ESE381 Embedded Microprocessor Systems Design II

Spring 2022, K. Short revised February 6, 2022 5:15 pm

Laboratory 02: Bitwise Logical Operations in C
To be performed the week starting February 6th.

Prerequisite Reading
(All items are posted on Blackboard.)

1. 330 ohm Resistor Network CTS SIP Data Sheet
2. Bargraph 10 Element LED Data Sheet

Overview
The purpose of this laboratory is to provide you with experience in configuring port pins and writing simple programs that require bit manipulation using C's bitwise logical and shift operators.

List of Design Tasks
Design Task 1: A Simple 8-bit Parallel Input Port and a Simple 8-bit Parallel Output Port
Design Task 2: Software Read-Modify-Write
Design Task 3: A Simple Combinational Function Using Bitwise Logic Operators and Shifting
Design Task 4: A Simple Combinational Function Using Named Bits

Design Tasks

Design Task 1: A Simple 8-bit Parallel Input Port and a Simple 8-bit Parallel Output Port
An 8-bit parallel input port and an 8-bit parallel output port are to be implemented. The input port is connected to one side of an 8 position SPST DIP switch. The other end of each switch is connected to ground. The input pins for this port must have their internal pull-up resistors enabled in software to convert each switch's position to a valid logic level. Thus, when a switch is open it will be read as 1 and when it is closed it will be read as a 0.

The output port drives 8 of the LED elements of a 10 element LED bargraph. The top two elements of the bargraph are not used. The cathodes of the bargraph LEDs are connected to the corresponding pins of the output port. Current limiting for the LEDs is accomplished using a 330 ohm SIP resistor network.

The basic operation of the system is to continually read the switches and, based on the logic levels read, turn ON the LEDs corresponding to the switches that are open (logic 1 output from the switch) and turn OFF the LEDs corresponding to the switches that are closed (logic 0 output from the switch).

We would prefer to have a sequence of eight consecutive pins to make up an 8-bit port. This makes the processing of the data easier. Using pins PA7 through PA0 for the input port would be ideal. This would be possible if you were designing our own printed circuit board. However, you are using the AVR128DB48 Curiosity Nano board. On this board, the board's pins labeled PA1 and PA0 are actually wired for their alternate function, which is to provide the two connections for a high frequency external crystal. The board's printed circuit traces are actually connected to a high frequency external (but on-board) crystal. You would have to physically cut these printed circuit board traces and make connections to PA1 and PA0 to use these port pins. You are **NOT** going to do that. Instead, you must use pins PC1 and PC0 in place of PA1 and PA0, respectively, for the input port. So, the 8-bit input port will consist of pins: PA7, PA6, PA5, PA4, PA3, PA2, PC1, and PC0.

For the output port use PD7 through PD0.

Design the required external logic. Draw a simple schematic for your logic. Bit 7 should be at the top of both the DIP switch and the 8 LEDs of the bargraph that are actually used. Do not show the Curiosity board symbol on your schematic. Just show to which AVR128DB48 pins the switch and LED circuits connect.

You must write a program to read the input port and for the input switches that are open (logic 1) turn ON the corresponding LEDs and for the input switches that are closed (logic 0) turn OFF the corresponding LEDs,

However, you are to use only the very basic port capability of the AVR128DB48. Thus, you are constrained to only use registers of type DIR, IN, OUT, and PINnCTRL. Write a program named `parallel_in_parallel_out_flat` that **uses only flattened names** (those with underscores) for the ports and other special function registers. Do not declare any variables in your program. Use C bitwise operators to create the output port value from the composite input port.

Write a second program named `parallel_in_parallel_out_stuct` that uses only structural names (those with dots) for the ports.

Submit your schematic and source files for the two programs as part of your prelab.

Design Task 2: Software Read-Modify-Write

An extremely common task in embedded system design requires changing the value of a group of bits in a register without changing the values of any other bits in that register. This is accomplished by reading the entire contents of the register, modifying the group of bits that need to be changed while leaving the other bits unchanged, and then writing the result back to the register. This set of operations is referred to, appropriately, as read-modify-write. Read-modify-write was originally accomplished only by using a sequence of software operations. Some more advanced microcontrollers, like the AVR128DB48, have SFRs that allow this to be done using special hardware and a single instruction.

At this point, we are interested only in accomplishing read-modify-write using software. The software approach can be applied to any register or memory location that can be read and written. The modification is to be done using masks and C's bitwise operations. Fortunately, a port configured as an output port can also be read, so that it is easy to do this for output ports.

This task uses the same hardware as used in Task 1 along with SW0 on the Curiosity board. You must write a program named `read_modify_write_sftw_sw0` to accomplish a read-modify-write operation. The read-modify-write operation will be done on VPORTD_OUT.

To be able to verify whether our read-modify-write operation works, we need to have a known value in VPORTD_OUT to start. So, when this program first starts execution, it reads the input port of Task 1 and uses that value to control the LEDs of VPORTD_OUT, just as in Task 1. After that, whenever SW0 is pressed, the program uses the values of bits 3 down to 0 of the input port to control LEDs 6 down to 3 of the output port. The other LEDs of the output port must not have their values changed. Do not use variables in your program. You do not need to debounce SW0.

Submit your source file for your program `read_modify_write_sftw_sw0` as part of your prelab.

Design Task 3: A Simple Combinational Function Using Bitwise Logic Operators and Shifting

The C bitwise logical operators that you have studied provide all the logical operations needed to carry out Boolean function computations. So, it would seem that you can compute any Boolean function of a set of inputs, and we can. However, the C bitwise operators operate on an entire byte at a time and can be cumbersome and inconvenient when we want to operate on individual bits in a register to compute Boolean functions. This task will illustrate that fact. There are better approaches for computing Boolean functions. like using structures that have bit field members. You will use that approach in the next task.

The function that you must implement for this task is basically a 3-input XOR gate or odd parity checker. This is a simple combinational function as expressed by its truth table.

XOR Function

C	B	A	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The output of this function is a 1 whenever the number of inputs that are 1s is odd. Zero inputs being a 1 is considered an even number of 1s. The inputs in the truth table are C, B, and A. You

must associate those inputs with pins PA7, PA6, and PA5, respectively. The output is F. You must associate the output with pin PD7.

The best approach to writing this program is to derive the canonical sum-of-products Boolean expression for this function. Then use shifting and the bitwise logical operations to compute the output.

The hardware for this design is the same as for Task 1.

Write a program named `xor3_logic_ops` that implements this function using C's bitwise logical operators (including the logical shifts). You must use only those operators. Do not use unions, structures, or pointers.

Submit your source file for this program as part of your prelab.

Design Task 4: A Simple Combinational Function Using Named Bits

In Lecture 4 we discussed how to use a union of an `uint8_t` and a bitwise structure to allow you to give names to each bit in a byte of data. Using this approach it is relatively easy to write (and compute) a sum-of-products expression in C. You are to use this approach to implement the function described in Task 3.

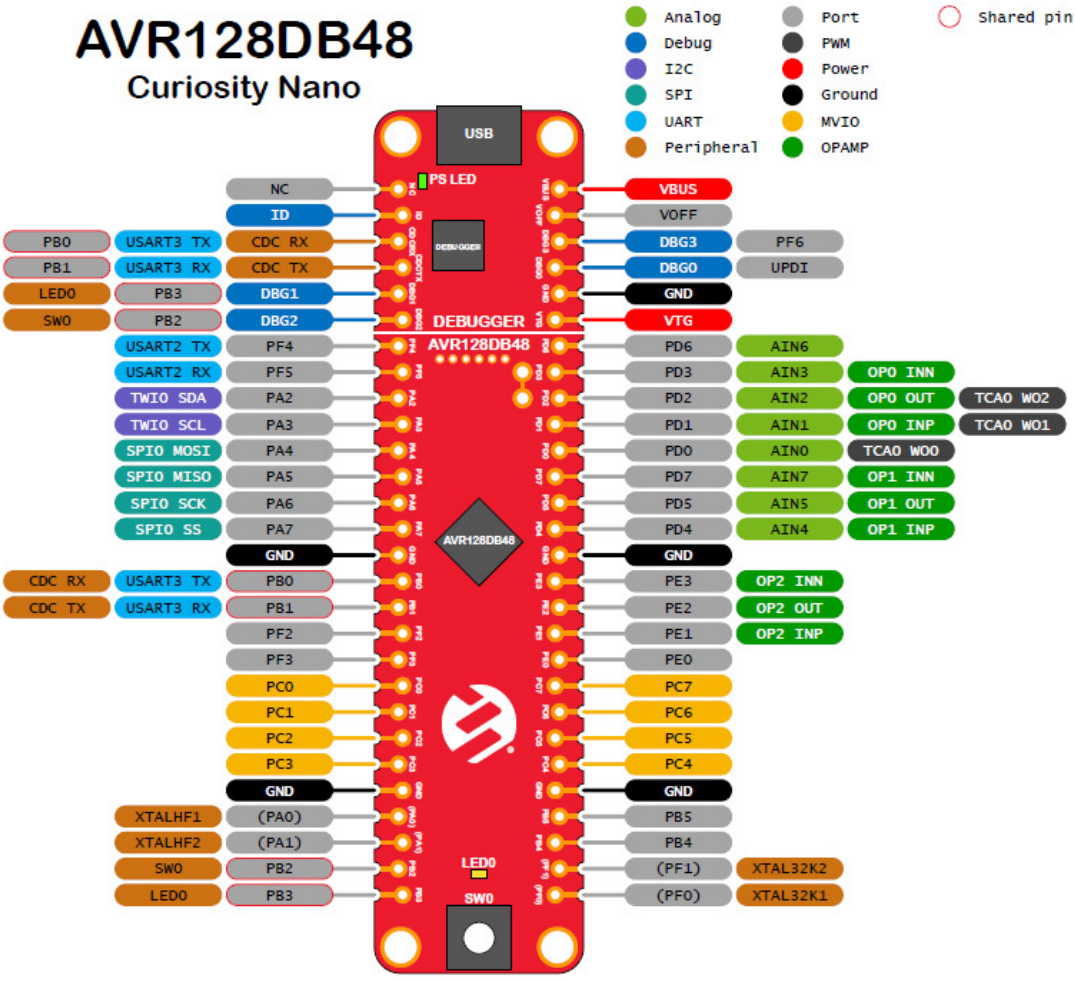
You can use the header file data.h from that lecture or modify it to have more descriptive names for this particular function.
The hardware for this design is the same as for Task 1.

Write a program named `xor3_named_bits` that implements this function using named bits via the bitfield approach.

Submit your source file for this program as part of your prelab.

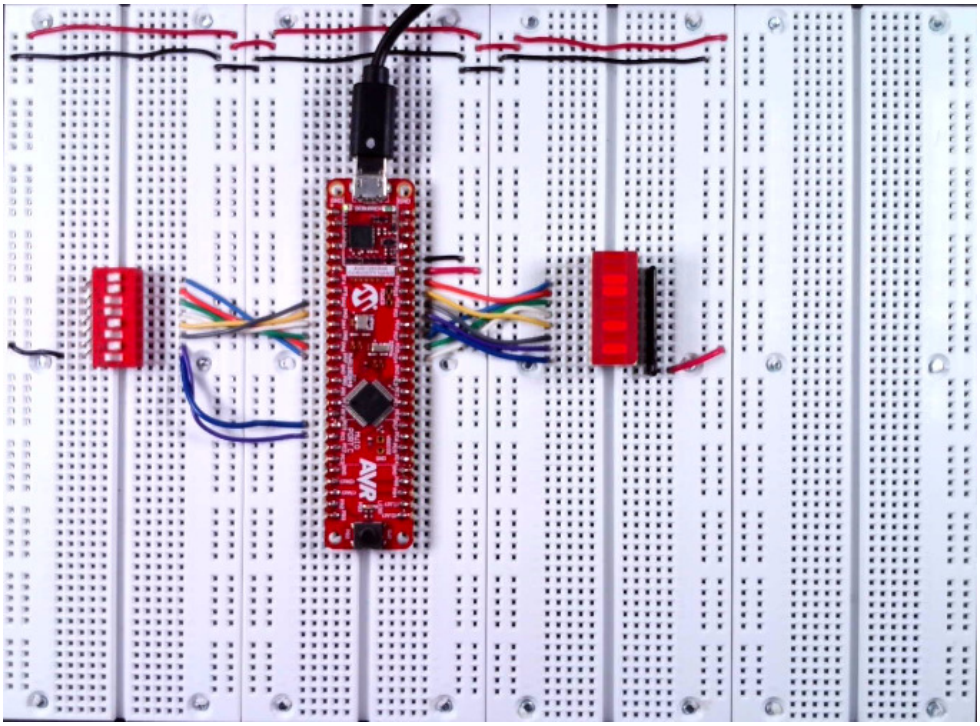
Laboratory Activity

During the in-laboratory portion of this laboratory you will be connecting external devices to the AVR128DB48 Curiosity Nano board.



Laboratory Task 1: A "Simple" 8-bit Parallel Input Port and a Simple 8-bit Parallel Output Port
Your breadboard consists of four smaller breadboards. Place the Curiosity board in the middle of

the second smaller breadboard from the left, as shown below.



Build your circuit. The pins on the Curiosity Nano that provide the bits of PORTA, PORTC, and PORTD are not in sequence, so be very careful not to use the wrong pin(s).

Wire the DIP switch so that the top switch is connected to pin PA7 and the bottom switch is connected to PC0. A DIP switch positioned to the right must generate a 0 output to the corresponding microcontroller input pin. Wire the bargraph LED so that the eighth LED (third from top) is connected to PD7 and the bottom LED is connected PD0. In other words, the top two LEDs of the 10 element LED bargraph are not to be used.

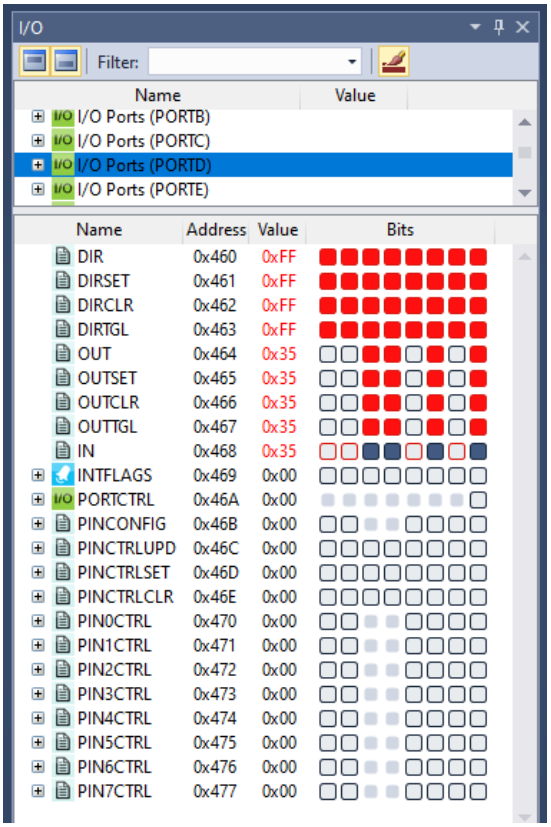
Have a TA check your wiring before connecting the Curiosity Nano board to the computer.

Connect the Curiosity board to your computer. Power up your circuit.

Follow the procedure from Laboratory 01 to create a new project. For the project's name use `parallel_in_parallel_out_flat`.

Enter your program. Set the compiler optimization to -Og. Build the program. Prior to single-stepping each instruction, determine what changes you expect to see in the values of the port registers and other special function registers in your program. Single step each instruction in the program. After each single step confirm that the changes you expected occurred in the debugger's I/O Win-

dow.



After you have single stepped through each instruction, run the program full speed. Verify that your system functions properly at full speed.

Have a TA verify that your program performs as required. Get the TA's signature.

Repeat this process for your program `parallel_in_parallel_out_struc`.

Have a TA verify that your program performs as required. Get the TA's signature.

Laboratory Task 2: Software Read-Modify-Write
Follow the procedure from Laboratory 01 to create a new project. For project name use `read_modify_write_sftw_sw0`.

Enter your program. Set the compiler optimization to -Og. Build the program. Prior to single-stepping each instruction, determine what changes you expect to see in the values of the port registers and other special function registers in your program. Single step each instruction in the program. After each single step, confirm that the changes you expected occurred in the debugger's I/O Window.

After you have single stepped through each instruction, run the program full speed. Verify that your system functions properly at full speed. If your program reads VPORTA_IN or VPORTC_IN

immediately after the associated pull-ups are enabled and you get an incorrect starting value when your program is run at full speed, some of the pull-ups may not have had time to take effect. That is, the voltages at some of the associated pins that should be pulled up to 1s have not had time to charge the parasitic capacitance associated with each pin and reach their logic 1 values. In this case, just repeat the first instruction that reads the input port. With the input instruction repeated, the first copy of that instruction acts as a delay, hopefully giving all of your pull-ups time the take effect.

Have a TA verify that your program performs as required. Get the TA's signature.

Laboratory Task 3: A Simple Combinational Function Using Bitwise Logic Operators and Shifting
You will be using the hardware from Task 1.

Follow the procedure from Laboratory 01 to create a new project. For project name use `xor3_logic_ops`.

Enter your program. Set the compiler optimization to -Og. Build the program. Prior to single-stepping each instruction, determine what changes you expect to see in the values of the port registers and other special function registers in your program. Single step each instruction in the program. After each single step confirm that the changes you expected occurred in the debugger's I/O Window.

After single stepping through each instruction, run the program full speed. Verify that your system functions properly at full speed.

Have a TA verify that your program performs as required. Get the TA's signature.

Laboratory Task 4: A Simple Combinational Function Using Named Bits
You will be using the hardware from Task 1.

Follow the procedure from Laboratory 01 to create a new project. For project name use `xor3_named_bits`.

Enter your program. Set the compiler optimization to -Og. Build the program. Prior to single-stepping each instruction, determine what changes you expect to see in the values of the port registers and other special function registers in your program. Single step each instruction in the program. After each single step confirm that the changes you expected occurred in the debugger's I/O Window.

After single stepping through each instruction, run the program full speed. Verify that your system functions properly at full speed.

Have a TA verify that your program performs as required. Get the TA's signature.

Leave the circuit you have constructed on your breadboard insert, it may be used in later laboratories.

Questions

1. You want to verify the implementation dependent effect(s) of the C programming language's >> operator on a signed value. Write a very short program to do this.

2. What is the difference between the & operator and the && operator in C?

3. What is the difference between declaring a variable with `uint16_t` and `unsigned int` in a C program. How would you find the width of an `unsigned int` in our C compiler version.?

4. Compare the assembler code generated by the compiler using an optimization level of None (-O0) and of -Og for the first program you wrote for Task 1. Does one program result in more object code that the other?

5. Compare the assembler code generated by the compiler using an optimization level of -0g for the two programs you wrote for Task 1. Does one program result in more object code that the other?