Lab 1 Writeup

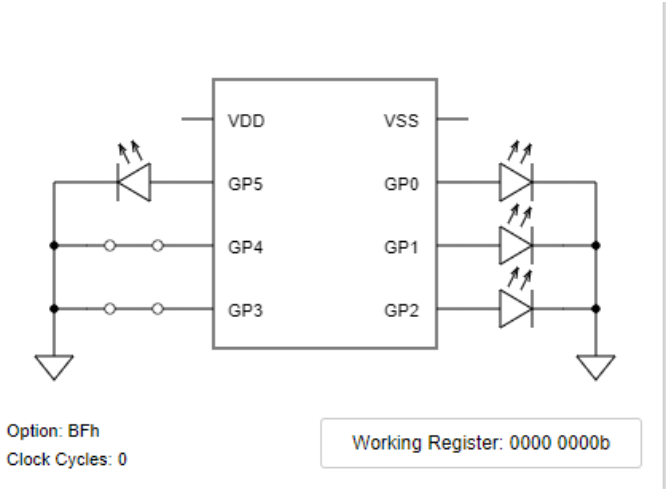
Orry Butler
Student ID: 919288162
University of California, Davis

Introduction

The purpose of this lab is to write a program in PIC12F508 assembly, which uses two inputs and four outputs in order to perform random dice rolls and then display the values of the dice rolls to the four outputs' pins. It then moves the values of each role into a size-sixteen circular buffer, which will save the last sixteen roles. We do this by first setting the mode of the role. We can choose between two modes: set and clear. The set mode rolls a dice that has sixteen sides, numbered 0 to 16. The clear mode rolls two dice, both with eight sides numbered 0 to 7, which will then be added - giving us a roll value from zero to fourteen. We then perform bit masking, which will be discussed later, in order to turn an 8-bit number into one 4-bit number. After performing bit masking, the value will be displayed to the four output pins, showing the 4-bit binary number displayed across the four pins and then added to the circular buffer. This lab writeup will discuss how we implemented all of these methods and the testing that went into it.

Theory of Operation

In this program, we use GP4 and GP3 as inputs and GPO, GP1, GP2, and GP5 as outputs that show as lights displaying the binary value of each of the rolls. GP3 is used to start and end rolls, while GP4 is used to select the mode for each set of rolls and is not a factor until rolls are rest. The first step we take in this program is to set up the input and output pins. We can do this by loading the value 00011000b into the working register, with zeros representing outputs and ones representing inputs. Then tris these values to set up our inputs and outputs.



We then go into the mode choice function which checks GP4. If GP4 is pressed, it is clear and we go to out function for the clear mode choice and load an even value into the variable mode. If GP4 is not pressed, we go to our function for mode choice set, which loads a value of one into the mode variable. This then takes us to check_start, which looks at the GP3 input. When GP3 is pressed, it begins a roll. But, when GP3 is not pressed, the function loops until GP3 is finally pressed and we can start the roll. We then can start the roll, which uses an 8-bit linear feedback shift register provided in the lab by Dr. Posnett. This generates our pseudo random numbers for each role that is performed and stores the value into a variable LFSR (Posnett). After the random number is generated, we branch to the continue function, which then once again checks GP3. If

GP3 is pressed, it continues rolling the dice. If GP3 is not pressed, it checks the mode choice by loading the mode into the working register and performing the bitwise operation andlw on the mode value. Since the mode is even for clear and 1 for set, when we andlw the mode the STATUS, Z register will be set for mode clear and we can perform bit masking for two dice rolls. However, if the mode is set, the operation will not return zero and thus STATUS, Z is not set and we can perform bit masking for one size-sixteen die roll.

We will first discuss the set bit masking, as it is much less complicated than the clear bit masking. The idea of bit masking here is to get rid of the first four bits of our binary number. We can do this by taking the value that is stored in the lfsr variable and using the command andlw 1111b to get a number that has a value (0 to 15) in order to match the die being used. This will return only the values where lfsr has ones in the first four bits, and then we can store this value back into the lfsr variable.

Now we will discuss clear bitmasking. For the clear mode, we know that there are two dice rolls, both of size eight with numbers zero through seven printed on the sides. So, we first need to mask the first three bits of the lfsr, using a similar method to the set bit masking but now andlw 111b to get a number (0 to 7) to match the sides of the die and store this value to a new variable dice_one. We then have to mask another three bits – I chose to do andlw 00111000b – in order to mask bits 4 through 6 so we have a completely new set of data from the first mask. As you can see from this, this operation will not give us a number from 0 to 7, so we have to rotate the bit right using the rrf command on the lfsr. If we rotate right three times, we end up with a bit that represents a number (0 to 7) in the first three bits, but will still have higher values on the left side of these bits due to the rrf function rotating into carry. We can then perform masking again by using the command andlw 111b, which will keep only the first three bits of the lfsr and we can add this number to the value stored in dice_one to get our final value for lfsr of the two dice when mode is set to clear.

We will now discuss the part of the project which displays the data. The data display is fairly simple, except for GP5. GP5 is more difficult than the other output pins because it corresponds to the 5-bit, which is not included in the 4-bit binary number. So, in order to output the first 3 bits of lfsr, we just move the lfsr into the GPIO register and it will output whatever is in the first three bits. Then, to output the fifth bit, we must andlw 1111b lfsr. If there is a 1 in the 3-bit position, GP5 will light up. If there is not a 1 in the 3-bit position, GP5 will not light up.

Last, we completed storing into the circular buffer, in which I used code for storing values into FSR and INDF from the stack example from microcontrollerjs.com as an example in order to make my circular buffer. This takes a buffer pointer, which stores a pointer into the FSR, and then stores the value of lfsr into INDF associated with that FSR and increments the value of buffer_pointer on each run unless equal to 15 (Whiteley). In order to check if the value of the buffer pointer is equal to 15, we perform the operation xorlw 1111b on the value of the buffer pointer. When the buffer pointer equals 15, this bitwise operation should return 0 setting the Z flag. Also, when the buffer pointer is 15, we assign a value of 0 to the buffer pointer and the buffer starts again. The program will then be looped through with each of these operations until

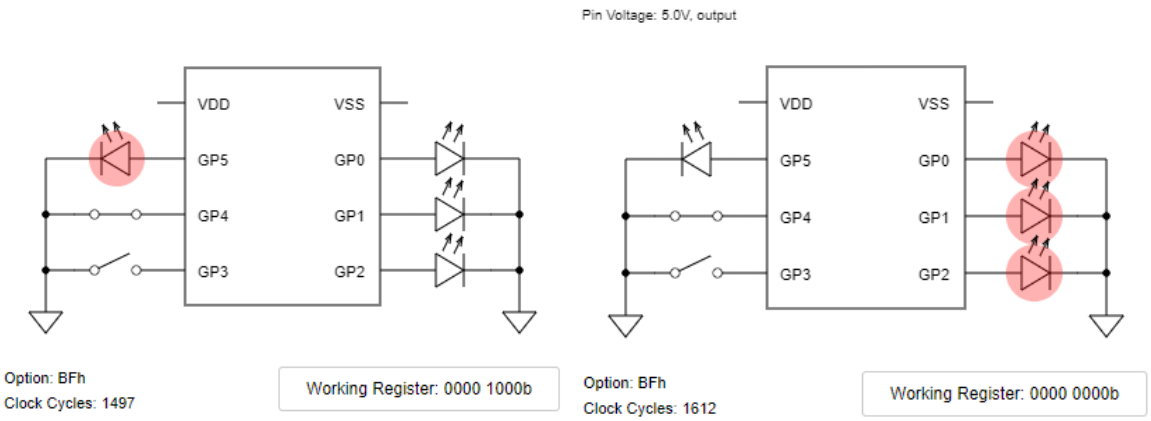
the game is reset.

Discussion

There was quite a bit of testing done to, and some tricky parts experienced in, this program. The first part of testing that I would like to discuss is on the buffer. The buffer part of the code was fairly simple, but I ran into some issues while writing it. In my initial testing of the code, I could never get the final value of the buffer to fill. However, after some debugging, I saw that I was checking the size of the buffer before I was putting values into the buffer–so I would always be skipping the last part of the buffer and going back to 0. After fixing this issue, the buffer now fills the last value, and I have not had any issues with the buffer fully filling or wrapping back around to the start of the buffer. Below is an image of the buffer fully filled.

0Eh	buffer_ptr	00h
0Fh	buffer_size	00h
10h	buffer_start	09h
11h		0Bh
12h		0Ch
13h		06h
14h		03h
15h		08h
16h		02h
17h		07h
18h		09h
19h		0Bh
1Ah		09h
1Bh		0Bh
1Ch		0Ah
1Dh		05h
1Eh		08h
1Fh	buffer_end	01h

Another part of the program in which a lot of the testing went into was displaying the values to the output pins. At first there were a significant amount of bugs in outputting due to a bad understanding of how GPIO worked. After I figured it out, I learned that GP5 is associated with the 5-bit, but since there was no 5-bit in the lfsr it would not output the 3-bit to lsfr. So, after a fix, it finally started to output lsfr to the four output terminals. Shown below are several outputs of the value of lsfr



The main bit of testing I had to conduct was on the clear bit masking function. On several occasions, I ran into issues of the code either not shifting bits or making numbers that were too large. For this function, I had to do debugging and perform the bit operations by hand for each one, and then cross reference with the values that my code was getting, and finally got to a point where the bit masking would output the correct values for the clear mode every time. This proved to be the hardest part of this program because of the difficulty in debugging it with little to no output to reference.

Conclusion

In conclusion, I implemented a program which had two inputs and four outputs in the PIC12F508 assembly language. This program used two modes for selecting die rolls: one choice was a single die roll of a sixteen-sided die including zero, and the other was rolling two die with eight sides including zero. We did this by choosing 8-bit pseudo random numbers and performing bit masking on them to turn them into 4- and 3-bit numbers, respectively. We used GPIO in order to output these numbers to light on our four output pins. Then, we were able to store each of these values into a circular buffer, which has a size of sixteen and continues storing the last sixteen roles until the game is reset. This project taught me a significant amount about PIC12F508 assembly and how to apply things, such as the STATUS register, and how to work with the size of the call stack and program memory.

Appendix

Code

```
// Basements and Byters
#GP0   led
#GP1   led
#GP2   led
#GP5   led

#GP4   switch
#GP3   switch

// Constants
$w      0
$f      1
$RESULT_W  0
$RESULT_F   1
//variables
@lfsr 0x0A
@mode 0x0B
@buffer_ptr 0x0E
@buffer_start 0x10
@buffer_end 0x1F

:origin
        // Setup GP0,GP1,GP2,GP5 as output, GP3,GP4 as input
        movlw 00011000b
```

```
        tris    6
        //move literal into lfsr for 8-bit shift register
        movlw 0x78
        movwf lfsr
//Check GP4 for mode choice clear go choice_clear
//set go choice_set
:mode_choice
        btfss GPIO, GP4
        goto choice_clear
        goto choice_set
@dice_one 0x0C
//set mode equal to 1 for set
:choice_set
        movlw 1
        movwf mode
        goto check_start
//set mode equal to even working register for clear
:choice_clear
        movwf mode
//check if GP3 set if clear start roll if not don't
:check_start
        btfss GPIO, GP3
        goto loop
        goto check_start
//Check if GP3 is set if clear continue loop if not
//Go to bit_masking for mode choice
:continue
        btfss GPIO, GP3
        goto loop
        movf mode, w
        andlw 0001b
        btfss STATUS,Z
        goto bit_masking_set
        goto bit_masking_clear
//code used from Dr. Posnett on canvas 8-bit linear feedback shift register(Posnett)
//https://canvas.ucdavis.edu/courses/765142/assignments/1012635?module_item_id=1484419
:loop
        call lfsr_galois
        goto continue
: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, RESULT_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
   //fun dispaly during the roll
        movwf GPIO
        movwf lfsr // save the new lfsr
        retlw 1
//bit masking for 2 die rolls
:bit_masking_clear
   //mask the first roll
        movf lfsr, w
        andlw 111b
        movwf dice_one
   //mask the second roll
        movf lfsr, w
        andlw 00111000b
        movwf lfsr
        rrf lfsr, f
        rrf lfsr, f
        rrf lfsr, f
        movf lfsr, w
        andlw 111b
   //add up the two rolls
        addwf dice_one, w
        movwf lfsr
        goto display_number
//bit masking for the set mode choice
:bit_masking_set
   //set lsfr to the first 4 bits of original random number
        movf lfsr, w
        andlw 1111b
        movwf lfsr
:display_number
   //display first 3 bits
        movwf GPIO
   //display the fourth bit
        andlw 1000b
        btfss STATUS, Z
        bsf GPIO, GP5
//referenced microntrollerjs.com while writing this segment Indirect memory
addressing(Whiteley)
//http://microcontrollerjs.com/sim/microcontroller.html?file=stack.asm
```

```
:put_buffer
   //store in each ptr location in the buffer
         movlw buffer_start
         addwf buffer_ptr, w
         movwf FSR
         movf lfsr, w
         movwf INDF
   //check the size if 15 reset the buffer ptr if not increment
   //the buffer_ptr by 1
         movf buffer_ptr, w
         xorlw 1111b
         btfsc STATUS,Z
         movwf buffer_ptr
         btfss STATUS, Z
         incf buffer_ptr, f
         goto check_start
```

Works Cited

Posnett, Daryl. *8-Bit Linear Feedback Shift Register*. 2023. https://canvas.ucdavis.edu/courses/765142/assignments/1012635?module_item_id=1484419

Whiteley, John. *Indirect Memory Addressing*. http://microcontrollerjs.com/sim/microcontroller.html?file=stack.asm