VERI Logbook

Information

Verilog Sources

All the code from this experiment can be cloned from https://git.skozl.com/e2-verilab

```
git clone 'https://git.skozl.com/e2-verilab'
```

Part 1

https://git.skozl.com/cgit.cgi/e2-verilab.git/tree/part_1

Exerise 1:

Timing Analysis

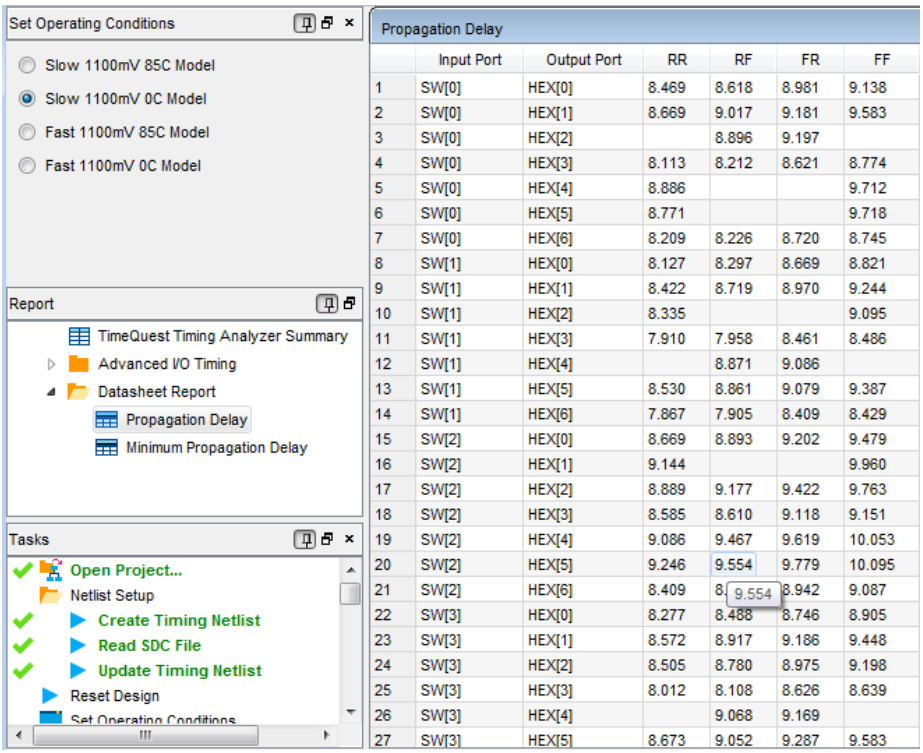
Timequest is a tool in the Quartus suit that allows us to see the delay between the inputs and the outputs of our system. The delay is in microseconds. For insatnce we can see that if we change SW0 it will take around 9 microseconds for the signals to propagate and reach one of the LED's of the 7-segment display.

The picture shows us that there is a timing depends on wether the input and output rise and fall. This difference arises because of the fact that the performance within the gates is different for the N-MOS and P-MOS elements.

Timequests geneates different reports for the slack depending on the temperature at which the device is running. slack varies based on **temperature and voltage** we run the gates at and they will hence perform differently. It might be the case that our timing is fine for 0C but when the device heats the timing breaks and we introduce glitches. Therefore it is important to consider a range of temperatures under which the device might operate..

Furthermore we can observe that our design uses 11 pins (7 for the display + 4 switches) and 4 logic units.

This is an interesting result since we have 4 inputs and 7 outputs, which can be implement using seven copies of a 4 input look up table, one for each segment. However since we use only 4 it is clear that the Quartus software is doing some sort of optimisations during the synthesis of our hardware definition.



	Input Port	Output Port	RR	RF	FR	FF
1	SW[0]	HEX[0]	8.469	8.618	8.981	9.138
2	SW[0]	HEX[1]	8.669	9.017	9.181	9.583
3	SW[0]	HEX[2]		8.896	9.197	
4	SW[0]	HEX[3]	8.113	8.212	8.621	8.774
5	SW[0]	HEX[4]	8.886			9.712
6	SW[0]	HEX[5]	8.771			9.718
7	SW[0]	HEX[6]	8.209	8.226	8.720	8.745
8	SW[1]	HEX[0]	8.127	8.297	8.669	8.821
9	SW[1]	HEX[1]	8.422	8.719	8.970	9.244
10	SW[1]	HEX[2]	8.335			9.095
11	SW[1]	HEX[3]	7.910	7.958	8.461	8.486
12	SW[1]	HEX[4]		8.871	9.086	
13	SW[1]	HEX[5]	8.530	8.861	9.079	9.387
14	SW[1]	HEX[6]	7.867	7.905	8.409	8.429
15	SW[2]	HEX[0]	8.669	8.893	9.202	9.479
16	SW[2]	HEX[1]	9.144			9.960
17	SW[2]	HEX[2]	8.889	9.177	9.422	9.763
18	SW[2]	HEX[3]	8.585	8.610	9.118	9.151
19	SW[2]	HEX[4]	9.086	9.467	9.619	10.053
20	SW[2]	HEX[5]	9.246	9.554	9.779	10.095
21	SW[2]	HEX[6]	8.409	8.	8.942	9.087
22	SW[3]	HEX[0]	8.277	8.488	8.746	8.905
23	SW[3]	HEX[1]	8.572	8.917	9.186	9.448
24	SW[3]	HEX[2]	8.505	8.780	8.975	9.198
25	SW[3]	HEX[3]	8.012	8.108	8.626	8.639
26	SW[3]	HEX[4]		9.068	9.169	
27	SW[3]	HEX[5]	8.673	9.052	9.287	9.583

Figure 1: timing for 85C

Exercise 3:

Trivial exerices requiring to instantate the module 3 times. The MSB 2 bits use a module by themselves however that is not a problem becaus Quartus will optimise out the redundant logic.

Experiment 4

The algoritm used to convert pure binary numbers to binary coded decimals is fairly trivial however it's implementation may not be. As I approached the challenge before looking at Proffesors Cheungs solution I implement my on which can be found it the repository. The pseudo code goes like so:

```
for (i=3; i>=0; i=i-1)
        begin
            if (BCD >= 5)
                BCD = BCD + 3;

            BCD = BCD << 1;
            BCD[0] = SW[i];
        end
```

From the reports we can see that our 10-bit binary to bcd out on the displays used 38 ALMS.

Part 2

https://git.skozl.com/cgit.cgi/e2-verilab.git/tree/part_2

Exercise 5:

Difficult part here is confugiring the ModelSim correctly once set up we can input commands.

As the clock is 50MHz, nd as we expect in modelsim we can see the clock cycles of length 20ns. Each of these causes the value of the counter to inrcease if enable is high. If enable is set to low the counter pauses at the last value and will then resume from the same once enable is high again.

Exercise 6:

Experiment involves chaining up exercises from part_1 with the previously created counter. The key thing here is to make the reset and enable to be active low by design.

When the button is pressed the counter counts to quickly for it to be visibly going trough the values, but based on our previous modelsim exercise we can assume so.

Introducing another counter of size log_2(50k) with a reset at 50k and feeding anding it with the enable signal we can see the 16 bit counter now counting up every millisecond.

From report we can see that: Design works using 76 ALMS, which is quite a sizable increase. However when you add the numbers up it makes sense. Each 7 segment decoder takes 4ALM's each, the 50k counter takes 16 registers, the binary to bcd converter will use around 30 ALMS for its shifting and the inputs need to be buffered. It now becomes hard to count exactly.

Predicted maximum frequency from the reports are: 0C is 411 MHz 85C is 425 MHz

We are running at 50MHz which is well belowe the maximum frequency so we should not be experiencing any glitches due to bad timing.

It is red because we have unconstrained output ports paths that can cause problems at high frequencies. Since our project does not interface with any other digital logic this is not a problem for us but in a big project with many different modules it is important to define some constraits which must not be broken.

Experiment 7 numbers

Printing the linear feedback shift registers in hex with the 7 segment decoders and writing implementing the LSFR we get:

```
SHIFT <= {SHIFT[5:0],SHIFT[6] ^ SHIFT[0]};
```

```
01
03
07
0F
1F
3f
7f
7e
7d
7a
75
6a
```

Working it out by hand this is what we expect.

Experiment 8 and 9

This challenge is probably the hardest part of the experiment as it took a lot of time to correctly implement the finite state machine and the top level with correctly function modules. It can be found in the git repository and is implemented exactly as suggested with one key difference. The desingn recommends a second divider by 2500 after the 50000 divider, which would give you a clock every 2.5s instead of 0.5s. Therefor the second divider is implemented as a divider by 500.

Experiment 9 is exactly the same except we include an extra state where we enable a counter and wait for input.

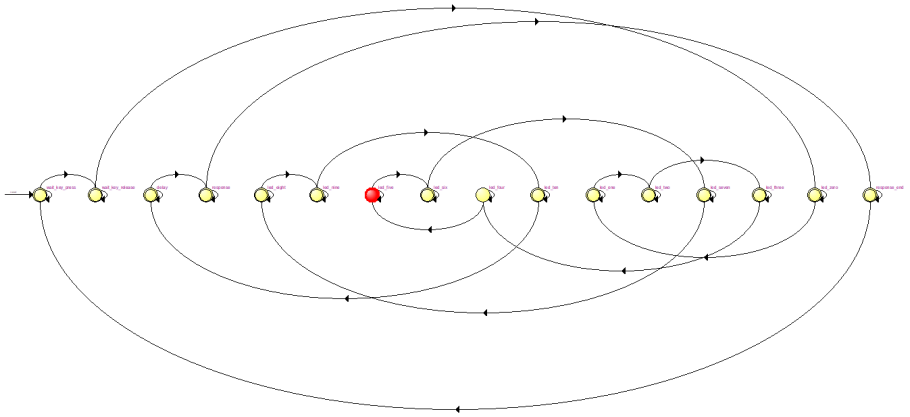


Figure 2: FSM for ex9

Part 3

https://git.skozl.com/cgit.cgi/e2-verilab.git/tree/part_3

Experiment 10

As CS goes low and LD goes high the following gets transmited over serial: * 3 bits of cmd, specifying: * zero = 0 * buffer = 1 * gain = 1 * power = 1 * 10 bits of our transmited value * h23b = 10 0011 1011 * 2 bits padding set to 0's = 00

So we transmit the sequence:

```
cmd  2    3   b   n
---- -- ---- ---- --
0111 10 0011 1011 00
```

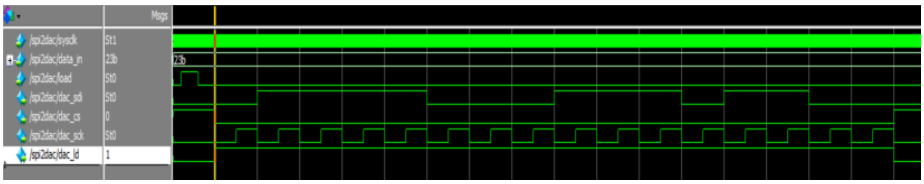


Figure 3: timing diagram

Experiment 10

We can meassure output voltage from 0 to 3.3V depending on the input, this is a DC voltage as we expect. Meassuring with scope we observe:

SCK is a train of ipulse at 10KHz SDI Changes according to state of switches

Experiment 11

We can now meassure both channels and see that they produce the same output range of 0 to 3.3V out and seem to be identical at DC.

The pwm and dac seem to produce the same output after the filter. Howevere if we meassure the pwm out before the filter we get a high frequency spikes (which is how pwm is produced).

Experiment 12

All the values match with first being 512 and last being 508.

Analysing the values in the rom with the switches we can see that is we increase the value of the switches the value displayed on the hex display goes up and down imitating the amplitude of a cycle of a sine wave.

Experiment 13

The output from the dac after the filter is staggered every 100us , which represents 10kHz, or in steps, while that of the PWM is smooth. Both produce a ~1khz sine wave.

Experiment 14

Instead of using a counter which adds a static one you can increment by a value as defined by the SW as you like, allowing you to skip samples.

The formula is as fololws:

SW * 10k / 1k

Which translates to a multiply by 0x2710 and a dividbe by 1000 or 2^10, which is the same as shifting right 10 bits, which from 24 gives you 14 left (the 14 MSB).

Part 4

https://git.skozl.com/cgit.cgi/e2-verilab.git/tree/part_4

Experiment 16

The adc digitalise values as positive numbers shifted up by 2^8 + 2^7 = 384 = 0x180 This is the value that we get on the 7 segment display when we have no input, and hence our wave is shifted by that amount.

Since our DAC is 10 bits we need to set the new offset to be at half way in our range which is 512. This is all that our processor does.

After implementing the described curcuit we get the sound we play in out. All we do is digitilise the sound and reproduce it.

After using the multiply module the sound observed is louder as expected. However it is not 4 times louder!

Experiment 17

```
# Pulsegen
Waiting data_valid
high for 1 cycle
Waiting for data_valid low
```

Had to create a finite state machine for this pulse_gen module.

To operate the fifo: Do not read until full, after it is full it will stay full and we should keep reading it. Every clock cycle keep wrreq high and feed it the data out. Take the output of the fifo shift it with sign extention and add it to the dac input.

To multiply times a factor beta we shift the number to reduce its amplitude we need to sine extend it, otherwise we will probably get distrtion/clipping and wrong volume for the echo.

Experiment 18

Same as 16, but we don't want positve feedback so we subtract (same amplitue but down). And switch ouptut and input, very minor modifications. You can steal hear the sound wave with the same amplitude just inverted. If we had positive feedback we would go to the rail and get stuck there.

Experiment 19

Why 0x666 multiplier: By multiplying by 1638 and then removing the least significant ten bits we effectively multiply by 1.64

Using all 8192 bits of ram would cause a delay of 0.819s (when all 9 switches are high). The coffecients we have to multiply to get ms is:

$$0.8192/511 * 1000 = 1.603$$

Which is close to what we are doing.