Lab 1 Report
DS-5 ARM Prime Number

Luan Vo, October 5th 2020 (Grace day usage 1 out of 10 (first use))

***Purpose & Introduction* — This lab focuses on getting students familiarized with the platform of DS-5 (ARM) assembly code. The report will be entailed how to convert a C/C++ code into ARM assembly language, and the logic behind the conversion. In this lab, I will attempt to convert a given C/C++ code for distinguishing between prime numbers and composite.**

Keyword: ARM, DS-5, leaf procedure, non-leaf procedure

I. PRE-LAB PRECAUTIONS

There is a difference between ARMv8 and LEGv8 that is the stack (SP) offset of 16 instead of 8. This is important as by giving an offset that is not divisible by 16, will cause an error categorized as "error: source not found".

Figure 1. *Source not Found Error (from Piazza)*

Some notes before we start, in ARM, sometimes user must put registers instead of immediate. Such as: SDIV X1, X2, X3. Therefore, there is a need to initialize some registers.

II. GIVEN C CODE

Below is the segmented C code that represents the code that needed to be converted to ARM assembly code, which includes:

A. isPrime:

To determine if the input is a prime number or not:

```
unsigned long long isPrime(unsigned long long n)
{
        unsigned long long i;
        //prime test. if d is 1, number is prime. if d is 0, number is composite
        for (i=2; i <= n/2; ++i) //for loop starting at I = 2
        {//loop stops if n/2 is LESS than I, increment I each iter
                if (n % i == 0)//composite if remainder is 0
                {
                        return 0; //return composite status
                }
        }
        return 1; //return prime status of input
}
```

B. primeIterator:

To assign the numbers' attribute accordingly to their respected arrays of prime or composite, results taken from isPrime function:

```
void primeIterator(unsigned long long a[], unsigned long long prime[], unsigned long long composite[], unsigned long long len){
        unsigned long long i = 0, j = 0, k = 0; //array indices
        unsigned long long d = 0; //prime or composite
        unsigned long long temp = 0; //temp value

        //iterate over elements of a, checking if each is prime
        for( i = 0; i < len; i++) { //for loop starting at I = 0
        //for loop stops if I < len, increment I each loop
                //initialize current element
                d = isPrime(a[i]); //get status of input

                if(d == 1){          //if prime status
                        prime[j] = a[i]; //store input to prime
                        j++;    //increment prime array index
                }
                else{      //if NOT prime (i.e composite) status
                        composite[k] = a[i]; //store to compo
                        k++;  //increment compo array index
                }
        }
        return;    //return nothing (i.e no return expected)
}
```

C. Inputs in Main function

To test the code output, we have defined the following variables within the main code:

unsigned long long arrayLength – length of input array.

unsigned long long a[] – input array.

III. CONVERTING GIVEN TO ARM ASSEMBLY

The equivalent conversion of the C code above in term of ARM assembly is given below:

Comments are denoted as // or /* */ (colored green)

A. ARM Assembly Equivalent

```
        .globl isPrimeAssembly
    isPrimeAssembly:
        SUB SP, SP, #160                // create stack
        //STORING CONSTANTS
        STUR X19, [SP, #0]              //i
        STUR X20, [SP, #16]             //j
        STUR X21, [SP, #32]             //k
        STUR X22, [SP, #48]             //d
        STUR X23, [SP, #64]//temp - SEEMS LIKE IT HAS NO USE IN C CODE, SO USE FOR HOLDING TEMP VALUES
        //STORING EXTRA
        STUR X24, [SP, #80]
        STUR X25, [SP, #96]
        STUR X26, [SP, #112]
        //STORE X30
        STUR X30, [SP, #128]    //X30 IS LR
        STUR X27, [SP, #144]

        //INITIALIZING CONSTANTS
        ADD X19, XZR, XZR               //I = 0
        ADD X20, XZR, XZR               //J = 0
        ADD X21, XZR, XZR               // K = 0
        ADD X22, XZR, XZR               //D = 0
        ADD X23, XZR, XZR               //temp = 0
        ADD X27, XZR, XZR
        //INITIALIZING EXTRA
        ADD X24, XZR, XZR
        ADD X25, XZR, XZR
        ADD X26, XZR, XZR

        Loop:
            //SUB X23, X19, #2              //for testing purposes
            SUB X23, X19, X3                // X23 = i - len
            CBZ X23, Exit                   // If i == len => branch to exit

            SUB SP, SP, #16                 //create stack for LR
            STUR X30, [SP, #0]              //save return address
            LSL X9, X19, #3                 //Shifting i by 1 in address
            ADD X9, X9, X0                  //shifting current address of array a
            LDUR X27, [X9, #0]              //x27 = a[i] (save in arguements /result register across calls)
            BL isPrime                      //jump to isPrime with parameter a[x0] (where x0 = i = x19)

            ADD X22, X5, XZR                // d = return from isPrime = isPrime(a[i]) (return reggister is x5)
            SUB X10, X22, #1                //temp = d - 1
            LDUR X30, [SP, #0]              //restore return address
            ADD SP, SP, #16                 //restore stack

            CBNZ X10, Else                  //if (d-1) != 0 (means d not equal to 1) jumps to Else

            //if d == 1
            LSL     X11, X20, #3
            ADD X11, X11, X1                //shifting current address of prime
            ADD X25, XZR, X27               //prime[j] = a[i] (note: a[i] stored in reg x27)
            STUR X25, [X11, #0]             //STORE BACK TO (SHIFTED) PRIME[j]
            ADD X20, X20, #1                //j = j + 1

            ADD X19, X19, #1                //Increment i++
```

```
            B Loop                          //jumps back to loop

            //else
            Else:
                LSL X12, X21, #3            //LSL x21 to become an address increment of 1
                ADD X12, X12, X2            //shifting current address of composite
                ADD X26, XZR, X27           //composite[k] = a[i] (note: a[i] stored in reg x27)
                STUR X26, [X12, #0]         //STORE BACK TO (SHIFTED) COMPOSITE[k]
                ADD X21, X21, #1            //k = k + 1

                ADD X19, X19, #1            //Increment i++
                B Loop                      //jumps back to loop

        Exit:   //EXIT function
            LDUR X19, [SP,#0]               //load i
            LDUR X20, [SP, #16]             //load j
            LDUR X21, [SP,#32]              //load k
            LDUR X22, [SP, #48]             //load d
            LDUR X23, [SP,#64]              //temp - SEEMS LIKE IT HAS NO USE IN C CODE, SO USE FOR
HOLDING TEMP VALUES
            LDUR X24, [SP,#80]              //load extra registers
            LDUR X25, [SP,#96]
            LDUR X26, [SP,#112]
            LDUR X30, [SP, #128]            //load return to main address
            LDUR X27, [SP,#144]
            ADD SP, SP, #160                //reduce stack to zero
            BR X30                          //jump back to main (with return)
    isPrime:
        ADD X5, XZR, XZR

        //Initializing constants
        ADD X13, XZR, XZR           //i = 0
        ADD X13, X13, #2            //i = 2
        ADD X14, XZR, XZR           //temp14 = divisor = 0
        ADD     X14, X14, #2        //set divisor to 2

        //Udiv = quotient = int(dividend/divisor) (WHERE X15 IS QUOT, X13 IS DIVISOR, X27 IS DIVIDEND - OR N)
        //Remainder = dividend - quotient*divisor
        Loop1:
            SDIV X15, X27, X14      //temp15 = n/2
            SUBS XZR, X15, X13      // (n/2) - i
            B.LO EXIT1              //if (n/2) LARGER/EQUAL than i, so branch when LESS than

            UDIV X15, X27, X13      //quotient = n/i
            MUL X15, X15, X13       //quotient = quotient * divisor
            SUB X16, X27, X15       //temp16 = dividend - (quotient*divisor)
            CBZ X16, IF1
            ADD X13, X13, #1        //increment i: i++
            B Loop1                 //jumps back to loop1, no return expected

            IF1:
                ADD X5, X5, #0  //return 0
                BR X30          //Branch back to Loop, with return (in isPrimeAssembly)

        EXIT1:
            ADD X5, X5, #1          //return 0
            BR X30                  //Branch back to Loop, with return (in isPrimeAssembly)
```

B. Special Notes

Although there is no need to initialize registers, as all registers are that not predetermined to be 0x0000 (if not in use). However, it is good practice to initialize any registers that are not predetermined, to avoid mishaps from happen since some register might not be initially 0x0000.

Additionally, a few registers that are reserved and is not advised to change their parameter in an undesirable manner are:

Table 1. *Predetermined Registers*

Register	Effect
X0	Array A address
X1	Prime array address
X2	Composite array address
X3	Length of input array

As described in the table, these registers hold the address of their respective arrays, which means that we can store/load value to these array by first performing left shift (by 8) and then updating the new address with their respective registers (which will be further discussed below).

IV. STEPS AND APPROACHES

A. Getting Started

The first step I took to solve this problem is to ***write everything down on paper and properly assign registers and their respective uses***. For example, registers 19 to 27 are specifically kept for cross call procedures, such as for a[i] in isPrime(a[i]). Furthermore, to keep consistency, these saved registers are assigned as for loop parameters as well as arrays (such as I, j, k etc). This convention may not be necessary, but for me, to have a benchmark or guideline will make converting easier and will also help in keeping track of which registers are already used.

Secondly, after assigning registers, ***I would go through line by line of C code then write down the logic of C code in human language (pseudo translation in section II. Part A and B).*** This will serve as the foundation for converting to ARM assembly, as ARM assembly is heavily influenced by operational logic that is usually shortened in C. For example, 3 to 4 lines of ARM code is needed just for "for (i=0; i<n/2,i++)", therefore, I need to be careful and precisely write down how each C instruction is achieved in human language (which is then rewritten as comments – in green – next to each ARM instruction in section III. Part A)

B. Converting Steps

With the above preparations, it is now possible to get started with the translation to ARM assembly. Similar to how I translated each line of C code into human language, now I would do the same for ARM assembly, namely by going line by line from human language back to ARM instruction. For example:

Given: for (i=0; i<n/2, i++)

Human language: starting at I = 0, for so long as I is less than (n/2), continue with loop increment of I = I + 1.

ARM Assembly: (taken from sect III. Part A)

```
Loop1:
        SDIV X15, X27, X14      //temp15 = n/2
        SUBS XZR, X15, X13      // (n/2) - i
        B.LO EXIT1      //if (n/2) LARGER/EQUAL than i, so branch when LESS than
        ….                      //omitted middle code

        ADD X13, X13, #1        //increment i: i++
        B Loop1 //jumps back to loop1, no return expected
```

The translation would start from the C function of "isPrimeIterator", then proceed line by line. The final production and code are provided in section III. Part A.

C. Special Case – SDIV, Finding Remainder and BR:

There are, in my opinion, three crucial aspects of this project, which are the conditional for I <= (n/2), n%i == 0 and the proper usage of Branching instructions such as BR, BL and B.

As noted in section III. Part A, and IV. Part B, there needs to be precaution when considering n/2 statement, since branch statement used for this condition is either greater (and equal) or less than (and equal). This means that, the registers are being compared by the B.LO (or B.HT) instruction. ***This means that the preceding subtraction must be done with signed. Thus, the reason why my code have SDIV and SUBS preceding B.LO (refer to Loop1 code in section III part A)***.

For finding remainder, we know that: Quotient = (Dividend/Divisor) + Remainder. However, in ARM instruction and unsigned UDIV = int(Quotient) = int(Dividend/Divisor). This means that, the program automatically rounded the result of quotient. Therefore, to obtain remainder from this, we perform:

Remainder = (Quotient*Divisor) – Dividend

Where:

Quotient = UDIV = int(Dividend/Divisor)

Table 2. *Roles of variables in C/ARM*

Variable	Effect
N	Dividend
I	Divisor
Quotient	int(n/i)

D. Branching Instructions

This section will describe how I picked certain branching instructions:

B – Branching to target (expect no return), this will be mostly used to jump back to a loop, as loops are not expected to return anything.

CBZ/CBNZ – used as a conditional, mainly for jumping to exit once iteration is done.

BR – Jump BACK to register address, this is used to jump back to the caller from callee (such as from isPrime to isPrimeAssembly, or from isPrimeAssembly to main).

BL – Jump to target (with linked caller address), this is used to jump from caller to callee (such as jumping from isPrimeAssembly to isPrime, while providing isPrime a link to jump back to isPrimeAssembly).

V. RESULTS FROM DS-5

A. General Output from DS-5

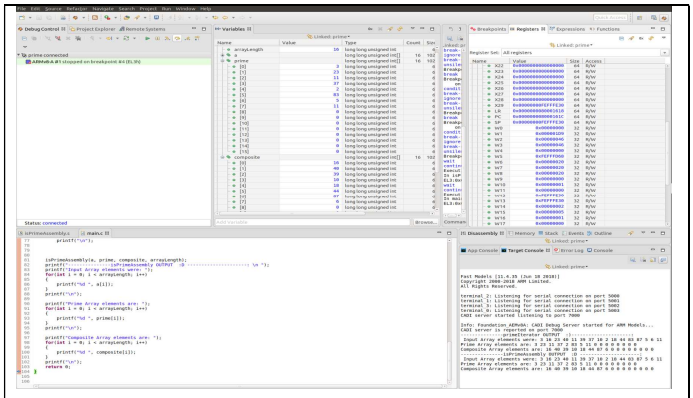
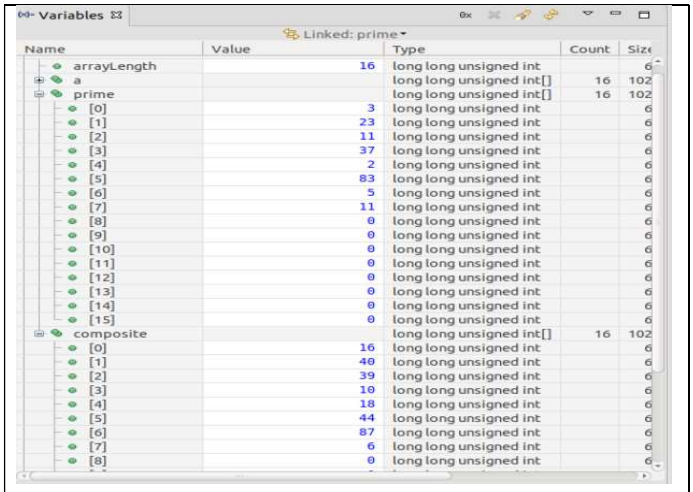
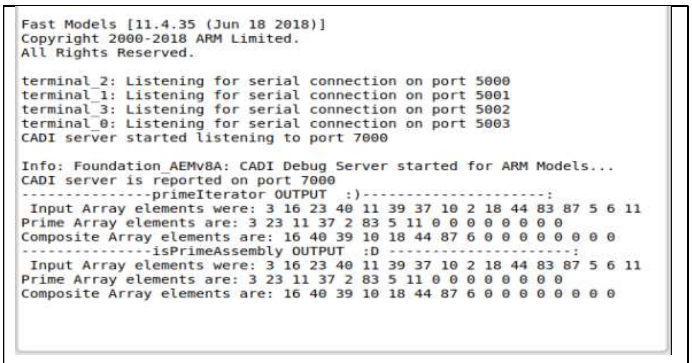


Figure 2. *Program output with given input array (might need to zoom in to see)*



Name	Value	Type	Count	Size
arrayLength	16	long long unsigned int		6
a		long long unsigned int[]	16	102
prime		long long unsigned int[]	16	102
[0]	3	long long unsigned int		6
[1]	23	long long unsigned int		6
[2]	11	long long unsigned int		6
[3]	37	long long unsigned int		6
[4]	2	long long unsigned int		6
[5]	83	long long unsigned int		6
[6]	5	long long unsigned int		6
[7]	11	long long unsigned int		6
[8]	0	long long unsigned int		6
[9]	0	long long unsigned int		6
[10]	0	long long unsigned int		6
[11]	0	long long unsigned int		6
[12]	0	long long unsigned int		6
[13]	0	long long unsigned int		6
[14]	0	long long unsigned int		6
[15]	0	long long unsigned int		6
composite		long long unsigned int[]	16	102
[0]	16	long long unsigned int		6
[1]	40	long long unsigned int		6
[2]	39	long long unsigned int		6
[3]	10	long long unsigned int		6
[4]	18	long long unsigned int		6
[5]	44	long long unsigned int		6
[6]	87	long long unsigned int		6
[7]	6	long long unsigned int		6
[8]	0	long long unsigned int		6

Figure 3. *Zoomed in Array Memory Storage of Fig. 2*

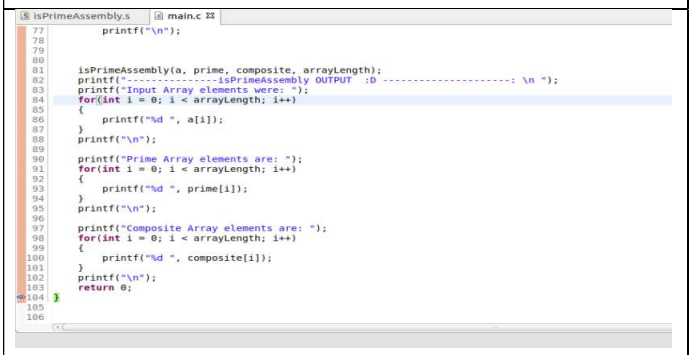


```
Fast Models [11.4.35 (Jun 18 2018)]
Copyright 2000-2018 ARM Limited.
All Rights Reserved.

terminal_2: Listening for serial connection on port 5000
terminal_1: Listening for serial connection on port 5001
terminal_3: Listening for serial connection on port 5002
terminal_0: Listening for serial connection on port 5003
CADI server started listening to port 7000

Info: Foundation_AEMv8A: CADI Debug Server started for ARM Models...
CADI server is reported on port 7000
---------------primeIterator OUTPUT  :)----------------------:
 Input Array elements were: 3 16 23 40 11 39 37 10 2 18 44 83 87 5 6 11
Prime Array elements are: 3 23 11 37 2 83 5 11 0 0 0 0 0 0 0 0
Composite Array elements are: 16 40 39 10 18 44 87 6 0 0 0 0 0 0 0 0
---------------isPrimeAssembly OUTPUT  :D ----------------------:
 Input Array elements were: 3 16 23 40 11 39 37 10 2 18 44 83 87 5 6 11
Prime Array elements are: 3 23 11 37 2 83 5 11 0 0 0 0 0 0 0 0
Composite Array elements are: 16 40 39 10 18 44 87 6 0 0 0 0 0 0 0 0
```

Figure 4. *Zoomed in Console Output of Fig. 2*



```
77          printf("\n");
78
79
80
81      isPrimeAssembly(a, prime, composite, arrayLength);
82      printf("----------isPrimeAssembly OUTPUT  :D ----------: \n ");
83      printf("Input Array elements were: ");
84      for(int i = 0; i < arrayLength; i++)
85      {
86          printf("%d ", a[i]);
87      }
88      printf("\n");
89
90      printf("Prime Array elements are: ");
91      for(int i = 0; i < arrayLength; i++)
92      {
93          printf("%d ", prime[i]);
94      }
95      printf("\n");
96
97      printf("Composite Array elements are: ");
98      for(int i = 0; i < arrayLength; i++)
99      {
100         printf("%d ", composite[i]);
101     }
102     printf("\n");
103     return 0;
104 }
105
106
```

Figure 5. *Zoomed in Console C code of Fig. 2*

B. Extra Testings

If inputs are:

unsigned long long arrayLength = 7;

unsigned long long a[] = {3,87,14, 101, 93, 977, 32}

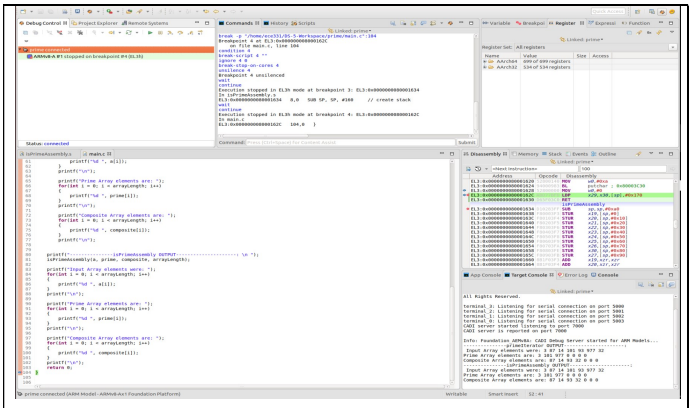


Figure 6. *Program output with customized input array (might need to zoom in to see)*

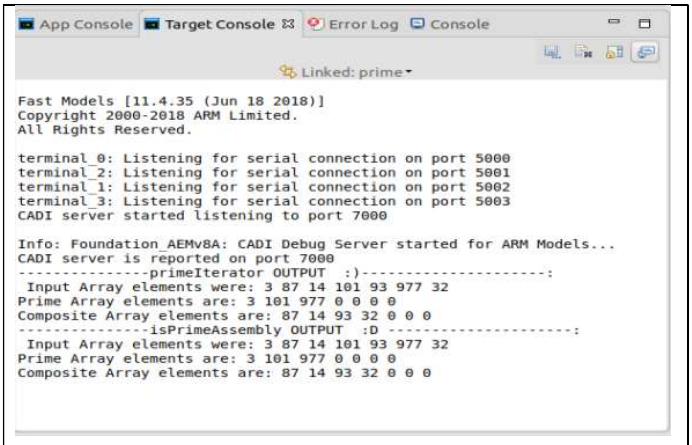


Figure 7. *Zoomed in Console Output of Fig. 6*

If inputs are:

unsigned long long arrayLength = 3;
unsigned long long a[] = {3,87,14}

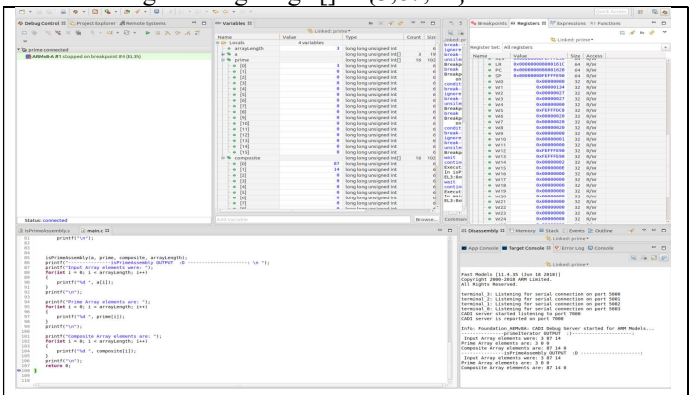


Figure 8. *Program output with customized input array (might need to zoom in to see)*

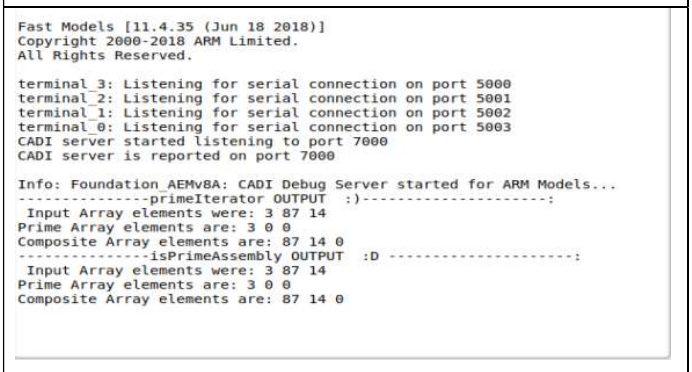


Figure 9. *Zoomed in Console Output of Fig. 8*

C. *Perculiar Behaviors*

When the system tries to have an array size of larger than 16, it seems to not work as expected. Is this normal? I assume it is, because the registers can only hold up to 16 values and thus was unable to make the desired changes that are larger than 16. For example, inspect the figure below:

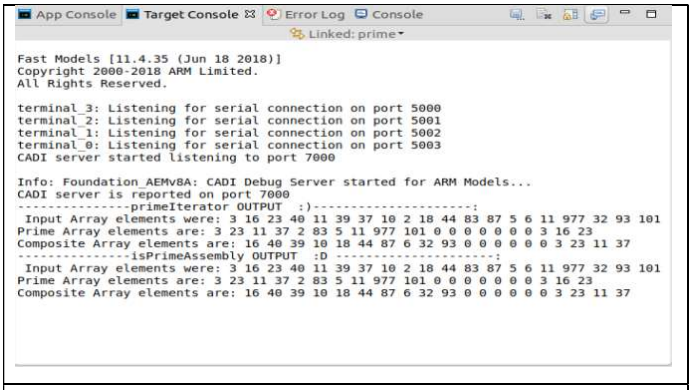


Figure 9. *If input is an array of length 20 – Peculiar*

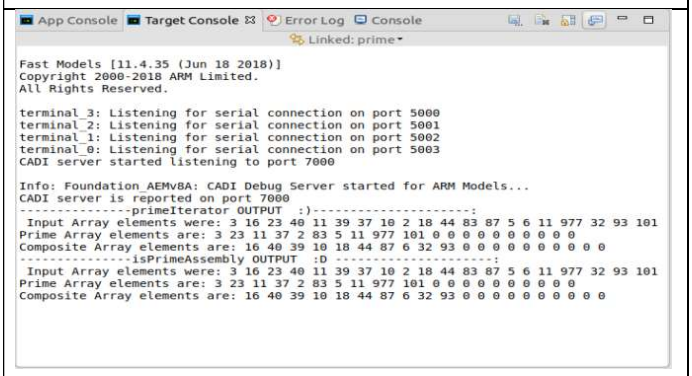


Figure 10. *If input is an array of length 20 - Fixed*

After mulling over this problem, ***I found that the peculiar behavior of array being overflowed (or repeated) is because when I initialized the arrays for prime and composite, they are only of length 16, whereas input array is of length 20*** (meaning that there are only 16 zeros being put into these arrays initially). Thus, after added more zeros to prime and composite arrays to match the length of input array, everything seemed to work out.

VI. SUMMARY & CONCLUSION

In general, the ARM code has (so far) been faithfully assigned integers to their correct respective categories (arrays of prime or composite). However, I have not ran tests on negative numbers (since they are not considered prime in the first place). In this lab, I have learnt a lot about how to – step by step – convert C code into ARM code, and I realized that it would take an enormous amount of time to write code in assembly language, it is simply because we have to state every mathematical operation, as well as assembly language being less intuitive when compared to C code (since many of them are reverse of our intuition.

VII. APPENDIX

isPrimeAssembly is attached with this assignment.