ECSE 324 LAB 2 REPORT

RAFID SAIF

Part 1a: Simple Drivers – LEDs and Slider Switches

Description:

The task for this part was to use the provided drivers to turn on/off the LEDs.

Approach Taken:

For this first problem, we are already given the code which reads the slider switches pressed and returns them in R0. We are also given the driver which takes in R0 the LEDs to be turned on. We simply branch through the same loop endlessly, where we first read the slider switches to R0 and then write the value in R0 to the memory address of the LEDs.

Challenges Faced:

None for this part, as code was provided.

<u>Part 1b: Advanced Drivers – HEX Displays and Pushbuttons</u>

Description:

The task for part 1b was to write drivers for the HEX displays and pushbuttons. Using these drivers, we write a program where the HEX displays display the number entered using the last 4 slider switches. This number is then displayed on the HEX display whose corresponding pushbutton is pressed and released.

Approach taken:

For this part, the pushbutton drivers were straightforward. The procedures in the instruction amounted to writing to and reading from the memory addresses of the I/O devices. Of more interest is the approach to the HEX display drivers. Firstly, we set the value of A3 to 1 for HEX_display and HEX_flood and 0 for HEX_clear. The reason for this shall be evident by the end of this section. We then set a register with 0's, 1's or an input number in our required indices. As each display requires 8 bits of data, we do a logical shift left of this value by 8 bits at every iteration of our loop. When we move from the index of HEX3 to that of HEX4, we adjust the last 8 bits of this value accordingly. We also have a variable to test which bits need to be adjusted. At each iteration, we shift this to the left by 1 bit. We then have a common subroutine for all three methods, 'adjust.' This subroutine, depending on whether the value of A3 we set before, performs a logical AND or logical OR operation. To clear bits, we do a logical AND with zeros in the bits we want to clear. To flood, we do a logical OR with ones in the bits we want to flood. To write to the display, we first clear the display we want to write to by calling the clear function before we enter the 'adjust' subroutine. The remaining steps are identical to flooding, so we perform logical OR with the number to be displayed in the required places.

Challenges Faced:

A challenge with the HEX subroutines was to perform the logical AND or OR operations correctly, while also updating the number we are comparing against. The attempt to use a common 'adjust' subroutine for all three methods complicated our implementation, because the function determines whether we perform a logical OR or a logical AND.

Possible Improvements:

For the HEX subroutines, several more parts could be unified into a common subroutine. We could also perhaps do this without a subroutine by simply using branching.

Part 2a: Timer Drivers and Configuration

Description:

The task for this part was to configure the timer and write subroutines to configure the timer, read interrupts and clear interrupts.

Approach Taken:

To configure the timer, we take two arguments- one for the load value and another for the control register. To read the interrupt, we simply read the F bit and to clear it, we write a 1 into the F bit. To make the timer, we bring over several subroutines from the code for parts 1a and 1b. In the main function, we first configure the timer, using a load value of 200000000. As the countdown frequency of the timer is 200 MHz, this means we reach a value of zero after 1 second. At this point, the F bit is set to 1. We check for this F bit in an endless loop, and when it is 1 we branch to part of our code to increment the timer. The first thing to do here is to clear the F bit, otherwise our timer will increment with every iteration of the loop. We then add 1 to the current value of the timer. If the value is less than or equal to 15, we use the HEX_display_ASM subroutine to display it in HEX0. We then move A2 into A1 and use that as the argument for LEDs_write_ASM. This causes the LEDS to switch on accordingly. If the value of the timer is greater than 15, we set the value back to zero and loop through again.

Challenges Faced:

The initial challenge was realizing that we are not directly using the counter value of the timer, but instead creating our own timer using the F bit. So instead, we use the load value to dictate when our F bit gets set to 1. Another challenge was to clear the F bit every time we read that it was 1.

Possible Improvements:

For the subroutines, we almost always used variable registers. This could be avoided in some cases, which would also allow us to avoid pushing and popping.

Part 2b: Polling Based Stopwatch

Description:

The task for this part was to use the code from the previous sections and create a stopwatch.

Approach Taken:

Firstly, we note that we increment in 10 milliseconds So, we load the value 2000000 into the load register of the timer. Like the previous section, we increment the register for HEX0 using the F bit. This represents our 10 milliseconds display. We have six such registers representing the count of each display. We increment every register when the register which represents the display immediately to its right exceeds the maximum value that it should represent. Finally, we loop back to polling for the F bit. We also poll for the values of the pushbutton Edgecapture register. With these values, we write into the control register for the timer to start, stop and restart the timer.

Challenges Faced:

No major challenges faced for this part. The only task was to implement 6 bits which increment based on the values of the next bit. One mistake made here was that we used registers A3 and A4, which were changed in other functions. We later changed them to variable registers.

Possible Improvements:

We did all the implementation of the timer manually. We could instead use a common subroutine which performed many of the common tasks.

Part 3: Interrupt Based Stopwatch

Description:

The task for this part was to implement the stopwatch from the previous section

Approach Taken:

Firstly, we note that we increment in 10 milliseconds. So, we load the value 2000000 into the load register of the timer. We also load a 1 into the interrupt enable register. We then check for interrupts using device IDs. The code provided initially only tests for pushbutton interrupts. We add code which, if the interrupt ID is not that of the pushbuttons, checks if it matches the ID of the timer. If it is neither, we simply execute an infinite loop. In the main code, we enable pushbutton interrupts as well. In the timer ISR, we write the F bit of the timer into the memory space which contains the timer interrupt value. In the pushbutton ISR, we write the value of the pushbuttons that are pressed into the memory space for pushbutton interrupts. We then poll for the values in the memory spaces. The rest of the code is identical to that in part 2b.

Challenges Faced:

Initially, the timer ran too fast, despite having the same load value as part 2b. We later noted that this happened because we did not clear the timer interrupt when we first received a 1, so the timer interrupt bit was always set to 1. This caused us to increment our timer with every iteration of the loop and not the actual ARM timer.

Possible Improvements:

This part is very similar to polling in that we have an endless loop where we are polling for the timer and pushbutton interrupt values and then perform functions within the main loop. We could instead write these functions into the ISRs.