# PA1 Write-up

Park Chaehyun

For each problem, I aimed to implement the reference code in C into assembly line-by-line.

Problem 1)

(1) Initialization

x5, where the sum of the digits will be stored, is initialized to 0. x7, which will be used as the divisor, is initialized to 10.

(2) Loop 1 (LHS)

If a0 is 0, exit the loop. If not, save (a0)%10 (1's digit of a0) to x6, and add x6 to the sum. And divide a0 by 10 and save to a0. This will remove existing 1's digit and move each digit on the left (if any) to the right by one position. And then repeat the loop. The loop repeats until a0 has one digit left, it will become 0 after the division and the loop will be exited. Then we have the sum of each digit of the LHS number.

(3) Loop 2 (RHS)

Same as loop 1, just with RHS number (a1), and add each digit to the sum. Now we have the sum of every digit of the two numbers.

(4) Return value

Save the sum (x5) to a0, which is where the return value should be stored.

Problem 2)

I assumed n is a positive integer, like in the reference code.

(1) Initialization

First, the return address (x1) and n (a0) are saved on the stack. And then it is checked whether n<= 2 or n>=3.

(2) If n<=2

The return value is 1 and saved to a0. The stack is popped and the procedure is returned.

(3) If n>=3

Load n from the stack. Substract 1 from n and save it (n-1) to a0, which is given as argument to th e subsequent call of fibonacci, executing fibonacci(n-1). The return value of fibonacci(n-1) (which a0 n ow holds) is saved to the stack. n is again loaded from the stack. Substract 2 from n and save it (n -2) to a0, which is given as argument to the subsequent call of fibonacci, executing fibonacci(n-2). Th en a0 now holds the return value of fibonacci(n-2). Load the return value of fibonacci(n-1) and add i t to a0. Now a0 holds fibonacci(n-1)+ fibonacci(n-2), the desired value. Pop the stack and return the procedure.

Problem 3)

1. Initialization

I initialized 'sum' (x5) to 0.

2. Save the root node to the queue array

Each element (*right, *left, val) of the root node (a0) is loaded to x29, and the value of x29 is saved to corresponding addresses (incremented from a1). I made use of the fact that each element takes 32 bits, so I used 4 as increment. So 0(a0) holds val (the first element in the struct), 4(a0) holds *left, 8(a0) holds *right.

3. Initialization

'cur' (x6) is initialized to 1 and 'head' (x7) to 0.

4. Loop

(i) how it works

Via this loop, the tree is traversed and each node's value is added to 'sum' (x5). That is done by evaluating every node in the queue. The node to be evaluated is determined by ‘head’. (‘head’ is incremented by 1 after the queue is accessed). It is initially zero so the root node is evaluated. Its left node (if it exists) is added to the queue, and so is its right node (if it exists). ‘cur’ is incremented every time a node is added to the queue, which is used as the next empty index for the queue. And the node’s value is added to the sum, and the loop repeats, evaluating the next node in the queue. So ‘head’ denotes the number of nodes that have been evaluated (its children added to queue, and its value added to sum) and ‘cur’ denotes the number of nodes that have been found. So the termination condition ensures that every node is evaluated.

(ii) implementation
First, it checks whether head==cur. If true, it exits the loop. If false, ‘head’-th node in the queue (queue[head]) is accessed by adding 12*’head’ to queue[0] (a1) because each node is 12 bytes (it contains 3 4-byte elements). The left node is accessed by offset 4 from the node, and the right accessed by offset 8. Each node is added to the queue, similar to [2]. And then the , ‘head’-th node’s value is added to ‘sum’ (x5).

5. Exit
When it exits the loop, ‘sum’ is loaded to a0, which stores the return value.