Assignment 1 (Due Date: September 20, 2025 by 11:59 PM)

(130 points)

Complete each of the exercises given and submit on BrightSpace as a Word document or pdf. Where necessary, clearly comment your code. Ensure that your code executes correctly before submission by using a simulator. When converting to RISC-V assembly, ignore the preprocessor directives and system calls.

(40 points) 1. Convert the following C code into RISC-V assembly. Clearly comment your code and mention the registers, the line of C code being converted and any other necessary detail.

a)
```
int x = 7, y = 20, z = 0;
   while (y > x) {
          y = y - 3;
        if (y & 1) {       // checking for odd number
               z = z + y;
        } else {
               z = z - x;
               x = x + 2;
        }
}
```

b)
```
int i, a = 0;
   for (i = 0; i < 12; i++) {
        if ((i & 1) == 0) continue;   // check for even "i" and skip
        a = a + i;
        if (a >= 16) a = a - 4;
        }
```

c)
```
int p = 5, q = 2;
  do {
        p = p + q;
        if (p < 12) q = q + 1;
```

```
        else      q = q - 2;
    } while (q > 0);
```

d)
```
int x = 0, y = 7, z = 0;
    while ((x < y) && (z < 10)) {
        x = x + 2;
        if (x >= 4) z = z + 3;
        else        z = z + 1;
    }
```

(10 points) 2. Each number in the Fibonacci series is the sum of the previous two numbers. Table 1 below lists the first few numbers in the series, ***fib(n)***.

Table 1 Fibonacci series

n	1	2	3	4	5	6	7	8	9	10	11	…..
fib(n)	1	1	2	3	5	8	13	21	34	55	89	…..

a) What is ***fib(n)*** for ***n = 0*** and ***n = –1***?

b) Write a function called ***fib*** in C that returns the Fibonacci number for any nonnegative value of ***n***. ***Hint:*** You probably will want to use a loop. Clearly comment your code.

c) Convert the C function of part (b) into RISC-V assembly code. Add comments after every line of code that explain clearly what it does. Use a simulator to test your code on ***fib(9)***.

(10 points) 3. Consider the following C code snippet:

```
// C code
void setArray(int num){
        int i;
        int array[10];
        for (i = 0; i < 10; i = i + 1)
                array[i] = compare(num, i);
}
int compare(int a, int b) {
        if (sub(a, b) >= 0)
                return 1;
```

```
        else
                return 0;
}

int sub(int a, int b) {
        return a - b;
}
```

a) Implement the C code snippet in RISC-V assembly language. Use ***s4*** to hold the variable ***i***. Be sure to handle the stack pointer appropriately. The array is stored on the stack of the ***setArray*** function. Clearly comment your code.

b) Assume that ***setArray*** is the first function called. Draw the status of the stack before calling ***setArray*** and during each function call. Indicate stack addresses and the names of registers and variables stored on the stack; mark the location of ***sp***; and clearly mark each stack frame. Assume that ***sp*** starts at ***0x8000***.

c) How would your code function if you failed to store ***ra*** on the stack?

(10 points) 4. Consider the following high-level function:

```
// C code
int f(int n, int k) {
        int b;
        b = k + 2;
        if (n == 0)
                b = 10;
        else
                b = b + (n * n) + f(n - 1, k + 1);
        return b * k;
}
```

a) Translate the high-level function ***f*** into RISC-V assembly language. Pay particular attention to properly saving and restoring registers across function calls and using the RISC-V preserved register conventions. Clearly comment your code. Assume that the function starts at instruction address ***0x8100***. Keep local variable ***b*** in ***s4***. Clearly comment your code.

b) Step through your function from part (a) by hand for the case of ***f(2,4)***. Draw a picture of the stack similar to the one and assume that ***sp*** is equal to ***0xBFF00100*** when ***f*** is called. Write the stack addresses and the register name and data value stored at each location in the stack and keep track of the stack pointer value ***(sp)***. Clearly mark each stack frame. You might also find it useful to keep track of the values in ***a0***, ***a1***, and ***s4*** throughout execution. Assume that when ***f*** is called, ***s4 = 0xABCD*** and ***ra = 0x8010***.

c) What is the final value of ***a0*** when ***f(2,4)*** is called?

(10 points) 5. Consider two strings: ***string1*** and ***string2***:

a) Write high-level code for a function called ***concat*** that concatenates (joins together) the two strings: ***void concat(char string1[], char string2[], char stringconcat[])***. The function does not return a value. It concatenates ***string1*** and ***string2*** and places the resulting string in stringconcat. You may assume that the character array ***stringconcat*** is large enough to accommodate the concatenated string. Clearly comment your code.

b) Convert the function from part (a) into RISC-V assembly language. Clearly comment your code.

(10 points) 6. Consider a function that sorts a 10-element integer array called ***scores*** from lowest to highest. After the function completes, ***scores[0]*** holds the smallest value and ***scores[9]*** holds the highest value.

a) Write a high-level ***sort*** function that performs the function above. ***sort*** receives a single argument, the address of the scores array. Clearly comment your code.

b) Convert the ***sort*** function into RISC-V assembly language. Clearly comment your code.

(10 points) 7. Consider the following snippet of C code:

// Modify the array in place through a pointer
int A[8] = { 5, -2, 7, 0, 9, -4, 3, -1 };

int n = 8;
*int *p = A;*

```
int sum = 0;
int count = 0;

while (n > 0) {
    int v = *p;
    if (v < 0) {
        *p = 0;              // negative entries replaced by zero
    } else {
        sum = sum + v;         // accumulate non-negatives
        if (v & 1) {          // test for odd number
            count = count + 1;  // count odd non-negatives
        }
    }
    p = p + 1;               // advance to next int
    n = n - 1;               // one fewer element left
}
```

After the above C code snippet executes, the program:

- has all negative numbers in ***A*** replaced by 0
- ***sum*** holds the sum of all original non-negative values
- ***count*** holds the number of odd non-negative values

Convert the given C code snippet to RISC-V assembly.

(10 points) 8. Consider the following snippet of C code:

```
// Each entry has a value and an "active" flag.
typedef struct {
    int value;
    int active;
} Entry;

Entry B[6] = { {5,1}, {-3,1}, {7,0}, {0,1}, {9,1}, {-4,0} };

int n = 6;
int sum = 0;
int count = 0;

for (int i = 0; i < n; i++) {
    int v = B[i].value;
```

```
    if (v < 0) {
        B[i].value = 0;            // clamp negatives to zero in place
    } else {
        sum += v;                  // accumulate non-negatives
        if (B[i].active && (v & 1)) { // count odd, active entries
            count++;
        }
    }
}
```

After the loop runs, ***sum***, ***count*** and ***B*** are modified in place.

Convert the given C code snippet to RISC-V assembly.

(10 points) 9. Consider the RISC-V machine code snippet below. The first instruction is listed at the top.

a) Convert the machine code snippet into RISC-V assembly language.

b) Reverse engineer a high-level program that would compile into the assembly language routine and write it. Clearly comment your code.

c) Explain in words what the program does. a0 and a1 are the inputs, and they initially contain positive numbers, A and B. At the end of the program, register a0 holds the output (i.e., return value).

0x01800513
0x00300593
0x00000393
0x00058E33
0x01C54863
0x00138393
0x00BE0E33
0xFF5FF06F
0x00038533

(10 points) 10. Repeat the process in 9 for the following machine code. ***a0*** and ***a1*** are the inputs. ***a0*** contains a 32-bit number and ***a1*** is the address of a 32-element array of characters (***char***).

0x01F00393
0x00755E33
0x001E7E13
0x01C580A3
0x00158593
0xFFF38393
0xFE03D6E3
0x00008067