Sofia Velasquez Sierra
260967314

March 16th 2023

LAB 2: Report

1. *Calculator*

The first part of the Lab consists of writing assembly code using drivers to create a calculator that supports addition, multiplication, and subtraction operations. Using switches and push buttons to input values and operations, and display the result on HEX displays. The calculator handles negative numbers and overflow, and the leftmost HEX display shows a negative sign if the result is negative. The calculator allows performing operations using the current result and clears the result and HEX displays when the clear button is pressed.

For 1 multiplication loop: Total program size: 896 bytes | Executed instructions: 399 | Data loads: 38 | Data stores: 36 | Total memory accesses: 473

Approach:

I approached this part by determining the sub-parts I had to implement for the calculator to function as a whole. I decided that I would start with implementing the polling; which automatically gives me two subroutines to implement: *PB_edgecp_is_pressed_ASM* and *released.*

The subroutine *PB_edgecp_is_pressed_ASM* was written in PushButtons exercise. It is a polling subroutine that waits for a PushButton to be pressed and then with BX LR takes the code back to the initial polling. To test this subroutine I just went line by line and checked if it was working correctly when a button had been pressed.

The subroutine *released* checks if the PushButton that was previously pressed has been released. If this one is released, the subroutine will first BL to *switch_input* which reads the input th switches and store them into two separate registers. *released* will then branch to the subroutine linked to each PushButton: PB0-clear, PB1-multiplication, PB2-subtraction, PB3-addition. I followed this by implementing each of the subroutines.

First, to implement clear, I directly branched to HEX_clear_ASM which clears all of the HEX displays. Then, I implemented the three operation subroutines in the same manner; storing the desired operation in a register, and branched to the subroutine *display* which displays the calculation onto the HEX displays. Later, while implementing the other specifications of the calculator I added the necessary code into each subroutine so that it would meet the desired result. Once again, the testing for this part was straightforward; consisted of inputting different numbers into the switches and checking that the code would branch to the right subroutine when needed and also outputted the correct answer.

I followed this by implementing the negative sign when an operation result was negative. This consisted of comparing the two inputs in the *subtraction* subroutine; if the first input was smaller, I inversed the subtraction order and added a negative sign to the last HEX display using the *negative* subroutine.

After this, in *switch_input* subroutine I added the lines of code that would check if there is a number on the HEX displays and use it as an input instead of using *m* and *n.* With this implemented, I was able to add a functionality into the *addition* subroutine; which compares the new input with the input in the display and if the latter is smaller the negative sign is removed. Testing this part involved more time and testing several times, when something did not work I had to go line by line from the beginning to find the problem and then address it.

Finally, I implemented the overflow, which appears on the HEX display once r is greater than 0x000FFFFF or less than 0xFFF00001.

In general, my approach to this exercise was to gradually incorporate each subroutine and make the required modifications as I progressed through the assignment. The testing was also done gradually, first separately for each of the sub-parts, and then the entire code using different inputs and using all the functionalities of the calculator one after the other making sure each one works correctly.

Challenges:

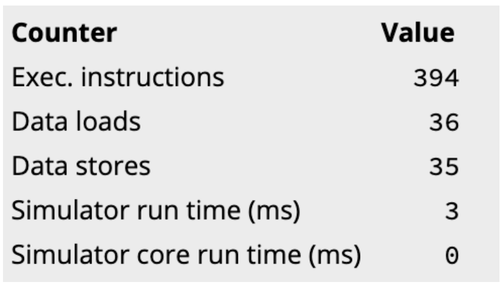
For this part of the lab, the main challenges I encountered were making the code work well together. I began testing each part or subroutine separately, and when it came to testing everything together I encountered a few problems and had to go line by line several times to debug the code. This demanded a lot of time and stress. I also had trouble getting my code to work without turning off the clobbered register and device-specific warning settings.

Shortcomings:

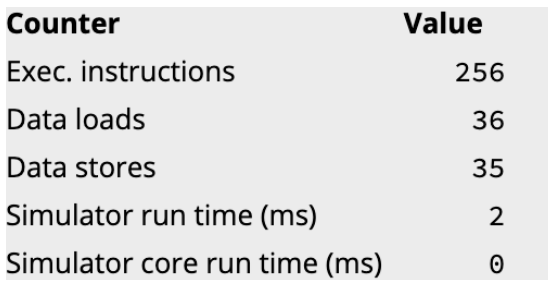
The shortcomings I mainly encountered for this part were the clobbered clobbered callee-saved register settings, which I was not able to fix.

Performance Analysis

An optimization for this part was to add BXEQ LR to each of the comparisons done in my *display* subroutine cmp. This enables the code to exit the subroutine faster, and can reduces the instructions executed as seen below:



Counter	Value
Exec. instructions	394
Data loads	36
Data stores	35
Simulator run time (ms)	3
Simulator core run time (ms)	0



Counter	Value
Exec. instructions	256
Data loads	36
Data stores	35
Simulator run time (ms)	2
Simulator core run time (ms)	0

My code could have also been optimised in other ways if time had allowed it. For instance, when converting the input to its corresponding HEX value, I could have implemented this in a faster way. Also, I could have tried different ways of randomising the moles, which could have optimised the code.

2. ***Timers***

The second part of the lab consisted of implementing a a Whack-A-Mole game using ARM A9 private timer, switches, push buttons, and HEX displays. Moles randomly appear on one of four HEX displays, and the user must press the corresponding switch to make the mole disappear and earn a point. The game lasts 30 seconds, and the score is displayed at the end. Control the game using PB0, PB1, and PB2, which start, stop, and reset the game. An infinite loop is used to poll the input registers and the ARM A9 private timer interrupt status register.

For a 3 second operation: Total program size: 1080 bytes | Executed instructions: 8346237 | Data loads: 3436480 | Data stores: 1963750 | Total memory accesses: 13746467

Approach:

I approached this part by implementing the timer first. The subroutines for the timer were written in the exercises preceding this part of the lab, so all I had to do was make a few modifications: implement the timer as a decrementation instead of incrementation and have it decrement from 30 instead of 15. This part was tested by checking that the counter was correctly being displayed and was also being decremented in the right way and timing.

I followed this by creating the mole and having it appear randomly in the first 4 HEX displays. The randomization was implemented using the last two bits of the counter to change the offset of where the mole was going to be stored each time: **AND R12, R11, #0b11** (R11 contains the current address) and **R12** will be the offset. I tested this by checking if the mole indeed appeared at different HEX display positions after each iteration.

Once this was implemented I used the *read_slider_switches_ASM* subroutine to implement the whacking of the moles. This consisted of branching to *whack_mole* when a switch was selected, *whack_mole* will then load the corresponding HEX display address if there is a mole and branch and link to *hit_mole*. *hit_mole* increments the score counter and using BL *clear_mole* the hit mole is cleared. The testing was done by checking if the mole hitting indeed worked, if it did indeed disappear and then reappeared at a different position. The testing was also done besides the timer, so that both of them were indeed working properly.

The PushButtons were then implemented, associating each one to a different subsequent subroutines: *stop* and *reset. stop* pauses the timer and the mole generation. *reset* resets the timer and the score counter to 0. This was tested by starting the timer and clicking the buttons making sure the desired functionality worked.

Once the subroutines mentioned above were implemented, I added the polling. It begins by first checking if a Pushbutton has been pressed. Then the poll branches to *ARM_TIM_read_INT_ASM* which gives the value of F. This one is checked back in the poll and there will be an infinite loop until F is different than 0x00000001. This being the case, the poll now BLs to *ARM_TIM_clear_INT_ASM, ARM_TIM_config_ASM and HEX_write_ASM,* which polls the timer by then decrementing the timer register by 1 at each iteration. The polling continues while the counter has not reached 0, when it does the score obtained is written in the HEX display. The polling was tested as integration testing, making sure every subroutine worked as expected, and that the code as a whole was doing what is expected.

<u>Challenges:</u>

The main challenge I encountered for this part was having the entire code work together, so the integration testing took me the longest to debug and test in different ways: i.e. pushing the stop button and restarting, pushing the reset button before the timer ends the count.

<u>Shortcomings:</u>

The main shortcomings I encountered implementing the milliseconds for the timer, which I should have implemented by initialising the counter 1/10 of the counter frequency. Also, I was not able to run the code run the code without turning off the clobbered register and device-specific warnings.

Performance Analysis

To randomise the mole appearance I used the last two bits of the counter to change the offset of where the mole was going to be stored each time. This way seemed to be random, since the mole would appear almost an equal amount of times at each HEX display. Maybe in a longer counter this would not be the case and the mole appearance would visibly not be random.

The triple polling is nonoptimal since it takes a significant amount of time for each of the pollings to execute and they are executed one at a time.

3. Interrupts

The task in this part was to modify the previously implemented Whack-A-Mole game.The interrupts for the ARM A9 private timer and pushbuttons need to be enabled. The IRQ handler needs to check both interrupts and call corresponding service routines. The KEY_ISR subroutine needs to be simplified, and a memory location designated to determine which pushbutton was pressed.

<u>Approach:</u>

This part of the lab was not implemented, but this would have been my approach if time allowed it.

For this part of the lab, we modify the whack-a-mole game from the previous section to use interrupts instead of polling. First of all, *_start* is modified by implementing the interrupts for the pushbuttons and private timer using the *enable_PB_INT_ASM* and *ARM_TIM_config_ASM* subroutines, that were used in part 2. The *IDLE* subroutine is created to implement the count down, which will resemble the counter from part 2. The *SERVICE_IRQ* subroutine is modified and the IRQ handler will be able to check the interrupts and branch to the corresponding ISR subroutine. The *ARM_TIM_ISR* and *KEY_ISR* will be linked to the PushButton and the timer, respectively, and are responsible for handling the interrupt requests. The *CONFIG_GIC* subroutine has to be modified to configure the PushButton interrupts and also the timer by passing the corresponding IDs to *CONFIG_INTERRUPT*. *KEY_ISR* will be significantly modified; it will write the contents of the PushButtons edge-capture register to the *PB_int_flag* memory (this will then allow the code to determine which PushButton was pressed), also clear the interrupts and then branch to a subroutine that will determine which PushButton was selected and branch to the according subroutine. The *ARM_TIM_ISR* subroutine will write a 1 into *tim_int_flag* when an interrupt is received, then clears the interrupt and then would branch to subroutine that will decrement the time.

A part from the changes mentioned above, the code should have stayed almost the same as in part 2.

<u>Challenges:</u>

The main challenge was that I did not manage to implement the code in a functional way. I found that understanding the way interrupts worked was a bit tricky and had a hard time understanding how to write a fully working code.

<u>Shortcomings:</u>

The main shortcoming was having time to really understand and implement this part correctly.

Performance Analysis

Part 3 was not fully implemented so I am unable to compare the performance differences between parts 2 and 3. However, I think that the interrupts are more efficient, it would lead to less instructions executed, and less data loads and stores.