MONASH UNIVERSITY

Department of Electrical and Computer Systems Engineering

ECE2072 ASSIGNMENT

Due Date:	Start of week 12. The precise deadline will appear on the Moodle submission link.
Submission:	Individual submissions via Moodle.
Assessment:	10% final mark. Plagiarism or collusion will result in zero marks for the copier ***and*** the originator.
Late Policy:	Loss of 1% of the final mark contribution of the assignment per day unless an official special consideration is granted.

Problem Description

Design a counter that produces an output every clock cycle. The output sequence that each student downloads from Moodle contains 15 Binary Coded Decimal (BCD) digits that are unique to each student. Let's call these output values V_0, V_1, V_2, .. V_{14}. The count sequence wraps around from V_{14} back to V_0. Here is the normal count sequence:

V_0 V_1 V_2 V_3 V_4 V_5 …V_{14} V_0 V_1 V_2 V_3 V_4 V_5 ….

There are additional inputs with these properties:

1. ***iSkip*** – when asserted the next output skips 4 values (eg from V_0 to V_5).
2. ***iRev*** - when asserted the next output is 2 previous values earlier in the sequence (eg goes from V_5 to V_3).
3. When both *iSkip* and *iRev* are 1, the counter skips 8 values (eg goes from V_0 to V_9).
4. When neither *iSkip* nor *iReverse* are 1, the counter follows the normal sequence.
5. ***iRst*** – when 1 the next output value is V_0.
6. ***iClk*** - Outputs are produced shortly after each rising *iClk* edge.

Design your counter using two modules. Module ***CounterSkipReverse*** implements the state sequence of the counter in sequential logic. The module ***StateToCountSequence*** is a combinational logic circuit that maps the state to your BCD *V* output. Since your

unique BCD *V* sequence necessarily contains repeated digits (there are 15 in the sequence) the BCD output cannot act as the state of the counter, as often occurs with conventional counters. The design challenge here is to work out what state representation is appropriate. Ensure your design is synthesised by compiling with Quartus.

Implement your design with a ***minimum*** number of flip-flops using Verilog compiled with Quartus with the standard project settings for a DE2 board. See the assignment FAQ on Moodle for Quartus settings that are required to ensure a minimum number of flip-flops are compiled when an FSM is detected in your code by Quartus.

You must also implement your design on the FPGA using LabsLand.

Question 1

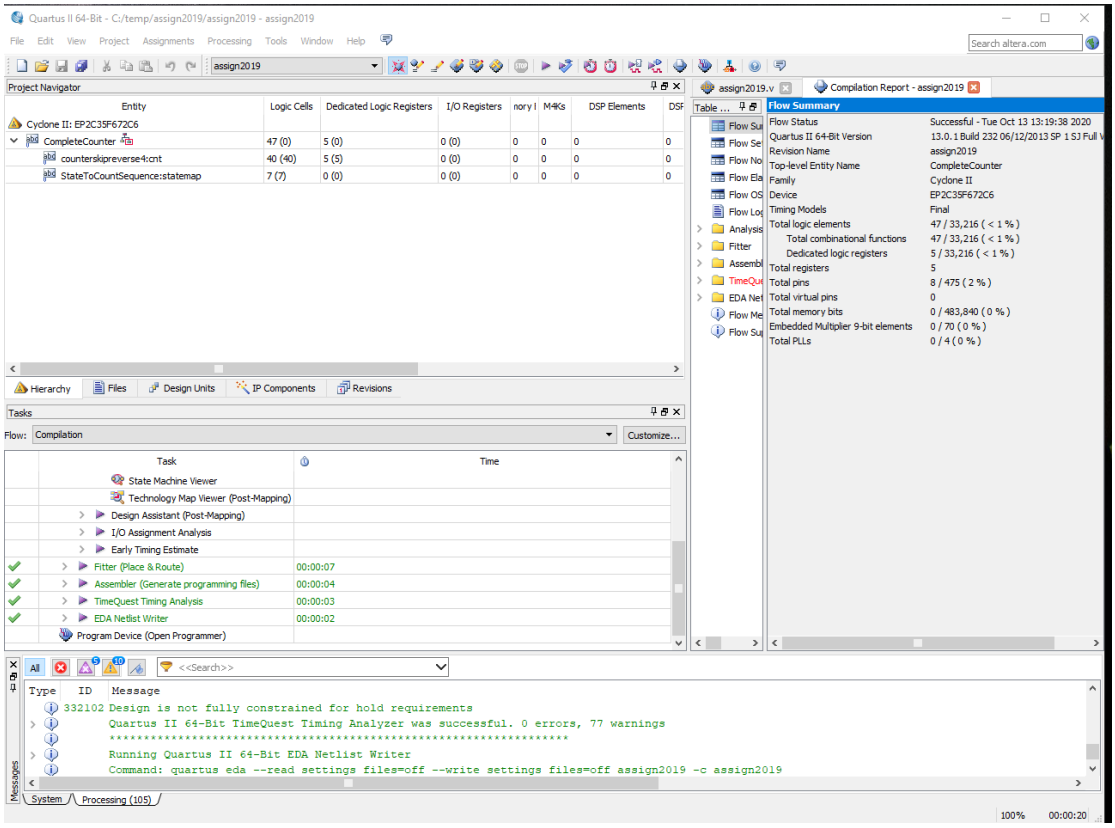
Download from Moodle your Verilog template file named *assignID.v* that defines your count sequence V_0 V_1 V_2 V_3 V_4 V_5 …V_{14} . Note that *ID* in *assignID.v* is your 8 digit ID number. This particular file must be used for your Verilog code to solve the above counter problem. Complete the modules *CounterSkipReverse*, *StateToCountSequence* and *CompleteCounter*. The file *assignID.v* must not be shared with, or shown to anyone else. Do not delete lines in this file since this may invalidate your answer during a preliminary automatic compilation, marking, plagiarism and collusion checking phase. Manual marking and checking will be also used after this. Download a new copy of your *assignID.v* file from Moodle if you accidently delete any lines.

Question 2

In your *assignID*.v file complete the testbench module *AssignmentTestBench* that enables ModelSim to check the correct functionality of your HDL design. Ensure that you test ***every transition from every valid state*** of your circuit. Since we have a reset action, we are not interested in the property of "self-starting" in this assignment. Include up to four screen captures from ModelSim showing the testbench simulation with inputs, outputs and state of your design. Use appropriate display options to maximise readability.

Question 3

Minimise the number of Logic Cells that your design requires after a synthesis compilation in Quartus with DE2 board settings. This will involve exploring different implementation strategies, checking them with your testbench and synthesising them in Quartus. Include a screen capture of Quartus that clearly shows the number of Logic Cells and "registers" (these are really flip-flops) after compilation for each module. This screen capture must also show the time and date displayed on your computer. Here is an example of how to show the registers in a Quartus project (ignore the actual numbers since they do not apply to your assignment):



Question 4

Perform a small research and explain how this design may be implemented, adapted, or applied into the real-world scenario. You must include references or citations to justify your explanations. The more references or citations that are included that are relevant, the stronger your explanation and justification will be.

Uploads (substitute your ID for ID below):

> **WARNING**: Using parts of anyone else's file or allowing someone else to access or copy your file constitutes a collusion and plagiarism offence and will be treated as cheating. This may result in you failing the assignment and even being excluded from the university.

Submit 6 or fewer files:

- *assignID.v* (Submit ONLY your minimised Logic Cell version from Q3).
- Q2_IDa.jpg , Q2_IDb.jpg, Q2_IDc.jpg , Q2_IDd.jpg
- Q3_ID.jpg

You may submit *.png* image files instead of the *.jpg* files above. A marking rubric will be released a week before the Moodle submission date. It will include marks allocated to the Good Style Guide below. Not meeting underlined guidelines below will result in higher mark deductions.

Good Style Guide

Follow these important 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 12 lectures. This does not apply to the 2020 assignment.
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 to an 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! They are fine for test benches and simulation.
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.