The recursive Fibonacci number calculator is a benchmark code designed to make use of the stack. Multiple calls to the function will save the program counters value to the instruction stack. Later, when returning from a function call, it can be restored so that the CPU can continue where it left off. In normal C++ code, when calling a function specified values must be passed to it. In Assembly there is no need for this, the function has access to the same registers. For this reason, whenever calling a function it is important to save the current state of the registers into the data stack (Only needs to be done on registers containing valuable data).

The Fibonacci code in assembly (pureassembly\_fibonacci.txt, Appendix) alters from the C++ version (figure 1, Appendix) in some ways. Some data initializing must be done before calling the function. A register must be set to 3072. This will act as a stack register, that holds a pointer to the head of the data stack. R7 was chosen for this purpose. The desired value of n is loaded into R1. R5 is incremented from 0 to 1. This will make setting y = 1 later much cheaper since a MOV instruction takes only 2 cycles, an LDI, on the other hand, takes 3. R2 will be used to store the result after a Fibonacci () call.

Inside the Fibonacci function, there is an if statement with the condition of n smaller than or equal to 1. To evaluate this using our ISA first the number 1 stored in R5 is moved to an empty register R4. The current value of n then can be subtracted from R4, with the result, 1-n, saved in the same register. If the difference is a negative number, it implies the condition is not met (, the program needs to jump to the “else” part of the function. SUB R4 R1 will set the N flag of the status register according to the result. This is immediately followed by a conditional JMR NS R6, which will jump to the address, where the “else” begins, initialized in R6. Otherwise, the R5 is moved to R2, setting y =1. Finally, an RTN will jump back to the address from where the function was called.

Entering the “else” section, the value of n is PUSH-ed to the stack since fib(n-1) is about to be called. After the program returned from the function, n is loaded back into R1. Before calling fib(n-2), the previous result is PUSH-ed to the stack. This ensures the result will not be overwritten by the next function call. After returning from fib(n-2) as well, Fib(n-1) is loaded into R3 from the stack. The two values are summed then the function exits with an RTN instruction.

This is a highly inefficient way of calculating the elements of the Fibonacci sequence. Even with a relatively small input number n, the program could enter more 1000 iterations, causing a stack overflow.