ECE331 Project1

Armv8 Assembly Language Coding and GCC/GDB

Due: Monday, November 4th 11:59 PM

Objectives

- Understand high-level language statements and their relationship with low-level assembly code.
- Understand procedure call mechanisms and resizing of the stack during procedure calls.
- Gain familiarity with the structure of a stack frame.
- Gain familiarity with converting a C language to assembly code.
- Gain experience with debugging assembly in GDB.

The Problem: Split an array of numbers into two separate arrays of primes and composites.

In this lab assignment, you will perform operations involving one array with 10 integers stored in the computer memory. You will write a program in Arm assembly to split this array into two arrays. One stores prime numbers, and the other stores composites. In earlier courses, you learned how to define and use groups of integers stored in an array. For example, in the C language an integer array is defined to store values as follows.

```
int intarray[]={1, 2, 3, 4, 5, 6};
```

In this assignment you will learn how to represent this array in memory using assembler representation and use a series of Arm assenbky instructions to perform the following tasks:

- Split the arrayA (array size = 10). Make two arrays (arrayPrime and arrayComposite) in prime and composite numbers from arrayA. arrayA has 5 prime elements and 5 composite elements. You should note that the values can be in any order. The program should work prime if the prime/composite values in the original array don't alternate. For example, consider arrayA as :

  ```
  int arrayA = {7, 16, 23, 40, 11, 39, 37, 10, 2,18 }
  ```

  The final split arrays, arrayPrime and arrayComposite are :

  ```
  int arrayPrime = {7, 2, 23, 11, 37}
  ```

  ```
  int     arrayComposite     =     {40,     16,     39,     10,     18}
  ```

- The starter code for this assignment can be found on Piazza. After following the instructions for installing the ece331 VM for your operating system or architecture of choice, download both `main.c` and `isPrimeAssembly.s` to the home directory (/home/ece331) or other convenient location. This terminal command will compile and assemble the C/Assembly source files into a debuggable executable called `isPrime`:

  ```
  aarch64-linux-gnu-gcc main.c isPrimeAssembly.s -g -o -static isPrime
  ```

- On the left pane of the browser, you will see a list of files. The one relevant for this exercise is main.c. Double click the file to see the source code.

- At the top of the .c file, you will see an 'extern' function prototype -- this points to your assembly code. More on this later.

- The next two functions implement the Prime algorithm. The first iterates through the input array and splits prime and composite numbers into two output arrays. Since the function arguments are arrays, these are actually pointers to the starting memory address of the arrays.

- In main(), the first few lines initialize the arrays mentioned above. The numbers are different than the example, and the output arrays are initialized to zero.

- After initialization, you will see a function call to the primeIterator() function, which in turn calls the isPrime() function. After this function returns, print statements display the contents of the arrays. isPrimeAssembly() is commented out. This is intentional, as you will uncomment this line to run your assembly instead of the C code.

- You can debug your code using GDB. Check out the video tutorial and read.me files for a crash course in GDB debugging. In the Arm VM, this is as simple as running "gdb isPrime" assuming you ran the gcc command above and produced an executable called isPrime. If you simply run ./isPrime, you will be able to see the printf console output of the C program, which sorts the prime and non-prime numbers. Later, this output may also be useful in debugging your Arm assembly program.

Implementation Using Arm Assembly

- In the left pane of the C/C++ perspective, there is a file called 'isPrimeAssembly.s'. This is where you will be writing your ArmV8 instructions to implement an equivalent of the C function. In the main.c code, comment the line that calls 'PrimeIterator()' and uncomment the line 'isPrimeAssembly()'.

- Navigate the editor to isPrimeAssembly.s. Here you will see stubs for the functions/procedures you had implemented in C. The first procedure is called 'isPrimeAssembly' and will perform the function of the primeIterator C function. You will see a second procedure label called 'isPrime:' which will implement the equivalent C function to see if an individual number is prime.

- There is a nop statement (no operation or dummy instruction) with a breakpoint for debugging purposes. The base addresses of a, prime, composite, and the array length are passed to the procedure using the conventions we discussed in class. This means that you can find them in X0, X1, X2, and X3. You can run to this breakpoint by issuing the GDB commands "b isPrimeAssembly.s:6" (assuming nop is on line 6 of our

assembly file) and then “run”. If you are using the 331 VM, you will be able to see a register summary when you hit the breakpoint in the GDB dashboard. If you are using a different VM, you can find instructions for installing it in the read.me file on Piazza.

Important note:

The text uses an Arm assembly language called LEGv8, while GCC uses Armv8. There are many similarities between these two assembly languages. One difference is that you cannot use the alias 'LR' for the link register, but instead you need to use the register number, X30.

There are several other critical differences and caveats. You can find a good summary here: https://bitbucket.org/HarryBroeders/legv8/wiki/Home

Materials for Submission

Each student is to submit a written report (in pdf format) that includes their assembly code listing with comments.

Each line of code (or group of lines as appropriate) should be commented to indicate what is happening in terms of the algorithm. It is unacceptable to simply copy and paste unoptimized, compiled C code with no understanding of how it works. Additionally, please include the following screenshots taken after your program is working:

1. The data memory after loading the values of arrayA, i.e., all the numbers of arrayA should be in the data memory. The area below the registers has a memory viewer which can be used to show the regions of memory that hold the contents of the arrays

2. The data memory after you finish splitting arrayA, i.e., the prime numbers and composite numbers of arrayA should be in arrayPrime and arrayComposite, respectively.

Note:

- For arrayA, you don’t have to use the set of numbers provided in the example. Any set of numbers is OK. The numbers in arrayA do not have to be in any particular order.

- Your code will be tested for grading with a new set of 16 numbers that are different from your used set of numbers for submission.

- There are several solutions to implement the C operation “x%2” (see the C code in section 5) in Armv8. All these solutions are acceptable. Figure out one among these several possible solutions that works best for you.

- This is NOT a group project; each student’s report and code is expected to be independent work.