Carleton University

Department of Systems and Computer Engineering

SYSC 3006 Fall 2016

Computer Organization

Lab #9

Prelab: There is **no Pre-Lab** for this lab.

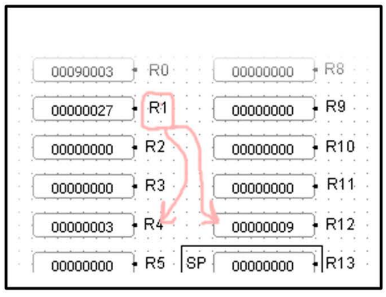
Read this document carefully before deciding what to do.

Lab Overview

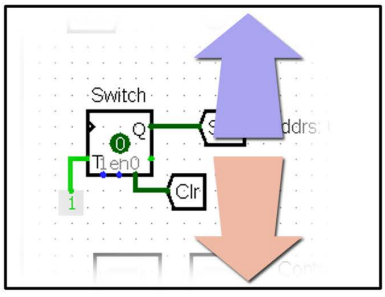
This lab will gently introduce the interactions between software and I/O devices. Specifically, you will develop an assembly program which displays a constantly changing value onto a two-digit display device.

The "display" part is where I/O comes in; but we'll start with the basics first. In Fragment 1, we'll take a given value and store it (after proper formatting) into two registers that we'll initially treat as our -very primitive- output devices. In Fragment 2, we'll introduce an I/O Switch component to interact with the software and control whether we're incrementing or decrementing that value. Finally, in Fragment 3, we'll display the value on the Digit Display I/O device, instead of just storing the values in registers (which are typically hidden from a human user).

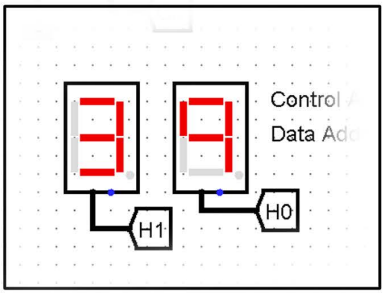
Lab progression is summarized below (all I/O devices can be found in the Debugger):



Fragment 1: Compute and store R1's decimal digits into registers.



Fragment 2: User input (via I/O Switch) to control if value is incrementing or decrementing.



Fragment 3: Decimal value output (via I/O Digit Display device).

Fragment 1

This fragment implements a simple loop that increments a value (R1) indefinitely. Think of it as a stopwatch that continues to increment without stopping. Following that analogy, we will treat R1 as a "Counter".

The final goal (in Fragment 3) is to display that number on the Digit Display I/O device. There are two requirements to pay attention to:

1. We want to display the value in decimal.

2. The Digit Display device is limited to two digits. So, we have to reset R1's value to zero whenever it reaches 100 (decimal).

The digit display device is composed of two separate single-digit displays. Hence, we need to format R1's value into a 2-digit binary-coded decimal (BCD) encoding. Recall from ELEC2607 that BCD encoding refers to representing each decimal digit in its own binary encoding. To see how BCD differs from a regular binary representation, consider the following example:

- Given R1 = 39 (decimal) = 0x27 = 0b00100111
- We need to somehow extract the tens digit "3" and the units digit "9" from the register's 32-bit representation (0b00100111)
  - Hint: use division to obtain the tens and the units digits.
  - A new DIV instruction is detailed at the end of this document.
- Once the the decimal digit values are known, R1's BCD-encoding can be either:
  - Packed into a single value (assuming 4 bits/digit): 0b 0011 1001; or
  - Split into two separate values: 0b0011 and 0b1001.

Fragment 1 uses the split approach, placing each BCD-encoded digit in its own register: the units digit in R12; and the tens digit in R4.

In-lab: Complete "Fragment1_toBCD_SRC.txt", assemble, and debug it. Set the Debugger's Tick Frequency to 4.1kHz, and verify that R1 increments continusly (mod 100), and that R4 and R12 display the correct (decimal) digits.

Fragment 2

Fragment 2 introduces our first interaction with an I/O device. Building on your Fragment 1 solution, this fragment queries the state of the Switch I/O component every iteration of the loop to decide whether to increment or decrement the value of R1. When the switch is off (0), R1 is incrementing; and when the switch is on (1) R1 should be decrementing.

You, the user, will be be able to interact with the software and change its behaviour *while* it's running, without altering its code. Notice that when interacting with the system through an I/O device, you become a "user". That is, you're no longer a "programmer" with access to the underlying software or architectural details. You simply have what the I/O interface fascilitates. [Real-life "user through I/O" example: you can press keys on your keyboard to control the behaviour of your video game character, but you don't have access to the specific signals, registers, fragments, etc. that process your key-presses].

The SYSC3006 Debugger uses special memory addresses to allow access to I/O devices. The address used to query the value of the Switch component is predefined as 0x80000200.

In-lab: Complete the provided "Fragment2_Trigger_SRC.txt", assemble it and debug it. Set the Debugger's Tick Frequency to 4.1kHz, and verify that R1, R4 and R12 operate correctly as stated above. You should be able to tap (poke tool) the Switch button to flip between incrementing and decrementing the counter, interactively while it's running.

Fragment 3

Building on your Fragment 2 solution, this fragment will display the digits on the Digit Display I/O device. That is, instead of just storing the digits in R4 and R12 (which are inaccessible to a "user"), the fragment will show the digits on the Digit Display device (also known as the "Hex Display" device).

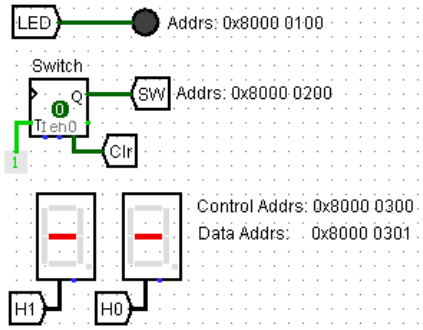
By default, the Digit Displays are turned off ("-" sign is displayed). The Fragment 3 source code contains the necessary subroutine to turn them on.

To display a value on the Digit Display device, the number must be packed as follows: Bits 0-3 represent the units digit; Bits 4-7 represent the tens digit. This is the same as the packed BCD format discussed in the Fragment 1 example given above. In Fragment 3, the number will be packed into R6 before being sent to the Display device.

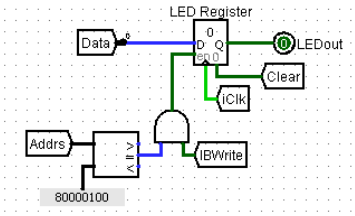
In-lab: Complete the provided "Fragment3_Display_SRC.txt", assemble it and debug it. Set the Debugger's Tick Frequency to 4.1kHz, and verify that R1, R4, R12, and the Display Device are operating correctly.

I/O Components (optional reading material)

The purpose of the I/O component in a computer system is to provide both an external interface for connecting external devices and a convenient internal interface for connecting the I/O component to the Interconnection Bus. The I/O component provides a program-accessible buffer between the Interconnection Bus and each of the devices connected at the external interface. The complexity of the buffer associated with a particular device depends on the nature of the information communicated with device, how the buffer supports control of the device. The Debugger's computer system includes an I/O component that connects three external devices: an LED, a switch (simulated using a T flip flop), and a 2-digit hex display. The LED and Hex display devices allow information to be output from the computer system. The switch allows information to be input to the computer system. In the computer system, the connected devices appear to the right of the Debug Control Panel as shown below:



An LED is a simple device that turns ON when presented with a logical 1 signal, and turns OFF when presented with a logical 0 signal. To connect an LED to a computer system, the system's I/O component must provide an output signal that can be connected to the LED to turn the LED ON and OFF. To allow a program to turn the LED ON or OFF, it must be possible to change the value of the output signal under program control. The Debugger circuit's I/O sub-circuit interface for the LED is shown below:



The heart of the interface is the 1-bit LED Register. The output (Q) from the register is connected to the LEDout pin, and the LED is connected externally to that pin. Once connected, the ON/OFF state of the LED will be determined by the contents of the register. The LED can be turned ON or OFF by writing a 1 or 0 (respectively) into the LED register. Recall that a memory word is just a register wrapped in some simple circuits to allow the selection of the specific word and to facilitate control for reading and writing. Not surprisingly, the design of the Interconnection Bus interface for the LED register shares many similarities with the interface used by Main Memory.

The circuit wrapping the LED Register accomplishes similar functionality to that used to interface memory words. The Data (D) input of the LED Register is connected to the least significant bit of the

Interconnection Data Bus and allows data on that bus to be presented to the register. The Addrs signal is connected to the Interconnection Address Bus to allow selection of the register using that bus. The comparator is used to select the register by comparing the Addrs signal to the constant value 0x80000100. The comparator sets the "=" output to 1 iff the values are equal. If Addrs = 0x80000100 AND IBWrite = 1, then the LED Register is enabled (en) and the data presented at the D input will be latched into the register on the falling edge of the System Clock. The interface allows a value to be written into the LED Register using a Bus Write Cycle. Therefore, an STR instruction can be used to write a value from a processor register into the LED register.

Note that the address used to select the LED sub-circuit is 0x8000 0100. In this lab, all I/O addresses are greater than 0x8000 000. This ensures that a word in main memory (from addresses 0x0000 0000 through 0x0000 0800) will not be selected when selecting the I/O component, and vice versa. Since address values requires a full 32-bits, they cannot be specified completely as immediate operands, and are out of the range of PC-relative access. One way to access the I/O component is to initialize a processor register to 0x8000 0000 and then use the Offset Immediate mode for LDR and STR instructions to interact with the I/O component.

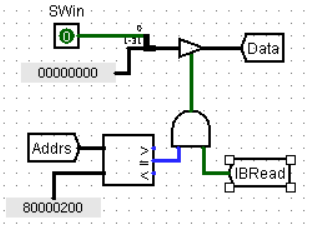
The following fragments assume that R4 contains 0x8000 0000. This fragment turns ON the LED:

```
MOV     R5, #1              ; value for turning LED ON
STR     R5, [ R4, 0x100 ]   ; turn LED ON
```

This fragment turns OFF the LED:

```
MOV     R5, #0              ; value for turning LED OFF
STR     R5, [ R4, 0x100 ]   ; turn LED OFF
```

A switch is also a simple device. The value from the switch is either logic 1 (when the switch is closed) or logic 0 (when the switch is open). In the Debugger circuit, a T flip flop is used to simulate an external switch. Use the Poke Tool to toggle the flip flop state between 0 and 1 just as you would toggle a real switch to change the output value. The I/O sub-circuit used to interface the switch is shown below:



The switch is connected externally to the SWin pin, and therefore that pin has the same state as the switch (0 or 1). The pin is connected to bit 0 of a 32-bit splitter, and the remaining splitter inputs are all held at 0. The splitter is connected to the input of a tristate buffer, and the output of the buffer is connected to the data bus. When the buffer is activated, the least significant bit on the data bus will reflect the state of the switch. The buffer is selected for activation in a manner similar to the way in

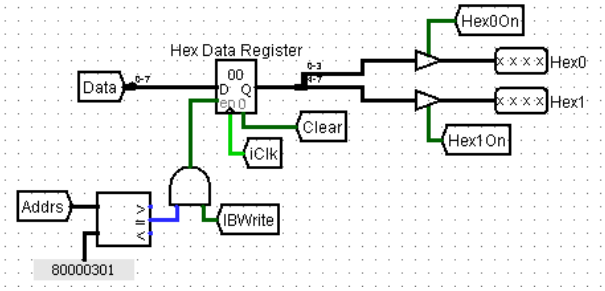
which the LED Register was selected. In this case, the comparator matches the address 0x8000 0200 instead of 0x800 0100, and the IBRead signal is used instead of IBWrite. The interface allows the value of the switch to be read using a Bus Read Cycle. Therefore, an LDR instruction can be used to read the switch value into a processor register.

The following instruction assumes that R4 contains 0x8000 0000, and reads the current state of the switch into R5:

```
LDR     R5, [ R4, 0x200 ]          ; read the switch state
```

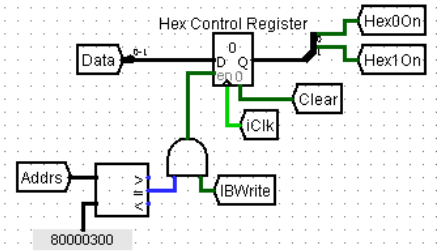
Once the switch state is in R5, it can be used appropriately.

The Hex Display device has been included as a more complex device (but only slightly more complex). Recall that a 7-segment display can be used to show the hexadecimal value corresponding to any 4-bit value. (See ELEC 2607 notes, pages 127 ff.) The 2 digits in the Hex Display device have been interfaced in the I/O component as shown below:

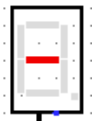


The heart of the interface is the Hex Data Register, and everything to the left of (and below) that register is very similar to the LED sub-circuit discussed above. The major differences are (1) the address 0x80000301 is used to select the Hex Data Register, and (2) the register holds the 8 least significant bits read from the Data Bus. The 8-bit value in the register contains the 2 hex digits to be displayed. The least significant nybble is output through the Hex0 pins, and the most significant nybble is output through the Hex1 pins. 7-segment hex digit displays are connected externally to each set of pins. Writing a data value into the Hex Data Register will cause the value to be shown on the connected hex displays … well … as long as the tristate buffers driving the output pins have been activated.

Before any data will appear on a hex digit display, the display must be enabled by activating its associated tristate buffer (see the Hex0On and Hex1On signals above). The interface provided in the Debugger circuit includes the Hex Control Register to allow each hex digit to be enabled or disabled under program control. The sub-circuit for the Hex Control Register is shown below:



The Hex Control Register is a 2-bit register, and each bit controls the enabling of one of the hex display digits. Values can be written into the Hex Control Register using address 0x8000 0300. Writing a 1 into the control register bit associated with a display digit will enable that display digit. When a display digit is enabled, it displays its associated value in the Hex Data Register. Writing a 0 into the control register bit associated with a display digit will disable that display digit. When a display is disabled, it displays a "minus sign":



DIV: Unsigned Division

Divides two unsigned 32-bit values to give an unsigned 16-bit result and the unsigned 16-bit remainder. Both the result and the remainder are given in a single 32-bit value with the result occupying the low 16-bits and the remainder occupying the high 16-bits.

FLAGS: DIV is a data manipulation instruction, and sets the flags in the following ways:

Z == 1 iff the result (including the remainder) is 0
C == 1 iff the result (or remainder) was larger than 16-bits, **in this case the result is meaningless**
N == x the N FLAG is set to a meaningless value by the DIV instruction
V == 0 the V FLAG is always cleared by the DIV instruction

Syntax and RTL:	DIV Rd, Rx, Ry	Rd ← Rx ÷ Ry
	DIV Rd, Rx, #imm16	Rd ← Rx ÷ 0-extended(#imm16)

Examples: DIV R9, R7, R0

; the low 16 bits of R9 are loaded with the 16-bit result of R7 ÷ R0
; the high 16 bits of R9 are loaded with the 16-bit remainder of R7 ÷ R0

DIV R2, R2, #10

; the low 16 bits of R2 are loaded with the 16-bit result of R2 ÷ 10
; the high 16 bits of R2 are loaded with the 16-bit remainder of R2 ÷ 10

EQU: Equate Directive

The EQUate (EQU) directive allows constant values to have associated names that are recognized by the Assembler. EQU is used to improve the read-ability (and write-ability) of programs. The EQU directive is not assembled directly into a memory location in the program image. The Assembler pre-processor simply replaces every occurrence of the defined name with its associated value.

The Assembler allows the names defined by EQU directives to appear anywhere that constant literals can appear.

Syntax:

EQU *<constant-name>*, *#constant-value*

Examples of EQU statements and the use of the defined names:

```
EQU   IObase, #0x80000000   ; "IObase" represents the Debugger's base I/O address

EQU   Next, #0x0A           ; offset to Next pointer in the linked list example from class

EQU   One, #1               ; this is possible but probably not useful

DCD   IObase                ; reserves a word of memory, and uses the "IOBase" name
                            ; defined by the EQU above to initialize the value
                            ; of the word to 0x80000000

LDR   R4, [ R4, Next ]      ; assumes R4 initially points to a linked list node, and
                            ; loads R4 with value stored in the node's Next pointer (this
                            ; uses the "Next" name defined by the EQU above)

MOV   R0, One               ; possible but probably not useful (this uses the "One" name
                            ; defined by the EQU above)
```