Robert Kulesza
November 27, 2018
Professor Erkan
Computer Organization & Architecture

Project Two:
A Simple Guessing Game

The project “A Simple Guessing Game” required us to write a simple LC-3 program that included basic input, basic output, and basic error checking. The program is required to store the value 6, which would be the solution to the “Guessing Game.” The program is required to ask the user to input a number between 0 and 9, inclusive. After the user enters a number into the keyboard, the program compares it to the solution (6) and outputs “Too Big”, “Too Small”, or “Correct! You took # guesses” depending on the result of the comparison. If the inputted guess was not an ascii digit then the program must output “invalid input” instead. (Note that # should be replaced with the actual number of guesses the user made.) Also, after 9 guess the program is required to output “Game Over. Correct answer is 6” and then halt. The instructions necessitated that we use TRAP x20 (getc) to read a character from the keyboard and suggested the use of the pseudo-op .STRINGZ to store strings in the program.

I increment a counter variable to keep track of how many guesses are made. When that counter is set to 9, the program outputs a “game over” message and halts. Then I prompt the user for input (the user’s guess). I check to make sure that the guess >= 30 and that the guess <= 39 (ascii values for 0 and 9). If the guess is not within this range, I print out “invalid input” and move to the next iteration of the loop. Then I check to see if the guess <, =, or > 6 by comparing it to 36 (ascii value for 6). If the guess < 6 then I print out “Too Low” and move to the next iteration of the loop. If the guess > 6 then I print out “Too High” and move to the next iteration of the loop. If the guess = 6 then I print out “Correct! It only took you x guesses!” where x = counter . Since the guess was correct and the game is over, the program halts.

```
.orig x3000

;load 30 into COUNTER (counter is in ascii)
                AND R1 R1 X0
                ADD R1 R1 XF
                ADD R1 R1 XF
                ADD R1 R1 XF
                ADD R1 R1 X3
                ST R1 COUNT

;Ask user to guess number n, 0 <= n <= 9
MAIN            LEA R0 PROMPT       ;ready prompt string to be printed
                PUTS                ;print prompt string

;User inputs their guess
                GETC                ;read user input into R0
                LD R1 COUNT         ;load counter into R1 (counts guesses)
                ADD R1 R1 X1        ;Increment counter (a guess was made)
                LD R2 GUESS_LIMIT;load -ASCII(9) into R2 (9 = max # guesses allowed)
                ADD R2 R2 R1        ;counter - max # guesses should be < 0
                BRz NINE_GUESSES;if counter - max # guesses == 0 then game over
```

```
                ST R1 COUNT         ;since counter < max # guesses, save counter

;if guess is not ascii between 30 and 39
;ASCII: if guess < ASCII_LOW then output “invalid input”
                LD R3 ASCII_LOW     ;load -30 into R3 (Ascii(0) = 30)
                ADD R3 R3 R0        ;guess - 30 >= 0
                BRn INVALID_INPUT ;skip to where we output “invalid input”
                LD R3 ASCII_HIGH    ;load -39 into R3 (Ascii(9) = 39)
                ADD R3 R3 R0        ;guess - 39 <= 0
                BRp INVALID_INPUT ;skip to where we output “invalid input”

;if guess > 6 then output “too big”
                LD R3 NSIX
                ADD R0 R0 R3        ;guess - 6
                BRp TOO_BIG         ;if guess - 6 > 0 then output “too big”

;if guess < 6 then output “too small”
                BRn TOO_SMALL       ;if guess - 6 < 0 then output “too small”

;if guess == 6 then correct
                LEA R0 CORR1        ;print correct message
                PUTS
                LD R0 COUNT
                OUT
                LEA R0 CORR2
                PUTS
                HALT                ;correct message printed so end program
TOO_SMALL  LEA R0 TOO_SMALL_STR ;print “too small”
                PUTS
                BR MAIN             ;next iteration of loop
TOO_BIG       LEA R0 TOO_BIG_STR ;print “too big”
                PUTS
                BR MAIN             ;next iteration of loop
INVALID_INPUT LEA R0 INV_INPUT_STR ;print “invalid input”
                PUTS
                BR MAIN

;After 9 guesses output “game over, correct answer is 6”
NINE_GUESSES LEA R0 GS_LMT_STR ;print message
                PUTS
                HALT                ;game over so halt game


;Constants – remember PUTS to output strings
```

```
NSIX            .FILL x-36                          ;stored value -36 (check usr guess)
GUESS_LIMIT     .FILL x-39                          ;num guesses to lose (neg/ascii)
ASCII_LOW       .FILL x-30                          ;ascii code for 0
ASCII_HIGH      .FILL x-39                          ;ascii code for 9
INV_INPUT_STR   .STRINGZ ”Invalid Input\n”          ;invalid input str
TOO_BIG_STR     .STRINGZ “Too Big\n”                ;too big – wrong guess msg
TOO_SMALL_STR   .STRINGZ ”Too Small\n”              ;too small – wrong guess msg
CORR1           .STRINGZ ”Correct! You took “       ;correct guess message p1
CORR2           .STRINGZ ” guesses!\n”              ;correct guess message p2
PROMPT          .STRINGZ ”Guess a number between 0 and 9:\n “
GS_LMT_STR      .STRINGZ ”Game Over. Correct Answer is 6\n”     ;guess limit breached
message

;variables
COUNT           .BLKW x1                            ;guess counter


.end
```

Hence, the project was to create a simple guessing game with a fixed solution. My solution follows all guidelines and instructions. It works properly with no bugs. There are improvements I could make - for example, I store 30 in count in the beginning in a very sloppy way. I could change that in a number of ways. Firstly, instead of storing and loading from a memory address, holding the value of count, I could increment count in a register I reserve for it. That way I never have to change the value at the memory address of count and can set the value using the pseudo-op .FILL. That would make the program run more efficiently because it would not be loading/storing from memory nearly as much which takes more time than dealing with registers. That said, it wouldn’t change the overhead because we still need the memory to store x30 to initiate count. In any case, a simple guessing