MONASH UNIVERSITY

Department of Electrical and Computer Systems Engineering

ECE2072 ASSIGNMENT

Due Date:	Friday 21st October 2022 11:55 pm.
Submission:	Online submission using a Moodle assignment. Individual submissions are required.
Assessment:	10% final mark, marked out of 60. Plagiarism or collusion will result in zero marks for the originator and the copier. A marking rubric is provided at the end of this document.
Late Policy:	Late submission will incur a penalty of 10% for each day late. Assignments submitted more than a week late will not be assessed. Apply for special consideration for late submission due to documented serious illness/circumstances outlined here: **https://www.monash.edu/exams/changes/special-consideration**

Assignment Feedback and Grading

The objective of the assignment is to provide an opportunity to design logic circuits and give you feedback on your understanding of the design of FSMs and their implementation in a Hardware Description Language.

A solution that appears to work will not necessarily receive full marks. A good design is more than a "working solution" - it is clear, uses consistently labelled signals, is concise, efficient, easily maintained and understood, well documented and states assumptions where necessary.

Use of high level behavioural Verilog style will be rewarded over low level Boolean logic expressions and individual flip-flop instantiations. Ensure you understand the ***Good Style Guide and Marking Rubric*** at the end of this document. Students who have participated and understood the ECE2072 laboratory experiments will have the skills to complete this assignment.

Submission Requirements (via campus specific dropbox):

Please use Moodle to submit the following ***five files*** and ***one video*** as a **zip:**

1. A single PDF file: containing your written answers to each part.
    - The questions to be answered are listed under "REPORT Part 1", etc, at the end of each section.
    - Each part must start on a ***new page*** and must be in the ***correct order: Part 1, Part 2, and then Part 3***.
    - Verilog code be "pretty printed" in colour - instructions on producing this will be on Moodle.
2. Three .sof files: your three binary files, forming the solution for each part:
    - They should be labelled *MyPart1.sof, MyPart2.sof* and *MyPart3.sof*
3. Video: Part of A video of 3 minutes or less of **you** demonstrating **your code** working on an FPGA
    - The video should have a length of 3-4 minutes
    - You must appear in your video
    - The (programmed) FPGA must appear in your video
    - You should demonstrate the final functionality of your design (MyPart3.sof) and discuss:
        i. your understanding of its functionality
        ii. your design methodology
        iii. any design decisions you made
4. A final file should contain all your ***Verilog code*** named with your student ID as ID.*v*.
    - **Do not include the modules below the FSM module since these are not to be changed and will generate false plagiarism alerts.**
    - You will **not** be able to upload this file separately, you must submit it as a zip, or Moodle will not accept it.
    - If your ID is 12345678 the filename would be ***1234568.v*** - do ***not*** pretty print this file since it must be readable plain text.
    - This file will be checked for plagiarism electronically against all other code using the Measure of Software Similarity MOSS: https://theory.stanford.edu/~aiken/moss/

Please note that both providers and recipients of any plagiarised reports or parts of reports will be penalised via Faculty of Engineering disciplinary proceedings – plagiarism is treated very seriously and can lead to exclusion from the university. Repeat students should note that this assignment is different to previous years and, in any case, it is not permitted to submit code submitted previously for assessment.

Introduction to the Design Problem

The assignment's aim is to design a ***cognition timer*** that measures the time for a user to process a request to complete a simple task. Cognition times are important for assessing our ability process information quickly and accurately. To keep things simple here we use a basic reading, processing, action planning and implementation task as follows: A pair of octal digits is displayed, each in the range 0-7. The user adds the numbers and enters the result as a 4 bit binary number using slide switches SW[3:0] on the DE2 board. The user indicates the answer is complete by changing the *submit* slider switch SW[17] to 1. The time from displaying the two octal digits to the switches are submitted is then displayed in 100 milliseconds. Should a wrong switch value be detected after SW[17] is 1 or the user does not respond quickly enough, an error is shown instead of the time. More details are provided below. To aid your understanding of the assignment requirements, *DemoPart1.sof, DemoPart2.sof and DemoPart3.sof* demonstration files are provided on Moodle for downloading to a DE2 board.

The parts 1 and 2 are intended to be an easy way to familiarise yourself with some of the basic Verilog modules that have been provided on the Moodle site. Part 3 is more challenging and requires you to develop a "puppeteer" style finite state machine (FSM) that controls the whole cognition timer design.

Part 1: A Simple Timer Counter and Display

Design a timer counter that provides a running display in units of 100 milliseconds. On startup or downloading of the *.sof* file the timer should initially display 0000 and then increment every 100 milliseconds until it reaches the decimal number composed of the the last 2 digits of your ID number + 100. For example, if your ID number is 98765432, your counter timer counts up to 0132 in 13.2 seconds. The counter then wraps back to 0000 and the whole process repeats. Your design should ensure each output value, including the largest (0132 in our example) persists for 100 milliseconds each.

Start by creating a new Verilog module named *MyPart1*for your design and instantiate this within the top level module named *assign2022* – ie the module that connects to the standard board signals such as CLOCK_50, SW[17:0] and KEY[3:0]. Within *MyPart1* instantiate the *Timer* and *DisplayTimerError* modules and add some logic of your own. Use SW[17:16] to enter error codes (see the provided module *DisplayTimerError*), SW[15] to enable the display, and SW[14] to freeze the output. Make HEX7 and HEX6 display the last two digits of your ID (rather than P1 as in the demo, so we know this is your demo!). Connect the reset iRst to ~KEY[0]. Test your design. It should be similar to *DemoPart1.sof* that counts to 0132 – check it out by downloading to a DE2 board.

REPORT Part1:

- **1.1:** Explain how your design has made the time duration spent displaying your last count value 100 millisecond.
- **1.2:** Include ***pretty printed*** Verilog just for your *assign2022* and *MyPart1* module – do not include other modules. How to “pretty print” is described on Moodle.

In your submission, you must include a file named *MyPart1.sof* (rename the file that Quartus produces for download to the FPGA).

Part 2: Display and Generation of Two Octal Digits

At the top level in the module *assign2022* instantiate the module *OctNumbersGenerationDisplay* that is provided. In part 2 the top level module *assign2022* should display a new pair of numbers, each in the range 0 to 7, on the seven segment displays HEX5 and HEX4. A new pair is generated on each *posedge iClk* when SW[0] is 1. The sum of the two octal numbers defines which switches the user should set in the final cognition timer design. We are not implementing the checking for the correct switches here in part 2. That is for part 3. We are only aiming to display and generate the two octal numbers in this part.

Use *SW[5]* connected to the port *iChooseRandID* to choose the source of the 6 bit number (ie two 3 bit octal digits). A random number generator is the source when *SW[5]*=1 and your ID number sequence is used when *SW[5]*=0. The ID sequence uses a module called *NextIDdualOctal*. You need to complete the implementation of *NextIDdualOctal* as follows:

- Module *NextIDdualOctal* has the standard *iClk* and *iRst* inputs for the system clock (usually CLOCK_50) and system reset (usually a synchronised ~KEY[0]). After a reset, followed by a request for the next ID digit *iNext*=1, the module output *oIDdualOctal* should produce, on the next *iClk* positive edge, two octal digits. That is two numbers in the range 0 to 7 are produced. The user will need to add these and enter the sum in binary in Part 3. In this part we have made the two octal digits depend on your ID number as the next point:

- Your ID number has eight digits. Each digit is used in turn - we move to the next ID digit when *iNext=1* and a *posedge iClk* occurs. The ID digit is converted to two octal digits by multiplying by 11 and truncating the result to 6 bits (that is the lowest 6 bits). In our example ID=98765432 the first would be 11*9 = 99 = 7'b1100011, truncated to 6 bits gives 6'b100011 which is displayed as the two octal digits 4 and 3. The second ID digit in our example should produce 11*8=88=7'b1011000, truncated to 6 bits is 6'b011000 or 3 and 0 octal digits in our example.

- On each successive positive *iClk* edge when *iNext=1*, the next ID digit*11 truncated to 6 bits should be produced on *oIDdualOctal*. Once the last ID number is reached, we start from the first again.

- Upon reset, the output is the ***last*** *ID digit*11 truncated to 6 bits,* as a consequence of the requirement that the first request (after a reset) for an ID digit corresponds to your ***first*** ID digit output. Note also that we would like the initial state after downloading the *.sof* file to be the reset state. The configuration file will, by default, initialise all flip-flops to 0, so the reset state should be all 0's.

Test your design by connecting the *iClk* of the module *OctNumbersGenerationDisplay* to debounced ~KEY[1] in the top level module *assign2022*. A manual clock allows you to slow down the action of *iNewOctNumsReq* and *iChooseRandID* to allow testing. A file *DemoPart2.sof* is provided for comparison and has an assumed ID = 98765432. When downloaded (or after a reset on a posedge clock), the output is initially 11*2=22 or in octal 6'o26. After compiling your design, ensure that no hardware multipliers are generated by your design, since all multiplications by 11 should be completed before synthesis by the compiler. Also ensure that you do not synthesise unnecessary flip-flops (called registers in Quartus) in your design.

REPORT Part2:

- **2.1:** How many states are there in a minimal state diagram for the FSM generated by a good solution for the *NextIDdualOctal* module*?*
- **2.2:** How many flip-flops should a minimal solution for module *NextIDdualOctal* generate? How many does yours generate?
- **2.3:** Include pretty printed Verilog for your *assign2022* and *NextIDdualOctal* modules – do not include other modules since ***they are not to be modified***.

In your submission, you must include a file named *MyPart2.sof* (rename the file that Quartus produces for download to the FPGA).

Part 3: The Cognition Timer

In this part the basic modules are put together into a module supplied called *CognitionTimer* which you need to instantiate at the top level. The *CognitionTimer* design consists of instantiations of the modules: *Timer, DisplayTimer, DualOctalGenerationDisplay* and *FSM.* Part 1 of this assignment has given you a good understanding of the first two modules and whilst Part 2 the third module. In this part 3 you need to control and monitor the first three modules using the finite state machine module called *FSM* that you need to complete yourself. It reads in the user input, keeps an eye on the time, outputs error codes (if the wrong switches are set or timing out with no switches set), outputs the cognition time, and displays the two octal numbers being requested at the appropriate times.

The FSM must be implemented as a Moore finite state machine – that is the outputs (ie the ports of the FSM module that begin with "o") must only dependent on the state and not the inputs. This requirement keeps your design simple and it guarantees that the outputs of FSM only change shortly after a *posedge CLOCK_50*. You need to work out what states are needed as you develop your solution. This will become clearer when you work through the sequence of steps that the *FSM* module produces:

0. Reset at start up *(iRst=1* or on download of a *.sof* file to the FPGA). This is usually where we initialise everything and since the FPGA down loads 0s into registers and flip-flops at the start, it makes good sense to use a state with a zero coding for the reset state.
1. Initially the display is blank for a fixed time of 2 seconds. This is the time the user is intensely watching for the appearance of the dual octal numbers to be added and put on the switches.
2. New dual octal numbers are requested by the FSM and displayed, prompting the user to respond by setting appropriate switches in SW[3:0] and then setting the *submit* switch SW[17].
3. The time from display to *submit* becoming 1, provided the correct sum of the octal numbers set, is the cognition delay and is displayed in 0.1 second units. The display persists until the submit switch SW[17] is reset by the user.

4. If the user sets the wrong switch(es) (error 1) or does not respond within 8 seconds (error 2) an error is displayed instead of the the cognition delay. In the case of error 2, the *submit* switch SW[17] needs to be set and then reset by the user to continue. For error 1, the error is removed when the submit switch SW[17] is reset by the user.
5. Go back to step 1.

It is good idea to develop the FSM module state by state and test as you add new states. Note that state transitions (ie from *state* to the *next_state* in the template code for *FSM* provided) can depend on inputs to the *FSM* and the current state. Use the two always blocks provided in the template code for the *FSM*, where the first block updates the outputs and next state with a combination logic circuit implemented as a case statement. Remember a combinational logic function represented by an always block needs to cover all input combinations to avoid inferred latches as discussed in the week 8 workshop. The second always block is provided and just updates the *state* from *next_state* on the rising clock edge.

You **must** use CLOCK_50 for the global clock, ~KEY[0] for the global reset, SW[17] for the submit switch, SW[16] for the iChooseRandID, SW[3:0] for the user entry of the sum in binary, HEX0-3 for the time/error display, HEX4 and HEX5 for the octal display of the numbers to be added on the switches. The other HEX outputs can be used for debugging such as observing the state of the FSM. The LEDR[17:0] should reflect the value of the SW[17:0] as provided in template code *assign2022*. A file *DemoPart3.sof* is provided for comparison and has an assumed ID = 98765432.

Helpful Hints (and time savers!): The slide switches SW[17:0] are not debounced. So when you slide a switch from 0 to 1, it can produce a sequence that bounces between 0 and 1 before settling on 1 (eg 010011000111010101011111). Likewise for 1 to 0. This will need to be considered in your design and testing of the FSM. In practice due to the mechanical design of switches there is an upper limit on the time between the first 1 and the last 0 in this sequence. You can design the FSM to cope with the switch bounce with minimal overhead and no extra modules.

REPORT Part3:

- **3.1:** Include pretty printed Verilog for your *FSM* module. Ensure that you document the Verilog code. This must include a description of each state and the use of parameters with brief state names that describe each state. State S1, S2 etc are NOT acceptable. Clearly the states need different values and the number of states will determine how many bits are needed for the state. All your state bits must be contained in the Verilog signal *state* in your code.

In your submission, you must include a file named *MyPart3.sof* (rename the file that Quartus produces for download to the FPGA).

Good Style Guide

Follow these guidelines when writing Verilog:

1. Use **[N-1:0] ordering** of multi-bit signals unless there is a good reason to use unconvention ordering.
2. **Non blocking** assignments <= should be used in posedge clock always blocks. Each bit in <= assignment can synthesise a flip-flop.
3. Do **not mix non blocking and blocking** assignments in the same always block.
4. Do **not drive the same signal in two different blocks** within the same module.
5. **Cover all input combinations** for defining combinational logic from always blocks. A default assignment at the top of the block is a good way to do this.
6. Think about whether **don't cares** should be used in the default case statements.
7. **Use synchronous resets** unless there is a good reason to violate timing constraints with asynchrounous resets.
8. **Synchronise all asynchronous inputs** to the system clock with two cascaded flip-flops to allow for metastable settling as discussed in week 11 lectures.
9. **Use just one global clock** **(eg CLOCK_50)** with *posedge* triggering in your designs. Multiple clocks and posedge with negedge triggering can cause timing problems. Clock enables (CE) can be used to select lower rate sampling than the global clock.
10. Ensure you have **enough bits** on signals connected with an in instantiation. The compiler will issue a warning message if insufficient bits are provided.
11. Use **named associations** in port mapping for instantiations with more than 2 ports –

    ```
    eg Dflipflop dff1(    .iClk(clk),
                          .iRst(rst),
                          .iD(data),
                          .oQ(out)
                        );
    ```

    rather than rely on order:

    ```
        DFlipflop dff1(clk, rst, data, out);
    ```
12. Check **all warning messages** after compiling in Quartus and ModelSim. See 10.

13. Use *i* and *o* as a prefix for port names for inputs and outputs of modules. Use _n as a suffix for asserted low or negative logic signals eg *iRst_n.*
14. Use all uppercase for *PARAMETER_VALUES*, capitalise *ModuleName* and lowercase for *signals*.
15. Avoid pointless parameters such as ONE, TWO, THREE for 1, 2, 3.
16. Beware of using operators % / * (modulus divide and multiply) because they synthesise to large circuits!
17. **Check your synthesised circuit** after compiling. See Quartus resource usage and Quartus:Tools-> netlist viewers ->RTL viewer.
18. Use an **appropriate radix** for readability – eg 4'd10 is the same as 4'b1010 but the former is usually more readable but this can depend on the context. Decimal is preferred unless individual bits are needed. This applies particularly to displaying signals in simulations.
19. Use **high level behavioural design** style where possible and avoid bit level instantiation of flip-flops and hand optimised logic gates. For example a multi-bit counter can be defined using addition as follows

    ```
    always (posedge clk)
            count <= count + 1'b1;
    ```

    rather than working with individual flip-flop instantiations and Boolean expressions for each D input.