Laboratory Exercise 5

Using Input/Output Devices including a Timer

The purpose of this exercise is to explore the use of devices that provide input and output capabilities for a processor. There are two basic techniques for sychronizing with I/O devices: program-controlled *polling* and *interrupt-driven* approaches. We will use the polling approach in this exercise, writing programs in the ARM* assembly language. Your programs will be executed on the ARM processor in the DE1-SoC Computer system, and emulated using the CPUlator web-based system emulator. Parallel port interfaces, as well as a *timer* hardware module, will be used as examples of I/O hardware.

In general, a parallel port provides for data transfer in either the input or output direction. The transfer of data is done in parallel and may involve from 1 to 32 bits. The number of bits, n, and the type of transfer depend on the specific parallel port being used. The parallel port interface can contain the four registers shown in Figure 1.

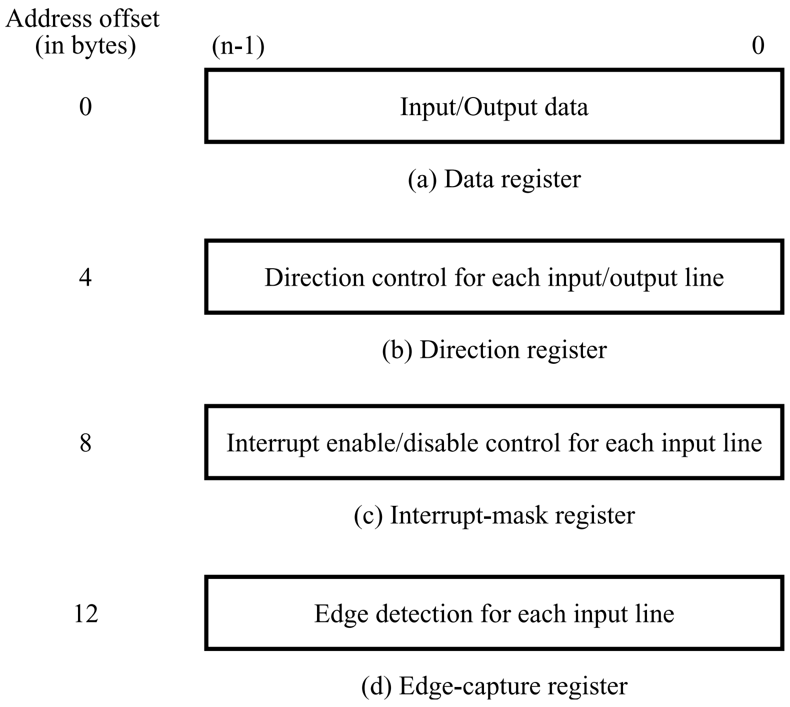

Address offset (in bytes)
(n-1)
0
0
Input/Output data
(a) Data register
4
Direction control for each input/output line
(b) Direction register
8
Interrupt enable/disable control for each input line
(c) Interrupt-mask register
12
Edge detection for each input line
(d) Edge-capture register

Figure 1: Registers in the parallel port interface.

Each register is n bits long. The registers have the following purpose:

- *Data* register: holds the n bits of data that are transferred between the parallel port and the ARM processor. It can be implemented as an input, output, or a bidirectional register.
- *Direction* register: defines the direction of transfer for each of the n data bits when a bidirectional interface is generated.
- *Interrupt-mask* register: used to enable interrupts from the input lines connected to the parallel port.
- *Edge-capture* register: indicates when a change of logic value is detected in the signals on the input lines connected to the parallel port. Once a bit in the edge capture register becomes asserted, it will remain asserted. An edge-capture bit can be de-asserted by writing to it using the ARM processor.

Not all of these registers are present in some parallel ports. For example, the *Direction* register is included only when a bidirectional interface is permitted by the parallel port. The *Interrupt-mask* and *Edge-capture* registers must be included if interrupt-driven input/output is used.

The parallel port registers are memory mapped, starting at a specific *base* address. The base address becomes the address of the *Data* register in the parallel port. The addresses of the other three registers have offsets of 4, 8, or 12 bytes (1, 2, or 3 words) from this base address. In the DE1-SoC Computer parallel ports are used to connect to SW slide switches, KEY pushbuttons, LEDs, and seven-segment displays.

Important Note on Subroutines and Parameter Passing for this Lab and ALL FUTURE LABS and EXAMS

In class we have discussed the ARM standard convention ('rules') for the use of registers for parameter passing to subroutines. These rules are formally called the ARM *procedure call standard* (PCS). In PCS, a subroutine is allowed to modify only ARM registers R0-R3, but it is not permitted to have changed the contents of ARM registers R4-R11 after execution of the subroutine. If you need to use registers R4-R11 during your subroutine, you should save their current values by pushing them onto the stack at the beginning of the subroutine, and restore them by popping them off the stack before the subroutine returns. The order of pushing and popping must be done carefully, if you were to do it with individual instructions. It is better to do it all at once with the push and pop instructions, which will use the correct order, provided exactly the same registers are pushed and popped. CPUlator will, by default, exit with an error and say that you have 'clobbered' one ore more of register R4-R11 if you over-write these registers in a subroutine.

Also, parameters are passed to the subroutine using registers R0 to R3. If you need to transmit more than 4 registers worth of parameters into the subroutine, then these should be passed using the stack. One way to do that is to have the *caller* (the code that called this subroutine) push those parameters onto the stack. The *callee* (the subroutine that was called) would then be responsible for popping them off of the stack.

Similarly, a single result is returned from a subroutine using register R0. If more results needed to be return, registers R1-R3 can be used and if there is a need for evern more than 4 results then these can be passed back to the caller on the stack.

Part I

You are to write an ARM assembly language program that displays a decimal digit on the seven-segment display *HEX*0 as described below. The other seven-segment displays $HEX5 - 1$ should be set to be blank.

The DE1-SoC Computer contains a parallel port connected to the seven-segment displays $HEX3 - 0$. The port is memory mapped at the base address 0xFF200020. A second parallel port is connected to $HEX5 - 4$, at the base address 0xFF200030. Figure 2 shows how the display segments are connected to the parallel ports.

The following functionality should be included:

- If KEY_0 is pressed on the board, you should set the number displayed on *HEX*0 to 0.
- If KEY_1 is pressed then you should increment the displayed number, but don't let the number go above 9 (i.e. pressing the key won't change the value if it is already at 9).
- If KEY_2 is pressed then decrement the number, but don't let the number go below 0 (i.e. pressing the key won't change the value if it is already 0).
- Pressing KEY_3 should blank the display, and pressing any other KEY after that should return the display to 0.
- The parallel port connected to the pushbutton *KEYs* has the base address 0xFF200050, as illustrated in Figure 3. In your program, use the *polling* I/O method to read the *Data* register to see when a button is being pressed.

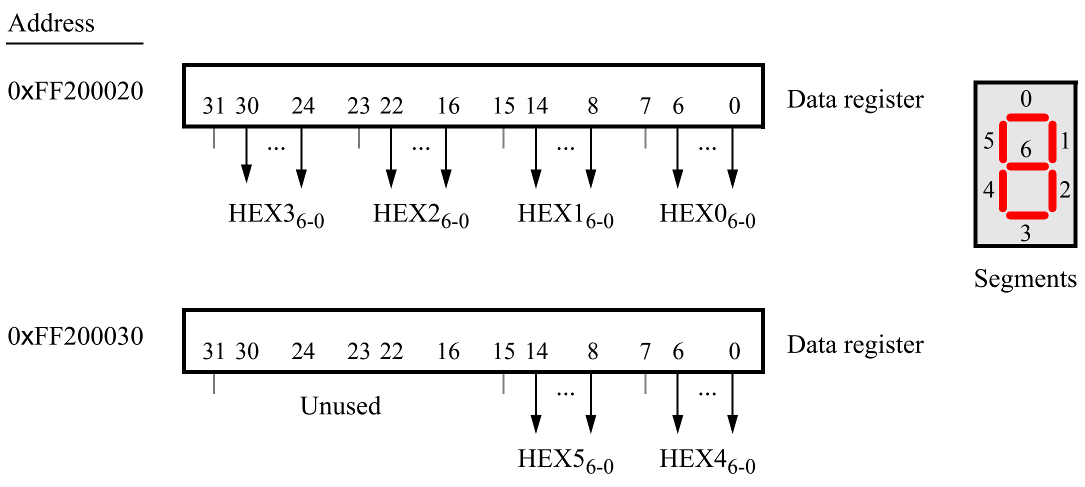
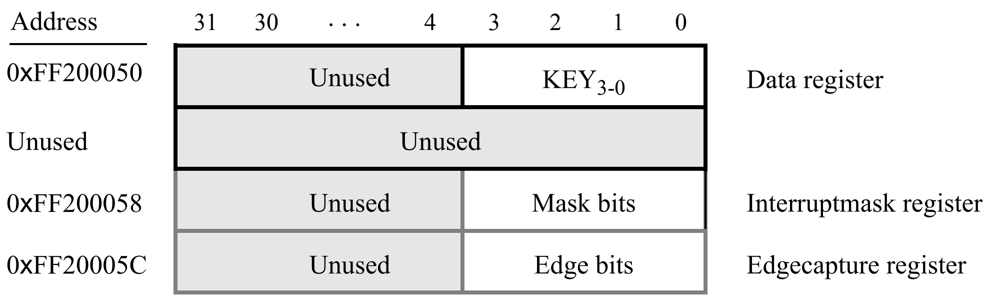


Figure 2: The parallel ports connected to the seven-segment displays *HEX*5 − 0.



Address	31 30 . . . 4	3 2 1 0	
0xFF200050	Unused	$\text{KEY}_{3\text{-}0}$	Data register
Unused	Unused		
0xFF200058	Unused	Mask bits	Interruptmask register
0xFF20005C	Unused	Edge bits	Edgecapture register

Figure 3: The parallel port connected to the pushbutton *KEYs*.

- When you are not pressing any *KEY* the *Data* register provides 0, and when you press KEY_i the *Data* register provides the value 1 in bit position i. Once a button-press is detected, be sure that your program waits until the button is released. You *must not* use the *Interruptmask* or *Edgecapture* registers for this part of the exercise.

Create a new folder to hold your solution for this part, put the code you create into a file called *part1.s*, and test and debug your program using CPUlator. You may want to refer to previous lab exercises for examples of assembly-language code, for displaying numbers on the seven-segment displays.

Part II

Write an ARM assembly language program that displays a two-digit decimal *counter* on the seven-segment displays *HEX*1 − 0. The counter should be incremented approximately every 0.25 seconds. When the counter reaches the value 99, it should start again at 0. The counter should stop/start when any pushbutton *KEY* is pressed.

To achieve a delay of approximately 0.25 seconds, use a delay-loop in your assembly language code. A suitable example of such a loop is shown below, which gives a good value for the delay in the CPUlator emulator. If you're using a real processor (which is faster) use the other number given.

```
DO_DELAY:  LDR   R7, =500000 // 500000 for CPUlator; use 200000000 on real hardware
SUB_LOOP:  SUBS  R7, R7, #1
           BNE   SUB_LOOP
```

To avoid “missing” any button presses while the processor is executing the delay loop, you should use the *Edgecapture* register in the *KEY* port, shown in Figure 3. When a pushbutton is pressed, the corresponding bit in the *Edgecapture* register is set to 1; it remains set until your program resets it back to 0. You reset an *Edgecapture* bit by writing a 1 into the corresponding position of the register.

Put your code into a file called part2.s, and test and debug your program.

Part III

In Part II you used a delay loop to cause the ARM processor to wait for approximately 0.25 seconds. The processor loaded a large value into a register before the loop, and then decremented that value until it reached 0. In this part you are to modify your code so that a hardware *timer* is used to measure an exact delay of 0.25 seconds. You should use polled I/O to cause the ARM processor to wait for the timer.

The DE1-SoC Computer includes a number of hardware timers. For this exercise use the timer called the ARM A9 *Private Timer*. As shown in Figure 4 this timer has four registers, starting at the base address 0xFFFEC600. To use the timer you need to write a suitable value into the *Load* register. Then, you need to set the enable bit E in the *Control* register to 1, to start the timer. The timer starts counting from the initial value in the *Load* register and counts down to 0 at a frequency of 200 MHz. (Note that CPUlator attempts to match this speed in its counters, which it can’t do when just executing code). The counter will automatically reload the value in the *Load* register and continue counting if the A bit in the *Control* register is set to 1. When it reaches 0, the timer sets the F bit in the *Interrupt status* register to 1. You should poll this bit in your program to cause the ARM processor to wait for the timer. To reset the F bit to 0 you have to write the value `1` into this bit-position.

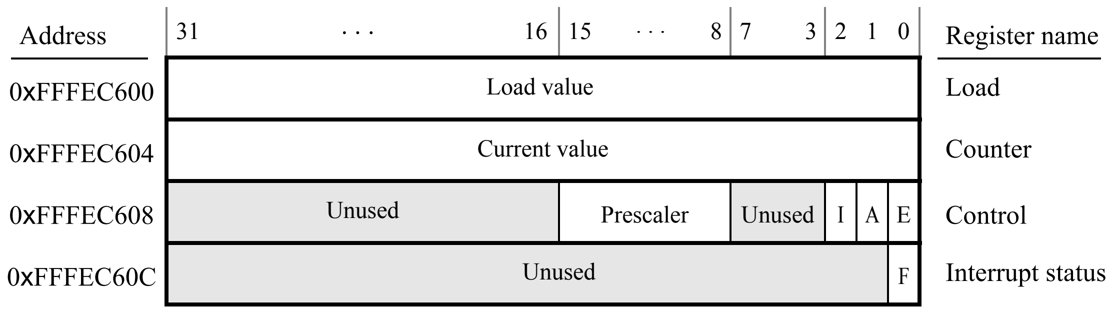


Figure 4: The ARM A9 Private Timer registers.

Put your code into a file called part3.s, and test and debug your program.

Part IV

In this part you are to write an assembly language program that implements a real-time clock. Display the time on the seven-segment displays $HEX3-0$ in the format SS:DD, where *SS* are seconds and *DD* are hundredths of a second. Measure time intervals of 0.01 seconds in your program by using polled I/O with the ARM A9 Private Timer. You should be able to stop/run the clock by pressing any pushbutton *KEY*. When the clock reaches 59:99, it should wrap around to 00:00.

Put your code into a file called part3.s, and test and debug your program.