Project 1: Divide and Conquer Report Draft

October 15, 2017

Sabbir Ahmed
Zafar Mamarakhimov

1 Description

A recursive, divide-and-conquer algorithm was developed and analyzed to multiply together lists of complex numbers. Two different multiplication methods were used to compute the same products to analyze the crossover point.

1.1 Background

A complex number z is given by a real part x and an imaginary part y,

$$z = x + iy,$$

where *i* is the imaginary unit $\sqrt{-1}$.

Multiplying two complex numbers are similar to multiplying polynomials. Let $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$ be two complex numbers. Then their product is

$$z_1z_2 = (x_1 + iy_1)(x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y2 + y_1x_2)$$

That is, the real part of z_1z_2 is $x_1x_2 - y_1y_2$ and the imaginary part is $x_1y_2 + y_1x_2$. The computation of a single complex product requires four real products and two real additions (subtraction is just addition with one operand negative and the "+*i*" doesn't count as an addition as this is really just notation to separate the real and imaginary parts).

As it turns out, there is a way to reduce the number of real multiplications needed to compute a complex product. It is based on the following observation, which is similar to how Karatsuba's method for multiprecision multiplication was derived:

$$(x_1 + y_1)(x_2 + y_2) = x_1x_2 + (x_1y_2 + y_1x_2) + y_1y_2$$

Let t denote the product $(x_1 + y_1)(x_2 + y_2)$; then if the real products $r = x_1x_2$ and $s = y_1y_2$ are computed, the complex product is just

$$z_1z_2 = (r - s) + i(t - r - s)$$

Now, this computation only requires three real multiplications, but increases the number of additions to five. Since multiplication is the more expensive operation, it is expected for the reduction in the number of multiplies to pay-off, at least if the numbers are large enough.

Simply multiplying two complex numbers would not require a divide-and-conquer solution. However, to multiply a list of n complex numbers, there is a natural recursive divide-and-conquer solution, which is to recursively multiply the left and right halves of the list, each of length approximately $\frac{n}{2}$, and then multiply together the results of the two recursive calls. The base case is a list of length one, for which the function simply returns the single value in the list.

When multiplying a list of numbers, the difference between three or four real multiplications per complex multiplication can make a significant difference in the running time, especially if individual real multiplications are expensive. Multiprecision arithmetic is required to handle the growth of the product as the numbers get multiplied. Since the cost to perform a single real multiplication will increase per iteration, the use of the "three-multiply" complex multiplication is expected to be faster than the "four-multiply" version. However, since the three-multiply version requires more additions, it may not pay-off until the numbers are large or the list of numbers is long.

The point at which the asymptotically better algorithm becomes faster is called the *crossover point*.

2 Implementation

The project was

3 Assembly Implementation

The program is written in the AVR Assembler in scripts described in Section 4. 3 registers are used to store the user inputs and compare with the expected inputs. Another register is used to keep track of the number of shifts required for the other registers to be compared accurately.

The following snippets with visual aids will demonstrate the algorithm used in the program. The following walkthrough sets the secret code as 0b00011011, and the user successfully guesses the sequence.

3.1 Initialization

As soon as the game is loaded, the directives and labels are loaded with immediate values and the registers are initialized. The ports are initialized as well - PORTB is used for the UP and DOWN joystick inputs, and the LED and buzzer outputs, PORTD is used for the reset push button, and PORTE for the LEFT and RIGHT joystick inputs.

Table 1: Values of registers at initialization

Register	Before	After
USER	NULL	NULL
CURSOR	NULL	NULL
REALSTATE	NULL	NULL
NSHIFT	NULL	NULL
UART	NULL	NULL

3.2 State Subroutines

The program then moves to STATE0, where the secret code is loaded to a register CURSOR, the value 3 is loaded to a register NSHIFT and it waits for the first input. The current state and a delimiter are also transmitted. All the state subroutines consist of identical instructions, with the exception of the immediate values being loaded into the registers and the UART sequences.

```
; STATE0 is the initial state of the game, where the machine waits for the
; user's first input. The correct input progresses the game to the next stat
; and an incorrect input results in the buzzer being triggered.
STATE0:         RCALL TRANSMIT_0            ; transmit '0' for STATE0
                RCALL TRANSMIT_COMMA        ; transmit ','

                LDI CURSOR, SECRET          ; load and mask the secret code
                ANDI CURSOR, 0b11000000     ; into the CURSOR register
                LDI NSHIFT, 3               ; to shift USER BY 3*2 bits
                RCALL RDINPUT               ; get user's input
```

Table 2: Values of registers before RDINPUT

Register	Before	After
USER	NULL	NULL
CURSOR	NULL	0b00011011 & 0b11000000 = 0b00000000
REALSTATE	NULL	NULL
NSHIFT	NULL	0b00000010
UART	NULL	0,

3.3 Reading Inputs

RDINPUT waits in a loop for the user's input. Once an input via the joystick is triggered, the program counter moves to the subroutine that has been specified to the input. As per this demonstration, JOYSTICK UP was triggered, leading the program counter to call the JOYSTICKUP subroutine.

```
; if joystick up was pressed, the UART transmits 'U,' and the current state
```

```
; register loads the code for UP which is then shifted
JOYSTICKUP:      RCALL TRANSMIT_U              ; transmit 'U'
                 RCALL TRANSMIT_COMMA          ; transmit ','

                 LDI USER, UP                  ; load joystick input code to U
                 RCALL LSHIFT                  ; shift left USER by NSHIFT*2 b
                 RJMP DEBOUNCEUP
                     RET

; waits for user to stop pressing and then returns
DEBOUNCEUP:      SBIC PINB, 6
                     RET
                 RJMP DEBOUNCEUP
```

Here, the UART is transmitted with the ASCII value for the initial of the input that was pressed, i.e. `U', followed by the comma delimiter. The USER register is then loaded with the immediate value that was hardcoded as the input, in this case, 0b00000000 for UP.

Table 3: Values of registers right after JOYSTICKUP

Register	Before	After
USER	NULL	0b00000000
CURSOR	0b00000000	0b00000000
REALSTATE	NULL	NULL
NSHIFT	0b00000010	0b00000010
UART	0,	0,U,

The LSHIFT subroutine is then triggered, where the program decrements NSHIFT until it is ≥ 1. The USER register is left shifted twice per these iterations. All the joystick input subroutines consist of identical instructions, with the exception of the immediate values being loaded to USER.

```
; left shift the user input to match the position of the states in SECRET
LSHIFT:          LSL USER
                 LSL USER                      ; left shift twice per iteratior
                 DEC NSHIFT                    ; decrement the number of shifts
                 CPI NSHIFT, 1
                     BRGE LSHIFT               ; if NSHIFT >= 1, keep looping
                     RET                       ; else, breaks
```

3.4 Evaluating Inputs

The program counter then returns to the STATE0 subroutine, and proceeds through the rest of the lines without requiring an input.

Table 4: Values of registers after DEBOUNCEUP

Register	Before	After
USER	0b00000000	0b00000000 « 3 = 0b00000000
CURSOR	0b00000000	0b00000000
REALSTATE	NULL	NULL
NSHIFT	0b00000010	0b00000000
UART	0,U,	0,U,

```
STATE0:         .
                .
                .
                CLR NSHIFT
                LDI NSHIFT, 3               ; shift REALSTATE to map the co
                MOV REALSTATE, CURSOR       ; of the joystick inputs
                RCALL RSHIFT
                RCALL EXPINPUT              ; check which input was expecte

                RCALL TRANSMIT_COMMA
                RCALL CMPINPUT

                RCALL TRANSMIT_S            ; transmit 'S' for success
                RCALL TRANSMIT_NEWL         ; transmit '\n' to proceed to n
                                            ; state
```

Table 5: Values of registers right before RSHIFT

Register	Before	After
USER	0b00000000	0b00000000
CURSOR	0b00000000	0b00000000
REALSTATE	NULL	0b00000000
NSHIFT	0b00000000	0b00000010
UART	0,U,	0,U,

The NSHIFT register is reloaded with the immediate value, this time, to indicate the number of right shifts required for the REALSTATE register. The RCALL subroutine provides an identical procedure as LSHIFT, except for the direction of the bit shifting.

The EXPINPUT subroutine is then called, which compares the now shifted REALSTATE to the coded joystick immediate values. Once a match is found, the UART gets transmitted with the ASCII character of the initial of the expected input. The program counter returns to STATE0, and transmits another comma delimiter to the UART.

```
; compares the current state to find the expected output
EXPINPUT:       CPI REALSTATE, UP           ; if current state is 'UP'
```

Table 6: Values of registers right before EXPINPUT

Register	Before	After
USER	0b00000000	0b00000000
CURSOR	0b00000000	0b00000000
REALSTATE	0b00000000	0b00000000 » 3 = 0b00000000
NSHIFT	0b00000010	0b00000000
UART	0,U,	0,U,U,1,S,\n

```
        RCALL TRANSMIT_U          ; transmit 'U'

    CPI REALSTATE, DOWN           ; if current state is 'DOWN'
        RCALL TRANSMIT_D          ; transmit 'D'

    CPI REALSTATE, LEFT           ; if current state is 'LEFT'
        RCALL TRANSMIT_L          ; transmit 'L'

    CPI REALSTATE, RIGHT          ; if current state is 'RIGHT'
        RCALL TRANSMIT_R          ; transmit 'R'

        RET
```

Finally, the CMPINPUT subroutine is called to compare the USER register with CURSOR. An equal comparison returns the program to the STATE0 subroutine, and proceeds to transmit a successful message to the UART followed by the next state. An unequal comparison branches to BUZZERON and resets the game to STATE0.

```
; compare user's input to the current state and returns if true, else, branc
; to BUZZERON to reset the game
CMPINPUT:       CP CURSOR, USER
                    RET                     ; if equal, return from subrouti
                BREQ BUZZERON               ; else, trigger buzzer
```

The program then proceeds through identical procedure for the remaining states, until the LEDON subroutine is called after the input of a successful sequence.

Table 7: Example of values of registers after STATE1

Register	Before	After
USER	0b01000000	0b01000000
CURSOR	0b01000000	0b01000000
REALSTATE	0b01000000	0b01000000
NSHIFT	0b00000001	0b00000000
UART	1,D,	1,D,D,2,S,\n

4 Code

The assembly code used for the implementation has been attached and provided at the end of the report.

4.1 main.asm

Contains the state machine implementation of the game and instructions to handle all the inputs and outputs.

4.2 uart.asm

Contains the specifications for the UART interface.

5 Testing and Troubleshooting

Because of hardware issues with the AVR Butterfly and its temporary supply issue, the program could not be tested until later in the design timeline. Testing was performed using supplies from peers, with limited availability. Early tests showed failures in the design where the inputs were incorrectly mapped to their ports. After correcting the I/O issues, complex subroutines such as loop shifting were added for elegant solutions and successful tests were performed. The LED was connected to PIN 1 of PORTB and the push button to PIN 7 of PORTD. Additional subroutines were added in the UART instructions later without the available hardware for testing.