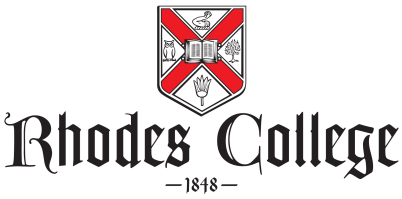
COMP 231
Introduction to Computer Organization
Lab 4



This lab consists of four subproblems. For each one, you will need to write a short ARM assembly language program and create a matching Altera Monitor Program project. You will submit your work with GitHub Classroom. Your lab submission should consist of four folders named `part1`, `part2`, etc.

The purpose of this assignment is to gain familiarity with hardware device control and usage.

Part 1: General Purpose I/O Devices

The switch, LEDs, key buttons, and seven-segment displays are all connected to the ARM CPU as **general-purpose I/O** devices (or **GPIO** devices) through a *parallel port*. So far, we've only used the base address of each device to read or write data to it. In reality, these devices consist of multi-word control and data registers, which allow us to use them in more sophisticated ways. This lab and the next lab will explore these uses. In general, a GPIO device consists of four registers as shown in Fig. 1.

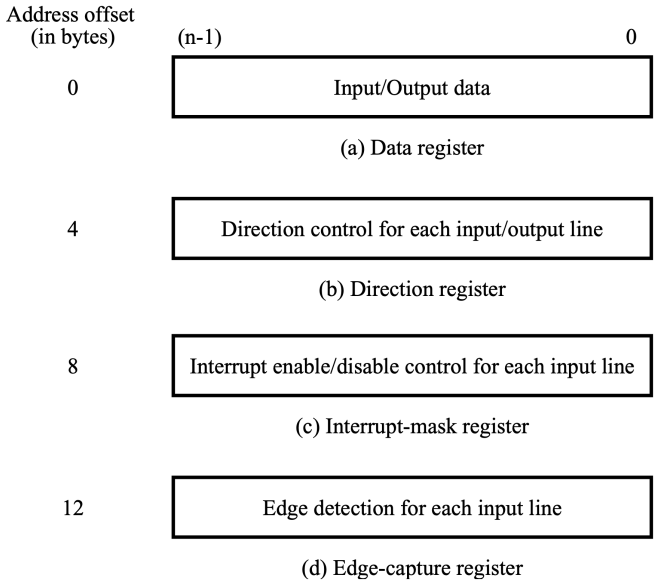


Figure 1: GPIO control and data registers

Each register is 32-bits wide and have the following purposes:

- **Data Register:** This register is used to read data from the device (if it is an input device) or write data to the device (if it is an output device). This register is located at the base address of the device.
- **Direction Register:** This register is used to define the direction of transfer for each bit, if it is connected to a bidirectional hardware device.
- **Interrupt-mask register:** Used to enable interrupts (if supported) on the device.
- **Edge-capture register:** This register indicates that the value in the data register has changed. Once set, this will remain set until a zero is written to it.

These parallel port registers are *memory mapped*, starting at a *base* address. The base address is the address of the *data register* of the parallel port. The addresses of the other registers are 4, 8, or 12 bytes (1, 2, or 3 words) from the base address. We will only be using the data register for this part.

The usage and legality of each of the parallel GPIO port registers depends on the device configuration itself. For example, there are only four keypress buttons on the DE10, and the device is read-only, so the device is setup as in Fig. 2.

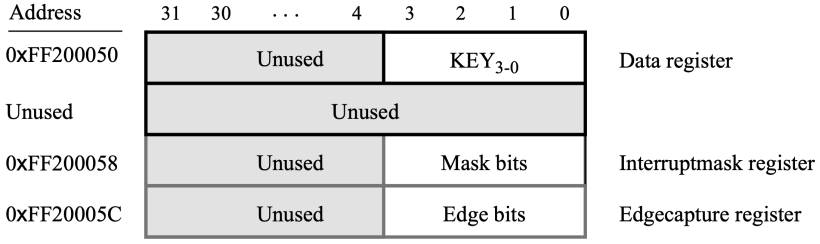


Figure 2: Parallel port connected to the pushbutton keys

For this part, you will display a decimal value on the seven-segment display, which will change when you press a keypress button.

1. If KEY_0 is pressed, you should display a zero on HEX_0. If KEY_1 is pressed, increment and display the updated number. If KEY_2 is pressed, decrement and display the number. If KEY_3 is pressed, the display should be blanked (i.e. no number displayed). This process should loop indefinitely.

2. To read a keypress button, you must consult the data register of the KEY GPIO port, located at address `0xFF200050`. As shown in Fig. 2, you will need to check each of the four bits (one for each key) in the data register. The bit will be set (1) when the button is pressed zero when not pressed. You will need to loop while the button is pressed in order to wait until the button is released.

3. You should copy code from Part 4 of Lab 2 to avoid having to re-write the code that displays numbers on seven-segment displays.

4. Write your program in a file named `part1.s` and create a new Altera Monitor Program project for this part.

5. Compile and test your program to make sure it works correctly.

Part 2: A simple timer

For this part, you will write an assembly language program that displays a two-digit decimal counter on the seven-segment displays, HEX_1 and HEX_0

1. The counter should advance approximately every 0.25 seconds. When the counter reaches 99, it should reset back to zero.

2. To wait 0.25 seconds, we will use a *delay loop*. This is an inefficient, but simple way to sleep for a specific duration. The code to do this is shown below (you can use any register):

```
do_delay: ldr  r7, =200000000 // delay counter
sub_loop: subs r7, r7, #1   // subtract one, set status
          bne sub_loop
```

3. The main complication with using the delay loop is that you may miss the time when keypress happens. To address this, we will use the *edge-capture register* on the GPIO port. When a button is pressed, the corresponding bit in the edge-capture register will be set until you reset it back to zero.

4. Create your program in `part2.s` and create a new project file to go along with your program.

5. Compile and test your program to make sure it works correctly.

Part 3: Hardware Timers

In Part 2, we used a delay loop to wait 0.25 seconds. This is inefficient because the program loops many, many, times until enough time has passed. This keeps the CPU at 100% utilization and we can't do anything other than wait for the time to pass. In this part, we will use a hardware timer so that we don't have to rely on a delay loop, and we also get a more accurate timer.

The DE10 has a number of hardware timers. For this part, we will use the *ARM A9 Private Timer* as shown in Fig. 3. This timer has four registers, starting at the address `0xFFFEC600`. Detailed information about this timer can be found in the *DE10-Standard Computer System with ARM Cortex A9* document that can be found on Canvas.

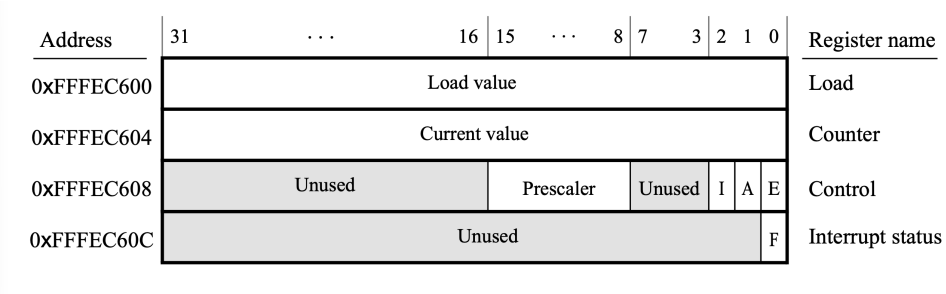


Figure 3: Parallel port for the ARM A9 Private Timer

1. To use the timer, you first write a value to the *load register*. This timer operates at a frequency of 200MHz, so once every clock tick, the *counter register* will decrement from the initial load value, down to zero.

2. The timer will only start timing when a 1 has been written to the *enable* bit (E) of the *control register*. The timer will stop if a zero is written to the enable bit.

3. When the timer reaches zero, it will set the F bit in the *interrupt status register* to 1. This will remain set until a zero is written to this bit.

4. If the *automatic* bit (A) of the control register is set, then the load value will be used to reset the timer and start timing again automatically.

5. Create your program in `part3.s` and create a new project file to go along with your program.

6. Compile and test your program to make sure it works correctly.

Part 4: A Real-time Clock

For the last part, you will write a program that implements a real-time clock. You should display the time on the four seven-segment displays ($HEX_3 \ldots HEX_0$) in the `SS:DD` format, where `SS` is seconds and `DD` are hundredths of a second.

1. Measure time intervals of 0.01 seconds, by polling the A9 Private Timer. Every time the timer elapses, update the time counter and display it on the seven-segmented displays.

2. If any KEY button is pressed, you should stop the timer if it's running and start it if it is not running. The timer should initially start off and will start with a button press.
3. When the clock reaches `59:99`, it should wrap around to `00:00`
4. Create your program in `part4.s` and create a new project file to go along with your program.
5. Compile and test your program to make sure it works correctly.

Submission

When you have completed this entire exercise and have a functioning program, submit your lab 4 project folder with four folders and four project files via GitHub Classroom. **Do not commit binary files that are generated by the compiler – only submit .s files, a project file, and a makefile.**