Programming Assignment 1

Group: Meghan Brandt, Paige King, and Anika Roskowski

Part 1: Find nth element of the Fibonacci sequence.

Code:

```
main        ;==================== MAIN =======================
    MOV R0, #16     ;Input (target index)
    MOV R1, #0     ;Current index
    B fib    ;start calculation

fib          ;================ Fibonacci =====================
    CMP R1, #1     ;current index <= 1 basecase
    BEQ fib_base_one
    BLT fib_base_zero

    B fib_return          ;calculate return value (aka fib(n))

fib_return    ;---------------- Return Value --------------------
    STMFD SP!, {R2}    ;flip R2 and R3 by using the stack
    MOV R2, R3     ; as a temporary placeholder
    LDMFD SP!, {R3}    ;R2 prevprev and R3 prev of fibonacci
    ADD R3, R3, R2       ;R3 = fib(n-2) + fib(n-1)
    B fib_next     ;begin calculation of next index.

fib_next     ;----------------- Next Index ---------------------
    CMP R0, R1     ;target match current?
    BEQ fib_done          ;target = current: end calculation

    ADD R1, R1, #1       ;if not done, calc next index
    B fib          ; and recall the program

fib_base_one    ;----------------- Base Cases --------------------
    MOV R2, #1     ;store value to R2
    B fib_return
fib_base_zero             ;initialize R2 and R3 to zero
    MOV R2, #0            ;This is not fully required
    MOV R3, #0            ;but it is the safer option
    B fib_return

fib_done     ;================ Program END ===================
    END            ; Calculated value is stored in R3
```

Explanation:

This code finds the nth element of the Fibonacci sequence, where n is defined on line 2. The code above has n as 16. *Main* initializes the target index, n, and the current index to registers R0 and R1 respectively and branches to *fib* to begin calculation of the Fibonacci sequence. In *fib*, first the base cases are checked for, then, if not in a base case, *fib_return* is called to calculate the return value at the current index. *Fib_return* uses the stack to swap the values stored in R2 and R3, this is done so that the oldest value can be moved to R3 and overwritten by the addition on line 17. This allows for simpler code, as R2 will always hold Fib(i-2) and R3 will always hold Fib(i-1) at the beginning of *fib_return* on any given call. *Fib_next* compares the current index with the target and will either end the program or will continue and increment the current index.

Register contents:

R0: Target index (n) specified by the user.
R1: Current index (i)
R2: Fib(i-2) at the start of *fib_return*. After *fib_return* this register holds Fib(i-1)
R3: Fib(i-1) at the start of *fib_return*. After *fib_return* this register holds Fib(i). Thus this is the output register when the program ends.

Testing:

This program was tested with various n values up to 16. The output of these values were compared to the table given in the lab write-up. Similarly, the values were manually checked and verified during calculation, as when n is 16, the Fibonacci value of every index under 16 is also calculated.

Part 2: Fill an array of the even numbers in the Fibonacci sequence

Code:

```
main                ;=========== MAIN ==============
    MOV R0, #6     ;Input (target index)
    MOV R1, #0     ;Current index
    MOV R5, #1    ;Create 'array'
    LSL R5, R5, #16 ;set array addressing (0x00010000)
    B fib    ;start calculation

fib          ;================= Fibonacci ======================
    CMP R1, #1      ;current index <= 1 basecase
    BEQ fib_base_one
    BLT fib_base_zero

    B fib_return          ;calculate return value (aka fib(n))

fib_return    ;---------------- Return Value --------------------
    STMFD SP!, {R2}    ;flip R2 and R3 by using the stack
    MOV R2, R3      ; as a temporary placeholder
    LDMFD SP!, {R3}    ;R2 prevprev and R3 prev of fibonacci
    ADD R3, R3, R2        ;R3 = fib(n-2) + fib(n-1)
    B value_test          ;check value conditions.

fib_next     ;----------------- Next Index ---------------------
    CMP R0, R6      ;target match current count
    BEQ program_end       ;target = current: end calculation

    ADD R1, R1, #1        ;if not done, calc next index
    B fib           ; and recall the program

fib_base_one    ;----------------- Base Cases ---------------------
    MOV R2, #1      ;store value to R2
    B fib_return
fib_base_zero             ;initialize R2 and R3 to zero
    MOV R2, #0            ;This is not fully required
    MOV R3, #0            ;but it is the safer option
    B fib_return

value_test    ;=============== Value Testing ====================
    TST R3, #1      ;test last bit
    BEQ value_even
```

```
    BNE value_odd
value_even         ;if even push to array
    B array_push
value_odd                ;if odd continue to calculate next fib
    B fib_next

array_push     ;=============== push to the 'array' ===============
    STR R3, [R5], #8    ;post increment by 8
    ADD R6, R6, #1        ;add to current counter
    B fib_next      ;continue

array_pop     ;=============== pop from the 'array' ===============
    SUB R5, R5, #8       ;Due to the post increment
    LDR R4, [R5]
    B pop_loop
pop_loop      ;======= loop, popping all values from array ======
      CMP R0, #0
      BEQ program_end     ;For quick output testing
      SUB R0, R0, #1                  ;array is placed backwards
      B array_pop                     ;into R6
;to use this function, replace "program_end" on line 24 with "pop_loop"

program_end    ;================= Program END ====================
    END            ; Calculated value is stored in R3
```

Explanation:

The above code creates an array of size n and fills it with only the even values returned by the Fibonacci sequence. There are only 3 changes in the first 36 lines of code. The first is the addition of the array index register R5 in *main*. Next the branch call in *fib_return* on line 20 changed to *value_test* opposed to the original *fib_next*. Lastly, the comparison on line 23 in *fib_next* changed from comparing R0 and R1 (the target index, n, and the current index, i) to comparing R0 and R6 (the target size of the array, n, and the current element count in the array).

The main addition in this program is *value_test* where the last bit of the value is tested. If this bit is set, the value is odd and is not stored in the array. If the bit is cleared, the value is even and gets added to the end array using *array_push*. In either case, *fib_next* is called after to continue the Fibonacci calculations if the condition is met or the program will end.

As the array is merely being set in this program and not being used, very little array handling is necessary. Most of the array is handled in *array_push* and the R5 (next open address) and R6 (element count) registers. For completion, *array_pop* was written but is not used outside of the testing function *pop_loop*. Note that *pop_loop* is not called in the code as it is given above, but can easily be used by replacing the label in the branch call on line 24 with *pop_loop*. This will iterate through the array and store each value to R4. *array_pop* removes the element from the array, so using *pop_loop* will empty the entire array, this is why it is not being called in this submission version.

Register contents:

R0: Target array size (n) specified by the user

R1: Current index (i) of the Fibonacci sequence

R2: Fib(i-2) at the start of *fib_return*. After *fib_return* this register holds Fib(i-1)

R3: Fib(i-1) at the start of *fib_return*. After *fib_return* this register holds Fib(i).

R4: Return value from *array_pop*, which is the last element of the array of even values.

R5: The address value of the next open space in the array of even values.

R6: The current element count of the array of even values.

Testing:

This code was tested with n set to the value of 6, using the *pop_loop* function to show the contents of the array. The output array with this input n value contained 0, 2, 8, 34, 144, 610 which matched the expected contents given the Fibonacci sequence in the lab write-up.

Part 3: Fill an array of the prime numbers in the Fibonacci sequence.

Code:

```
main                ;=========== MAIN ==============
    MOV R0, #7     ;Input (target index)
    MOV R1, #0     ;Current index
    MOV R5, #1    ;Create 'array'
    LSL R5, R5, #16 ;set array addressing (0x00010000)
    B fib    ;start calculation

fib          ;================= Fibonacci ======================
    CMP R1, #1     ;current index <= 1 basecase
    BEQ fib_base_one
    BLT fib_base_zero

    B fib_return         ;calculate return value (aka fib(n))

fib_return    ;---------------- Return Value --------------------
    STMFD SP!, {R2}    ;flip R2 and R3 by using the stack
    MOV R2, R3     ; as a temporary placeholder
    LDMFD SP!, {R3}    ;R2 prevprev and R3 prev of fibonacci
    ADD R3, R3, R2       ;R3 = fib(n-2) + fib(n-1)
    B value_test         ;check value conditions.

fib_next     ;----------------- Next Index ---------------------
    CMP R0, R6     ;target match current count
    BEQ program_end      ;target = current: end calculation

    ADD R1, R1, #1       ;if not done, calc next index
    B fib          ; and recall the program

fib_base_one    ;----------------- Base Cases ---------------------
    MOV R2, #1     ;store value to R2
    B fib_return
fib_base_zero            ;initialize R2 and R3 to zero
    MOV R2, #0           ;This is not fully required
    MOV R3, #0           ;but it is the safer option
    B fib_return

value_test    ;=============== Value Testing ====================
    TST R3, #1     ;test 1's place
    BEQ value_even      ;if cleared its even.
    BNE value_odd              ;if set its odd.
value_even         ;even numbers cannot be prime
    CMP R3, #2           ;2 is the only exception!!
    BEQ array_push            ;handle exception
    BNE fib_next               ;continue if not 2.
value_odd                ;odd numbers need to be checked for prime
```

```
    CMP R3, #1            ;1 is not prime
    BEQ fib_next

    B prime_calc

prime_calc    ;=============== Check if Prime ===================
    LSR R11, R3, #1    ;divide by two (less to check)

    MOV R9, #3     ;set initial iterator of for loop
    B prime_for_loop

prime_for_loop
    CMP R11, R9    ; target vs current
    BLE array_push        ;if (R11 <= R9) end as prime

    MOV R10, R3    ;temporary copy of value R3 to divide

prime_sub_loop    ;aka division
    SUBS R10, R10, R9
    BEQ fib_next          ;ever sutracts to 0, end as not prime
    BLT prime_for_next    ;subtracts to negative, next continue for loop
    B prime_sub_loop    ;else keep subtracting

prime_for_next
    ADD R9, R9, #2        ;add to iterator - only check odd
    B prime_for_loop

array_push     ;============== push to the 'array' ===============
    STR R3, [R5], #8    ;post increment by 8
    ADD R6, R6, #1        ;add to current counter
    B fib_next     ;continue

array_pop     ;============= pop from the 'array' ===============
    SUB R5, R5, #8        ;Due to the post increment
    LDR R4, [R5]
    B pop_loop
pop_loop     ;======= loop, popping all values from array ======
    CMP R0, #0
    BEQ program_end       ;For quick output testing
    SUB R0, R0, #1                    ;array is placed backwards
    B array_pop                 ;into R6
;to use this function, replace "program_end" on line 24 with "pop_loop"

program_end    ;================= Program END ====================
    END            ; Calculated value is stored in R3
```

Explanation:

The above code creates an array of size n and fills it with only the prime values returned by the Fibonacci sequence. This code uses the exact same Fibonacci calculation method as shown in part 1 and part 2. The first

Value_test checks if the current value of the Fibonacci sequence, Fib(i), is even or odd. If it is even, the only possible prime number is 2 and is specifically checked for on line 42. If the value is odd, the value 1 must be checked for as it is a special case and will never be prime, any other odd value can be checked whether it is prime or not with *prime_calc*. *Prime_calc* checks if

Fib(i) is divisible by any odd value less than Fib(i)/2. By limiting the search to odd values less than Fib(i)/2 we are able to dramatically increase the efficiency of the program. Efficiency matters here because there is no instruction for division; instead values must be repeatedly subtracted and checked in highly repeated loops causing very poor performance of the program.

The *prime* family (all labels starting with *prime*) effectively works as a C for loop, iterating through odd values from the initial value 3 until the iterator is greater or equal to Fib(i)/2 which is stored in R11. The iterator for this loop is stored in R9. These values are all initialized in *prime_calc* and the actual for loop is implemented in *prime_for_loop*, where the conditions are checked and the temporary version of the current Fibonacci value to be used in the *prime_sub_loop* is set. *Prime_sub_loop* is the subtraction loop used to replicate division, repeatedly subtracting the iterator value from the temporary Fibonacci value, until the return value is either zero or negative. If the return value is zero, the iterator is a divisor and the value is not prime. If the value is negative, the iterator is not a divisor and the for loop is continued by branching to *prime_for_next*. Here the iterator is simply incremented to the next odd number, and branches back to *prime_for_loop*.

There are a few helper functions that are defined in this program that are not naturally called. The first of which is *array_pop*, which pops the last value from the array and returns it to R4. Note that similar to popping from a stack in C, this removes the last element from the array. The other function, *pop_loop* uses *array_pop* to iterate through the entire array and store each value to R4. These functions were only used to conveniently check the array contents at the end of the program. As the comment after this function states, to use this function for testing the label in the branch instruction on line 24 must be changed to *pop_loop*.

Register contents:

R0: Target array size (n) specified by the user

R1: Current index (i) of the Fibonacci sequence

R2: Fib(i-2) at the start of *fib_return*. After *fib_return* this register holds Fib(i-1)

R3: Fib(i-1) at the start of *fib_return*. After *fib_return* this register holds Fib(i).

R4: Return value from *array_pop*, which is the last element of the array of primes.

R5: The address value of the next open space in the array of primes.

R6: The current element count of the array of primes

R9: For loop iterator used in the *prime* family

R10: Temporary copy of Fib(i) (R3) used for the repeated subtraction.

R11: Iterator limit for the for loop, defined as Fib(i)/2

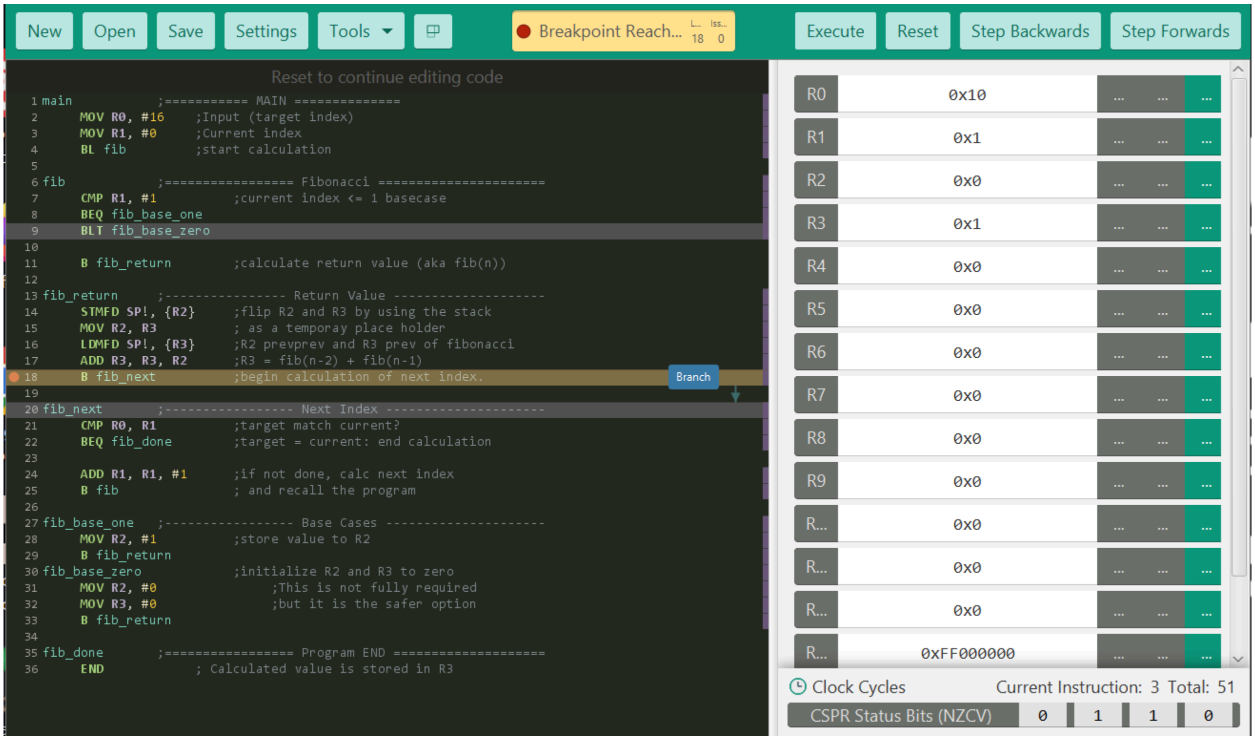
Note: The array of primes starts at address 0x00010000 and is set on line 4 and 5. There is no register that contains the base address of the array, but it can be calculated by subtracting (R6 * 8) from R5.

Testing:

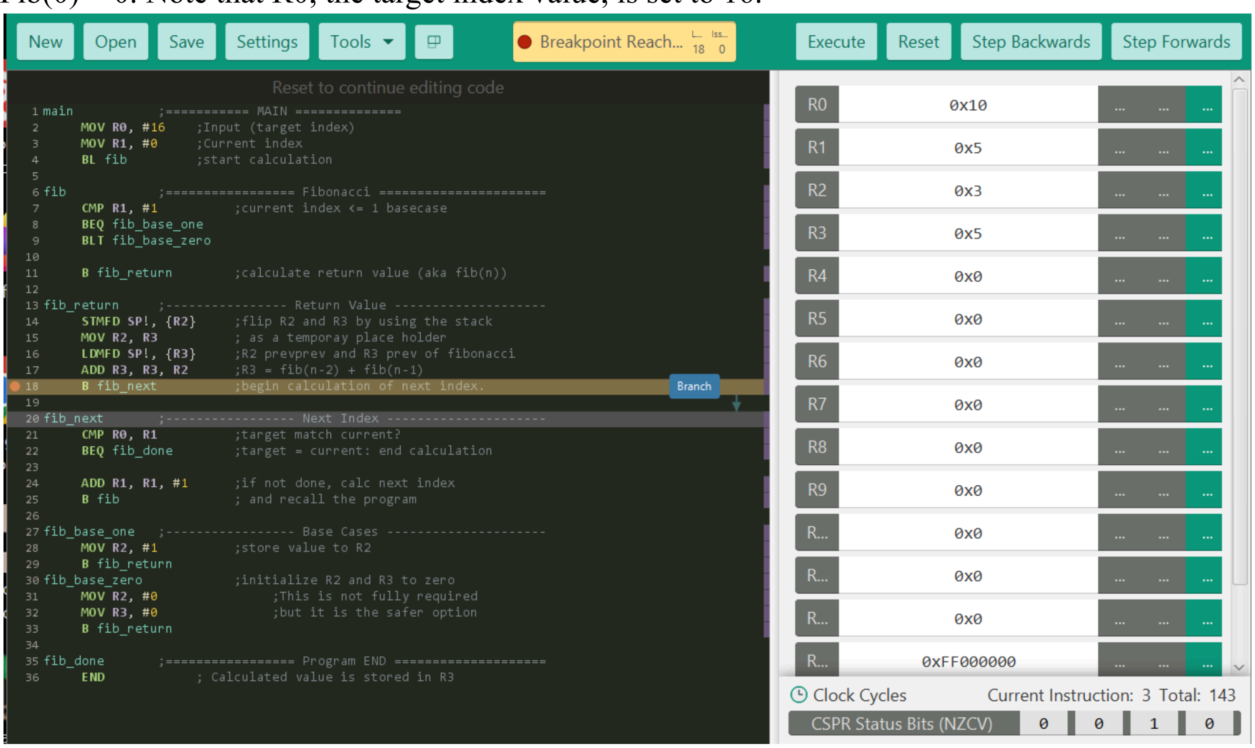
This code was tested with n set at 5, and the entire array was checked and verified using the *pop_loop* function. Using this n value, at the end of the program the array of primes contained 2, 3, 5, 13, 89, 233, 1597 which matched the expected output given in the lab write-up. Further testing was performed iteratively while writing the code, to ensure proper function of branch conditions.

Screenshots of executing code

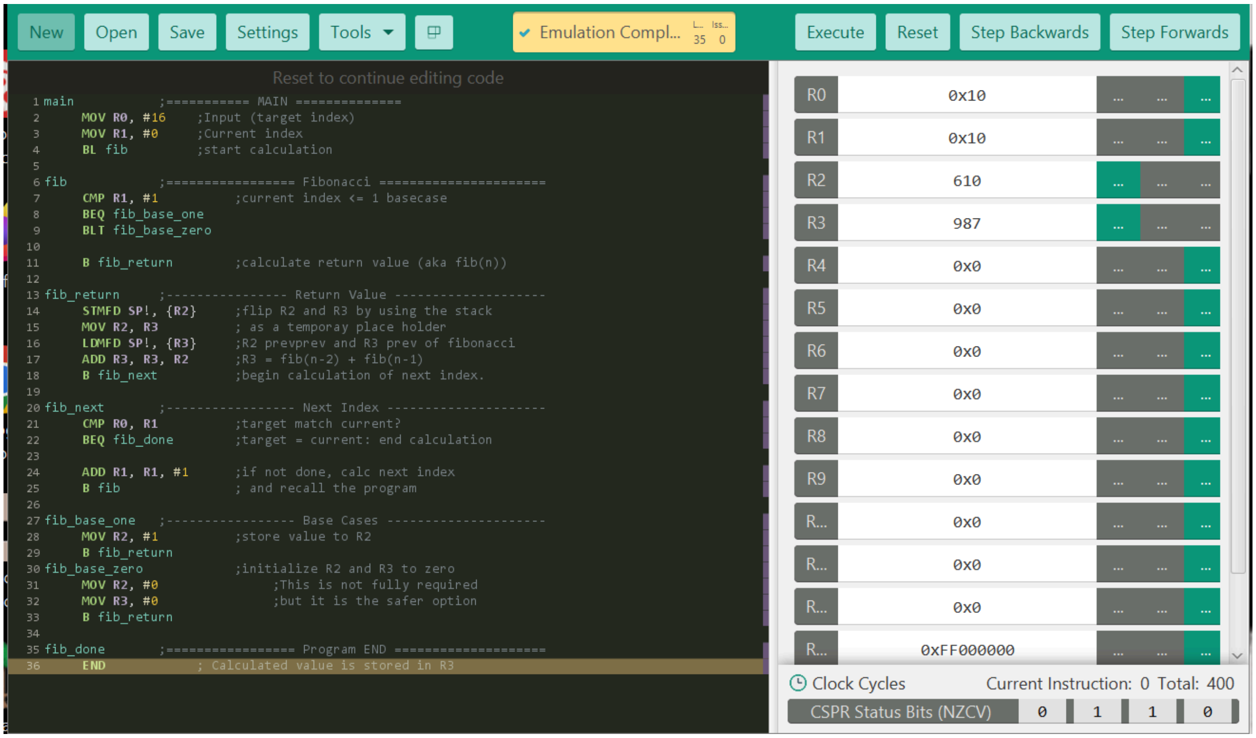
Part 1:



Picture 1: End of the second iteration, paused before *fib_next* is called to increment and begin the next iteration. As this is after *fib_return* has completed, R3 contains Fib(1) = 1 and R2 contains Fib(0) = 0. Note that R0, the target index value, is set to 16.

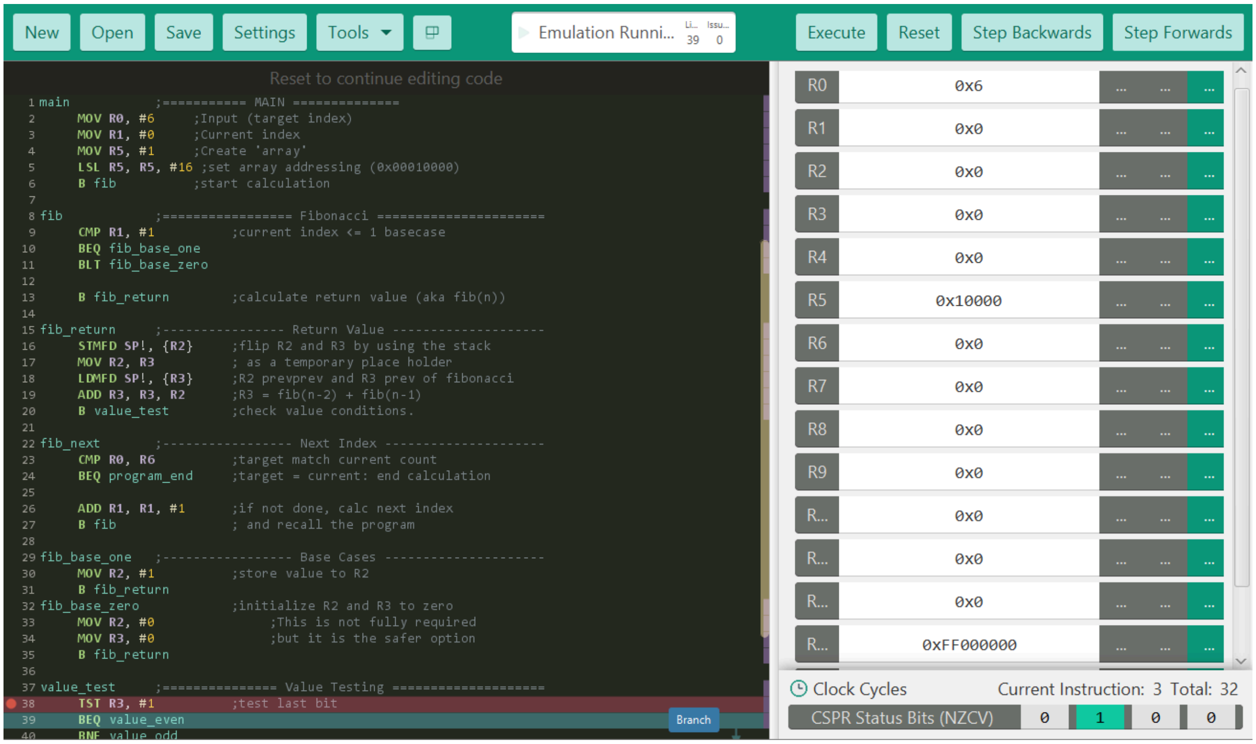


Picture 2: End of the 6th iteration. Here we see that R3 = Fib(5) = 5 and R2 = Fib(4) = 3



Picture 3: Shows the ending state of the program. The output register R3 is showing the correct value of Fib(16) = 987. Similarly, R2 shows Fib(15) = 610. Note that between this picture and the last, R2 and R3 have been switched to display in Decimal instead of Hex.

Part 2:

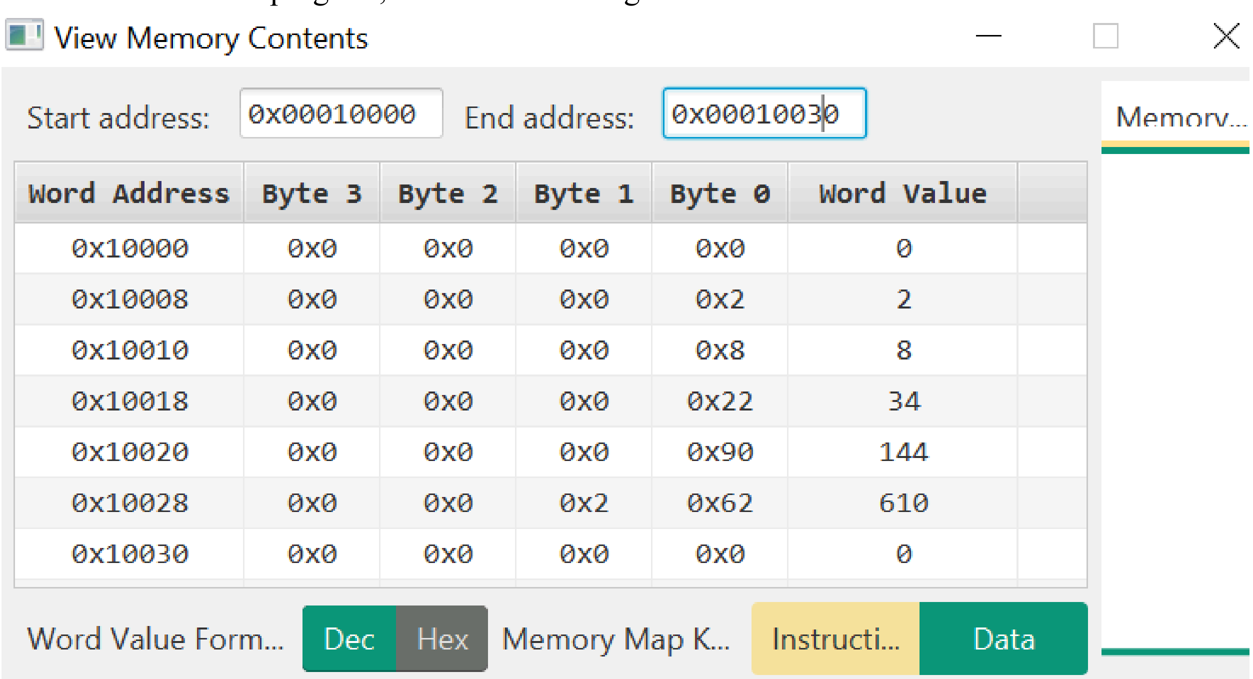


Picture 1: Taken part way through the first iteration, where R2 and R3 have been set and the value R3 is now being tested. R3 = 0 is even, and as you can see the even branch is being taken. Note n = 6 here.

Picture 2: Taken near the end of the 6th iteration, where R3 = 5 which is an odd number and the appropriate odd branch is being taken. Note that R5 has changed to 0x10010, this is because

there has been two pushes to the array so far, 0 at Fib(0) and 2 at Fib(3). This is confirmed looking at R6 which contains the value 2, for these two elements.

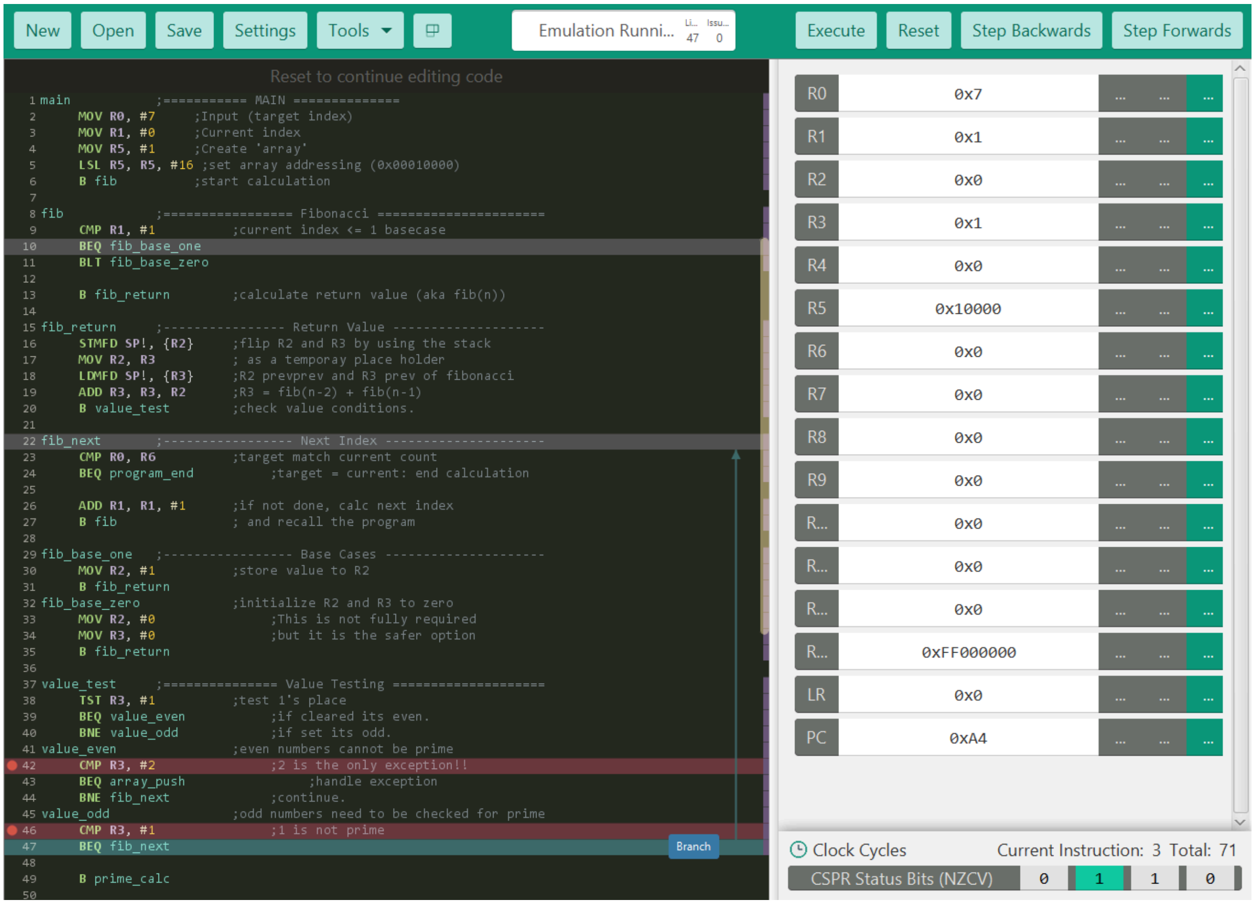
Picture 3: End of the program, with associated register values in Hex.



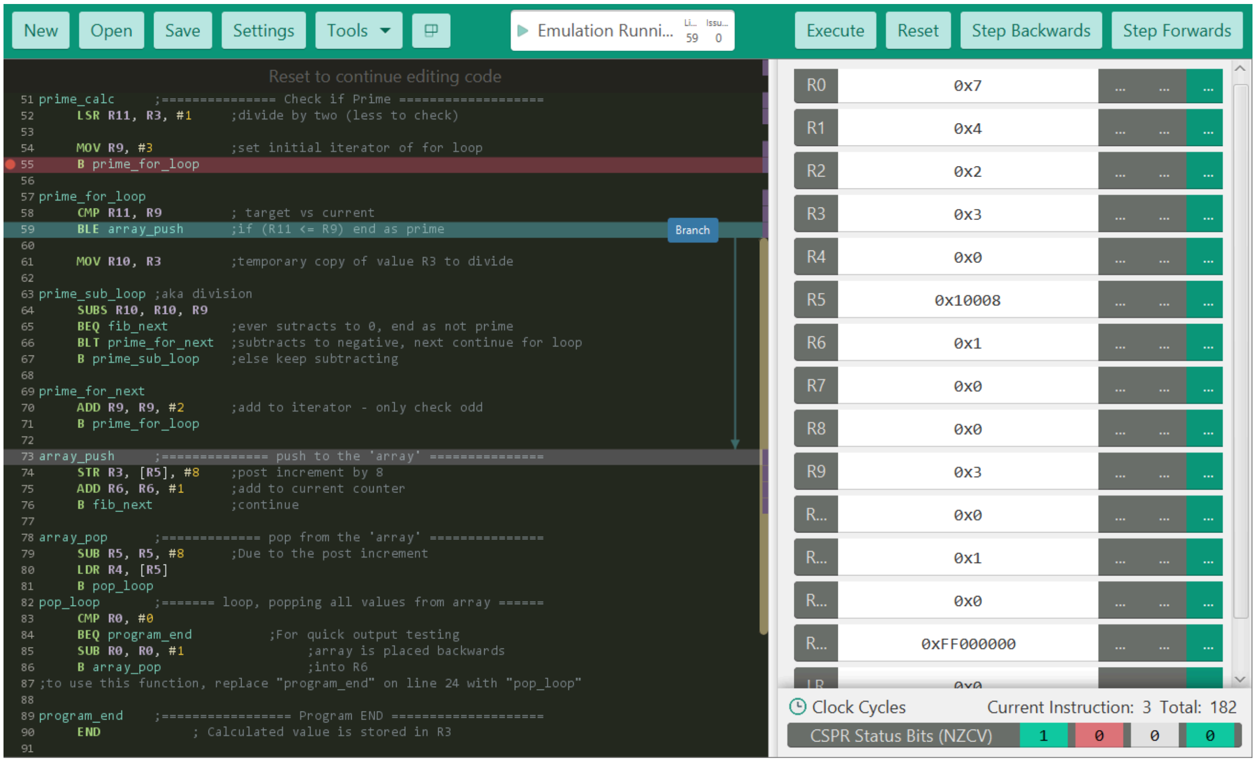
Word Address	Byte 3	Byte 2	Byte 1	Byte 0	Word Value
0x10000	0x0	0x0	0x0	0x0	0
0x10008	0x0	0x0	0x0	0x2	2
0x10010	0x0	0x0	0x0	0x8	8
0x10018	0x0	0x0	0x0	0x22	34
0x10020	0x0	0x0	0x0	0x90	144
0x10028	0x0	0x0	0x2	0x62	610
0x10030	0x0	0x0	0x0	0x0	0

Picture 4: Memory contents of the array at the end of the program, with the first 6 even numbers calculated by the Fibonacci sequence.

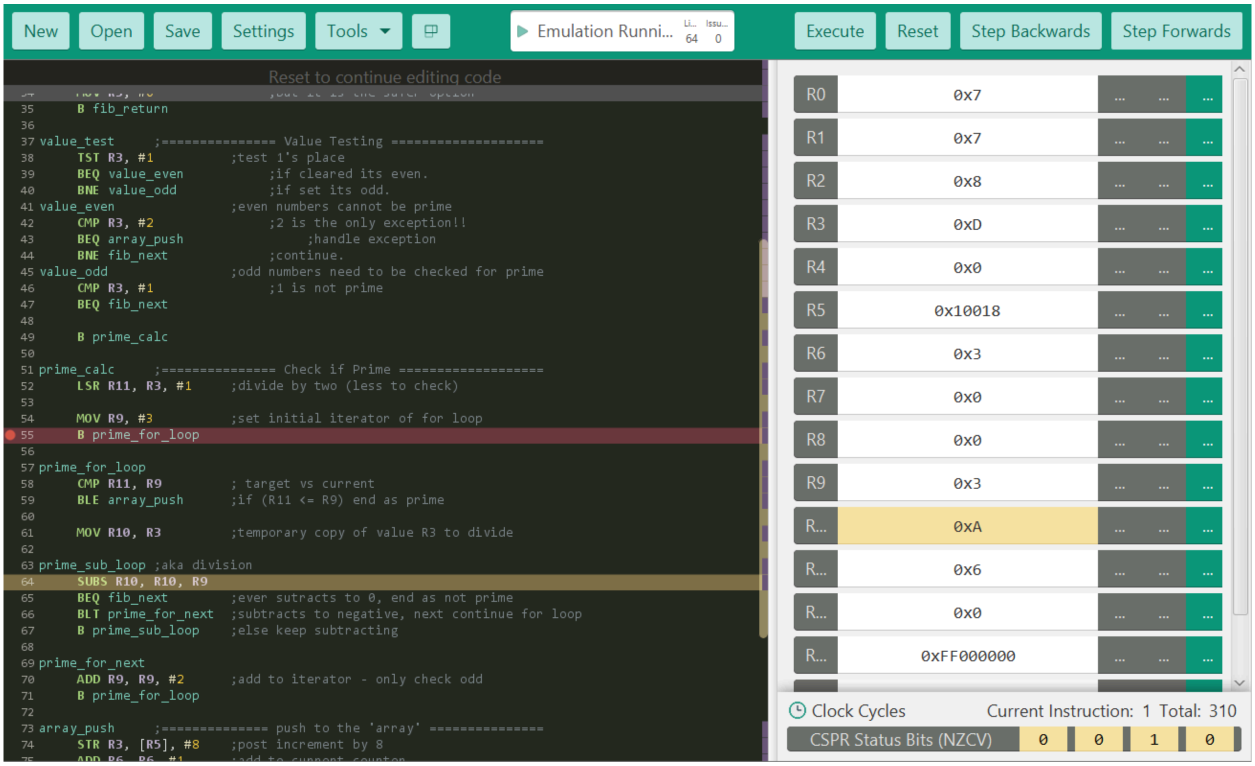
Part 3:



Picture 1: At the end of the 2nd iteration, R3 = 1 is a special case that is specifically handled which can be seen with the branch on like 47. There is another special case branch for the value 2 in *value_even* that pushes the even prime to the array. Note that n = 7 in this example, thus at the end of the program it is expected that the array will be populated with 7 prime elements.



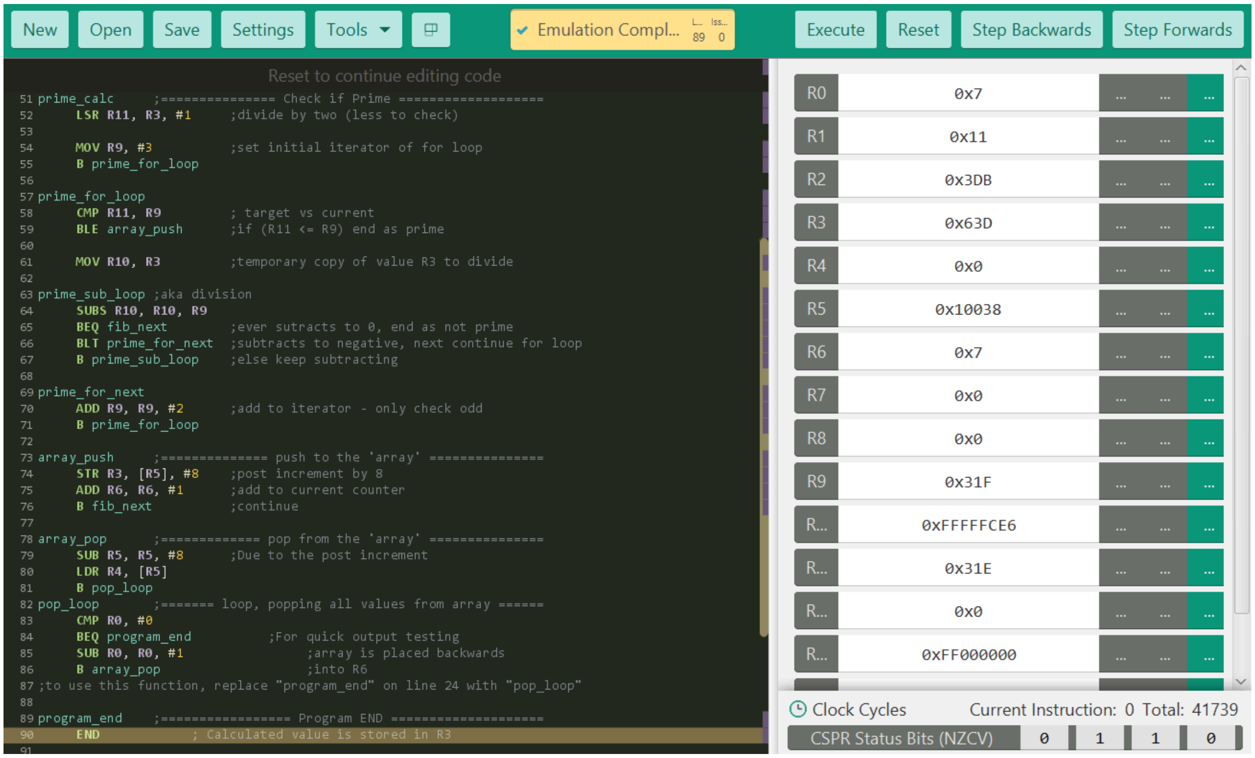
Picture 2: This is the first case the *prime* family is being called. From R1, it's clear this is the 4th iteration. R3 = 3 which is an odd number not equal to one so it is correct that this value is getting to this point. We see that R11 = 1, which is the correct integer division of 3/2 which is gotten via the logical shift right on line 52, which is less than the initial iterator value 3 in R9, which means the Fibonacci value 3 is immediately prime and the for loop is not needed to check this. The value 3 is pushed to the array.



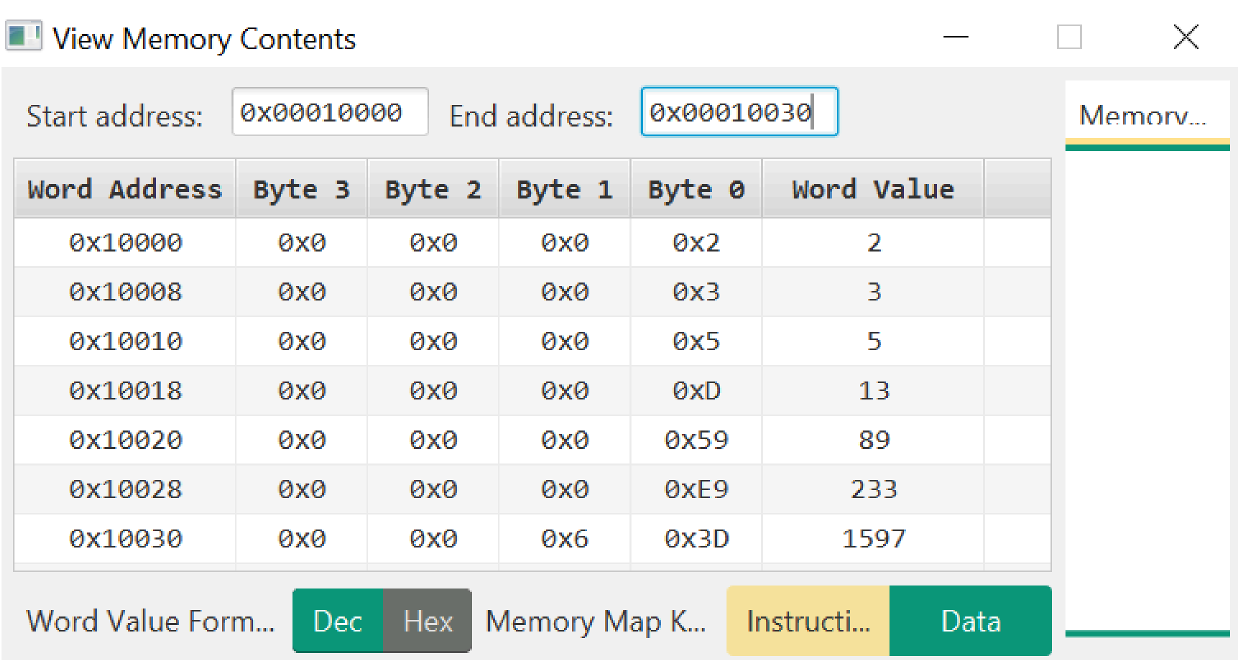
Picture 3: Here R3 = 0xD = 13 which needs to be checked in case it is prime. R11 = 6 which is greater than R9 = 3 which satisfies the condition and *prime_for_loop* can run. Thus *prime_sub_loop* is finally called for the first time in the program. The result of the first subtraction can already be seen in R10 as 0xD - 0x3 = 0xA or 13 - 3 = 10 in Decimal. Note R6 is already at 3 here.

Picture 4: Next iteration of *prime_for_loop*. R10 is reset to 0xD and R9 has increased to 0x5

Picture 5: Next iteration of *prime_for_loop*. Here R10 has not been reset so it still contains its negative value left over from *prime_sub_loop* of the previous iteration. R9 has increased to 7 which is greater than 6 in R11 thus the for loop ends and R3 = 13 is pushed onto the array. This process will continue until the first n (in this case 7) prime numbers are found.



Picture 6: End of the program with all related register values. Note that even after limiting the amount of potential prime numbers to check, the instruction count is still over 40,000 and the program takes multiple seconds to complete.



Word Address	Byte 3	Byte 2	Byte 1	Byte 0	Word Value
0x10000	0x0	0x0	0x0	0x2	2
0x10008	0x0	0x0	0x0	0x3	3
0x10010	0x0	0x0	0x0	0x5	5
0x10018	0x0	0x0	0x0	0xD	13
0x10020	0x0	0x0	0x0	0x59	89
0x10028	0x0	0x0	0x0	0xE9	233
0x10030	0x0	0x0	0x6	0x3D	1597

Picture 7: Memory contents of the array at the end of the program. This matches the array given in the lab write up where n = 7.

Instructions of Use

1. Set input value of R0 on line 2
    a. This will set the index value of the Fibonacci sequence to fine, as in part 1, or it will specify the size of the filtered array of values from the Fibonacci sequence, as in part 2 and
2. Execute.
    a. A warning may pop up about an infinite loop when running part 3, press ignore and continue. There is no infinite loop, *prime_sub_loop* runs through thousands of iterations to correctly replicate division.
3. Check the output
    a. Part 1: Output will be in R3
    b. Part 2 and 3: Output is in a contiguous block of memory starting at address 0x00010000. This can either be seen in the "View Memory Contents" window, or can iteratively be shown using *pop_loop* as previously explained.