

# Programming Lab 5: ID Card Data Parser

EE 306: Introduction to Computing

Professor: Dr. Al Cuevas

TAs: Apurv Narkhede, Nic Key, Ramya Raj, Jerry Yang

**Due: 12/10/2018 at 12pm**

## 1 Overview

This lab is intended to familiarize you with the stack and interrupts. By the end of this lab, you should be able to:

1. write and debug the lines of assembly code you end up writing.
2. describe how an interrupt works and why interrupts are useful.
3. configure an interrupt in LC3.

Late submissions will not be accepted; grades will be published within 24hrs of the due date. There are no regrade requests for this lab.

*Note:* You may NOT show anyone other than the TAs or Dr. Cuevas your code.

**Any violation of this rule constitutes academic dishonesty.**

## 2 Background

One of the neat little things about your upgraded ID card is that it can get you into EER and ECJ after the doors lock at night. As an engineering student, you've probably used this power many times already, and if you haven't figured out you could do this, you aren't a real engineering student yet.

To you, the user, all you do is wave your wallet in front of the little scanner, and it beeps and unlocks the door for you. How does it do that? In fact, your upgraded ID card has a low-frequency RFID chip inside it such that when the scanner sends out a small signal, it activates the chip in your card, which uses the same signal to transmit your information (EID, student identification number, etc.) to the card reader. The card reader then parses this information and determines if you can enter or not.

(Fun fact: if you swipe your card on a card reader that connects to your computer, you can see what data is actually stored on your ID card. I discovered this myself last year...very interesting to swipe my credit cards too.)

In this lab, you will be designing a parser in LC3 to aid the card reader in parsing incoming data (in the form of a string) that it has collected from a read card. As each character of the string is received, the card reader verifies that the character is a valid ASCII character before passing it into the LC3 console (read: KBDR). Since there is no way to know when a card will be read, or how long the check of each character will take, the best approach to solving this problem is through implementing an interrupt-based solution.

## 2.1 Interrupts

Interrupts are useful when you want your code to do  $X$  when a stimulus  $Y$  happens but you don't know when  $Y$  will happen. In your phones, interrupts occur for every process that you ask it to run. For example, an interrupt is triggered when you push the home button on your smartphone, or when you open a new app. When you are not using your phone, your phone is running the "WFI - Wait For Interrupt" command, waiting for you, the user, to tell it to do something. Since the phone cannot know when you want something done, it uses interrupts to control its functions. Different kinds of interrupts exist, but most externally triggered interrupts are configured to interpret unpredictable events in the outside world.

An interrupt service routine (ISR) is the user-defined subroutine that is called/executed when an interrupt is triggered. Unlike a subroutine, there is no way to know when the ISR will be called in your code, and there is no way to restore the machine state when an interrupt is triggered. This presents a problem: how can we save and restore the machine state (PC and PSR - processor status register, which contain your condition codes and whether the machine is on or not) before and after the ISR is called?

Enter the stack. The stack is accessible from everywhere in memory as long as we know where it is - and we know where it is because it is at a fixed location we set in the main program. Most interrupt-based systems therefore use a two-step process for calling an interrupt:

1. When the interrupt is triggered, push the machine state onto the stack (usually PC, PSR, and a couple of other stuff in more complex systems).
2. Execute the ISR.
3. When the ISR is done, pop the machine state off the stack and restore all state registers back to their original values pre-ISR.

This process allows the machine state to be completely preserved at any point in time when an interrupt is called.

Another problem also arises in interrupt-based systems. If we have multiple interrupts, how can we know which one occurred? To solve this problem, we rely on the Interrupt Vector Table (IVT). The IVT works in the same way that the TRAP vector table in LC3 works: when a specific interrupt is called, the computer will decode the interrupt, look for the corresponding entry in the IVT, and branch to the location given by that entry. For example, in LC3, when a keyboard interrupt is triggered, LC3 goes to location x0180 - the particular vector number assigned to keyboard interrupts - in its IVT and uses the contents of that memory location as the location of the ISR. It then branches to the ISR. Note that this implies that ISRs can be stored anywhere in memory, even adjacent to your main program.

## 2.2 LC3 Interrupts

The LC3 only has one kind of interrupt: the keyboard interrupt. When enabled, every keypress in the console window will trigger an interrupt in LC3. There are two main things you need to know in terms of setting up an LC3 interrupt: initialization and ISR structure.

### 2.2.1 LC3 Interrupt Initialization

To set up (aka “initialize”) the keyboard interrupt, you need to do the following in your main program:

1. Set R6 to be the stack pointer. We usually call R6 the SP after this.
2. Store the location of your ISR into the IVT. Since the keyboard interrupt vector is x0180, you will store the location of your ISR into x0180.
3. Set the interrupt-enable (INTEN) bit in the KBSR. (This is KBSR[14].)

Below is an example of an interrupt initialization routine in which the stack begins at x3000 and the ISR begins at x1500.

```
; set up ISR
LD  R6,SP      ; 1. Set stack to x3000
LD  R1,KBISR   ; 2. Load location of ISR into R1
LD  R2,KBISRU  ; Load keyboard IUT location into R2
STR R1,R2,#0   ; Store location of ISR into IUT
LD  R1,INTEN   ; 3. Load bitmask for interrupt-enable bit, KBSR[14]
LD  R2,KBSR    ; Load address of KBSR
STR R1,R2,#0   ; Set KBSR[14] (technically we would OR it with the contents
                ; of KBSR to be "friendly" but you'll learn all about that in 319k)

; other main program stuff
; don't forget the bonus opportunity in the submission section!

HALT

SP .FILL x3000
KBISR .FILL x1500
KBISRU .FILL x180
KBSR .FILL xFE00
INTEN .FILL x4000
```

### 2.2.2 ISR Structure

When you write your ISR, you will write it in a separate file with a separate .ORIG, .END, and separate code. This is to ensure that your main program and your ISR do not overwrite each other when they are loaded into LC3. Furthermore, it is generally impossible for you to place your ISR away from your main program yet have both in the same file (it is doable in LC3, but not in practice).

Below is the structure for your ISR file. Note that the .ORIG value must match what you store in the IVT, or you'll get unpredictable behavior. Also note the RTI instruction at the end of the ISR. The RTI instruction restores the machine state and branches back to the main program for you so you don't have to.

**You cannot use TRAPs in your ISR. Doing so may cause unpredictable behavior.**

```
.ORIG x2500 ; where the ISR will be located in memory and what goes into your IUT

; ISR code

RTI ; Return from interrupt instruction - manages the stack for you

; ISR data fields

.END
```

### 3 Lab Specifications

In this lab, you will design a data parser in LC3 to help the ID card reader parse incoming data that it has collected from a read card and determine if the card holder can enter the building. To simulate the card reader feeding data into your program, you will be testing your parser by entering characters into the console and observing the output. Each read results in a string of 16 characters that is fed in, character by character, to the console.

Your program's job is to determine if the read card contains one of four access codes: 16100, 16105, 16110, and 16115. These access codes indicate that the card holder is a valid person and can enter the building. If a string containing one of these access codes is input to the console, you should print "Access Granted!" to the console on a new line and disregard all characters in the string that follow the access code (do not display them either) until the next card read begins. If the string does not contain one of these access codes, you should print "Access Denied!" and wait for the next card read. The access codes can appear anywhere in the string.

#### 3.1 Interrupt Service Routine (ISR)

Your ISR will begin at xA000. It will read the character typed into the console and store it in memory location x8000. It will also count the number of characters that have been input to the console and store the count in memory location x8001. Remember that an ISR is a subroutine, so you must save and restore any register you use. Note that it is not required to save the link register since it is already on the stack. You may not implement any other functionality in the ISR.

**Do not use TRAPs in your ISR. Access KBSR/KBDR manually in your ISR.**

#### 3.2 Main Program

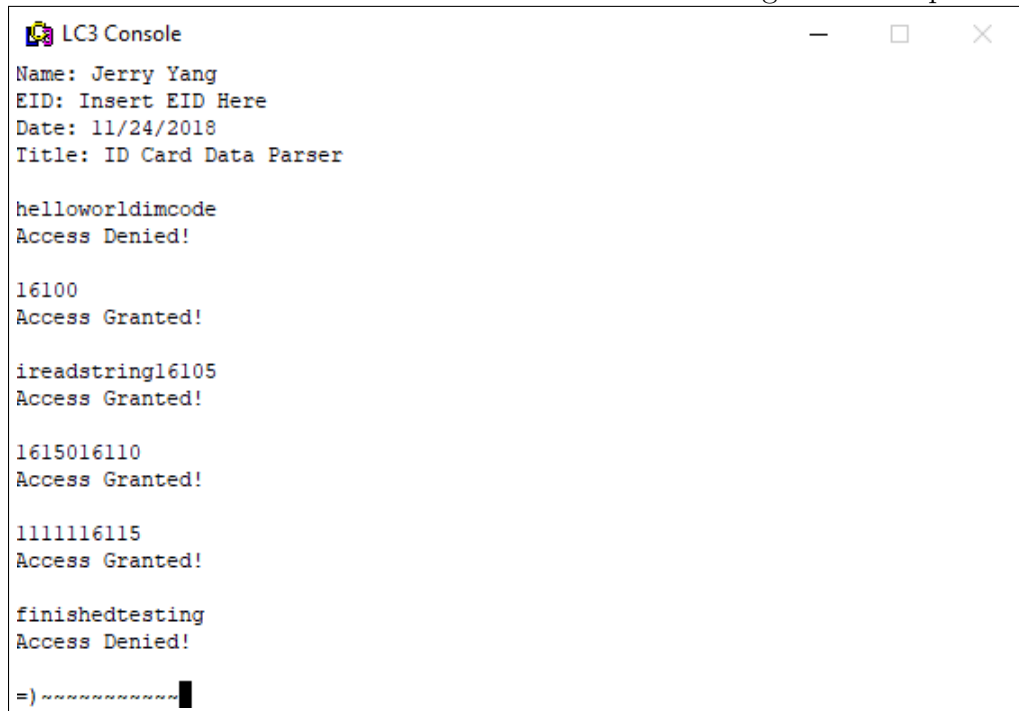
Your main program should implement the functionality described above. In your interrupt initialization routine, set the stack to xF000. Your program should constantly check x8000 to see if a character has been received by the console. Once it receives a character, it will clear x8000 to indicate it has been read. Your main program should also make use of the character count in x8001, clearing it when appropriate. Note that your program should never halt.

One way (and probably the easiest way) to approach this problem is to design and program a finite state machine (FSM), given the character and character count as inputs and console output as output. In your FSM, you should specify what should be done in each state using pseudo-

code, which will make the actual coding much easier. You can (and should) also write your own subroutines as necessary.

## 4 Test Cases

Here is a screenshot of the console that shows a working card data parser.



```
LC3 Console
Name: Jerry Yang
EID: Insert EID Here
Date: 11/24/2018
Title: ID Card Data Parser

helloworldimcode
Access Denied!

16100
Access Granted!

ireadstring16105
Access Granted!

1615016110
Access Granted!

1111116115
Access Granted!

finishedtesting
Access Denied!

=)
```

## 5 Submission

1. Reproduce the test cases as shown above in the console, replacing your name, EID, and date, as appropriate. Screenshot the console, your main program in the editor, and your ISR in the editor, and compile the screenshots into one PDF document. You should have a maximum of 5-7 screenshots, depending on how long your code is (if your code is long, consider cutting down). Bonus points if you screenshot where I got the “access codes” from (shhh don’t tell anyone, they’re fake anyway).
2. Check your syntax to ensure it will work with the autograder. (See autograder syntax document on Canvas.)
3. Rename the Lab5.asm file “EIDLab5.asm”, replacing “EID” with your EID.
4. Rename the isr.asm file “EIDisr.asm”, replacing “EID” with your EID.
5. Name the PDF file “EIDLab5.pdf”, replacing “EID” with your EID.
6. Submit the Lab5.asm, isr.asm, and the .pdf file to the Lab 5 Canvas assignment. Do NOT include any other files, and do NOT zip them up. There is a button in Canvas to add files:

File Upload

Upload a file, or choose a file you've already uploaded.

File:  No file selected.

Comments... ..

7. If you do not follow these submission instructions exactly, the autograder will spit out a 0!