Daniel Ackuaku

21st February 2019.

ENGR 304 Lab 2

Prof. Andrew Jo

Pseudocode for the Fibonacci sequence

initialize the variables term1 = 0, term2 = 1, nextTerm = 0, counter = 0, limit = 40.

initialize the array[limit] to hold the values

while limit < 40 {

nextTerm = term1 + term2;

store t1 into the array[counter];

term1 = term2;

term2 = nextTerm;

increment the counter by one;

}

Printout of the assembly code

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Name: <Daniel Ackuaku> \*/

/\* Course: Engineering 304L \*/

/\* Lab: <Lab Number 2> \*/

/\* Date: < 18th February 2019.> \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* The ".include nios\_macros.s" assembler directive includes NIOS macros for use in the program \*/

.include "nios\_macros.s"

/\* The word-aligned address of the reset vector, value is taken from the

cpu configuration in SOPC builder. \*/

/\*TEMPLATE: replace 0x00 with the appropriate address \*/

.equ RESET\_VECTOR, 0x00

/\* The ".text" assemlber directive indicates the beginning of the code section of the program \*/

.text

/\* The ".org RESET\_VECTOR" assembler directive places the main routine at the reset address \*/

.org RESET\_VECTOR

/\* The ".global \_start" assembler directive exports the "\_start" label as an external symbol \*/

.global \_start

/\* The "\_start" label identifies the program start location for the debugger \*/

\_start:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\* MAIN \*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\* init \*\*\*\*/

FIBONACCI\_INIT:

movia gp, MyArray /\* load the address of the array into the gp register \*/

mov r10, r0 /\* clear register 10 for the sum \*/

addi r11, r0, 1 /\* initialize register 11 to 1 \*/

mov r12, r0 /\* clear register 12 for the sum \*/

mov r13, r0 /\* clear register 13 for the sum \*/

mov r14, r0 /\* clear register 14 for the sum \*/

addi r14, r0, 40 /\* initialize register 11 to 1 \*/

/\*\*\*\* run \*\*\*\*/

FIBONACCI:

add r12, r10, r11 /\* add term1 and term2 to nextTerm \*/

stw r10, 0(gp) /\* store t1 into the memory \*/

add r10, r0, r11 /\* term1 = term2 \*/

add r11, r0, r12 /\* term2 = nextTerm \*/

addi gp, gp, 4 /\* move the pointer 4 bytes \*/

addi r13,r13, 1 /\* increment the counter by one \*/

bne r13, r14, FIBONACCI /\* while counter !+ r14 (40) continue \*/

/\*\*\*\* destroy \*\*\*\*/

FIBONACCI\_END:

br FIBONACCI\_END /\* infinite loop to keep program from going into the weeds \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\* DATA \*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* The ".data" directive identifies the section of the program that defines global variables \*/

.data

/\* The "MyArray" label has an address equal to the address of the first element in the

array of words immediately following the label, this program uses it as arbitrary

zero-terminated example content to sum in the main program. A ".word" array ensures each

element has 4 bytes of storage space \*/

MyArray:

.skip 40\*4

/\* The ".end" assembler directive indicates the end of the program and

all following lines are discarded \*/

.end

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

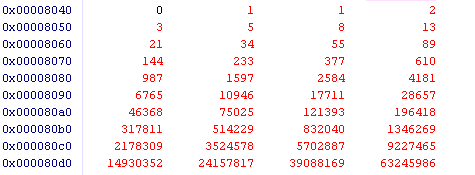


Figure 1 Results from the assembly code showing the Fibonacci sequence memory array.

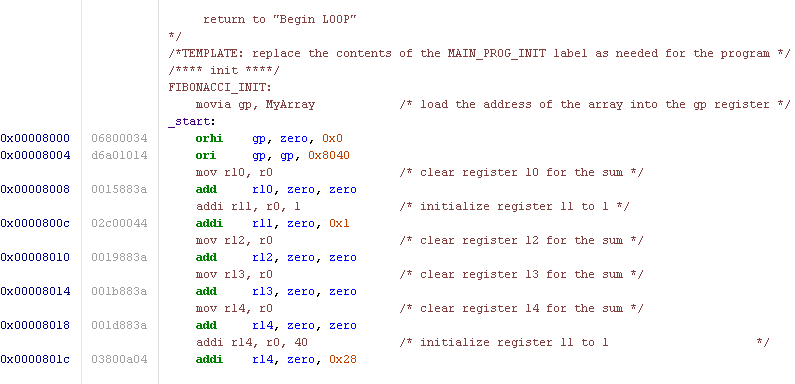


Figure 2 Debugger window of the Fibonacci Assembly code 1.

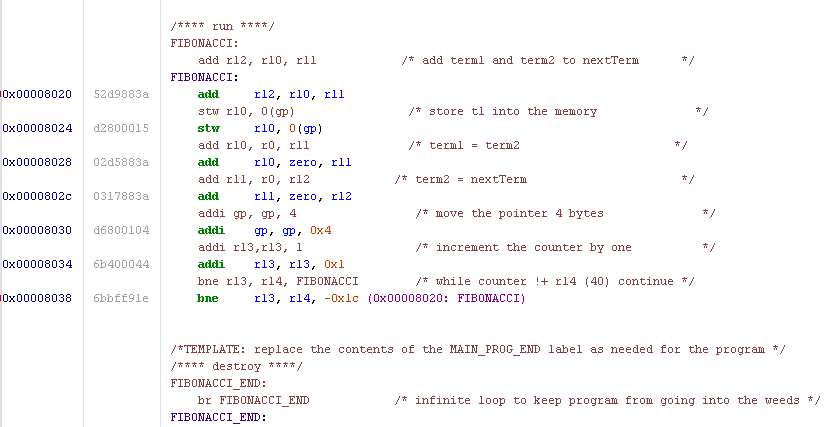


Figure 3 Debugger window of the Fibonacci Assembly code 2.

Code to calculate the first 40 terms in the Fibonacci sequence.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Name: <Daniel Ackuaku> \*/

/\* Course: Engineering 304L \*/

/\* Lab: <Lab Number 2> \*/

/\* Date: <19th February 2019> \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* This program calculates the first 40 numbers in the Fibonacci sequence \*/

int Fib[40]; /\* global since declared outside main() \*/

int main()

{ Fib[0] = 0;

Fib[1] = 1;

int i;

for ( i = 2; i < 40; i = i+1)

{

Fib[i] = Fib[i-2] + Fib[i-1];

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

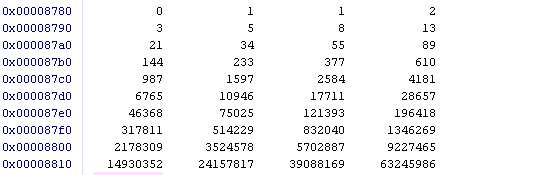


Figure 4 Results from the C code showing the Fibonacci sequence memory array.

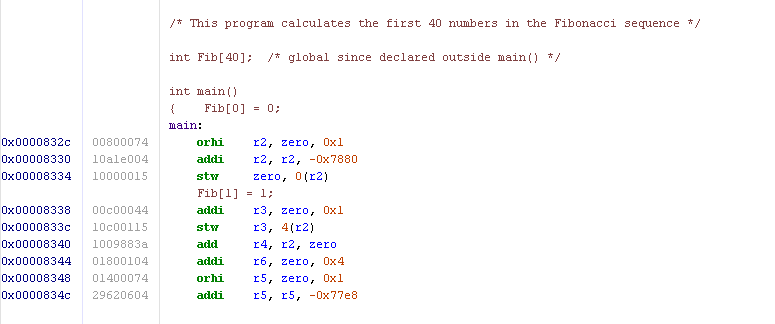


Figure 5 Debugger window of the Fibonacci main C code 1

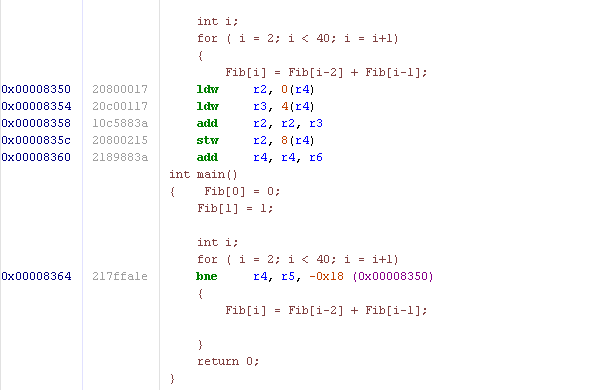


Figure 6 Debugger window of the Fibonacci main C code 2

Table showing the number of instructions executed in each version of the code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | # non-loop instructions | # loop instructions | Total executed in the loop code | Total main function instructions executed |
| Assembly | 8 | 7 | 7 \* 40 = 280 | (7 \* 40) + 8 = 288 |
| -O0 | 10 | 31 | 31 \* 39 = 1209 | (31 \* 39) + 10 = 1219 |
| -O1 | 9 | 6 | 6 \* 39 = 234 | (6 \* 36) + 9 = 243 |
| -O6 | 7 | 6 | 6 \* 39 = 234 | (6 \* 38) + 7 = 241 |

Out of the four versions compared, the best was the version -O6. It accomplished the task with the fewest number of instructions. Version -O6 used 241 lines of code, version -O1 was not far behind with 243 lines of code, the assembly version used 288 lines of code, and the worst was version O0 with 1219 lines of code.

The C code version O1 utilized the resisters within the range r0, r2, r3, r4, and the code version O6 utilized the resisters r0 -r3, r7, r8 and the code version O0 utilized the resisters within the range r0 -r3, r7, r8 whereas the assembly code written utilized the registers r0, r10, r11, r12, r13, r14.

Versions -O1 and -O6 had a similar loop structure that consisted of two sections the first section had 5 lines of code and the second section had 1 line of code. Similar to this loop structure the assembly code also had 2 distinct sections within the loop structure, the first being 5 instructions long and the second having 2 lines of code. Version -O0 also had 2 sections however in this implementation the first section had 25 lines of code and the second section had 6 lines of code.

The handwritten assembly took 41 more instructions than the highly optimized code of version -O6.