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

Problem 1. Two’s Complement Arithmetic (8 points)

(A) (4 points) Express both 37 and 11 in 8-bit two’s complement as well as hexadecimal.

37 in 8-bit two’s complement (0b): ______________

37 in hexadecimal (0x): ______________

11 in 8-bit two’s complement (0b): ______________

11 in hexadecimal (0x): ______________

(B) (2 points) Compute 37 + 11 in 8-bit two’s complement.

37 + 11 in 8-bit two’s complement (0b): ______________

(C) (2 points) Compute 37 – 11 in 8-bit two’s complement.

37 - 11 in 8-bit two’s complement (0b): ______________
Problem 2. Boolean Expressions (12 points)

(A) (8 points) Simplify the following Boolean expressions by finding a minimal sum-of-products 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. \((a \cdot b \cdot c) + \overline{a} \cdot \overline{b} \cdot \overline{c}\)

2. \(((a \cdot b) + (c \cdot d)) \cdot (e \cdot f \cdot g)\)

(B) (4 points) There are some Boolean expressions for which no assignment of values to variables can produce True (e.g., \(a \cdot \overline{a}\)). These Boolean expressions are said to be *non-satisfiable.*

Is the following Boolean expression satisfiable? If the expression is satisfiable, give an assignment to variables that makes the expression evaluate to True. If the expression is non-satisfiable, explain why.

\((\overline{x} + yz) \cdot (\overline{y}x + z) \cdot (\overline{z}y + x)\)
Problem 4. Combinational Logic in BSV (25 points)

(A) (4 points) The following BSV function f performs a specific arithmetic operation on n-bit operands a and b. We want the function f2 to implement f in a single line of code. Fill in the blank with an expression of the form “a operation b” to make f2 equivalent to f.

```bsv
function Bit#(1) f(Bit#(n) a, Bit#(n) b);
    Bit#(TAdd#(n,1)) x = 0;
    for (Integer i = 0; i < valueOf(n); i = i+1)
        begin
            x[i+1] = x[i] | (a[i] & ~b[i]) | (~a[i] & b[i]);
        end
    return x[valueOf(n)];
endfunction

function Bit#(1) f2(Bit#(n) a, Bit#(n) b);
    return (____________)? 1 : 0;
endfunction
```

(B) (4 points) Write the truth table for the combinational device described by this function.

```bsv
function Bit#(2) f(Bit#(1) a, Bit#(1) b, Bit#(1) c);
    return (case ({a, b})
        0: ((c==1) ? {a, b} : 3);
        1: {c, a};
        2: 1;
        3: (2'b01 ^ {c, c});
    endcase);
endfunction
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>out[1]</th>
<th>out[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(C) (6 points) The following BSV function \( f \) performs a specific arithmetic operation on \( n \)-bit operands \( a \) and \( b \). We want the function \( f_2 \) to implement \( f \) in a single line of code. Fill in the blank with an expression of the form “\( a \) operation \( b \)” to make \( f_2 \) equivalent to \( f \).

```bsv
def function Bit#(1) f(Bit#(n) a, Bit#(n) b);
    Bit#(TAdd#(n,1)) x = 0;
    for (Integer i = 0; i < valueOf(n); i = i+1) begin
        x[i+1] = (case ({a[i], b[i]})
            2'b00: x[i];
            2'b01: 0;
            2'b10: 1;
            2'b11: x[i];
        endcase);
    end
    return x[valueOf(n)];
endfunction

function Bit#(1) f2(Bit#(n) a, Bit#(n) b);
    return (__________)? 1 : 0;
endfunction
```

(D) (5 points) For \( n = 2 \), manually compile the BSV function \( f \) above into a combinational circuit. You can use inverters, AND, OR, XOR, NAND, and NOR gates, as well as multiplexers. Please label the inputs and outputs bit by bit (\( a[0] \), \( a[1] \), \( b[0] \), \( b[1] \), \( out[0] \), and \( out[1] \)). You do not need to optimize or simplify the circuit.
(E) (6 points) Show that half-adder devices can be used to implement any combinational circuit by implementing an inverter, an AND gate, and an OR gate using only half-adder gates. Make sure to clearly label the output. You may tie inputs to 1 or 0 if necessary, and may use multiple half-adder gates.

\[ C = A \& B; \]
\[ S = A ^ \text{B}; \]

Logic diagram of inverter implementation using half-adders:

Logic diagram of AND gate implementation using half-adders:

Logic diagram of OR gate implementation using half-adders:
Problem 1. Assembly Language (20 points)

(D) (6 points) You are given the following C code along with a partial translation of this code into RISC-V assembly. Please complete the assembly implementation of this code by fully specifying the 2 missing RISC-V instructions just before label L1. These must be actual RISC-V instructions, you may not use any pseudo-instructions.

```c
int A[1000];
for (int i = 0; i < 1000; i = i + 1) {
    A[i] = A[i] ^ 0x12345; // ^ is bitwise XOR in C
}
```

// Translation into RISC-V assembly:
```c
addi x11, x0, 0  // x11: i
addi x12, x0, 1000  // x12: loop termination condition
addi x16, x0, 0x100  // x16: array base
missing instr
missing instr
L1: bge x11, x12, L2
    slli x13, x11, 2 // byte offset into array
    add x17, x13, x16 // address of A[i]
    lw x14, 0(x17)  // x14 = A[i]
    addi x11, x11, 1
    xor x14, x14, x15
    sw x14, 0(x17)
    jal x0, L1
L2: ...
```

A: ... // contents of array A
A[0] = 0x100

For the RISC-V instruction sequences below, provide the hex values of the specified registers after each sequence has been executed. Assume that each sequence execution ends when it reaches the `unimp` instruction. Also assume that all registers are initialized to 0 before execution of each sequence begins.

(B) (6 points)
```c
li x11, 0x242
slli x12, x11, 12
li x13, 0xff
or x14, x12, x13
unimp
```

Value left in x11: 0x_____________
Value left in x12: 0x_____________
Value left in x13: 0x_____________
(C) (8 points)

```
. = 0x0
   li x11, 0x200
   lw x11, 0(x11)
   blt x11, x0, L1
   addi x12, x11, -1
   jal x13, L2

L1: sub x12, x0, x11
L2: xori x12, x12, 0xfff
    unimp

. = 0x200
X: .word 0x87654321
```

Value left in x0: 0x________________

Value left in x11: 0x________________

Value left in x12: 0x________________

Value left in x13: 0x________________
Problem 2. RISC-V Assembly and Calling Conventions (18 points)

Our RISC-V processor does not have a multiply instruction, so we have to do multiplications in software. The C code below shows a recursive implementation of multiplication by repeated addition of unsigned integers (in C, unsigned int denotes an unsigned integer). Ben Bitdiddle has written and hand-compiled this function into the assembly code given below, but the code is not behaving as expected. Find the bugs in Ben’s assembly code and write a correct version.

C code for unsigned multiplication

```c
unsigned int mul(unsigned int x, unsigned int y) {
    if (x == 0) {
        return 0;
    } else {
        unsigned int lowbit = x & 1;
        unsigned int p = lowbit? y : 0;
        return p + (mul(x >> 1, y) << 1);
    }
}
```

Buggy assembly code

```assembly
mul:
    addi sp, sp, -8
    sw s0, 0(sp)
    sw ra, 4(sp)
    beqz a0, mul_done
   andi s0, a0, 1  // lowbit in s0
    mv t0, zero  // p in t0
    beqz s0, lowbit_zero
    mv t0, a0
    slli a0, a0, 1
    jal mul
    srl a0, a0, 1
    add a0, t0, a0
    lw s0, 4(sp)
    lw ra, 0(sp)
    addi sp, sp, 8
    mul_done:
    ret
```

Rewrite assembly code so that it behaves as expected and follows the RISC-V calling convention:
Problem 3. Procedures and Stacks (20 points)

You are given an incomplete listing of a C procedure and its translation to RISC-V assembly code (shown on the right):

```c
int f(int a, int b) {
    int tempa = a + a;
    int tempb = ***;
    if (a < b)
        return a + f(tempa, tempb);
    else return ???;
}
```

(A) (3 points) Give the HEX encoding of the ‘jr ra’ instruction at the end of the program.

Hex encoding of `jr ra`: 0x __________

(B) (2 points) Give the HEX encoding of the ‘bge a0, a1, L1’ instruction at the beginning of the program.

Hex encoding of `bge a0, a1, L1`: 0x __________

(C) (2 points) What is the missing C expression corresponding to the ‘***’ in the above program?

(give C code expression)

(D) (2 points) What is the missing C expression corresponding to the ‘???’ in the above program?

(give C code expression)

(E) (2 points) Suppose the instruction ‘jal ra, f’ were replaced by the instruction ‘j f’ in the above program. What would be the result? Mark exactly one answer:

- The procedure would still work just fine: __________
- The procedure would compute the proper value, but not restore registers properly: __________
- The procedure would no longer compute f(a,b) properly: __________
The program calls \( f(2, 5) \) via the instruction ‘\( \text{jal ra, f} \)’. The program is interrupted just prior to an execution of the \( \text{bge} \) instruction at label \( f \). The diagram in the right shows the contents of a region of memory. All addresses and data values are shown in hex. **The current contents of the SP are 0x128.**

(F) (3 points) What are the arguments to the current (most recent) call to \( f \)?

Current arguments, \( a = \) _________; \( b = \) __________

(G) (2 points) What value was in SP just prior to the initial call to \( f \)?

Initial contents of SP: 0x _________

(H) (2 points) What is the address of the ‘\( \text{jal ra, f} \)’ instruction that made the original call to \( f(2, 5) \)?

Address of original call to \( f(2, 5) \): 0x _________

(I) (2 points) What is the hex address of the instruction at L2?

Address of instruction at L2: 0x _________

END OF PRACTICE QUIZ 1!