Sheffield Hallam University
Semester I of academic year 2021/22

"SoC architectures and FPGA prototyping"

Lab 3 -
Digital design using synchronous procedural Verilog

Assignment 1: synchronising inputs and assessing speed of operation

The code that will display last three decimal digits of your student ID number using the HEX2..HEX0 displays. You will enter these three last digits in binary code using the on-board slide switches. in order to convert your last three decimal digits into binary either use Windows calculator in programmer's mode or any online decimal to binary converter (or paper and pencil).

Double click on the file `DE10_Lite.qpf` from the folder ***Lab3_1***.

You will instantiate three copies of SevSeg modules to be driven by four-bit wires **`Dig3`**, **`Dig2`**, **`Dig1`**. Declare these wires first then instantiate the modules that will take these wires as inputs and drive **`HEX2`**, **`HEX1`** and **`HEX0`** respectively.

Write behavioural Verilog code for converting a binary number into three decimal digits, replicating the following C code with Verilog, and replacing **`number2convert`** with **`SW`**

```
Dig3 = number2convert / 100;

Dig2 = (number2convert - Dig3*100) /10;

Dig1 = number2convert - Dig3*100 - Dig2*10;
```

This code utilises hardware division which is not recommended in textbooks or on the Internet (e.g., https://tinyurl.com/y32vwgh3). However, for low number of bits and fixed divisor, Quartus compiler does a good job by instantiating an appropriate quite efficient IP. Compile the design and run it on board. Enter the three last decimal digits of your student ID number, take a photograph of the board and screenshot of the resource utilisation.

Explore the ***Quartuses' Tools -> Netlist viewers -> RTL viewer***. A new window will open showing the schematic diagram of the synthesised circuit. You can see that the Quartus compiler implemented subtraction using adders, and there is a hardware IP for every operation specified in the behavioural code. Take a screenshot for the report. You may also want to click at the plus sign in the top left corner of the instantiated **`SevSeg`** modules. This will expand the associated rectangle and show the schematic diagram of this module featuring two decoders and eight OR gates driving the display's segments.

This design demonstrates a digital hazard specific to asynchronous digital design (namely, structural hazard when, because of the different delays, experienced by the input signals, the output changes a few times before settling to the correct value). When you try to set all the switches down in order to enter 0, the display will react on move of every slider rather than wait until you finish sliding, and only then eventually show 000. You may try to move all the switches at once, but this is difficult to achieve, and in practice you will notice some random output before the display starts showing 000.

This undesirable behaviour can be eliminated by using synchronous design. We will connect the slide switches to a register (collection of separate D flip-flops for every bit used) and take value when the clock signal changes from 0 to 1 only (positive or rising edge). We will use KEY[0] to generate this clock, i.e. when the KEY[0] is pressed, the register will store its input:

```
reg [9:0] rSW;
always @ (posedge KEYn[0]) begin
     rSW <= SW;
end
```

Note that we no longer use `*` in the sensitivity list of the **`always`** procedural block, and we use non-blocking assignment (**<=**) to clock the input to the register. These two features are specific to the synchronous procedural blocks.

Please replace **`SW`** in the behavioural code to calculate **`Dig3..Dig1`** with **`rSW`**.

Run the design again. Notice that moving slide switches no longer affects the displays. Their reading changes only when you press KEY[0], creating the rising edge that clocks the **`rSW`** register.

Another feature of synchronous design, that is of a great practical value, is a possibility of calculating the maximum clock frequency for a design. This can be assessed if not only the input but also the outputs of the design (here **`Dig3..Dig1`**) are registered.

Declare the following four-bit registers **`rDig3, rDig2, rDig1`**.

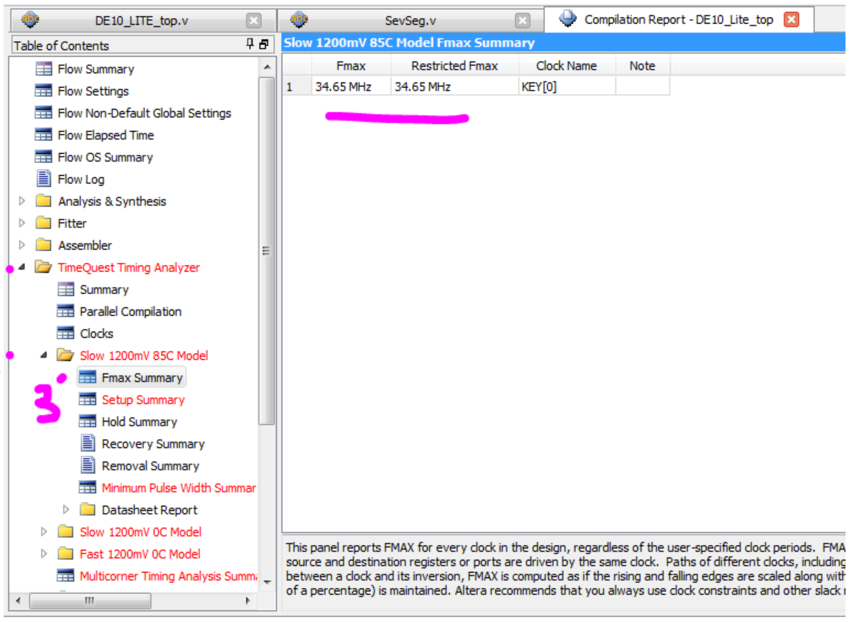
Assign these inside the above synchronous procedural block to the output wires **`Dig3..Dig1`** respectively using the non-blocking assignments (**`rDig3 <= Dig3;`** etc).

In the instantiations of the SevSeg replace the **`Dig3..Dig1`** with **`rDig3..rDig1`** respectively.

Compile the design and run it on board. Notice that two **`KEY[0]`** presses are required in order to propagate changed input values through to the display. The first press clocks the inputs into the **`rSW`** register. The second press updates the output registers **`rDig3..rDig1`**, displaying the new value.

Take screenshots of the resource utilisation and schematic diagram for the report.

As both the input and outputs are registered, it becomes possible to assess the worst-case clock frequency as shown below:



	Fmax	Restricted Fmax	Clock Name	Note
1	34.65 MHz	34.65 MHz	KEY[0]	

This panel reports FMAX for every clock in the design, regardless of the user-specified clock periods. FMA
source and destination registers or ports are driven by the same clock. Paths of different clocks, including
between a clock and its inversion, FMAX is computed as if the rising and falling edges are scaled along wit
of a percentage) is maintained. Altera recommends that you always use clock constraints and other slack

Figure 1. Screenshot for accessing the Fmax Compilation report

Take a screenshot for the report and screenshot of the code.

If you clock the design with the frequency, lower than the reported one, you are guaranteed to have all the outputs updated before the arrival of the following clock edge. This eliminates all the possible digital hazards (static, dynamic, structural) by design.

Assignment 2: improving maximum clock frequency of the design (I)

Copy your completed lab3_1 project folder and rename it to lab3_2. Double click on the **`DE10_Lite_top.qpf`** file to launch Quartus for this project.

If you compared the two schematic diagrams for the unregistered and registered designs, you may have noticed that quite a long chain of connected IP blocks is used to produce outputs. Each of these blocks is associated with some propagation delay; these combined delays limit the maximum operating frequency of the circuit.

In this assignment you will try to rearrange registers in order to achieve faster performance (higher clock frequency).

Comment out declarations of wires **`Dig3..Dig1`** and their continuous (**`assign`**) assignments.

Replace non-blocking assignments in the procedural block

```
rDig3 <= Dig3;
rDig2 <= Dig2;
rDig1 <= Dig1;
```

with the corresponding behavioural equations used to calculate **`Dig3..Dig1`** like

```
rDig3 <= rSW/100;
```

Make sure that you only use **`rDig3..rDig1`** in these equations.

Compile the design and run it on board. Notice that it now takes four "clock cycles" (presses of KEY[0]) to propagate the correct result to the display. That is because one extra clock cycle goes for calculating the **`rDig3`**, used for the other outputs, and one more – for calculating the **`rDig2`**, that is required to calculate the **`rDig1`**.

Take a screenshot for the maximum clock frequency for the report and notice its substantial increase. The increased number of the clock cycles required for the output to settle (called ***latency***) is the price to pay for this increase.

Take the screenshot for the developed code for the report.

Take the screenshot for the schematic diagram of the design for the report.

Take the screenshot for the resource utilisation for the report.

Assignment 3: improving maximum clock frequency of the design (II)

Copy your completed lab3_1 project folder and rename it to lab3_2. Double click on the **`DE10_Lite_top.qpf`** file to launch Quartus for this project.

You will try to further increase the maximum frequency by employing one more register and re-arranging the behavioural statements. It is a common knowledge that division and multiplication took way more time than addition/subtraction, and you will try to make sure that in every non-blocking assignment you only have a single division or multiplication. To achieve this, analyse the equations and select part of these that appear more than once and feature multiplication. Declare additional ***seven-bit*** register **`rTmp`** and assign this common equation to it using non-blocking assignment inside the procedural block. Use **`rTmp`** in the other non-blocking assignments instead of the common equation.

Compile and run the design on board. Note that the latency stays the same at four clock cycles as it was for the previous design.

Take a screenshot for the maximum clock frequency for the report and notice its substantial increase.

Take the screenshot for the developed code for the report.

Take the screenshot for the schematic diagram of the design for the report.

Take the screenshot for the resource utilisation for the report.

Note 1. Digital design technique that inserts registers within the path of data (***datapath***) in order to shorten the propagation delay of individual stages is widely used and called ***pipelining***. Here pipelining was not done in the right way as erroneous outputs could be observed on the display, but you will learn how to do it correctly at a lecture.

Note 2. you may have noticed that the registered display always starts from a value 000. This is because all the FPGA registers, unlike ASIC registers, are initialised when the bitstream is being uploaded. By default, all the registers (including **`rDig3, rDig2, rDig1`**) are initialised to zero, leading to the observed initial display state. In fact, any value on power up is allowed, and you can state it in the register declaration line like

```
reg [3:0] rDig3 = 3, rDig2 = 2, rDig1 = 1;
```

Please try it on your own as an optional exercise, but remember these initial values are only applicable for FPGAs on power up, and not available for general ASIC (silicon SoC) design.

Note 3. If you try to enter a number exceeding 999, it will not display correctly. If your **`rTmp`** register in lab3_3 is less than seven bits wide, some numbers will not display correctly. Can you figure out why as another optional exercise?

Report for the lab 3: answer questions at the end of the report template for the lab 3.