Assembly Language Exercises

SENAI CIMATEC - QuIIN
Equipe de Pós Processamento

April 2, 2025

Exercise	**Difficulty**	**Topics**
`sum_registers`	Easy	Arithmetic operations
`counting_bits`	Easy	Bit manipulation
`binary2dec`	Easy	Bit operations, Conversion
`max_of_arr`	Easy	Load/store operations
`reversing_string`	Easy	Strings, Load/store
`simple_stack`	Easy	Stack operations
`fibonacci`	Medium	Loops, Recursion
`bubble sort`	Medium	Loops, load/store
`printlen`	Medium	load/store, strings
`Ackermann`	Medium	Loops, Recursion
`Matrix Multiplication`	Medium	Loops, Arithmetic operations
`jumptables`	Medium	jumptables

Exercise 1: Sum of Two Registers

Problem Statement: Create an assembly program that adds the values stored in two registers and stores the result in a third register.

Example: a0 = 5, a1 = 3 therefore a2 = 8

- **Input:** a0 = first operand, a1 = second operand
- **Output:** a2 = result

Exercise 2: Counting Set Bits

Problem Statement: Write an assembly function that counts the number of set bits (1s) in a given 32-bit integer.

Example: Input: `0b1101` (13 in decimal) should output `3`.

- **Input:** a0 - 32-bit integer
- **Output:** a0 - Count of bits set to 1

Exercise 3: Binary to Decimal Conversion

Problem Statement: Develop an assembly function that converts a binary number (represented as an array) into its decimal equivalent.

Example: Input: 1011_2 should output 11_{10}.

- **Input:** a0 - Pointer to a binary string (null-terminated)
- **Output:** a0 - Decimal representation

Exercise 4: Maximum of an array

Problem Statement: Write an assembly function that finds the maximum in an array of integers (32-bit).

- **Input:** a0 - Pointer to the array.
- **Input:** a1 - Length of the array.
- **Output:** a0 - The largest element in the array.

Exercise 5: Reversing a String

Problem Statement: Implement a function in assembly that reverses a given string.

- **Input:** a0 - Pointer to the beginning of the string.
- **Output:** The string should be reversed in-place (i.e., the reversed string is located at the same memory location).

Exercise 6: Simple Stack

Problem Statement: Implement a stack data structure in assembly with basic operations like Push and Pop.

- **Push**: adds an element to the stack.
- **Pop**: removes the last element and returns it.
- Assume that the stack only supports 32-bit values and the memory space will not cause an overflow.
- `sp` points to the last element in the stack.
- **Input:** a0 - 0 for Push, 1 for Pop.
- **Input:** a1 - The value for Push, if applicable.
- **Output:** a0 - The value for Pop, if applicable.

Exercise 7: Bubble sort

Problem Statement: Implement the bubble sort algorithm. It should sort an array of n bytes. You may assume that the values are signed and that the array contains at least one entry. The sorting should be done in-place, i.e. you should not create a new array.

- **Input:** a0 - Pointer to the array.
- **Input:** a1 - Length of the array.
- **Output:** Array sorted

Exercise 8: Fibonacci Sequence

Problem Statement: Implement a function in assembly that calculates the Fibonacci sequence up to the N-th term using loops.

Example: Input: `N = 6` should output `0, 1, 1, 2, 3, 5`.

- **Input:** a0 - Integer N (number of terms)
- **Output:** a0 - Fibonacci sequence up to N

Exercise 9: printlen - printing a null-terminated string

Problem Statement: Write an assembly function that prints a null-terminated string and then adds a line-break. The function should also return the length of the string. **Obs:** The string ends with a null byte (`0x00`), and an ASCII line break (`0x0A`) should occur at the end of the output.

- **Input:** a0 - Pointer to the beginning of the string
- **Output:** a0 - Length of the printes string (excluding the null terminating byte and the line break)

Exercise 10: Ackermann Function

Problem Statement: Write an assembly function that calculates the Ackermann function $A(m, n)$. Your function should print the input values m and n before performing the calculation.

Definition:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Example: Inputs $m = 2$, $n = 3$ should first output `m = 2, n = 3`, and then return $A(2, 3) = 9$.

- **Input:** `a0 = m, a1 = n`
- **Output:** `a0 = Ackermann(m, n)`

Exercise 11: Matrix Multiplication

Problem Statement: Implement an assembly function that multiplies two matrices A (size $m \times p$) and B (size $p \times n$), storing the result in matrix C (size $m \times n$). Assume matrices are stored in row-major order in memory.

Example: For matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

The resulting matrix C will be:

$$C = A \times B = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Input/Output Specifications:

- **Input:**
  - `a0` - Pointer to matrix A
  - `a1` - Pointer to matrix B
  - `a2` - Pointer to result matrix C
  - `a3` - Integer m (number of rows in A)
  - `a4` - Integer p (number of columns in A, rows in B)
  - `a5` - Integer n (number of columns in B)
- **Output:**
  - Matrix C populated with the result of $A \times B$

Exercise 12: Jumptables - Struct to String

Objective: Develop an assembly function that translates numerical values stored in a struct to human-readable textual descriptions. Consider the following struct definition:

```
struct car {
    int color;
    int type;
    char *idString;
};
```

Instructions:

- The `color` variable takes one of five values:
  - 0 (Black), 1 (Blue), 2 (Green), 3 (Red), 4 (White)
- The `type` variable takes one of four values:
  - 0 (Convertible), 1 (Supercar), 2 (SUV), 3 (Minivan)
- The `idString` variable points to a string identifying the car. (You may ignore it in the final assembly task.)

The exercise consists of three tasks:

1. Determine the size of the given struct in bytes.
2. Determine which register holds each part of the struct when passed as an argument. Start with the first declared variable.
3. Implement the function `num_to_string` using two jumptables to produce a string formatted as: `A [color] [type]!` Afterwards, print the resulting string using the provided subroutine.

Input/Output:

- **Input:** Struct passed to the function (see task 1)
- **Output:** Printed string, no explicit return value.

Exercise 13: Into the CPU

Objective: The goal of this exercise is to apply your knowledge of computer architecture to optimize an assembly code for matrix operations.

To do this, you will need to access the following website: https://riscv.vercel.app/. In the **Examples** tab, select **Matrix addition & multiplication (C = C + A * B) example**.

Steps to Follow

1. Analyze and Summarize

- Carefully examine the provided assembly code and break it down into its main components.
- Understand the role of each section—this will help you identify potential optimizations.

2. Optimize the Code

- Look for ways to reduce the number of instructions, stalls, or clock cycles per instruction.
- Consider instruction reordering or register usage improvements to enhance efficiency.

Condition

Any optimizations must satisfy the following criteria:

- The correctness of the final result must be preserved. The values stored in memory addresses must remain unchanged after execution.
- The optimized code must achieve better performance than the baseline statistics below:

```
======== [Kite Pipeline Stats] =========
Total number of clock cycles = 3966
Total number of stalled cycles = 1676
Total number of executed instructions = 1518
Cycles per instruction = 2.613
```

Expected Memory State (Only Accessed Addresses)

(1600) = 6	(1608) = 5	(1616) = 1	(1624) = 8
(1632) = 2	(1640) = 5	(1648) = 9	(1656) = -6
(1664) = -9	(1672) = 0	(1680) = 0	(1688) = 2
(2400) = 2	(2408) = -8	(2416) = 1	(2424) = 3
(2432) = 7	(2440) = 5	(2448) = -5	(2456) = -5
(2464) = 2	(2472) = -7	(2480) = -1	(2488) = -3
(2496) = 7	(2504) = -2	(2512) = 5	(2520) = 4
(2528) = -7	(2536) = -4	(2544) = 6	(2552) = 5
(3200) = 62	(3208) = -127	(3216) = -52	(3224) = 79
(3232) = 61	(3240) = -3	(3248) = -26	(3256) = 58
(3264) = -34	(3272) = -8	(3280) = -16	(3288) = 53
(3296) = -23	(3304) = -9	(3312) = -56	(3912) = 20
(3920) = 19	(3928) = 18	(3936) = 8	(3944) = 15
(3952) = 14	(3960) = 13	(3968) = 12	(3976) = 11
(3984) = 10	(3992) = 0		

Tips for Optimization

Leverage Register Usage Efficiently:

- Identify which registers can be used and which already hold the necessary values to minimize redundant loads and stores.

Optimize Instruction Pipelining:

- Analyze the execution pipeline and consider reordering instructions to maximize performance and minimize stalls.

By following these guidelines, you should be able to improve the efficiency of the given assembly code while ensuring that the final results remain correct.