Theodore Peters 260919785
Ecse 324, Winter 2022
Lab 2 Report

Part 1: Basic I/O.

HEX_clear_ASM and HEX_flood_ASM work nearly identical, except that clear will zero the selected bits, while hex flood will one them. Thus, the common functionality of determining from the input which bits should be affected was factored out into a common function hex_mask, which takes as an argument in a1 the bit vector of which displays should be affected, and returns in a1 and a2 the bitmasks for overwriting the four low and two high seven segment displays, respectively. Then, in HEX_clear_ASM and HEX_flood_ASM it is only necessary to call hex_mask (thus requiring lr be pushed to the stack to be popped later), and then bitwise clear or bitwise or the appropriate memory locations corresponding to the seven segment display's readout.

HEX_write_ASM first requires encoding the digit which ought to be displayed into the corresponding bit vector representing which digits of the seven segment display to activate.

```
segment_encode:
.word 0b01001111010110110000011000111111
.word 0b00000111011111010110110101100110
.word 0b01111100011101110110111101111111
.word 0b01110001011110010101111001011000
```

These display configurations are stored in program memory under the label segment_encode. Each configuration requires only seven bits of data, but they are padded with an extra zero to make each 8 bits long, byte aligning them. Because we wish to display digits 0-F, this requires 16 bytes total, or four words. The bytes are arranged ascendingly in terms of the numerical value they represent, so the correct seven segment display setting for any digit can be retrieved by a ldrb instruction using the digit as an offset.

```
    ldr a3, =segment_encode
    ldrb a2, [a3, a2] // where a2 is the digit to display
```

Then, bit field insertion is used conditionally (bfine) to set the corresponding segments of the display based on the bit vector in a1.

```
    ldr a3, =SEG_ADDR // get memory address of seven segment device
    ldr a4, [a3] // load the 4 lower displays

    tst a1, #1 // checks whether the first display should be set
    bfine a4, a2, #0, #8 // sets the display
```

This will be repeated for all the four lower displays (HEX_0-HEX_3), then the two higher displays (HEX_4-HEX_5), which are stored in a separate word in device memory, offset 16 bytes from =SEG_ADDR.

The pushbutton drivers are all fairly simple, and their operation closely follows the provided templates for LEDs and switches. A few things to note: for PB_data_is_pressed_ASM and PB_edgecp_is_pressed_ASM, it is not sufficient to merely load and and the device info into a1, as the function specification requires that #1 be returned if the corresponding button is/has been pressed; thus, it is necessary to then set the a1 to #1 for the true case, rather than simple the device data, which would be #0b1000 instead if the fourth button was pressed, for example. Here is the data pressed function, edgecp pressed is identical save for the offset of loaded memory.

```
    ldr a2, =PB_ADDR
    ldr a2, [a2] // push button pressed states
    ands a1, a1, a2 // check if specified push button is pressed, move 0 result to
a1
    movne a1, #1 // if pressed, make sure to return 1 rather than 1, 2, 4, or 8
```

enable_PB_INT_ASM and disable_PB_INT_ASM function by loading the interrupt mask register and orring or bicing by a1 respectively, as a1 already 1-hot encodes which buttons to affect in the same format as is stored in device memory.

For PB_clear_edgecp_ASM, #0xF is moved into the edgecp registers to clear them. This was done rather than moving the read value because using this value would occasionally cause device errors in the case where a push button was set in the cycle between loading and storing memory, complaining that setting zero to the edgecp bits would not clear them. This method also slows a more expensive memory access. This is more an issue with the emulator than a real chip, as the emulator's clock is clearly far slower; this shouldn't be a reproducible issue on a real chip without insanely lucky timing.

The main function's polling occurs in two overlapping loops. One loop is for when SW9 is pressed, in which case the low displays are cleared and the pushbutton edge presses are continuously discarded (this is so buttons pressed while SW9 will have no effect, rather than being queued and all activating once SW9 is released). In the second loop, when SW9 is not pressed, the edge button presses are polled, and if any have been pressed they are cleared and the corresponding displays are set. In both of these loops, the switches need to be polled in order to set the LEDs. Flooding the two leading displays occurs only once, before the polling loop, as they are never affected again. Note that if there are two button presses in close proximity to one another, between the reading of the edgecp registers and their being reset, a button press can be lost. Again, this is a result of the emulator's slower clock speed and is less likely to occur on a real chip.

Because this deliverable was dependent on I/O, testing had to be done manually; it was confirmed that, with the appropriate switches, a display could be set to the appropriate value 0-F, that all four displays could be set, that SW9 would clear the displays and prevent them from being set, that the LEDs would be set based on the switches regardless of whether SW9 was pressed, and that after SW9 was released the displays could once again be set.

Part 2: Polling Stopwatch.

The drivers for controlling the ARM timer are fairly self explanatory, they behave in a manner near identical to drivers for push buttons already discussed, but load the device memory for the timer instead.

A new driver for setting the seven segment displays was created for the stopwatch. This was done because the previous HEX_write_ASM would be inefficient for this use case; it would need to be called for each digit that needed to be set, but for each call it would perform comparisons and conditional operations because it could potentially set every display. It would be possible to reduce the number of calls by only writing digits when they are actually updated (eg. the hours don't need to update every second), but that requires more conditional branching as the clock value is updated. Instead, a new method was devised, HEX_set_all_ASM, which takes, in a1, a word containing a 6 digit hex number in the last 3 bytes, and sets all 6 seven segment displays accordingly. Below, the setting of the first seven segment display, which will be stored in a4.

```
HEX_set_all_ASM:
    push {v1}
    ldr v1, =SEG_ADDR
    ldr a3, =segment_encode

    ubfx a2, a1, #0, #4 // select digit from a1
    ldrb a2, [a3, a2] // encode digit to seven segment display
    bfi a4, a2, #0, #8 // set digit in seven segment display
    ...
```

To facilitate easy interface with this funcion, the clock is stored as a single word in memory `clock: .word 0`, which records each time field in its own nybble. The increment of this is controlled by `inc_clock`. Because the clock is in a single register, when testing for whether the value should be carried, it is insufficient to use `cmp`, as then the values of greater digits would affect lower ones. Thus, two `tst` functions to test whether both bits corresponding to 8 and 2 were set to carry when 10 had been reached (4 and 2 for 6).

```
    tst a1, #0x8
    tstne a1, #0x2
    bicne a1, #0xF
    addne a1, #0x10
```

The `tstne` is run only when the first is successful, so for the `bicne` and `addne` to run both tests must be successful. This is done for each digit of the timer, comparing by 8 or 4 based on carrying over on 10 or 6. The timer display has a range of 0:00:00.0 to 15:59:59.9 (displayed as F59599), after which it will reset back to zero, though the clock will continue to store the number of hours in the first byte and nybble of the word, up until of 4095 hours, at which point it will overflow back to 0.

`clock_reset` will set `clock` to `0`, and will write 0 to all the seven segment displays. `clock_start` enables the clock and sets it to auto repeat by setting the appropriate E and A flags in the control register, and sets the countdown to `20000000`, which corresponds to 100 ms due to the clock's frequency (from De1-SoC manual). `clock_stop` sets the E bit to 0. `clock_start` and `clock_stop` rely on the `ARM_TIM_config_ASM` device driver to accomplish this. Additionally, they both call `ARM_TIM_clear_INT_ASM`, to ensure that as the timer is started and stopped there is no skew by 100ms from an unhandled interrupt flag from previous runs.

In the main polling loop, the push button edgecps are read, and cleared if any have been pressed. Then, the appropriate clock control functions are called to start, stop, and reset it. Finally, `ARM_TIM_read_INT_ASM` is called, and if the clock device indicates that 100ms have elapsed, `ARM_TIM_clear_INT_ASM` is used to reset the flag, then `inc_clock` is called to update and display the elapsed seconds.

Testing was performed manually: it was verified that the stopwatch could be started and stopped multiple times, and reset both when running and when stopped, multiple times in the same run. It was tested that the timer correctly displayed and carried over digits up to a minute, and assumed carrying for times up to an hour would function equivalently. Unit tests were also performed on `HEX_set_all_ASM`, by calling it directly from the Disassembly pane with various entered register values in a1.

Part 3: Interrupts stopwatch.

Mostly identical to part 2, but the templated code was configured to allow button interrupts to control clock flow rather than fetching their values through polling, and clock interrupts were used to increment the stopwatch instead of polling the interrupt register. The methods used in the previous deliverable remain largely unchanged, except `clock_stop` and `clock_start` now set the `tim_int_flag` rather than the timer interrupt register to zero, and `clock_start` ensures the I flag is set to 1. The idle loop now polls `tim_int_flag` and `PB_int_flag` rather than the device registers.

In the provided template code, `CONFIG_GIC` was modified to additionally service interrupt ID 29, the timer, as well as push buttons (73); and `SERVICE_IRQ` was modified to recognize both of these interrupts and branch to `ARM_TIMER_ISR` and `KEY_ISR`, respectively. These two ISR functions clear the interrupt device registers, and set the corresponding int_flags.

The same testing protocol was used here as to test Part 2, as the desired functionality is identical and only the implementation differs.