

CPE 325: Intro to Embedded Computer System

Lab01

Caesar Cipher and Prime Factorization

Submitted by: Gianna Foti

Date of Experiment: 22 August 2024

Report Deadline: 29 August 2024

Lab Section: CPE353-02

Demonstration Deadline: 29 August 2024

Introduction

In this lab, we had two programs to write. The first program required us to write a C program that implemented a Caesar cipher to encrypt. We were to take a hardcoded string and shift the ASCII value of alphabetical characters right by 3. For the second program, we were given a program with errors. We were supposed to fix the syntactic and logical errors of the code to compute the prime factorization of a hardcoded integer value.

Theory Topics

1. CCS Tools/Features

a. Memory window

The memory window displays the contents stored in memory on the target device. It allows the user to see how much memory is being used visually on their device when debugging. By default, the addresses are given in hexadecimal. In the search box, a specific address can be to jump the view to begin at a given address. When stepping through a program, changed values appear as red.

b. Console window

The console window displays readable text information such as build output, program output, or serial port information. It allows the user to see compiler messages, debugging information, and program output when in the debugging state. This is where we output the results for programs in the lab.

c. Variable window

The variable window displays and tracks the names, types, values, and location of variables as a program is executed. It allows the user to monitor the state of the variables in real-time. If applicable, the location column lists the memory addresses, which can be used alongside the memory window to pinpoint the exact memory location where the

variable is stored.

d. Breakpoints

Breakpoints allow you to pause the execution of your code at specific points. When a breakpoint is set, the program will halt execution when it reaches that line of code. This allows the opportunity to inspect variables, memory, and register states at that moment. This is particularly useful for diagnosing issues, understanding program flow, and ensuring that your code is performing as expected. You can set breakpoints at various locations, such as specific lines, functions, or conditions, to closely monitor how your code is executed in real-time.

2. CCS Commands

a. Run

Run executes the entirety of the program.

b. Resume

Resume begins or resumes the execution from the current execution point. It only stops when it reaches a breakpoint or finishes execution.

c. Step Into

Step Into iteratively steps through the code. It allows for stepping through the program line by line.

d. Step Over

Step Over is similar to step into but treats function calls as a single step, executing the entire function in one go without stepping through each line inside it.

e. Step Return

Step Return continues until the current function is done executing or until it returns.

f. Terminate

Terminate stops the entirety of the program. This command halts the entire program's execution immediately

Results & Observation

Program 1:

Program Description:

This program implements a Caesar cipher to encrypt a string of words. It takes a hardcoded string and shifts the ASCII value of alphabetical characters right by 3.

Process:

1. I hardcoded a string and set the str 'msg' to "Hello all, welcome to CPE325 Fall 2024!".
2. I then initialized the index to 0 and calculated the size = strlen(msg).
3. I then created a while loop for if the index was less than the size

- a. If true, I assigned `x = msg[index]` to get the current character in the string.

Condition: `(x <= 90 && x >= 65) || (x >= 97 && x <= 122)`

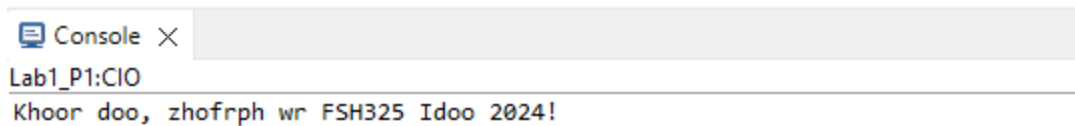
- True: Shift the character x by 3 positions to the right.
- False: No action

I then stored the updated character and saved the shifted character back to `msg[index]`

I then increment the index and went back to the loop condition

- b. If false, exit the loop.
4. End

Program Output:



```
Console X
Lab1_P1:CIO
Khoor doo, zhofrph wr FSH325 Idoo 2024!
```

Figure 1: Program 1 Output

Report Questions:

No lab questions for Program 1.

Program Flowchart:

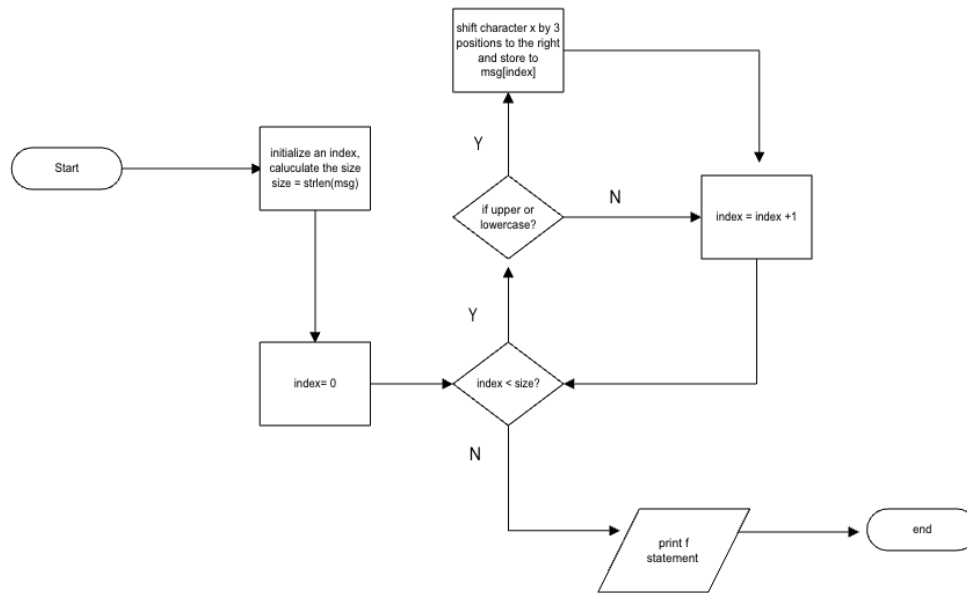


Figure 2: Program 1 Flowchart

Program 2:

Program Description:

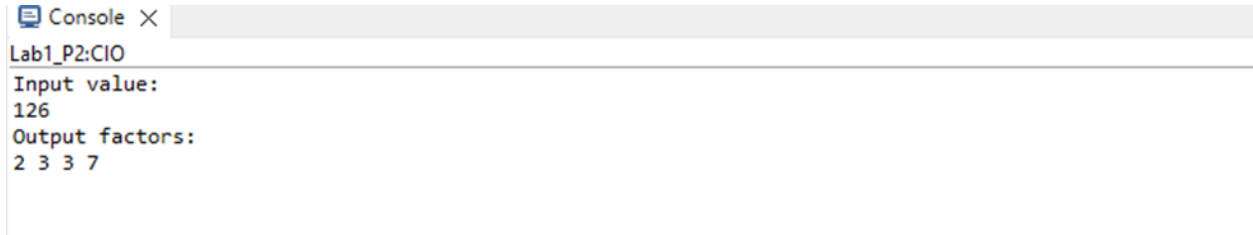
This program determines the prime factorization of a hardcoded value. This program had several issues in the code. It was our duty to find and debug these issues. The fixed solution output showed the output factors of the input value.

Process of Debugging:

1. I started in the main function and fixed any syntactic issues that I encountered. I then worked my way into the `get_prime_factors` functions.
2. After the syntactic issues were fixed and the program was able to be built, it was time to look at the output. However, I noticed there were issues with the output. Using the debugger, I deduced there was a logical error within the code.
3. To fix this, I made the following changes:
 - The loop in the first statement now ends when the value becomes less than 1.

- The second loop now ends when the factor is greater than or equal to the value.
 - The if statement now correctly runs when the modulus of the value and factor equals 0.
4. The code increases the factor (starting at 2) until it finds one that evenly divides the hardcoded number. Once the value is reduced to 1, the code stops.

Program Output:



```
Console X
Lab1_P2:CIO
Input value:
126
Output factors:
2 3 3 7
```

Figure 3: Program 2 Output

Report Questions:

1. **How many clock cycles does the function call to `get_prime_factors()` take to complete, for each input measured?**
 - a. Input Value: 126
 - i. $7,469 - 4,613 = 2,846$ clock cycles
 - b. Input Value: 130
 - i. $8,329 - 4,613 = 3,716$ clock cycles
 - c. Input Value: 145
 - i. $10,849 - 4,613 = 6,236$ clock cycles
2. **How does the input value affect the number of cycles taken?**
 - a. Increasing the input value increases the number of clock cycles taken.
 - b. Decreasing the input value decreases the number of clock cycles taken.

Program Flowchart:

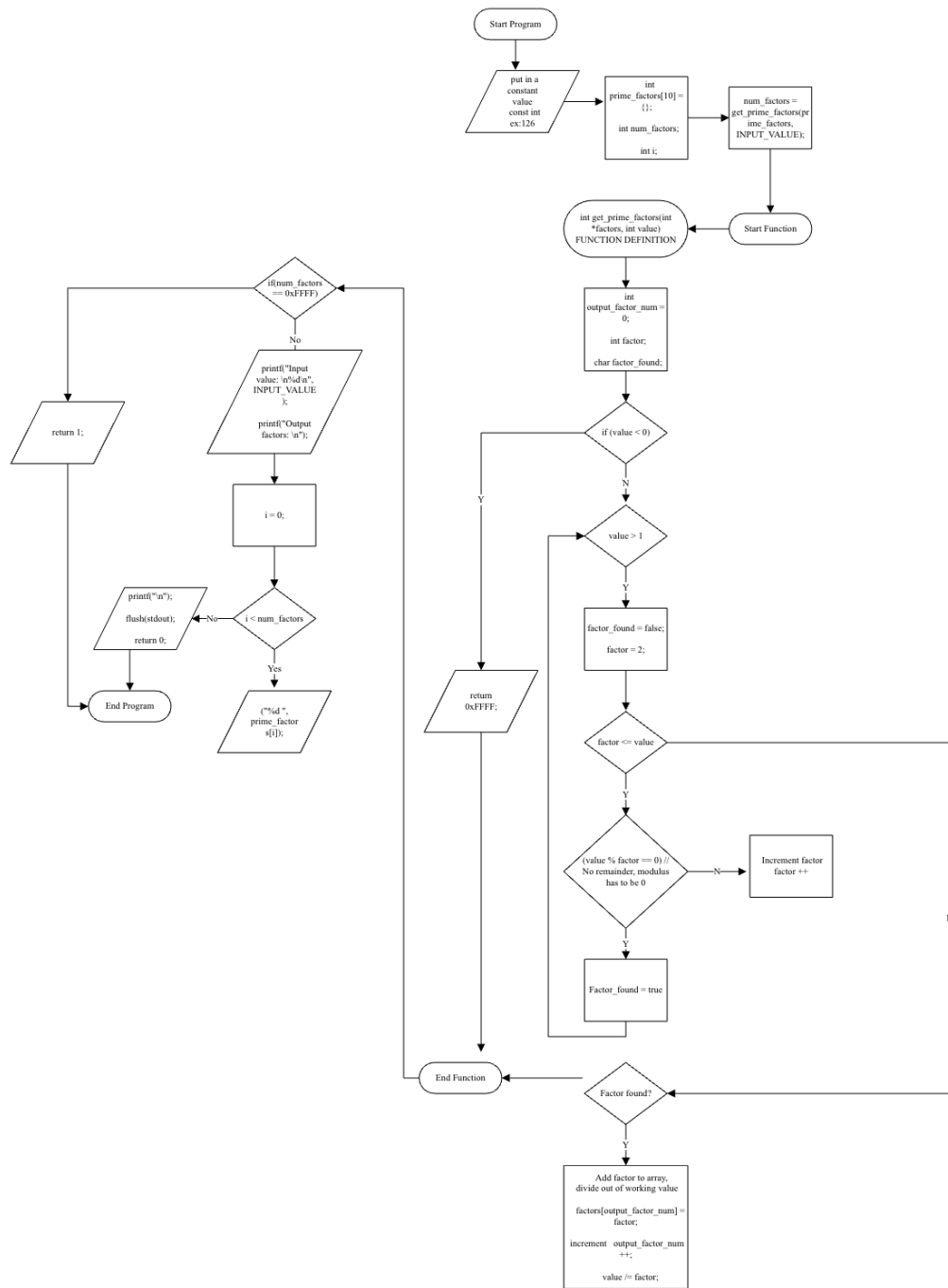


Figure 4: Program 2 Flowchart

Conclusion

In conclusion, this laboratory introduced us to Code Composer Studio and taught us the basics of how to navigate the application. It taught me how to use debugging tools such as the memory window, console window, breakpoints, and many others. It refreshed us on our C coding skills when we had to write a program using a Caesar Cipher which was much needed. Additionally, using the debugging tools helped me to debug the prime factorization program. After fixing the prime factorization program, I learned that by increasing the input value, the clock cycle increased as well. In the reverse, I learned that by decreasing the input value decreased the clock cycle as well. I did not have any major issues with project one and learned a great deal. Overall, this was a good introduction into Code Composer Studio and MSP430.

Appendix

Table 1: Program 1 source code

```
/*-----  
* File: Lab01_P1.c  
* Function: Implements a Caesar cipher to encrypt a string  
* Description:  
* Input: any string of characters  
* Output: the shifted ASCII value of alphabetical characters right by 3  
* Author(s): Gianna Foti  
* Date: 29 August 2024  
*-----*/  
  
//include statements  
#include <msp430.h>  
#include <stdio.h>  
#include <string.h>  
  
//main function  
int main()  
{  
    //this ends the Watchdog Timer  
    WDTCTL = WDTPW + WDTHOLD;  
  
    //hard coded string  
    char msg[]="Hello all, welcome to CPE325 Fall 2024!";  
  
    //setting the index to 0  
    int index = 0;  
  
    //the size of the message  
    int size = strlen(msg);  
  
    //while loop  
    while (index < size)  
    {  
        //element in the array  
        char x = msg[index];  
        //we use 90, 65, 97, and 122 due to the ASCII characters  
        //90 is z, 65 is A, 97 is a, 122 is z  
        //checking to see if the character is upper or lower case  
        if ((x<=90 && x>= 65) || (x>=97 && x<=122))  
        {  
            //shifting to the right by 3  
            msg[index] = x + 3;  
        }  
        //incrementing the index after processing  
        index++;  
    }  
  
    //this line was given to us to print a single line  
    printf("%s\n", msg);  
  
    return 0;  
}
```

Table 2: Program 2 source code

```

/*-----
* File:    Lab01_S2.c
* Description: This program finds the prime factorization of a hard coded value
* Board:    5529
* Input:    Hardcoded short int number
* Output:    Factors of input value printed to console (ordered low to high)
* Author:    Gianna Foti, ghf0004@uah.edu
* Date:      May 13, 2024
*-----*/
#include <msp430.h>
#include <stdio.h>

#define true 1
#define false 0

// This function finds all of the factors in `value`
// .. Factors of `value` are output as elements of `factors`
// .. Function return value is number of factors found (0xFFFF for error)
int get_prime_factors(int *factors, int value)
{
    //Had to uncomment this line out
    int output_factor_num = 0;
    int factor;
    char factor_found;

    if (value < 0)
        return 0xFFFF;

    // Loop while remaining value is not prime
    // changed to value < 1, the while it was not true was not working
    while (value > 1)
    {
        factor_found = false;
        factor = 2;

        // Get lowest remaining factor of `value`
        while (factor <= value)
        {
            //changed from 1 to 0
            if (value % factor == 0)
            { // Is `factor` a factor of `value`?
                // Factor found
                factor_found = true;
                break;
            }

            // Factor not found
            factor++;
        }

        if (!factor_found)
            break;
    }

```

```

        // Add factor to array, and divide out of working value
        factors[output_factor_num] = factor;
        output_factor_num++;
        value /= factor;
    }

    return output_factor_num;
}

int main(void)
{
    // stop watchdog timer
    //this ends the Watchdog Timer, had to add
    WDTCTL = WDTPW + WDTHOLD;

    // Input value for factorization
    const int INPUT_VALUE = 126;

    // Output array
    // Array must have at least as many elements as factors of INPUT_VALUE
    int prime_factors[10] = {};

    int num_factors;
    int i;

    // Calculate prime factors, and check for function error
    num_factors = get_prime_factors(prime_factors, INPUT_VALUE);
    if (num_factors == 0xFFFF)
        return 1;

    // Print input value & output prime factors separated by spaces
    printf("Input value: \n%d\n", INPUT_VALUE);
    printf("Output factors: \n");
    for (i = 0; i < num_factors; i++)
    {
        printf("%d ", prime_factors[i]);
    }

    printf("\n");
    fflush(stdout);

    return 0;
}

```