

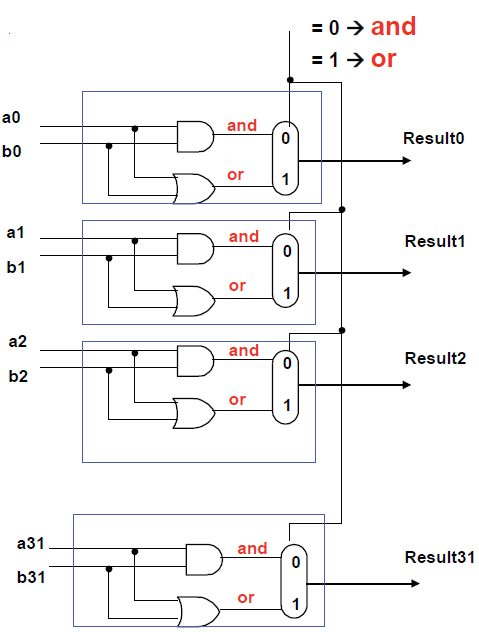
Computer Organization – Lab Exercises

Demonstrate your solutions and get them evaluated by one of your TAs in the respective lab session.

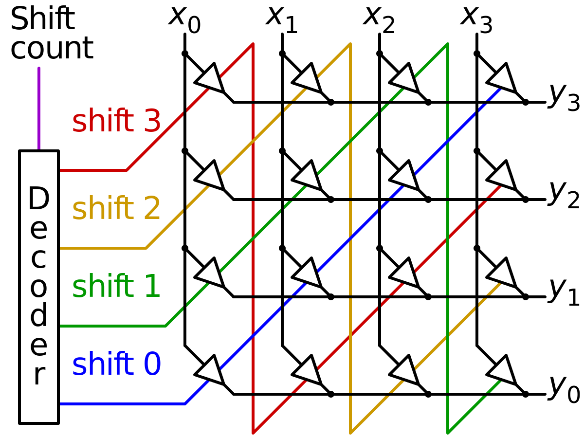
Completion of assignment does not guarantee marks, you need to convince the TA that you did own the code by answering TA’s questions!

**Logisim Exercises:**

1. Install Logisim. Complete Beginners’ Tutorial and Sub-circuits provided in the *Help→User Guide*
2. Using Logisim implement the following:
   * 8-bit AND logic
   * 8-bit OR logic
   * 8-bit NOT logic
   * 8-bit EXOR logic
3. Control every output of Exercise-2 by a 1-bit *tristate logic*
4. Create simple 4-to-1 multiplexor. This one is due for next lab.
5. Create a single selectable implementation of all the logic completed in previous exercise, similar to the AND-OR logic below.



1. Using Logisim implement the 4-bit barrel shifter with the corresponding 2-bit decoder as shown in the following figure. Note, barrel shifter is developed with an array of tri-state gates. The **decoder** part of the barrel shifter is operating with 2-inputs and 4-outputs – the output of 0 (Blue), 1 (Green), 2 (Yellow), and 3 (Red) are produced for the **decoder input (Shift Count)** values of 00, 01, 10, and 11 respectively. When the decoder output is 0 (Blue), the output of the Barrel Shifter is unshifted. That is, Y0 = Y0, Y1 = X1, Y2 = X2, and Y3 = X3. When the decoder excites Green line, X0X1X2X3 are shifted exactly by one bit position. Thus, Y1=X0, Y2=X1, Y3=X2, and Y0=X3. And so on. You could make use of the decoder that you built in Exercise 5.



1. Create 8-to-1 Multiplexor. Using this implement 3-input AND gate. Then using the same 8-to-1 Mux implement 3-input OR gate.
2. Create a full adder then using this create a 4-bit Ripple Carry Adder using Logisim
3. Multiplier assignment (optional): <http://www.eecg.toronto.edu/~ahouse/mirror/engi3861/multiplier_example_web.pdf>

**File I/O Exercises:**

1. Learn the features offered by C Programming File I/O library
2. Review the contents of ASCII table – Identify the ASCII values for the following: *New Line, Carriage Return, Bell, NULL, Back Space, and Space*.
3. Develop a simple C program that Opens a file in Read mode and dumps the content of the file to STDOUT - line by line. Keep the file size small, have exactly 3 lines. How did your code handle *end of file* condition?
4. Download and install *hex-editor-neo* (it is also listed in my *Box* cloud folder). Using this tool view the content of the file that you used in Ex 11. Identify the ASCII values of *Carriage Return* and *New Line* characters. Understand their history and role!
5. Develop a simple C program that creates a folder with a single file in it. The file contains your name on one line and your roll number on the next line.
6. Develop another simple C program that opens the file that you created in Ex 3 and inserts the following line in after the 1st line: “2nd Year CED/COE. Assumption: you don’t know the content of the 2nd line in the file.
7. Modify the above program as follows: The line to be inserted is taken from STDIN.
8. Is there any use case for *ASCII Back Space character* in File I/O – explain.
9. Open a MP3 audio file with HEX dump tool, review the content, specifically at the top and end of the file.

**ARMSIM Exercises:**

1. Install ARMSim on your laptop. Do arithmetic and logical operations on the contents of two registers. Learn the features of ARMSIM, no need to exercise them all for now.
2. Write an ARM assembly program to WRITE to a file. Write 1 line of text to this file. Make sure you close the file after you write to it.
3. Write an ARM assembly program to read from the file that you created in Problem 1. Open this file in READ mode, READ and dump its contents to STDOUT.
4. Write an ARM assembly program to receive 3 different Integers (Comma separated) inputs from a file. After receiving all the three values, print them on STDOUT on separate lines**.**
5. In some single threaded embedded system applications, we use timers built using assembler sub-routine. This type of routines is sufficient and reasonably accurate for many applications. Develop a subroutine “*Stimer*” that can create 1000 (approximately) Clock cycles delay. Using this subroutine write another subroutine “*Ltimer*” that can create delay which are multiples of 10000 (approximately) cycles. The *Ltimer* is simple and no need for other timers. The *Ltimer* can be used to create periodic scan of I/O devices or any periodic activity.
6. Write ARM assembly code that can make the two red LEDs in *Embest Plugin of ARMSIM* to glow alternatively at **observable** rate. Your code should be user configurable for various rate. Use timer developed in problem 4.
7. Write ARM assembly code that can make the 8-segment display to go from 0 to 9 at an observable rate. Your code should be user configurable for various rate.
8. Write an ARM assembler program that increments the elements of a vector of size 100. After incrementing, copy the vector to a different part of the memory and add the two vectors and store the result where first vector was in. Repeat the above procedure three or more times once with Multiplication (instead of ADD) by 5, once with divide by 4, and finally by adding 16384. Finally copy the vector to a file. Optimize code for execution time using appropriate cache configuration and choosing the appropriate assembly instruction. Solution with best execution time will be recognized suitably.
9. Plot your observations from problem 7 using Excel Spread Sheet.
10. Run the program that you wrote for Exercise 7 with ***unified cache with direct mapping and Write back****.* For the following combination of cache size and block size measure the *miss rate*. Plot your results and provide your observation. **Due on March 3rd or 4th.**

|  |  |  |
| --- | --- | --- |
| cache Size | Test 1 | Test 2 |
| Bock Size | Block Size |
| 128 | 16 | 32 |
| 256 | 16 | 32 |
| 512 | 16 | 32 |
| 1024 | 16 | 32 |
| 2K | 16 | 32 |
| 4K | 16 | 32 |
| 8K | 16 | 32 |

1. Run the program that you wrote for Exercise 14 with ***split cache with direct mapping and Write back.***For the following combination of cache size and block size measure the hit and *miss rate*. Plot your results and provide your observation.

**Verilog & GTKWave Exercises:**

1. This exercise requires you to install Verilog development tools on your laptop and demonstrate Verilog “Hello World” Code. Use Icarus Verilog tools.
2. Demonstrate Verilog module with test bench using Icarus. This does not require you to write any code. Download code from Internet. Make sure you understand this code.
3. Develop Verilog code for behavioral design of ***Full Adder*** along with an appropriate Testbench. Demonstrate your code. Try all types of test benches: Simple testbench ♣ Self-checking testbench ♣ Self-checking testbench with test vectors
4. **GTKWave is a simple, lightweight application designed to read and display waveform files. The program is designed to help the IC designed view the signal transitions on the established timeline, as well as the interaction of signals from different designs. It can easily read and translate the hardware description language in waveforms.Follow the instructions in this web site to complete the Simulation with gtkwave tool. Follow the instructions in this web site to complete the Simulation with GTKWave tool**

<https://www.swarthmore.edu/NatSci/mzucker1/e15_f2014/iverilog.html>