Jerry Xia, 260917329 | Lab 2 Report

Part 1.1

A brief description of each part completed (do not include the entire code in the body of the report).

- This program involves using two inputs, the LEDs and slider switches, to write the corresponding values from the switches into the LEDs, in a 1 to 1 mapping based on the location of the slider switch.

The approach taken (e.g., using subroutines, stack, etc.).

- This was constructed using a simple loop that would repeat over and over again, reading in the value of the slider switches and storing it into a register, r0 (the return register). Once this was done, that same value would be used to write to the LEDs, given the address of the LEDs, which would correspondingly light up the display of LEDs.

The challenges faced, if any, and your solutions.

- There were little challenges faced in this procedure, since the code was given in the document. All that needed to be done was using that code, branching and linking to it, and repeating the process over and over again until the program was manually terminated by the user.
- The lab manual, however, had to be thoroughly understood to really know what was happening. Once this was accomplished, the program was written rather feasibly and without much issue.

Possible improvement to the programs.

- Since this program is optimal, given what is required in the description, one can say that it is optimal. However, this is not truly the case. One can use interrupts to configure this program as well, with an IDLE loop. Using this would involve less reading, and would simply skip to the interrupt service routine to deal with the corresponding interrupt (that is, when a slider switch is pressed).

Part 1.2

A brief description of the part completed.

- This is a program that uses the HEX lights display, and lights up the HEX displays with the slider switches, while simultaneously lighting up the LED displays. It uses slider switches 0-3 (0-indexed), and activating slider switch 9 will reset the program. This also involves another new functionality to consider, that is, the pushbuttons. Pressing down on the push buttons and releasing them will allow data to be entered into the edge capture register of these pushbuttons. Using this, we can use one-hot encoded strings to parse the data and write to the correct HEX displays. Note that this will occur at the same time when we write to the LED displays.

The approach taken (e.g., using subroutines, stack, etc.).

- This program has two "starting" points, one where it is essentially a setup, and HEX 4 through 5 are lit up. The other HEX displays are dependent on the input of the user.
- Upon entering the second starting point, _START_, which is also the location of the main loop, the program first reads the slider switches, writes to the LEDs, pushes the return value on the stack to use for later, reads the pushbutton data, pops the return value from the stack, and uses those two values (the slider switches and the pushbutton data) to write to another subroutine called HEX_clear_ASM and HEX_write_ASM;
    - The former of which, clears the value at the HEX display before writing to it. This way, the data can be successfully written to the display without overlap of the current value of said display. Thus, this eliminates the issue of the display showing obscure, unrecognisable outputs on the display of each HEX.
    - After this, we clear the edge capture register, and begin the _START_ program again.

  Further Details:
- When slider switch 9 is pressed and released, the program branches to END.
- To know which digit (0-f) is being used, several cmp and beq instructions were used in succession.
- After this, depending on that digit the corresponding value that needs to be written to the display was placed in register r0, replacing the real value of the digit.
- Then, the program iterates through the HEX displays, using r1 as it's guide. Once a value is written to a specific HEX display, done through arithmetic bit shifting (given the user manual's description of the address locations of the HEX display), that bit is subtracted from r1, and we read r1 again, which now contains the one-hot encoded information for the location of the other displays that need to be written to.
- After this is all done, it will BX LR back to the _START_ subroutine (the main loop).

The challenges faced, if any, and your solutions.

- At first, I had accidentally written HEX_write_ASM first before all other HEX functions (e.g., clear and flood), but this actually played to my advantage. I ended up using a special binary string, “0x10”, which if given to HEX_write_ASM, would clear the HEX display that corresponded to the input r1 (what displays to write to). This was very helpful!

- As well, the idea of clearing and then flooding the displays was vital in my quest to complete this program. Without this tactic, I thought I just had to write to the displays, but if I load a value, add a new value to it and store it back, if that original value is not cleared, then problems will arise. This was certainly the case with the HEX displays, and the corresponding binary encoding for each digit (0-f).

Possible improvement to the program.

- Again, interrupts could be used instead of continuous polling.
- Perhaps I could have used BX LR to return to where I was in the original program when I had matched the one-hot encoding in r1, and this way I wouldn’t have to perform redundant checks upon writing to each HEX display (if I already wrote to it, I wouldn’t need to check if I would have to again).

Part 2.1

A brief description of the part completed.

- This program involves the input from the last program, except the value of the digit is only written to the first HEX display.
- This program displays digits 0-f on the first HEX display, and restarts this process again.
- This occurs at intervals of 1 second. LEDs are written to depending on the digit.

The approach taken (e.g., using subroutines, stack, etc.).

- To configure the clock to decrement at intervals of one second, the value 20,000,000 was written into r0, given to the configuration of the clock and thus, because the clock frequency is 200 MHz, this would equate to 1 second. (However, due to the processing power of the simulator, this 1 second interval is longer than expected and can vary at times).
- The program first configures the timer in the setup, then branches to the main loop;
    - We clear the HEX display.
    - We write the current value of r1, our counter, to the display.
    - We also write to the LEDs with this value.
    - Once this is done, the interrupt is reset in the clock so that it can begin again.
    - The count increments by one each time. When it reaches 16, it resets to #0x0.
- The procedure for writing to (and clearing) the HEX display is the same as part 1.2 of this lab.

The challenges faced, if any, and your solutions.

- I had a very troublesome time figuring out how the clock worked. I thought I was doubting basic algebra when I was configuring the prescaler, but then I realized that it was not the prescaler I had to change, but the actual count that the clock would start at.
- Other than that, I had to thoroughly read the lab manual to understand how the timer worked. Everything else in this part was done before in previous parts.
- I thought the interrupt couldn't be used but, because it is the only way to know when a clock cycle was completed, I ended up using it. This helped a lot as well.

Possible improvement to the programs.

- Again, interrupts could be used. But this is not a requirement of the problem.
- Other than that, since the only real configuration I had to perform was the clock counting down at intervals of 1 second. All the other code was cut and pasted from part 1.2.
  Larger issues
- The timer did not increment correctly. The last HEX display would only light up as 0 the entire time, because I misinterpreted the question; I assumed that it would count in increments of 10 milliseconds, so all I had to do was change HEX displays 1-5.I unfortunately lost marks for this. However, the simple fix would be to change the default load value, and start the program at the 0-th bit.

Part 2.2

A brief description of each part completed.

- This program is a polling-based stopwatch that has pause, reset and start functionalities. By starting the stopwatch, a count increments by one every clock cycle (this time reduced to 10ms) and this has a cascading event on the rest of the HEX displays, thus creating a polling-based stopwatch.

The approach taken (e.g., using subroutines, stack, etc.).

- This uses code, again, from part 1.2 of the lab.
- Since the clock has a frequency of 200MHz the cycle was reduced to 50,000,000 or #0x02faf080. This was used to configure the timer.
- After this, all displays were flooded.
- A poll was used to start the timer. This was only used at the start. If data was read that JUST the first push button was pressed and released, then the program would branch to the main function.
- Once data is read that a push button is pressed, then the program would loop until it was released. This ensured that the push button was always pressed and released and this data was read in the edge capture register. This would have been better implemented with an interrupt but considering that we had to poll the push button edge capture register, this was the only way I could come up with that successfully removed this problem.
- A main loop _START_ would be able to poll the data from the push buttons and the interrupt as the timer continued to cycle.

The challenges faced, if any, and your solutions.

- There was a problem with understanding how pause and start worked. I had previously assumed that just by pressing and holding the pause push button, that the program would pause. However, it was crucial to actually pause the process once the push button was released, and start again when the start button was pressed and released. I had very little issues with the reset push button.

Possible improvement to the programs.

- The part where the enable_PB_edgecapture was implemented but never used. This could have helped when, say, a push button was pressed, and we are waiting for the value to be returned but other push buttons are being pressed. Once the loop would start again, this data would be read and would confuse the program, throwing it off course.
- However, to compensate for this, I always cleared the edge capture register once a pause, reset or start action was performed, so that the program was always stable. The user would simply have to press and release again to allow the program to use the information that has been passed (of which being pause, reset and start).

Part 3

A brief description of each part completed.

- This program is an interrupt-based stopwatch that has pause, reset and start functionalities. By starting the stopwatch, a count increments by one every clock cycle (at 10ms) and this has a cascading event on the rest of the HEX displays.
- The use of several ISRs built the foundation for this part of the lab. This includes push buttons 0 - 3 (0-indexed) and the ARM A9 Private Timer.
- This stopwatch has the same functionalities as the previous stopwatch, but with interrupt-based handling.

The approach taken (e.g., using subroutines, stack, etc.).

- This uses code, again, from part 1.2 of the lab; the clock cycle is the same as part 2.2.
- Again, all displays were flooded.
- Interrupts were used to interrupt the IDLE procedure for which the program looped on. Once this was done, the program would be able to branch to the ISR and complete the functionality needed.
- At first, the program would need to configure the arm timer (and thus it's interrupt enable bit) before enabling IRQ interrupts in the processor.
- In CONFIG_GIC, the address for the timer also had to be included so that the GIC could forward the interrupt to the CPU (which I chose as the same as the push button CPU; CPU 0 in the ICDIPTRn register).
- Set the enable bit in the CPU Interface Control Register (ICCICR).
- Set the enable bit in the Distributor Control Register (ICDDCR). This enables forwarding of interrupts to the CPU Interface, CPU0 for both interrupts.
    - This above process also branched to CONFIG_INTERRUPT, where I didn't modify the code, but arithmetic was performed to ensure that the Interrupt Set-Enable Registers (ICDISERn) were configured. This also applies to the Configure Interrupt Processor Targets Register (ICDIPTRn)
- The interrupt mask bits for the push buttons were already set for the program, so I did not use enable_PB_ASM or disable_PB_ASM.

The challenges faced, if any, and your solutions.

- There was a problem with incrementing the counter every time that count would increment. I had to make sure that my process of incrementing the count, checking if the count reached a bound of 6 or 10 was reached and then I would reset the count. The initial count, for which the 10 milliseconds display shows, was following a different stream of logic and thus I had to correct this for both part 2.2 and 3 of this lab.
- Otherwise, understanding the lab manual was a challenging task. Several parts were not needed to be manipulated in this lab, so knowing which parts to leave alone (such as

CONFIG_INTERRUPT) and which to modify (the part before the IRQ enable bits is set and also in CONFIG_GIC) were crucial to create a working program.

Possible improvement to the programs.

- Again, enable/disable_PB_INT_ASM was not used. This could have helped with the implementation of several push buttons being pressed, and thus only reading the ones that are currently pressed right now so that only one push button is read at a time. However, just like in part 2.2, I compensated for this by using PB_clear_edgecp_ASM.
- Since a lot of boilerplate code is already provided, and the only code that was added is the same as part 2.2, no different possible improvements stem from those already given in part 2.2 of this report.
  Larger issues
- Again, the timer was configured incorrectly. If I had started at the 0-th HEX display the program would function correctly. Unfortunately, it didn't. This is explained in further detail in Part 2.2's "possible improvement to the programs."