ECS 50

Lab 1: Basements and Byters

February 8, 2023

Ryan Swift

INTRODUCTION

The task assigned in this lab was to create a device which could simulate a modified version of a dice roll. The title of the lab, Basements and Byters (BnB), is an allusion to the popular role-playing game Dungeons and Dragons (DnD) and its reliance on rolling dice as a proxy for randomness. The key difference in the BnB version of a dice roll is that each face of a die displays binary digits, rather than decimal. Additionally, BnB can be played in two different variations of a dice roll: either a single 16-sided die (denoted as 1d16z), or two, 8-sided die (denoted as 2d8z). The difference between the two modes is the statistical frequency of the different possible values; in 1d16z mode, each value (0-15) is equally likely to occur, whereas in 2d8z mode, the values in the middle of the range (ex. 6, 7, 8, 9) are more likely to occur than the extreme values (ex. 0, 1, 14, 15). The simulated hardware at our disposal was a PIC12F508 microprocessor with two input switches and 4 output LEDs, and the program was to be written in the PIC12F508's assembly language. The simulator used in this project is from microcontrollerjs.com, which limits the maximum number of instructions to 64 (0x00 - 0x3F), and thus the program created to simulate a dice roll in BnB must be 64 instructions or less.

THEORY OF OPERATION

The dice roll simulator (DRS) contains two switches attached to the bottom left and middle left of the PIC12F508, and four LEDs- one in the top left, and in all three locations along the right side of the CPU. The middle left switch (GP4) is the mode selector switch. When the switch is closed, the roll is simulated in 2d8z mode (two dice, each with 8 sides), and when the mode switch is open, the roll is simulated in 1d16z mode (one 16-sided die). While the mode selector switch may be opened and closed at any point during the roll, the roll will be simulated using the choice of mode set at the start of the roll, when the on/off switch closes. The bottom left switch (GP3) is the on/off switch which starts and ends the roll. When the switch is closed, the roll is started. While the dice are "rolling", the top left LED (GP5) will blink until the roll ends. To end the roll, open the on/off switch. The result of the roll will be displayed using the LEDs as binary bit indicators: on = 1, off = 0. The bit assignments are: GP0 = bit 0, GP1 = bit 1, GP2 = bit 2, and GP5 = bit 3. For example, if the LEDs at GP5 and GP1 are on and the others are off, this is representative of the unsigned binary number 1010, which is 10 in decimal. A circular buffer has been implemented to store the most recent 16 rolls. Though there are limitations to this method- the 17th roll will overwrite the first roll- this implementation allows for continuous storage for as long as the game is played.

DISCUSSION

Without a doubt, the most difficult aspect of this project for me was writing a program which did not exceed the maximum limit of 64 instructions. The initial draft of the DSR program contained around 90 instructions and thus, regardless of whether or not it was functional, it simply could not run in the simulator. My philosophy in programming style tends toward transparency and clarity, at the cost of efficiency and succinctness. This was a significant challenge that I was forced to overcome in the process of creating this program, as it turned out that my definition of clarity does not quite fit within 64 instructions of PIC12F508 assembly language.

In the majority of cases, the sole purpose of revisions were to eliminate unnecessary variables and instructions. One example of eliminating variables was optimizing the subroutine which computed the number to be output from the dice roll. In the 2d8z dice roll, I had created variables for both dice, in addition to the roll_num variable which would eventually be output. This made sense to me, as anyone reading the code would have no trouble understanding where the values for each of the two dice were stored, and how the output was related to them. After a couple of revisions, I realized that I did not actually need *any* of those extra die variables, let alone two. The solution that I came up with was to store the first die in roll_num and store the second die in the working register, and subsequently summing roll_num and the working register and storing it back in roll_num. This ended up shaving off four instructions, as well as the memory allocations for two unnecessary variables.

Another significant revision I made to my program was in the overall structure. This was another case of trying to make the program clear and readable, as each routine in the first revision would jump to the one written directly below it, and the final routine would jump back to the top. This version of the program was essentially a group of distinct subroutines grouped together by "goto" instructions, with no "main function" or primary routine which would call the rest of the subroutines and return there. As it turns out, I was able to save somewhere around four instructions by restructuring my program to include a routine akin to a "main" function which implemented calls to, and returns from, subroutines.

By far, the most significant savings in instruction count came from reducing the complexity of my razzle_dazzle function. The first revision of this function was far more interesting than the final version, but was also about three times longer. It involved clearing all of the LEDs at the start of each roll, then sequentially turning them all on, then all off. It also included a check to determine whether the on/off switch had been opened in between every individual LED instruction, so that the cessation of the roll and light show would be as responsive as possible to the user's input. However, despite the appeal and interest it generated, this version of razzle_dazzle contained a lot of unnecessary and expendable instructions, and thus it was repeatedly cut down. I decided not to completely eliminate the subroutine because I felt that it was necessary to inform the user when the roll was happening and when it was not, and thus I resorted to simply blinking the GP5 LED as a roll indicator.

Though it initially seemed quite daunting, I found that the implementation of the circular buffer was not too particularly difficult. As soon as I read about it, I knew that I would be using the FSR and INDF to move a pointer and access different memory locations. Implementing the circularity was probably the most challenging aspect of the buffer, and I was relatively satisfied with the solution I created. Since I knew I was going to need to wrap the buffer back to the beginning, it was clear that incrementing the FSR was not going to be sufficient on its own for circularity, as incrementing would eventually exit the region of memory addresses allocated to the buffer. Instead, the incrementation had to be supplemented with masking the upper nibble of the memory addresses, and then forcing the upper nibble to always take the value 0001. I'm sure there are may ways to do this, but I found that using logical operators performed both of those jobs well. After incrementing the FSR, ANDing it with 00001111 (0x0F) ensured circularity within a set of 16 consecutive memory locations, and then ORing the resulting value with 00010000 (0x10) forced those 16 values to be 0x10 through 0x1F. Although the OR which set the upper nibble to 0001 could have been replaced by adding 0x10 to the result of the AND, I thought that the OR operation was a more elegant option, as all it required was a logical comparison and a bit flip, rather than arithmetic. Evidence of the circular buffer working as intended have been included in the Appendices. Appendix A includes screenshots of the state of the circular buffer after 16 rolls in 1d16z mode (Figure 1), as well as after 17 rolls (Figure 2). The 17th roll demonstrates that the buffer will, indeed, wrap around to the initial memory address. Appendix B

contains the 2d8z counterparts, with Figure 3 illustrating the circular buffer after 16 rolls, and Figure 4 depicting the buffer after the 17th roll in 2d8z mode.

One of the easiest revisions to the initial program was in the extraction of the bit values from roll_num. Despite already being familiar with using rotation through the carry to extract a bit value, I decided to try using logical operators instead, as I was quite happy with the masking capability of the AND instruction that I discovered while working on the circular buffer. The initial implementation of the bit extraction was to iteratively mask all but a single bit, check the value of the bit, then either turn on the corresponding LED if the bit was a 1, or do nothing if it was a 0. I repeatedly tried to come up with a subroutine for this, but I could not develop a method which could iterate over both the bits of roll_num and the GPIO pins while also using fewer instructions. However, I eventually realized that extraction via rotation saved an instruction per LED, so I revised the program to used that method instead. In addition, since the rotations effectively destroyed the output value in the process, I had to rearrange the structure of the program such that the value was stored in the circular buffer prior to the process of being output.

Regarding the implementation of the linear feedback shift register, I would like to extend my gratitude to our instructor, Dr. Posnett for providing us with the code for it [1]. The linear feedback shift register (LFSR) was used to generate pseudorandom 8-bit numbers, and served as the core mechanism behind the DRS's approximation of randomness, which- when combined with extracting only a subset of those eight bits for use- resulted in a similar probabilistic distribution to what one might expect from an authentically random process. In Appendix A, the values from 1d16z mode are almost all unique, and include values close to the extreme ends of the range, such as 01h and 0Ch. As depicted in Appendix B, values produced from 2d8z mode are much more repetitive, though they do seem to be skewed a bit toward the lower end of the range. This is likely a result of the imperfection of the implementation of randomness in the DRS, but is likely sufficient for the purposes of playing BnB.

Despite the DRS performing all of the required tasks, there are a few details which I would have loved to modify or include if I had the ability to include more instructions. First, I really would have liked to implement a more interesting version of razzle_dazzle. Ideally, the program would clear the previous output once the next roll started and then perform the aforementioned sequence of illumination with checks after every step for maximum responsiveness. While the current implementation is sufficient, it is mostly unremarkable, but that was a choice that I had to make. Also, for some reason, whenever I test the program on the microcontrollerjs.com simulator in 2d8z mode, the page creates a pop up warning about the program counter wrapping. Given that the last instruction in the 2d8z subroutine is in the last location of program memory, my hypothesis is that this warning was programmed to appear whenever the last instruction in program memory is executed. After dismissing the warning, the program continues to run as expected and produces and stores all of the correct outputs, but it is still quite bothersome that it happens. I spent some time trying to take just a singular additional instruction out of the program, but I could not readily find a way without a significant refactoring of the program, which was simply not an option.

CONCLUSION

As someone interested in cybersecurity, being able to read and write programs in assembly is a more relevant and valuable skill, as compared to those primarily interested in software development. While this lab was obviously quite helpful in developing skills unique to understanding assembly language

as a general concept, there were also two other takeaways from the project which I felt were of particular significance. The first was that, relative to more verbose programming styles, keeping a program concise- particularly when programming in an assembly language- requires a much more thorough understanding of what is happening beneath the surface of the code. The project not only improved my conceptual and practical grasp on PIC12F508 assembly, but also taught me a lot about being a more efficient programmer and understand when, where, and how to revise a program to eliminate unnecessary instructions or variables. With the exception of a few small changes, my initial 90-ish instruction revision would have accomplished the exact same task as my final revision; however, it would have taken more time to run and used more system resources than my final revision.

The second, and arguably more important takeaway from completing the lab was becoming more familiar with designing a program whose behavior has been defined only at a relatively high level of abstraction. While it was not a new concept, it was quite helpful to gain more experience making decisions about the functional aspects of a program on a granular level, and practice defending and justifying those decisions. Though certainly frustrating at times, the constraints surrounding the creation of the program (time management, maximum number of instructions, etc.) served as a relatively effective simulation of an environment one might expect to experience as a software engineer in industry. Though it can be more work and effort to have to devise and defend certain aspects of a program, I happen to also perceive it as a creative outlet and form of expression.

Throughout the process of completing this lab assignment, I have no doubt that I have become a more effective and well-rounded computer scientist. I improved my ability to read and write programs in assembly; my capacity to write shorter, more efficient programs; and gained experience making decisions in software development and justifying them. All of these skills are extremely relevant to my future professional goals and aspirations, and I hope to continue improving upon them in future projects.

APPENDIX A: Buffer Screenshots 1d16z

0Fh		00h
10h	circbuf0	08h
11h	circbuf1	04h
12h	circbuf2	02h
13h	circbuf3	01h
14h	circbuf4	08h
15h	circbuf5	0Ch
16h	circbuf6	0Eh
17h	circbuf7	07h
18h	circbuf8	03h
19h	circbuf9	01h
1Ah	circbuf10	08h
1Bh	circbuf11	04h
1Ch	circbuf12	02h
1Dh	circbuf13	09h
1Eh	circbuf14	04h
1Fh	circbuf15	0Ah

Figure 1: Screenshot of 1d16z full circular buffer in Data Memory in Microcontrollerjs simulator

0Eh		00h
0Fh		00h
10h	circbuf0	0Dh
11h	circbuf1	04h
12h	circbuf2	02h
13h	circbuf3	01h
14h	circbuf4	08h
15h	circbuf5	0Ch
16h	circbuf6	0Eh
17h	circbuf7	07h
18h	circbuf8	03h
19h	circbuf9	01h
1Ah	circbuf10	08h
1Bh	circbuf11	04h
1Ch	circbuf12	02h
1Dh	circbuf13	09h
1Eh	circbuf14	04h
1Fh	circbuf15	0Ah

Figure 2: Screenshot of 1d16z circular buffer after buffer pointer has wrapped back to the initial memory address

APPENDIX B: Buffer Screenshots 2d8z

0Fh		00h
10h	circbuf0	04h
11h	circbuf1	03h
12h	circbuf2	04h
13h	circbuf3	0Dh
14h	circbuf4	04h
15h	circbuf5	04h
16h	circbuf6	03h
17h	circbuf7	06h
18h	circbuf8	0Bh
19h	circbuf9	0Ah
1Ah	circbuf10	01h
1Bh	circbuf11	00h
1Ch	circbuf12	04h
1Dh	circbuf13	09h
1Eh	circbuf14	05h
1Fh	circbuf15	03h

Figure 3: Screenshot of 2d8z full circular buffer in Data Memory in Microcontrollerjs simulator

0Fh		00h
10h	circbuf0	06h
11h	circbuf1	03h
12h	circbuf2	04h
13h	circbuf3	0Dh
14h	circbuf4	04h
15h	circbuf5	04h
16h	circbuf6	03h
17h	circbuf7	06h
18h	circbuf8	0Bh
19h	circbuf9	0Ah
1Ah	circbuf10	01h
1Bh	circbuf11	00h
1Ch	circbuf12	04h
1Dh	circbuf13	09h
1Eh	circbuf14	05h
1Fh	circbuf15	03h

Figure 4: Screenshot of 2d8z circular buffer after buffer pointer has wrapped back to the initial memory address

APPENDIX C: Source Code

```
// Program implements a BnB dice roll of either two 8-sided dice,
// or one 16-sided die. The outcome of the roll is displayed using
// LEDs on GP0/1/2/5. If an LED is on, then the bit represented by
// that GPIO pin is a 1, and if the LED is off, the bit is a 0.
// The value of the roll can be computed using the formula:
// roll = 1*GP0 + 2*GP1 + 4*GP2 + 8*GP5


// Define standard constants
$w     0
$f     1

// Reserve register to store mode
@mode  0x08

// Reserve registers to store roll outcome
@roll_num      0x09 // stores the value to be output from either mode

// Reserve register to store output of lfsr
@lfsr 0x0A

// Reserve registers to store circular buffer
@circbuf0      0x10
@circbuf1      0x11
@circbuf2      0x12
@circbuf3      0x13
@circbuf4      0x14
@circbuf5      0x15
@circbuf6      0x16
@circbuf7      0x17
@circbuf8      0x18
@circbuf9      0x19
@circbuf10     0x1A
@circbuf11     0x1B
@circbuf12     0x1C
@circbuf13     0x1D
@circbuf14     0x1E
@circbuf15     0x1F

:origin
       // Set GP0/1/2/5 to output, GP3/4 input
       movlw  00011000b
       tris   6

       // Set starting address for FSR/INDF
       movlw  0x10 // first buffer address
       movwf  FSR

       // Load seed value into lfsr
       movwf  lfsr // any non-zero value will work (0x10 in this case)



:start
       btfsc  GPIO, GP3      // clear=0->start->break waiting loop
       goto   start          // repeat loop until start is pressed

       // first thing to do once button pressed is read mode
```

```
movlw  0              // 0 = two, 8-sided dice
btfsc  GPIO, GP4      // clear = mode 0 = switch closed
movlw  1              // 1 = one, 16-sided die
movwf  mode   // store initial mode selection (ignore later changes)
call   razzle_dazzle //-----------------------
call   lfsr_galois

// Determines mode and completes appropriate roll outcome computation
// Will always compute 1d16z case because that requires fewer instructions
// than choosing the correct one to go to
call   get_num_1d16z  // if mode=1, we have one 16-sided die
movf   mode,f                // trigger zero flag
btfsc  STATUS,Z              // if mode=0, the zero flag will be set
call   get_num_2d8z   // if mode=1, Z is clear, this will be skipped
                                          // and 1d16z num will be kept, else get
                                          // 2d8z num


// Store output in circular buffer
// Copy current roll value into address of current buffer pointer
movf   roll_num,w
movwf  INDF

// increment buffer pointer (implement circularity)
incf   FSR,f  // move buffer pointer to next position
movlw  0x0F
andwf  FSR,f  // mask everything except bottom 4 bits (0-15)
movlw  0x10
iorwf  FSR,f  // effectively adds 16 (0001 0000), which
                            // maintains range of 0x10 thru 0x1F


// Set GPIO LEDs to display output
// pin(bit): GP0(0), GP1(1), GP2(2), GP5(3)
// GP0 = 1's, GP1 = 2's, GP2 = 4's, GP5 = 8's
// GPIO: bit set = off, bit clear = on
// zero flag: bit set = 0, bit clear = nonzero
// AND is used in output to mask every bit other
// than the single bit of interest

// Start by setting lights off (bit = 0), turn on if bit is 1
bcf    GPIO,GP0       // turn off light = roll bit 0 = GPIO bit clear
bcf    GPIO,GP1
bcf    GPIO,GP2
bcf    GPIO,GP5

// Rotates rightmost bit into carry each time
// Ruins roll_num during process, which is why it was
// stored prior to output

// get 1's bit
rrf    roll_num,f
btfsc  STATUS,C
bsf    GPIO,GP0       // turn on light = roll bit 1 = bit set

// get 2's bit
rrf    roll_num,f
btfsc  STATUS,C
bsf    GPIO,GP1

// get 4's bit
rrf    roll_num,f
btfsc  STATUS,C
bsf    GPIO,GP2
```

```
        // get 8's bit
        rrf     roll_num,f
        btfsc   STATUS,C
        bsf     GPIO,GP5

        goto    start // roll again!

//---------SUBROUTINES BELOW------------

:razzle_dazzle
        // Give 'em the ol' razzle dazzle
        // as long as button still pressed
        // 0 = on = clear| 1 = off = set
        btfsc   GPIO, GP3      // check if button released
        //-------------------------------------------------
        retlw   0       // if it is released, generate random number
        bcf     GPIO, 5        // light off
        bsf     GPIO, 5        // light on
        goto    razzle_dazzle


//————————————————————————————————————————————————————————————————-
Title: lfsr_galois subroutine source code
Author: Dr. Daryl Posnett
Date: Wednesday, 8 February 2023
Availability:
https://canvas.ucdavis.edu/courses/765142/assignments/1012635?module_item_id=1484419
//————————————————————————————————————————————————————————————————-


// lsfr_galois implements an 8-bit linear feedback shift register
// that can be used to generate a pseudo random binary number sequence
// The output sequence can be viewed in the register lfsr.

:lfsr_galois
        // First clear the carry so that we know that it is zero. Now, shift the lfsr right
        // putting the LSB in carry and copying the lfsr into the working register for more
        // processing

        bcf     STATUS,C
        rrf     lfsr, w

        // If the carry (LSB) is set we want to
        // xor the lfsr with our xor taps 8,6,5,4 (0,2,3,4)

        btfsc   STATUS, C
        xorlw   10111000b
        movwf   lfsr // save the new lfsr
        retlw 0


// Generate single value from lfsr by taking the bottom 4 bits (0-15)
:get_num_1d16z
                movlw   0x0F
                andwf   lfsr,w
                movwf   roll_num
                retlw   0



// Generate 2 values from 2 lfsr values by taking bottom 3 bits
// from current lfsr, then generate another lfsr and repeat, then add them
:get_num_2d8z
        // get outcome for 1st die
```

```
// (stored in roll_num to reduce instruction count)
movlw  0x07
andwf  lfsr,w
movwf  roll_num

// generate 2nd random number
call lfsr_galois

// get outcome for 2nd die
movlw  0x07
andwf  lfsr,w

// Sum the two dice and store it
// (outcome from 2nd die is still in W)
addwf  roll_num,f
retlw  0
```

APPENDIX D: Works Cited

[1] Posnett, D (2023) lfsr_galois subroutine code [Source code] https://canvas.ucdavis.edu/courses/765142/assignments/1012635?module_item_id=1484419