Lab 5:
Fundamentals of Assembly Language Part 1: Data Movement

ELEC1710

1 Introduction

This lab teaches the basics of data movement in ARM Thumb assembly. The task is to implement a look up table (LUT) which emulates a combinational logic circuit or ROM device in software.

In this case we will be emulating the CD74HC4511 chip used in Lab 2 and so the LUT will contain 7-segment decoder data. However, here we will support all digits 0-9, A-F allowing a full hexadecimal display to be built.

2 Program Algorithm

The program will be reading the 4 least significant bits of `GPIOA` and using these to address a LUT with 16 entries. Each LUT entry will be 8-bits (1 byte) wide. The data read from the LUT will be written to `GPIOB` pins 3-9 (inclusive) which can be connected to a 7-segment display. The Lookup table is provided for you. You will need to write instructions to read from GPIOA, decied which entry of the table is required, and write the corresponding table data to GPIOB. Listing 1 shows pseudo-code for the program which is required to be translated into ARM assembly.

Listing 1: LUT Algorithm

```
LOOP Forever
BEGIN
    Load GPIOA port pins A3->A0 into a variable A.
    Load into X the 8-bit data which exists in LUT at line A.
    Output X to GPIOB pins 3 to 9.  // this will require a shift of the data in X so
            //...that the correct 7 bits from the table align with the correct pins.
END
```

3 Assembly Reference

You will find the following instructions useful in this lab:

3.1 Load: `ldr` and `ldrb`

The `ldr` instruction copies data from memory (RAM, program data, GPIO pins, etc) to a CPU register. In this lab `ldr` will be used to read `GPIOA_IDR` (GPIOA input data register) while `ldrb` is used to read bytes from a LUT located in flash memory.

Note that the registers and memory addresses are 32 bits and memory addresses specify a byte's location. The `ldr` instruction loads 4 bytes (32 bits), starting from the specified address and finishing at the address + 3. The `ldrb` instruction only loads 1 byte (8 bits) and fills the remaining 3 bytes of the destination register with zeros.

Important: the look-up-table must be *word-aligned* (ie: start on an address which is a multiple of 4) or loading a 32-bit word from the LUT will result in an *exception* which causes the CPU to jump to the *default exception handler* and loop forever (ie: the CPU crashes and needs to be rebooted).

The data alignment is achieved with the assembler directive `.align 4`.

Listing 2: Loading a register with a constant using `ldr`.

```
ldr r3, =0x12345678
```

Listing 3: Loading data from `GPIOA` using `ldr`.

```
.equ   GPIOA_IDR, 0x40010808 // Address of GPIOA_IDR from datasheets
// ... other code
ldr r0, =GPIOA_IDR  // This constant needs to be defined in the code
ldr r1, [r0]        // Load the data at address GPIOA_IDR into r1
```

Listing 4: Loading a register with an address from code memory using `ldr`.

```
ldr r3, =LUT

.align 4
LUT:
    .byte 0x01
    .byte 0xFF
    .byte 0x34
```

Listing 5: Loading a register with a byte code memory using `ldr`.

```
ldr r3, =LUT
ldrb r4, [r3]    // Loads 0x01 into r4 from LUT

.align 4
LUT:
    .byte 0x01
    .byte 0xFF
    .byte 0x34
```

Listing 6: Loading a register with a byte from code memory `ldr` with a constant offset.

```
ldr r3, =LUT
ldrb r5, [r3, #2] // Loads 0x34 from LUT+2 into r5

.align 4
LUT:
    .byte 0x01
    .byte 0xFF
    .byte 0x34
```

Listing 7: Loading a register with a byte from code memory `ldr` with a register offset.

```
ldr r3, =LUT
ldr r2 = 0x01
ldrb r5, [r3, r2] // Loads 0xFF from LUT+1 into r5

.align 4
LUT:
    .byte 0x01
    .byte 0xFF
    .byte 0x34
```

3.2 Store: `str`

The `str` (store) instruction moves data from registers to a memory address. In the context of this lab it will be used to write data to `GPIOB`.

It has several forms however the syntax required for this lab is:

`str Rt, [Rn] // Copy the data from Rt into the address in Rn`

Example:

Listing 8: Store example.

```
ldr r0, =GPIOB_ODR
ldr r1, =0x0
str r1, [r0]     // Move the value 0x0 to GPIOB_ODR
```

3.3 Unsigned Bit Field Extract: `ubfx`

The `ubfx` instruction performs an *unsigned bit field extract* operation. This involves copying a given set of bits within a *word* (32-bit value) to to least significant bits of a destination. The syntax is:

`ubfx Rd, Rn, #lsb, #width`

It moves bits from `Rn` into `Rd`, starting at bit `#lsb` and ending at bit `#lsb+#width-1`. Note that bits are numbered from 0 to 31. For example:

Listing 9: Extracting bits using ubfx.

```
ldr r3, =0x12345678
ubfx r1, r3, #4, #4   // After execution r1 would contain the value 0x07
```

3.4 Logical Shift Left: `lsl`

A logical shift left moves each bit in a word one place to the left. The least significant bit is filled in with a `0` and the most significant bit is lost. **Fun fact:** This is the equivalent of multiplying by 2.

The syntax required for this lab is:

`lsl Rd, Rm, #n`

This instruction takes the data in `Rm`, shifts it to the left by `#n` bits and stores the result in `Rd`. This will be using for shifting a byte read from the LUT into the correct bit position for writing to `GPIOB`.

Example:

Listing 10: Logical shift left example.

```
ldr r0, =0x37
lsl r1, r0, #5 // After execution r1 would contain the value 0x6E0
```

4 Hardware Configuration

4.1 Hardware Requirements

- Breadboard

- STM32F103C8T6 development board

- 4x Tactile push button switches

- 4x 10k Ω resistors

- Various jumper cables

- Saleae logic analyser

- (Optional) 7x 330 Ω resistors

- (Optional) 1x 7-segment display

4.2 Hardware assembly

Perform the following construction steps:

1. Ensure that the STM32 board is plugged in at one end of the breadboard with the ST-Link debug connections mounted at the edge of the breadboard.

2. Confirm that the ST-Link debugger is connected correctly. See Lab 4 for connection details.

3. Connect the "V3" pin of the STM32 board to the breadboard's Vcc rail. With the debug port pins facing up the V3 pin is located on the lower left of the board.

4. Insert 4 push button switches and their associated pull-down resistors to create a source of a 4-bit number. See labs 1 and 2 for details regarding the push buttons.

5. Connect each push button's output to port pins A3, A2, A1 and A0. Note that the left most push button should be the MSB and connect to A3.

6. Connect 7 test leads from the Saleae analyser to `GPIOB` pins `B3` to `B9`.

7. (Optional) You may wish to skip this step until the software has been written and confirmed working with the Saleae analyser.

   Connect the 7-segment display to the STM32 via 330 Ω resistors.

   Port pins B3, B4,..., B9 drive segments a, b, ... , g respectively. User jumper cables as necessary.

5 Programming Tasks

5.1 Template Code

Listing 11 is a template for this lab. It contains the LUT data required for driving a 7-segment display.

NB: One of the table entries contains a bug. This is intentional, you should find and correct it.

Listing 11: Code listing for `sseglut.s`.

```
    .syntax unified
    .cpu cortex-m3
    .thumb
    .global sseglut

.equ    GPIOA_IDR, 0x40010808
.equ    GPIOB_ODR, 0x40010C0C

sseglut:
            // Write your code here
            // Read GPIOA_IDR
            // Extract bits required
            // Get data from LUT
            // Shift data to the correct bits
            // Write data to GPIOB_ODR
        b sseglut   // Jump back to sseglut, repeat forever

.align 4
ssegdata:   // The LUT
    .byte 0x3F  // 0
    .byte 0x06  // 1
    .byte 0x5B  // 2
    .byte 0x4F  // 3
    .byte 0x66  // 4
    .byte 0x6D  // 5
    .byte 0x7D  // 6
    .byte 0x07  // 7
    .byte 0x7F  // 8
    .byte 0x67  // 9
    .byte 0x77  // A
    .byte 0x7C  // B
    .byte 0x29  // C
    .byte 0x5E  // D
    .byte 0x79  // E
    .byte 0x71  // F
```

5.2 Programming Setup

A template for this lab is located under the Lab 4 folder on Blackboard. The code contains a `main.c` file which can, with minimal modification, be used to call any of several assembly `.s` files.

To perform this lab uncomment line 42 of `main.c` so it reads "`sseglut();`". This will cause the program to jump to the `sseglut:` label inside `sseglut.s`.

5.3 Programming Task

Using the instruction reference data in Section 3 implement the algorithm in Listing 1 using Thumb assembly inside `sseglut.s`. Confirm that your code works by observing the state of the appropriate `GPIOB` pins with the Saleae analyser as you input different binary numbers with the push buttons.

5.4 Delay Measurement

Use the Saleae analyser to measure the delay between changing the input on `GPIOA` and the corresponding bit pattern appearing on `GPIOB`.

Compare the STM32's delay to the delay specified in the CD74HC4511 chip's datasheet of around 20 ns.

5.5 Debugging

One of the table entries is incorrect. Find and correct this error.

5.6 Program Memory Size

Project part C will have some marks allocated for optimising code size. Measure the code size (in bytes) of all instructions between the `sseglut` and `ssegdata` labels. Do this by putting the debugger into "single instruction stepping mode", Figure 1, to bring up the disassembly view and obtaining the memory addresses of the first and last instructions in the `sseglut` section, Figure 2. You may use an online calculator to perform the hexadecimal subtraction. Note that you need to use the first address of the `ssegdata` section to include the size of the last instruction in `sseglut`.
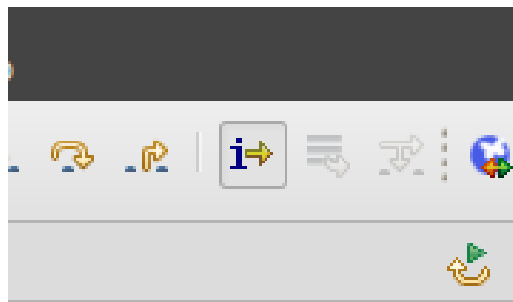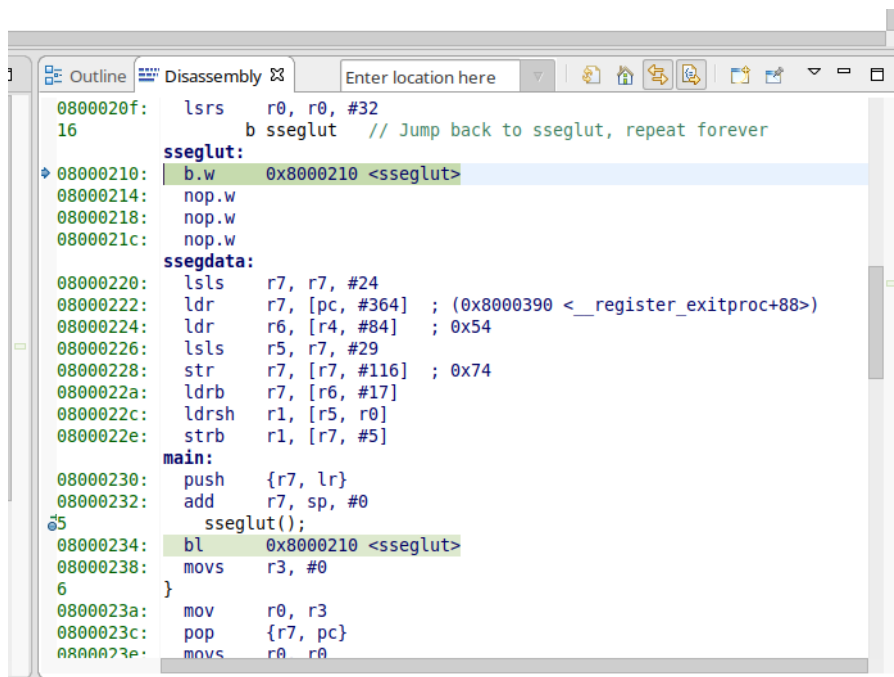

Figure 1: Click the "`i`" icon to enable single instruction stepping mode and enable the disassembly view.

Figure 2: Dissasembly view with memory addresses for each instruction.