Department of Computer Engineering
Digital Hardware Systems
CpE 3202 - Computer Organization & Architecture

Laboratory Exercise #6
“Input/Output”

Instructions: Redesign the I/O Memory based on the I/O organization discussed (address decoding, latches & buffers).

1. Refer to Fig. 2 for the structure and organization of the I/O Memory. At this point, it is important to note that the I/O Memory should not be compared to the Main Memory in terms of organization. Logic wise, the same modules functions the same in terms of read and write operations. In this context, the “I/O Memory” are actually latches and buffers which are directly connected to the I/O devices. Each latch or buffer has designated address (I/O Memory address). The I/O address is decoded and enables the latch or buffer that is being referred to by the address which means only one (1) latch/buffer are enabled at any time. The decoder output is multiplexed with the output enable (OE) control signal for latches. The latches are also controlled by the IOM and RW control signals. For the buffers, it is enabled by the signal from the decoder and the IOM and RW signals. All output devices are connected to the latches and input devices to the buffers. The address of the latches (input) starts at 0x000 to 0x00F and the buffers (output) starts at 0x010 to 0x01F. The programmer has to take note of the address to properly access the I/O device.

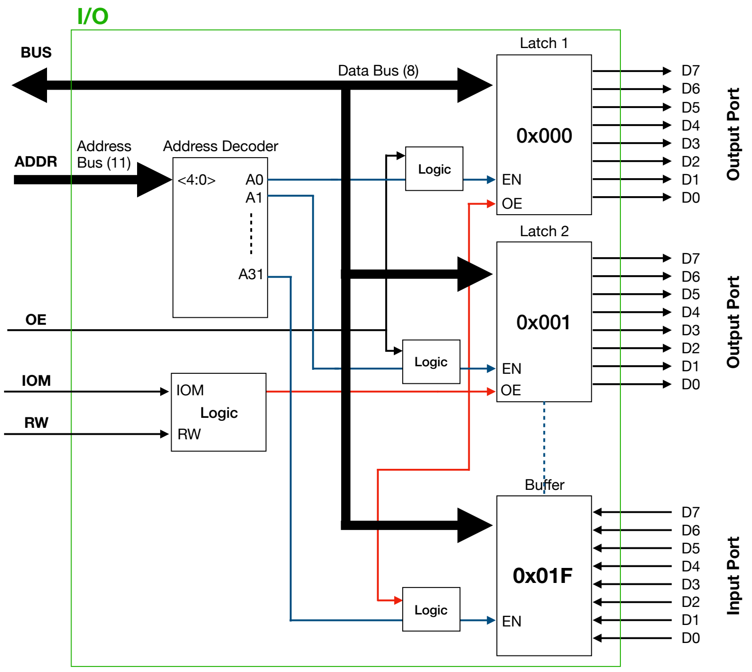

Figure 2. I/O Memory Organization

2. To write data to an output device, data will be set to the BUS, then address at MAR be loaded to ADDR. ADDR will be decoded and the correct latch will be enabled then the 8-bit data on the bus will latched. Reading from the I/O module, requires the same steps but this time the data on the buffer will be loaded to the BUS. Refer to the following source code for the declaration of the I/O memory.

```
…
/* declaration of IO memory (latches and buffers) */
unsigned char iOData[32];    // 8-bit data x 32 (16 latches and 16 buffers)
…
```

The declaration above are the combined 16 latches and 16 buffers. Below is sample code for writing and reading data to/from the latches. Do not use anymore the declaration “unsigned char iOMemory[32]”.

```
…
void IOMemory(void)
{
    if(OE)                       // check if output is enabled
    {
           if(RW && !IOM)    // check if memory write and IO Memory access
           {
               if(ADDR=>0x000 && ADDR <= 0x00F)  // check the address if valid
                      iOData[ADDR] = BUS;          // write data in BUS to IO Memory
           }
           else
           {
               if(ADDR=>0x010 && ADDR <= 0x01F)  // check the address if valid
                      BUS = iOData[ADDR];          // load data to BUS
           }
     }
}
…
```

Notice that the IOMemory( ) does not return anymore the data but directly load the data to the BUS. This follows closer to the logic of the Main and IO memory. Revise the MainMemory( ) function to load directly the data to the BUS instead of returning it. The function prototype will be *"void MainMemory( )".* Modify the following in the control unit to incorporate the changes made to the MainMemory( ) and IOMemory( ):

a) Instruction Fetch
b) Read from Memory (RM)
c) Read from I/O (RIO)

3. Based on the IO configuration in Fig. 3, write an assembly program that will send data to latch with address 0x001 and subsequently read buffer with address 0x01F. Using the SWAP (part 1) and WM instruction, move the read data from I/O to memory address 0x406. Verify the data by reading back memory address 0x406 using the RM instruction. This will demonstrate the output write and input read. Write your code using the semantics similar to the assembly code in Exercise 4. Convert the code to machine code and write to the memory using the method in part **2**, section **k**. Test your program and validate it using the following example:

   Data written to 0x01: 0xCA (11001010)
   Data read from 0x1F: 0x53 (01010011)

   Write the following function in your code to simulate the I/O connection in Fig. 3.

```
void InputSim(void)
{
    unsigned char data;

    for(i=7; i>=0; i- -)
    {
        /* load source data */
         data = iOData[0x001];
        /* shift bit to LSB */
         data = data >> i;
        /* mask upper bits */
         data = data & 0x01;
        /* position bit */
         data = data << (7 - i);
        /* write bit to dest buffer */
        iOData[0x01F] = iOData[0x01F] | data;
    }
}
```

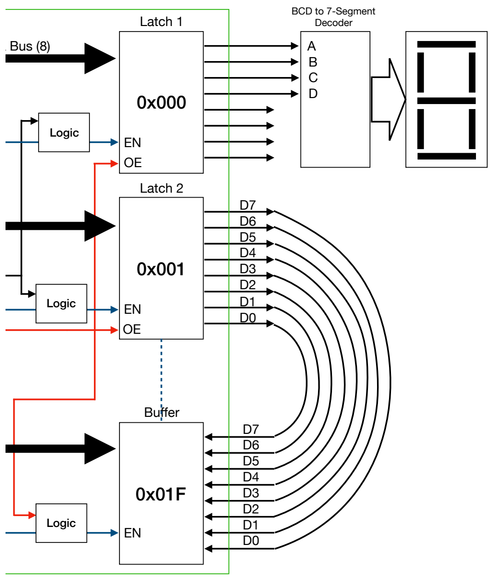

Figure 3. I/O Example

The function above will simulate the input connectivity of the buffer at address 0x1F. Temporarily call this function during the write to I/O instruction (WIO). After verifying the the assembly program, you can comment or delete the function call in the WIO operation in the control unit.

4. Assuming that a 7-segment display with a BCD to 7-segment decoder (see Fig. 3) is connected to a latch (0x00), write an assembly code that will display the value of a decade count-down counter. The code for the 7-segment decoder and display can be downloaded in the activity page in Canvas. Copy the function `SevenSegment()` to your code. Call the `SevenSegment()` every time a write to I/O (WIO) is performed. The function will display the corresponding data on latch at address 0x000. The program will count down from 9 to 0 once, then terminates.
5. Write your assembly code on a text editor including the machine code and save "CountDown.txt".
6. In your emulation program, initialize the program memory with the assembly program in #5.
7. Submit source code for the emulation program as "CPU+MEMORY+IO.c" and the assembly program "CountDown.txt" in Canvas. Follow the correct order of the submission.

Assessment

Criteria	Outstanding (10 pts)	Competent (8.5 pts)	Marginal (7.5 pts)	Not Acceptable (5 pt)	None (0 pt)
Logic	CPU+Memory+IO logic demonstrated clearly and following exactly the architecture.	CPU+Memory+IO logic demonstrated clearly with slight deviation to the architecture.	CPU+Memory+IO logic demonstrated clearly with several deviations the architecture.	CPU+Memory+IO logic was not demonstrated, did not adhere to the architecture.	
Emulation	Emulation of the CPU+Memory+IO is very close to the actual.	Emulation of the CPU+Memory+IO is slightly close to the actual.	Emulation of the CPU+Memory+IO is very far from the actual.	Emulation of the CPU+Memory+IO is not demonstrated.	
Coding	Coding is neat, systematic, logical and followed accepted coding standards.	Coding is logical and somewhat followed some coding standards.	Coding is logical followed little coding standards.	Coding is messy and did not follow any coding standards.	

Copyright Information

Author: Van B. Patiluna (vbpatiluna@usc.edu.ph)
Contributors: none
Date of Release: April 12, 2021
Version: 1.0

Change log:

Date	Version	Author	Changes
November 8, 2019	1.9	Van B. Patiluna	Original laboratory guide for CpE 415N.
April 12, 2021	1.0	Van B. Patiluna	Initial draft.