

## Prelab for Lab #4: GPIOs in Assembly Language

Week of 14 February 2022

### Pre-lab

This lab will be re-implementing Lab#1 in Thumb-2 assembly language. As a reminder, Lab#1 was the one where we turned on and off the LEDs via the joypad.

### Part A – Textbook Readings / Videos

The following might be helpful in preparing for the lab.

1. Textbook Chapters 4, 5, and 6 cover Thumb-2 assembly programming
2. The classnotes are also posted to the course website.

### Part B – Prelab assignment

#### Setting Registers in Assembly

A lot of this lab will be setting up the peripheral registers. In C that looks something like the following:

```
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;
```

That one C statement actually breaks down into a number of low-level operations.

1. First it gets the pointer to the RCC peripheral register. This is set up for you in provided code via pre-processor directives, as seen below. The end result is that `RCC` is a pointer to the address `0x40021000`. Why that address? (That's where the manual says it is).

```
#define PERIPH_BASE      ((uint32_t)0x40000000)
#define AHB1PERIPH_BASE (PERIPH_BASE + 0x00020000)
#define RCC_BASE        (AHB1PERIPH_BASE + 0x1000)
#define RCC              ((RCC_TypeDef *) (RCC_BASE))
```

2. Our example is then accessing the `AHB2ENR` subregister found in the `RCC` block. In C this is done as a struct access. Deep down C structs are just offsets from a base pointer. In our case `AHB2ENR` is at offset `0x4c` from the beginning. (Again, you can find this info in the manual). So to access `RCC->AHB2ENR` We want to access the location pointed to by `0x40021000 + 0x4c`
3. To actually set the value in ARM assembly language requires 3 operations because ARM is a load-store architecture (meaning you can't operate directly on memory addresses, you have to bring values into registers first). Once we get the current value at that location we can bitwise-OR it with the bit pattern we want to set, then store it back out.
  - (a) Load the current value into a register.
  - (b) Perform the OR operation
  - (c) Store the updated value back out.

The equivalent Thumb-2 assembly will look like this:

```
ldr    r1, =RCC_BASE           ; point R1 to base register
ldr    r3, [r1, #RCC_AHB2ENR]  ; load 32-bit value at base+offset
orr    r3, #RCC_AHB2ENR_GPIOBEN ; bitwise OR with our value
str    r3, [r1, #RCC_AHB2ENR]  ; store value back to base+offset
```

1. The first instruction loads the value 0x40021000 into r1. (RCC BASE is set up for you, similar to the pre-processor, using EQU directives). We discussed the use of = to load 32-bit constants in class. r1 was arbitrarily chosen, for this lab you can use any registers r0 - r10.
2. The next instruction loads into r3 the value found at r1 (holding RCC\_BASE) plus the constant offset RCC\_AHB2ENR, which is 0x4C (the header defines that for you).
3. The orr instruction bitwise ORs r3 with the constant value RCC\_AHB2ENR\_GPIOBEN. Note the # is needed to indicate it is a constant. Also note we could have written that as

```
orr    r3, r3, #RCC_AHB2ENR_GPIOBEN
```

A common shortcut is to leave out the first source register if it's the same as the destination register.

4. Finally our updated r3 is written back to memory, to the same location we got it from.

You'll find a lot of the register initialization in this lab is just these blocks of load/modify/store values. You'll probably want to cut-and-paste them a lot, as we haven't learned how to do function calls yet.

### If/Then/Else

Your code will have some if/then/else segments, where you read the joystick values on GPIOA and then set the LED in one case and clear the LED in the other. You might want to think about what this code will look like in advance of the Lab.

### Loops

Your code will also have a loop, but an infinite loop. What does that look like in assembly?

## Part C – Questions

1. When loading the pointer value into r1 we use = to do a PC-relative load from a value in the literal area. Why can't we just specify a 32-bit immediate target to a move instruction like the following?

```
mov r0, #0xdeadbeef
```

2. In the example code we orr with an immediate constant pre-pended with #. What would happen if your immediate constant didn't fit? How could you work around that issue?