ECE 350L – Spring 2016

Final Project Documentation: Pipelined Processor with A5/1 Cipher Unit

Yi Yan Tay (yt61) and Anthony Yu (amy8), 27 April 2016

Contents

List of Figures

Note to TAs: We updated our project to more fully use our processor after the demo. To differentiate between what we did in the demo and what our project can actually achieve, we have two separate "modes" for our A5/1 unit

1 Introduction

In this project, we implemented a 5 stage pipeline, full-bypass processor with a A5/1 encryption unit.The hardware was implemented on the Altera DE2-115 development board.

In the A5/1 protocol, a 64 bit key is used to generate a pseudorandom keystream that encrypts 128 bits of message data. We used PS/2 as input and a Hitachi HD44780 LCD screen as output. It was verified that encryption and decryption work fully.

1.1 A5/1 Cipher

The A5/1 is a symmetric key, stream cipher which encrypts plaintext using a pseudorandom cipher digit stream (keystream). As opposed to block ciphers, stream ciphers encrypt the plaintext one bit at a time. This makes it particularly useful in real time scenarios, where data has to be encrypted on the fly. In A5/1 the encryption operation is XOR. A5/1 is used for GSM communications, which means that is it used by virtually all cellphones. While it has recently been proven to be cryptographically insecure, its widespread use reflects the fact that the cipher is very fast when implemented in hardware.

A5/1 is based around a combination of three linear feedback shift registers (LFSRs) with irregular clocking. The three shift registers are specified as in **figure 1**. During irregular clocking, registers are only clocked when their clocking bit (orange in figure) is the same as the majority clocking bit. To initialize the A5/1 unit (i.e. get it ready to produce a cryptographically secure keystream), a few steps need to be taken.

1. 64 Cycles of XORing key (64 bits) into 3 LFSRs.
2. 22 Cycles of XORing frame (22 bits) into 3 LFSRs. The frame is a known value for the cipher and we used the standard value of 22'h000134.
3. 100 Cycles of irregular clocking as described above.
4. Output is now valid for 228 cycles (but we use only 128 cycles/bits for demo mode).

2 Design Specifications

2.1 Overview

Our implementation of the A5/1 Cipher takes in a 64-bit key and a 128-bit message through PS/2 keyboard input, displaying them on a 16 x 2 LCD. The key and message are stored in shift registers, then fed into the A5/1 Cipher where an output stream unique to the key is produced and combined with the message to form an encrypted output. This encrypted output is once again displayed on the 16 x 2 LCD. Switches are used to control the keyboard's input target, to initiate the key stream output after the inputs have been loaded, and to reset the cipher. 7 Segment Displays track the character counter for both key and message, while several indicator LEDs show the current states of the encryption.

2.2 Processor Changes

The encryption unit can run in either demo mode or full-processor mode.

2.2.1 Demo Mode

In demo mode, a simple MIPs instruction (`a51start` see ISA Changes Section) activates the A5/1 unit. The unit starts accepting PS/2 input. A hardware switch controls whether you are entering into the message or key register. Another hardware switch controls when the unit starts encrypting. A last hardware switch resets the unit. The A5/1 unit processes PS/2 input independently of the processor and outputs the result on the LCD screen. This mode is the fastest possible implementation of A5/1 encryption and decryption as it is optimized for hardware.

2.2.2 Full-Processor Mode

In Full-Processor mode, the encryption process happens entirely within memory and the processor. A single mips instruction(`a51start`) causes the A5/1 unit to generate a keystream. The programmer can then make use of `loadkeystream` to load the keystream into registers within the regfile. An xor instruction was added to the processor to allow 32 bit XORing of the message. Since messages are 128 bits, this operation has to be done 4 times. For an example of a program that does this, see our attached assembly code.

2.3 I/O

2.3.1 Main I/O Components

We had four I/O components:

1. 7 Segment Display
   We used three of these. Two of them are used to display in hex the character count for the message, which goes up to 32 characters (1a). The last is used to display in hex the character count for the key, which goes up to 16 characters (f). Both counters decrement when backspace is pressed, allowing the user to see when they have reached the maximum key and message length.

2. PS/2
   The provided PS/2 Interface and Controller modules in the skeleton were implemented in this project. The PS/2 keyboard enabled the user to enter a message and key in **Demo Mode** and for the user to enter a key in **Full-Processor Mode**. For reasons discussed in the Challenges section, we decided on allowing HEX input. So only 0-9 and A-F were allowable characters from the keyboard. The keyboard scan codes for these 16 characters were converted into both ASCII codes printable by the LCD, and to the 4-bit binary representation of their hex values for the actual data encryption. We had to implement both a debouncer and several look up tables for this to work properly. We found that implementing the keyboard for data storage was significantly more difficult than implementing it for key detection, as debouncing and extensive conversion was a necessity.

3. LEDs
   We used these to display several status messages: error(LED14), enterToKeyNotData(LED17), startKeyStreamGen(LED16), KeyStreamDepleted(LED15), message_to_LCD_done(LED14).

4. LCD Display
   The provided `LCD.sv` module in the skeleton was implemented in this project. We used the LCD display to display the key, the message and the encrypted message in **Demo Mode**. It is also used to display the key in **Full Processor Mode**. The use of two tristate buffers determines whether the PS/2 input or the encrypted output is being displayed on the LCD.

2.3.2 Key Entering

When the `enterToKeyNotData` hardware switch is set to 1, the PS/2 input is being fed into the keyframe register 4 bits at a time. We index into this register using a counter and decoder. A detailed specification is shown in **figure 2**.

2.3.3 Message Entering

When the hardware switch corresponding to `enterToKeyNotData` is at 0, the PS/2 input is being piped 4 bits at a time into a regular register. We index into this register using a counter and decoder. A detailed specification is shown in **figure 3**.

2.4 ISA Changes

A few instructions were added to our processor to allow A5/1 encryption in full-processor mode.

1. `a51start`
   Opcode 11111. Usage: a51start.
   Required for both demo and full-processor mode. This basically starts keystream generation. How the keystream is used depends on the mode of usage (XORed with PS/2 input in demo mode or XORed with contents of memory in full processor mode). This instruction stalls the processor for about 300 cycles (until the a51 unit has fully initialized the keystream).

2. `loadkeystream`
   Opcode 00000, AluOp 01001. Usage: loadks $rd, N.
   The A5/1 Unit generates a keystream which is accessed using this instruction. N can be 0, 1, 2 or 3. For instance, `loadks $r1, 1` loads the 33rd to 64th bit of the keystream into register 1.

3. `xor`
   Opcode 00000, AluOp 01000. Usage: xor $rd, $rs, $rt.
   32-bit XOR performed in ALU. Used for XOR of 128 bit message stream with 128 bit keystream.

3 Challenges

3.1 PS/2

The first challenge was understanding how the PS/2 keyboard gave outputs from the `last_data_received` and `ps2_key_pressed` outputs. Eventually we learned that on each physical key press, the `ps2_key_pressed` 1 bit output returns high for every make and

break code sent by the keyboard. We also discovered that using the `last_data_received` output instead of the `ps2_key_data` output was creating offsets between the key presses and the corresponding scan codes. The major challenges we faced were converting these key scan codes to ascii codes readable by the LCD display, debouncing the keyboard outputs, and storing the keyboard outputs to be used within the cipher unit.

Our original intent for our cipher was to have the user input the message and key by keyboard in standard ASCII, while also displaying the characters on the LCD. This would involve converting each of the keyboard's scan codes to their corresponding 8-bit ASCII codes, which could both be displayed and stored for encryption. However, encryption of printable (able to be typed) characters would lead to many 8-bit ASCII codes that are unprintable and not able to be typed, making both the display of the output and the subsequent decryption impossible.

We then decided to switch to entering in a message in hex, using only the keys 0-9 and A-F. This ensured we would be able to display the encrypted output, while also allowing the user to enter any 8-bits with two hex codes, as opposed to having some characters be unprintable. We decided to reduce the message size to 32 hex characters, or 128 bits, leaving the key at 16 characters, or 64 bits. We implemented two converters from 8-bit scan codes, one to the 8-bit ASCII representation of its corresponding hex digit, and another to their 4-bit binary value.

With the keyboard reading 3 outputs for each key press, a LCD was required to only store the single desired output. We first used an asynchronous counter to display every third code, but despite correct logic this counter produced various unexplainable issues, such as breaking the key and data counters. When this debouncing was switched to a J/K Flip-Flop implementation, which only recognizes an output after a break code is detected, the problems ceased. From this J/K Flip-Flop, the 3 code sequence of make break break only registers the third code, which is consistently the desired output.

Storing each hex digit's corresponding 4-bit binary representation on every key press required an asynchronous counter, since the data is not updated based on clock. Two asynchronous counters were created, one for the key and one for the message. These are triggered on the negative edge of `ps2_key_pressed`, as well as the positive edge of `reset`. The counters are responsible for incrementing the indexes that decide which 4 bits in the data and key + frame registers the output will be written to. Designing registers whose indexes enable multiple bits at a time was another challenge, as was figuring out which other control signals should also be involved.

3.2 Backspacing & LCD

Implementing the backspace function to our PS/2 input was another challenge that we undertook. To make backspacing functional, we needed to detect when backspace is pressed and decrement our asynchronous counters, as well as clear the last character from the LCD display. The backspace detection and counter decrementing was relatively straightforward, but the LCD functions were much more difficult. After some research, we found that the LCD seemed to have a set of commands which included shifting the cursor and potentially backspacing. However, `LCD.sv` was not designed to incorporate these commands, and our attempts to edit it resulted in strange bugs that we did not anticipate, such as cursor flickering and an inability to shift. We eventually concluded that the design of `LCD.sv` looped through all 32 positions on every print to the LCD, meaning that the cursor could not be easily shifted. Instead, we decided to print a distinctive backspace character `<` to alert the user of a backspace occurrence.

3.3 Processor Integration

We had problems with this before the demo. The A5/1 cipher is optimized for hardware because it uses systems of LFSRs. For us, it was very counter-intuitive to take something suitable for a hardware implementation and make it so that it is more "software-based". Thus, our implementation initially had a high level of hardware complexity but very little assembly code and very little usage of the actual processor (other than being able to start the unit using the processor). Afterwards, we realized that we could create custom instructions in the processor which allow it to fetch output from the A5/1 unit and XOR them using its own ALU. This led to us having two modes of operation as discussed above. While our assembly code is definitely not as long and complex as the code for a game, we believe the hardware complexity that we faced makes up for this.

3.4 Circuit Timing

After the A5/1 cipher has finished generating the keystream, there are several steps before the final output is pushed to the LCD display. When we first tested our full encrypter, there was the consistent problem of having two 0's at the beginning of every encrypted output. This issue remained as we sorted through many other input/output related bugs, and we wondered what issues with our counter or registers could be preventing our output from pushing the last two 4-bit units to the hex-to-ascii converter.

We found that the real culprit was timing issues; on the final clock cycle of our key generator when `KeyStreamDepleted` is asserted, a final value is still being written into the output. Then, another clock cycle must pass as the output is pushed into a keystream shift register which will push the keystream to a final xored output register in the next cycle. Only after all these things have taken place can the final counter begin, which increments the index pushing the final encrypted output to the LCD screen. To create these delays, D Flip-Flops are used to store and output the appropriate data at the correct clock cycle. Hardware edge detection is done by comparing the values of two adjacent delay Flip-Flops.

4 Testing

4.1 Component Simulations

For most of our submodules we created testbenches in either Quartus or Modelsim to test the relevant functionality. This testing was particularly important for modules such as our `a51_keygen` as well as the many counters and register/decoder systems utilized in this project.

4.2 Hardware Testing

Testing the hardware required a full compilation of the A5/1 and all relevant modules, followed by an upload to the DE2 board using the programmer tool. The hardware switches were used to switch input modes and to initiate the encryption, while the LEDs were used to check different control signals at various stages of the process. We frequently conducted experiments on our project in order to test specific functionalities or to isolate bugs. For example, we removed our PS/2 debouncing module in order to view the unadulterated output of the PS/2 keyboard. This allowed us to see when each make and break code is sent and in what order in relation to our physical key presses.

5 Code Structure

5.1 yt61_amy8_FinalProject.v

This file contains the processor and miscellaneous modules used in the processor. Submodules of note include:

5.1.1 a51

Interfaces with the A5/1 keystream generator and processor. Important inputs include:

1. `ps2_key_data` and `ps2_key_pressed`, which interface with the PS/2 input from the user.
2. `reset` which allows for both a reset using a flip switch and for the processor to reset the unit after it is used.

Important outputs of this module include:

1. Various LED indicators which show the status of the encryption
2. `data_to_LCD_out`, `LCD_reset` and `LCD_enable`, which outputs character data to the LCD and controls how and when it displays output.
3. `keyindex_out` and `dataindex_out`, which output the character count for both the key and message so that they can be display on our hexadecimal 7 segment displays.

5.1.2 a51_keygen

Keystream generator which handles the logic behind keystream generation. Important inputs to this module are the key and frame bits and a signal that starts the module (which is idle otherwise). An important output is a signal that indicates that the module has finished encrypting.

5.1.3 LFSR

Module used for all the Linear Feedback Shift Registers, handles registers of variable length.

5.2 Assembly Code

The assembly code needed to use our A5/1 unit in **Full-Processor Mode** is pretty simple as most of the complex logic is already hidden inside the processor and A5/1 unit. We essentially initialize a valid 128 bit keystream using `a51start`. After a few hundred cycles of `nop` due to our stall logic, we can load the keystream from the A5/1 unit into our regfile using `loadkeystream`. Since the A5/1 unit provides up to 128 bits of keystream, we can index into these bits using the RS field in our custom instruction. (`loadkeystream` is R-Type but does not use an RS register). Subsequently, we load the message (which is stored in dmem) into up to four registers. We then `xor` using the ALU. Lastly, the result can be stored for further processing or transmission using `sw`.

```
.text
```

```
a51start
loadkeystream␣$r1,␣0
loadkeystream␣$r2,␣1
loadkeystream␣$r3,␣2
loadkeystream␣$r4,␣3
lw␣$r5,␣0($r0)
lw␣$r6␣,1($r0)
lw␣$r7␣,2($r0)
lw␣$r8␣3($r0)
xor␣$r9,␣$r5,␣$r1
xor␣$r10,␣$r6,␣$r2
xor␣$r11,␣$r7,␣$r3
xor␣$r12,␣$r8,␣$r4
sw␣$r9,␣5($r0)
sw␣$r10,␣5($r0)
sw␣$r11,␣5($r0)
sw␣$r12,␣5($r0)
halt

.data
msg1:␣.word␣0x0000AAAA
msg2:␣.word␣0x0000BEEF
msg3:␣.word␣0x0000FEED
msg4:␣.word␣0x0000EEEE
```

6 Figures and Tables

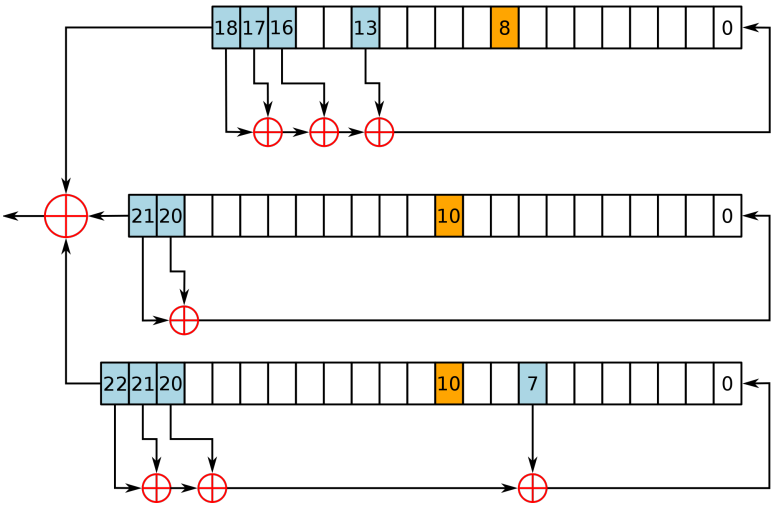

Figure 1: LFSR Configuration in A5/1 Cipher

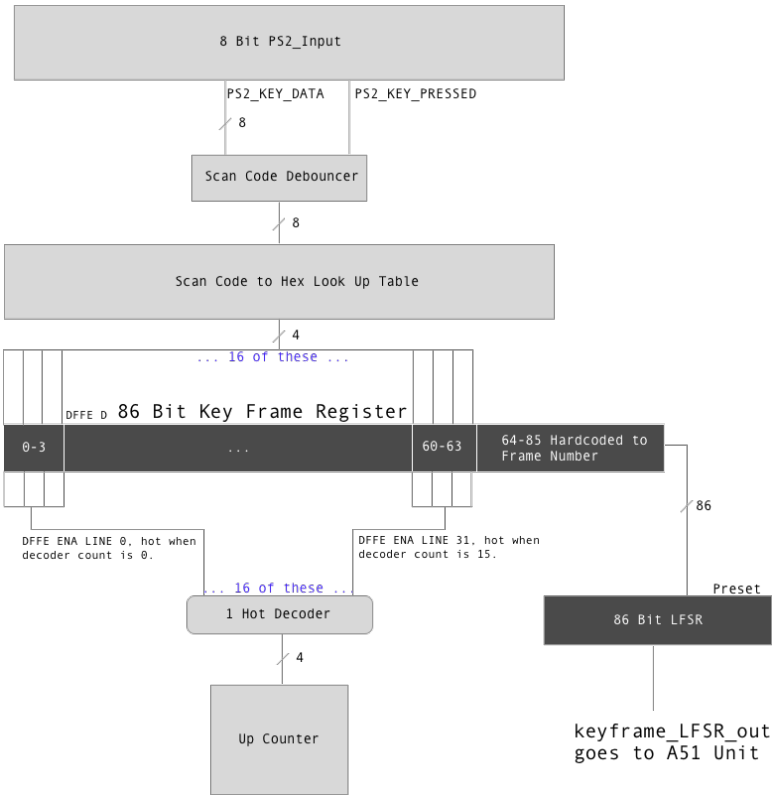

Figure 2: Hardware Schematic: Handling Key and Frame Entry into A5/1 Module

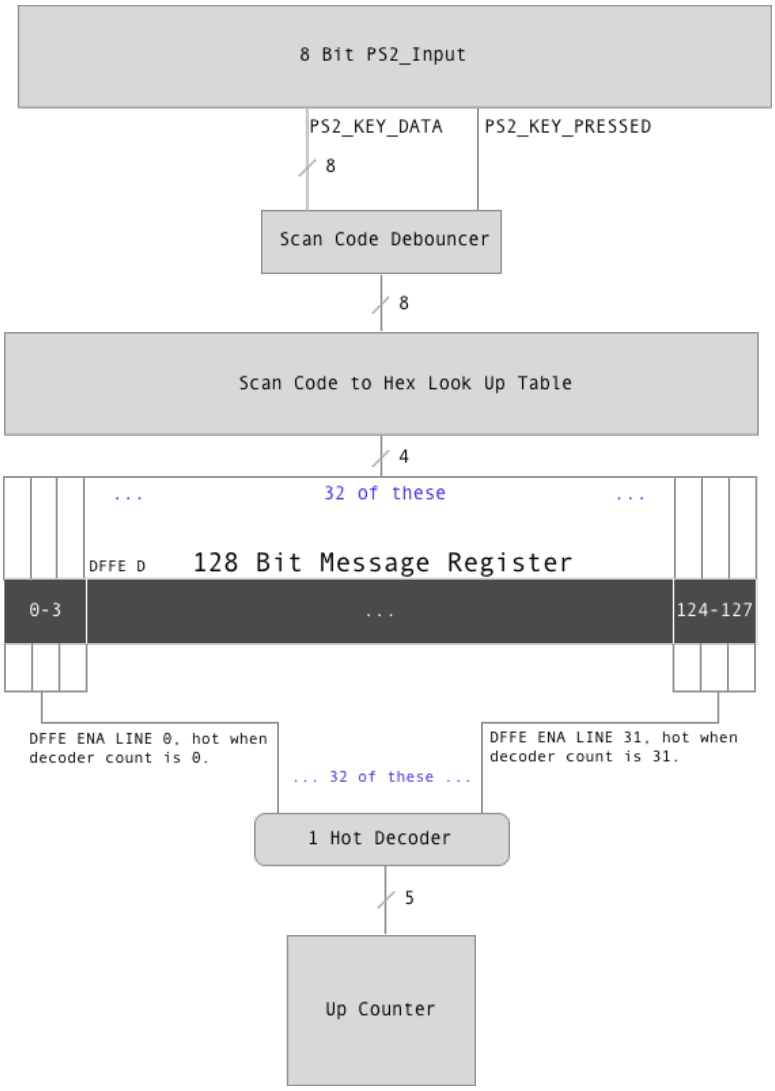

Figure 3: Hardware Schematic: Handling Message Entry into A5/1 Module

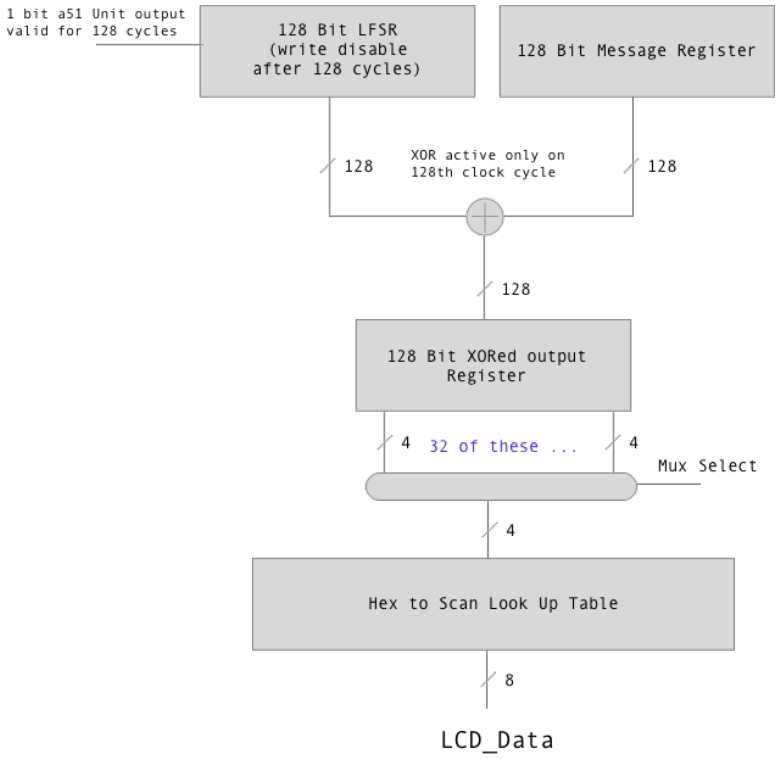

Figure 4: Hardware Schematic: Handling output of encrypted message to LCD in **Demo Mode**

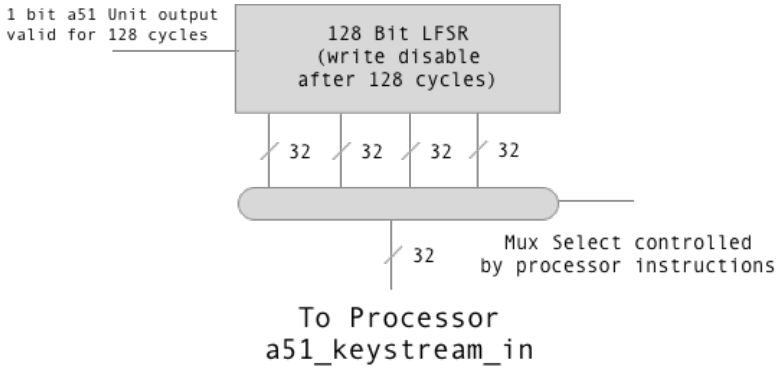

Figure 5: Hardware Schematic: Handling output of encrypted message to LCD in **Full Processor Mode**

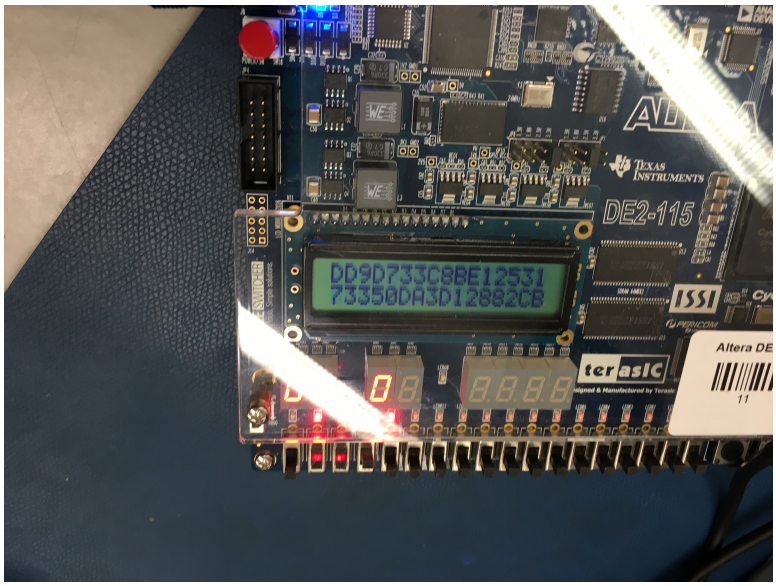

Figure 6: LCD Output showing result of Message = {128{1'b1}} & Key = {64{1'b1}}

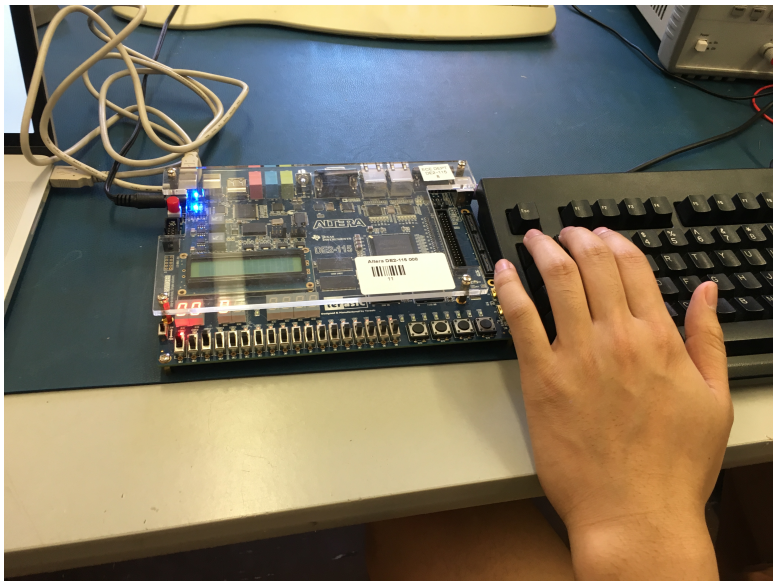

Figure 7: PS/2 input in Demo Mode

7 Acknowledgements

Many thanks to Dr. Dwyer, Dr. Sorin, Justin and all our TAs for helping us throughout the semester.