NOT and << is bitwise LEFT SHIFT on 8-bit operands? Provide your result in both 8-bit binary and in hexadecimal.

Result in binary (0b): \[1100 \hspace{0.5cm} 1000\]

Result in hexadecimal (0x): C8

(B) (4 points) What is 0b1011 multiplied by 0b0101? You can treat both numbers as unsigned, and the result will be more than four bits. Show how you computed this with binary multiplication. What is this number in decimal?

0b1011 multiplied by 0b0101 (show your work) (0b): \[110111\] (in decimal): 55

(C) (2 points) What is 0b10110101 in decimal assuming unsigned encoding (only positive integers)? What is it in decimal assuming two’s complement encoding?

0b10110101 assuming unsigned encoding (in decimal): 181

0b10110101 assuming two’s complement encoding (in decimal): -75
Problem 2. Boolean Logic (12 points)

(A) (6 points) Simplify the following Boolean expressions by finding a minimal sum-of-products (minimal SOP) expression for each one. (Note: These expressions can be reduced into a minimal SOP by repeatedly applying the Boolean algebra properties we saw in lecture.)

1. \((r + a) \cdot y \cdot ((r + \overline{a}) \cdot \overline{y})\)
   \[= \overline{r} \cdot \overline{a} \cdot y \cdot ((r + \overline{a}) + y)\]
   \[= \overline{r} \cdot \overline{a} \cdot y\]

2. \((x + y) \cdot z + \overline{z} \cdot x + y \cdot (\overline{x} + z)\)
   \[= x \cdot z + y \cdot z + \overline{z} \cdot x + \overline{x} \cdot y\]
   \[= x + y \cdot z + \overline{x} \cdot y\]
   \[= x + y\]

(B) (6 points) Consider the truth table on the right, which defines function Y out of three input variables (A, B, and C).

1. Find a minimal sum-of-products expression for Y.

   Minimal sum-of-products for \(Y(A,B,C) = \) ______________ 
   \[a \cdot \overline{b} + \overline{b} \cdot c\] ______________

2. Using only combinational circuits built from gates specified by the truth table above, which of the following statements are true?

   (A) We can implement any Boolean function using only these gates.
   (B) We can’t implement every Boolean function using only these gates.
   (C) We can only implement functions with the same truth table as above.

   Circle all true answers: A B C
Problem 4. Combinational Logic (28 points)

(A) (5 points) The following BSV function \( f \) performs a basic operation using \( a \) and \( b \). We want \( f_2 \) to implement the same function as \( f \). Fill in the blank in \( f_2 \) to make the two functions equivalent. Write a single-line expression that uses the ternary operator \( (?:) \).

\[
\text{function Bit#(n) } f(\text{Bit#(n) } a, \text{Bit#(1) } b);
\text{Bit#(n) } x = 0;
\text{for (Integer } i = 0 ; i < \text{valueOf}(n) ; i = i+1) \text{ begin}
\text{x}[i] = a[i] ^ b;
\text{end}
\text{return x;}
\text{endfunction}
\]

\[
\text{function Bit#(n) } f_2(\text{Bit#(n) } a, \text{Bit#(1) } b);
\text{return (b==1) \ ? \ \neg a : a ;}
\text{endfunction}
\]

(B) (5 points) Write the truth table for the combinational device described by the function below.

\[
\text{function Bit#(2) } f(\text{Bit#(1) } a, \text{Bit#(1) } b, \text{Bit#(1) } c);
\text{Bit#(2) ret = zeroExtend(a) + signExtend(b);
\text{case } \{a,b\}\)
\text{0: ret = \{1, c\};
2: ret = \{a ^ b, a & b\};
3: ret = \neg signExtend(c);
\text{endcase}
\text{return ret;}
\text{endfunction}
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>ret[1]</th>
<th>ret[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
(C) (5 points) The following BSV function \( g \) performs a specific arithmetic operation on \( n \)-bit operands \( a \) and \( b \). We want the function \( g_2 \) to implement \( g \) in a single line of code. Fill in the blank with a single expression to make \( g_2 \) equivalent to \( g \).

```vhdl
function Bit#(1) g(Bit#(n) a, Bit#(n) b);
    Bit#(2) ret = 'b10;
    for (Integer i = valueOf(n)-1 ; i >= 0 ; i = i - 1)
        begin
            if ({a[i], b[i]} == 'b01) ret = {0, ret[1] | ret[0]};
            else if ({a[i], b[i]} == 'b10) ret = {0, ret[0]};
        end
    return ret[1] | ret[0];
endfunction

function Bit#(1) g2(Bit#(n) a, Bit#(n) b);
    return __________;
endfunction
```

(D) (5 points) Finish the following circuit diagram to implement function `computeB`, given below. You may only use 32-bit 2-to-1 multiplexers, constants (0, 1, 2, 3, ...) and logic gates (AND, NOT, OR, XOR). We have provided three 32-bit greater-than-or-equal (\( \geq \)) comparators for you.

```vhdl
function Bit#(32) computeB(Bit#(32) in);
    Bit#(32) out = 0;
    if ( in >= 1 ) out = 1;
    if ( in >= 5 ) out = 5;
    if ( in >= 10 ) out = 10;
    return out;
endfunction
```
(E) (8 points) Show that one-bit comparators can be used to implement any combinational circuit by implementing an inverter, an AND gate, and an OR gate using only comparator gates. You may tie inputs to 1 or 0 if necessary, and may use one or multiple comparator gates. Clearly label all inputs and outputs.

\[
\text{EQ} = \overline{A} \cdot B \quad (A \text{ equals } B)
\]

\[
\text{LT} = \overline{A} \cdot B \quad (A \text{ less than } B)
\]

Logic diagram of inverter implementation using one-bit comparators:

Logic diagram of AND gate implementation using one-bit comparators:

Logic diagram of OR gate implementation using one-bit comparators:

(other answers are possible)
Problem 1. Assembly (18 points)

(A) (6 points) For the RISC-V instruction sequence below, provide the hex values of the specified registers after the sequence has been executed. Assume the program is located beginning at address 0x0 in memory and that all registers are initialized to 0.

```
= 0x0
lui x4, 0x2
jal x5, L1
ori x4, x4, 0x3
L1: addi x0, x4, 0x16
    addi x6, x0, 0x200
```

Value of x4 (in hex) 0x200
Value of x5 (in hex) 0x8
Value of x6 (in hex) 0x200

(B) (4 points) You are given the following C code along with a partial translation into RISC-V assembly that is missing a single instruction (the first one). Please complete the assembly implementation by specifying the single missing instruction. Assume that x and y are signed 32-bit integers (defined as int x; int y;)

```
if (x > y) {
    x = 5;
} else {
    y = 10;
}
```

```
// Partial assembly implementation
// Registers - x: x1, y: x2

bge x2, x1, else

li x1, 5
j endif
else:
    li x2, 10
endif:
```

(C) (8 points) Consider the following code sequence. This code is located beginning at address 0x0 in memory. Note that this code may modify itself.
. = 0x0
  bge x1, x0, L1
  lw x2, 0xC(x0)
  sw x2, 0x10(x0)
L1:  addi x1, x1, 2
    addi x1, x1, -1

(i) What is the address of label L1?
0xC

(ii) What are the final values of the register x1, when this code is run with the following initial values for x1?

If initial value of x1 = 5  →  Final value of x1 = _____6_____

  bge taken, final x1 = 5 (initial value) + 2 (addi) – 1 (2nd addi)

If initial value of x1 = -5  →  Final value of x1 = ____-1_____

  bge not taken, lw and sw overwrite 2nd addi at 0x10 with addi at 0xC
  final x1 = -5 (initial value) + 2 (addi) + 2 (addi written by sw)
Problem 2. RISC-V Assembly and Calling Convention (16 points)

(A) (6 points) The **factorial** C procedure given below implements the factorial function (e.g., factorial(5) = 5! = 5 x 4 x 3 x 2 x 1 = 120). This implementation uses a loop to compute the factorial. Since our RISC-V processor does not have a multiply instruction, the code uses a multiply procedure, `mult`, which multiplies two signed integers.

Ben Bitdiddle decides to implement this procedure in RISC-V assembly. But his assembly code, shown below, is not behaving as expected. Please fix the bugs so that his code works correctly and obeys the RISC-V calling convention.

The implementation of the `mult` procedure is not provided to you, but it does obey the calling convention.

```c
int factorial(int n) {
    int prod = 1;
    for (int i = 1; i <= n; i++) {
        prod = mult(prod, i);
    }
    return prod;
}
```

Provide the correct assembly code here. Your implementation should obey the RISC-V calling convention. **Note that there may be multiple errors in the code. Assume** \( n \geq 1 \).

```assembly
factorial:
    addi sp, sp, -16
    sw s0, 0(sp)
    sw s1, 4(sp)
    sw s2, 8(sp)
    sw ra, 12(sp)
    mv s0, a0
    li s1, 1 // set up prod
    li s2, 1 // set up i
loop:
    beq s0, s2, done
    mv a0, s1
    mv a1, s2
    jal ra, mult
    addi s2, s2, 1
    j loop
done:
    mv a0, s1
    jr ra // ret

mult: ...
```
(B) (10 points) The `sumSquares` C procedure given below computes the sum-of-squares function. Specifically, the `sumSquares` procedure takes a single argument `n` and computes $\sum_{i=1}^{n} i^2$ (e.g., `sumSquares(5) = 5^2 + 4^2 + 3^2 + 2^2 + 1^2 = 55`). This implementation is recursive. Again, since our RISC-V processor does not have a multiply instruction, the code uses a multiply procedure, `mult`, which multiplies two signed integers.

Ben Bitdiddle decides to implement this procedure in RISC-V assembly. But his assembly code, shown below, is not behaving as expected. Please answer the question below and fix the bugs so that his code works correctly and obeys the RISC-V calling convention.

The implementation of the `mult` procedure is not provided to you, but it does obey the calling convention.

```c
int sumSquares(int n) {
    if (n == 0)
        return 0;
    return sumSquares(n-1) + mult(n,n);
}
```

Ben’s buggy assembly code is given below. Provide the full instruction that the PC points to after executing `jr ra` (i.e., `ret`) for the first time in the given buggy code. Assume $n \geq 1$.

```
mv s0, a0
```

Provide the correct assembly code here. Your implementation should obey the RISC-V calling convention. Note that there may be multiple errors in the code.

```
sumSquares:
    beqz a0, done
    addi sp, sp, -12
    sw a0, 0(sp)
    sw s0, 4(sp)
    sw ra, 8(sp)
    addi a0, a0, -1
    jal ra, sumSquares
    mv s0, a0
    lw a0, 0(sp)
    addi sp, sp, 4

mult:
    ...
```

```
sumSquares:
    beqz a0, done
    addi sp, sp, -12
    sw a0, 0(sp)
    sw s0, 4(sp)
    sw ra, 8(sp)
    addi a0, a0, -1
    jal ra, sumSquares
    mv s0, a0
    lw a0, 0(sp)
    mv a1, a0
    jal ra, mult
    add a0, a0, s0
    lw s0, 4(sp)
    lw ra, 8(sp)
    addi sp, sp, 12

done:
    jr ra // ret
```
Problem 3. Procedures and Stacks (18 points)

Consider the C procedure below and its translation to RISC-V assembly code, shown on the right.

```c
int f(int a) {
    if (a == 1)
        return 0;
    if (a & 1 == 0) {
        // a is even
        return 1 + f(a / 2);
    } else {
        // a is odd
        return 1 + f(a * 3 + 1);
    }
}
```

(A) (3 points) Give the HEX encoding of the 'slli a1, a0, 1' instruction.

```
00000000|shamt| rs1 |001| rd |0010011
00000000|00001|01010|001|01011|0010011
0x 0 0 1 5 1 5 9 3
```

Hex encoding of slli a1, a0, 1: 0x 00151593

(B) (2 points) How much total stack memory does the program use to calculate f(3)?

```
f(3) → f(10) → f(5) → f(16) → f(8) → f(4) → f(2) → f(1)
```

1 word each 0 words

Number of words pushed onto the stack for f(3)? __7______
The program’s initial call to function $f$ occurs outside of the function definition via the instruction ‘jal ra, f’. The program is interrupted at an execution (not necessarily the first) of function $f$, just prior to the execution of li a1, 1 at label f. The diagram on the right shows the contents of a region of memory. All addresses and data values are shown in hex. The current value in the SP register is 0xFBC and points to the location shown in the diagram.

(C) (4 points) Is the argument to the initial call to $f$ even or odd? Circle your choice:

Even  Odd  Can’t tell

First recursive call has ra = 0x60, following ones have ra = 0x80, both jal instructions are 16 = 0x20 bytes away $\rightarrow$ first jal (for odd argument) is at address 0x5C, second jal (for even argument) is at 0x7C, and first recursive call takes the jal for odd (ra = 0x60).

(D) (3 points) What value was in SP just prior to the initial call to $f$?

Initial contents of SP: 0x FCC

(E) (3 points) What is the address of the ‘jal ra, f’ instruction that made the initial call to $f$?

Saved ra of initial call is 0x100 $\rightarrow$ jal ra, $f$ is at 0x100 – 4 = 0xFC

Address of instruction that made initial call to $f$: 0x FC

(F) (3 points) What is the hex address of the instruction at label EVEN?

Address of instruction at label EVEN: 0x 70

END OF QUIZ 1!