Brown University
School of Engineering
ENGN1640 Design of Computing Systems
Professor Sherief Reda
LAB 03 (100 points).
Due in lab to TA by Friday, March 4.

(1) [50 points] Your objective is to design a programmable stack-based calculator, which is also known as a postfix calculator, that can do 8-bit integer arithmetic. You have to include five instructions:

Instruction	**Meaning**
push *value*	push the immediate *value* into the top of the stack.
add	pops the two top elements in the stack and push the result of the addition to the stack.
sub	pops the two top elements in the stack and push the result of the subtraction to the stack.
mult	pops the two top elements in the stack and push the result of the multiplication to the stack.
halt	**Stops execution and displays the top of the stack on the 7 segment display in decimal.**

Here are two examples that show the operation of the calculator

Example A: 8-5*3	**Example B: 5 + ((1 + 2) * 4) – 3**
push 8 push 5 push 3 mult sub halt	push 5 push 1 push 2 add push 4 mult add push 3 sub halt

A. Design an instruction set format of this machine.
B. Translate the two example assembly programs into machine code using the format you created in part (A)
C. Write the Verilog code for this machine.
D. Implement the stack machine on the DE2 board. Assume the following:
   - The machine has a stack of exactly 10 8-bit entries (users will never create programs that require more than 10 items in the stack).
   - One LED will turn on if there is an arithmetic overflow.
   - The machine code for programs is stored in a ROM.
     i. **For the DE2 board**, You will need to instantiate a ROM from the megafunction wizard and initialize its content from a .mif file.

The .mif file can be created and edited by choosing File → New → Memory Files → Memory Initialization File. Interface the ROM to your CPU module using appropriate address, data and control busses.

ii. **For your Verilog testbench**, the ROM will be part of your testbench and your testbench will supply the instructions over the busses to the CPU and will receive the output directed for the 7-segment displays.

- When the machine boots, it starts executing instructions from address 0 until a halt is encountered.

E. Please report:
   - your Verilog testbench and simulation results using $display or $monitor outputs giving the results from the execution of the two examples;
   - the total logic and routing resources using by your circuit;
   - RTL circuit view;
   - the actual critical path of your circuit and the propagation delay of your circuit critical path.

F. 10 points of this question will be determined upon your final design size in terms of LEs. You should not use any RAM/ROM elements to substitute for logic functions implemented in LEs. $10 \leq 160$, $9 \leq 170$, $8 \leq 190$, $7 \leq 200$, $6 \leq 220$, $5 \leq 250$, $4 \leq 400$.

(2) [10 points] The NOR instruction is not part of the ARM instruction set, because the same functionality can be implemented using existing instructions. Write (and validate using the ARM simulation) a short assembly code snippet that has the following functionality: R0=R1 NOR R2. Test your code by NORing 0x7F with 0x10 and NORing 0x7F12 with 0x1017. Use as few instructions as possible.

(3) [15 points] Write an ARM program to reverse (not inverse!) the bits in a register (e.g., 10111→11101). Assume the register of interest is R0. Initialize R0 to test your code. Use the ARM simulator to test your code.

(4) [25 points] Write ARM code that tests whether a given input string is a palindrome. The procedure should set R11 to 1 if the string is palindrome; otherwise, 0. (Recall that a palindrome is a word that is the same forward and backward. For example, the words “wow” and “racecar” are palindromes). Write an assembly program that test the procedure by calling it three times using the strings “wow”, “racecar”, “x” and “sunshine” as inputs. Use the ARM simulator to test your code. To store the string in your program, include the following line at the end of the program `test_string: .asciz "wow"` which creates **zero-terminated string** in the memory and then you can load the starting address of the string in a register by using `LDR R1, =test_string.` Recall that to load the individual character you can use LDRB. Do not assume that the default values of registers are zero; make sure to initialize to zero if that is the intent. Use the fewest number of static instructions. 5 points will be allocated for the usage of minimum number of instructions. 5 <=15 instructions, 4 <= 17 instructions, 3 <= 19 instructions, 2 >= 20 instructions.