Abbas Yadollahi – 260680343
He Qian Wang – 260688073

October 27, 2017
Team A6

Lab 2 Report: gA6 RANDU

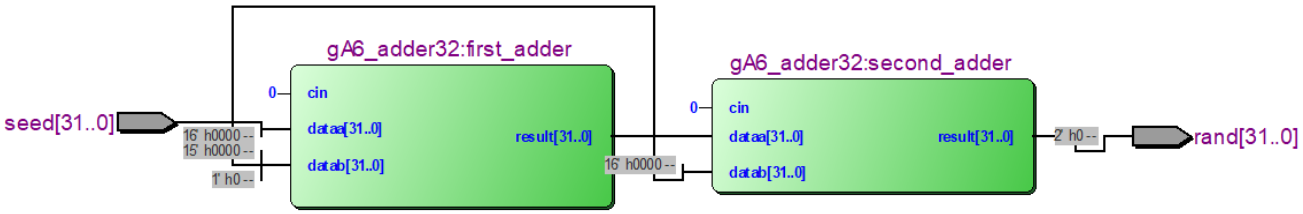
PART 1: 32-BIT RANDU

Objective

The first part of the lab required us to build a 32-bit random number generator using IBM's RANDU logic. To create the circuit, we used two instances of the 32-bit adder that is included in the LPM library. The circuit only requires one 32-bit input *seed* and results in one 32-bit output *rand*. Using the RANDU logic isn't the best for a random generator considering it is easy to guess the following number since it uses a simple equation $rand = mod(a * seed + b, c)$, where $a = 65539 = 10000000000000011_2$, $b = 0$ and $c = 2^{31}$.

Schematics

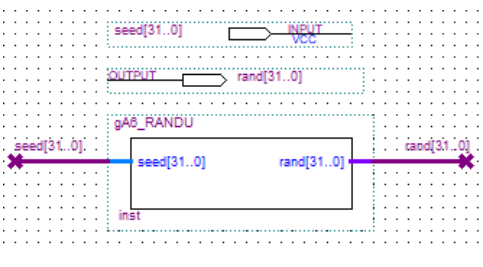
32-Bit RANDU



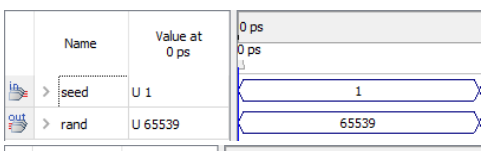
To implement the RANDU equation as a digital circuit, first we needed to multiply the input *seed* by 65539. In binary, this is equivalent to adding *seed* with *seed* shifted left once with *seed* shifted left 16 times. To create the multiplication operation, we circuit used two simple 32-bit adders from the LPM library in sequence. The input dataa and datab of the first adder are the two shifted versions of *seed* and the input c_{in} will always be 0. For the second 32-bit adder, we use the 32-bit output result of the first adder as the input dataa, the non-shifted version of *seed* for the input datab and, once again, the input c_{in} will always be 0. The second step to the RANDU circuit consists of doing a modulo of 2^{31} to the result of the second 32-bit adder. To do so, we use the simple fact that computing the modulo of a number with respect to its base to a power can be done by setting all bits beyond the power-1 to zero. In our case, since we're doing a modulo 2^{31}, we set all bits past 30 to 0, giving us our final 32-bit output *rand*.

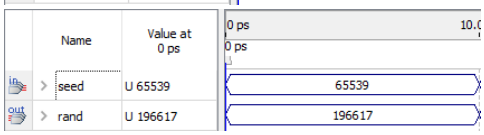
Simulation

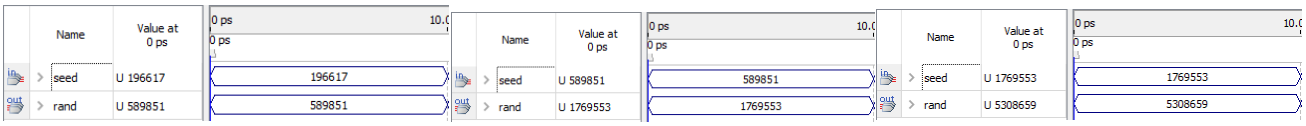
Waveforms



Simulation	Seed	Rand
1	1	65539
2	65539	196617
3	196617	589851
4	589851	1769553
5	1769553	5308659







To test our logic, we simulated the circuit starting with an input value of 1 for *seed*. Afterwards, we would take the result of that test and use it as the input to the next simulation. We repeated this five times. The results of each simulation can be seen in the tables and graphs above. Comparing the outputs with the results using the initial equation for RANDU, we can confirm that the digital circuit follows the desired random number operation.

Remarks

Like previously mentioned, the RANDU logic is a terrible random number generator since it follows a specific pattern. As a result, it can be proved that the modulo 2^{31} of three consecutive generated numbers (using the output as the next input) will always equal zero. The relationship can be described with the following equation $mod(rand_n - 6 * rand_{n-1} + 9 * rand_{n-2}, 2^{31}) = 0$. We can even confirm this using the values found in the simulation, where $rand_n = 589851$, $rand_{n-1} = 196617$ and $rand_{n-2} = 65539$. As we can see, the final result does, in fact, give zero.

The main limitation from this circuit is due to the usage of the output as the input of the next randomly generated number. It is possible that the modulo of the input gives 0, meaning that for every following generated number, the output will always give 0. Although, since we're using a very high modulo (2^{31}) relative to the max input (2^{32}), it will rarely occur.

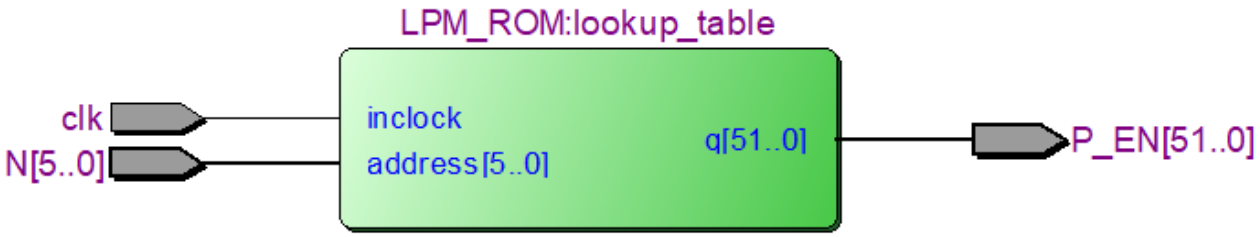
PART 2: 6-BIT POP ENABLE

Objective

The second part of the lab required us to build a 6-bit input circuit which would output a specific number (card) depending on the input. To implement the pop enable operation, we will use the ROM Lookup Table from the LPM library which allows a one to one to link for input and output. The entire component will only require a clock input *inclock*, a 6-bit input *N* and will output a 52-bit number *P_EN*. The output bits will depend on the input such that each bit of *P_EN* from 0 to *N*-1 will be 0 and the rest be 1. For all values of *N* above 51, all bits of the output will be 0.

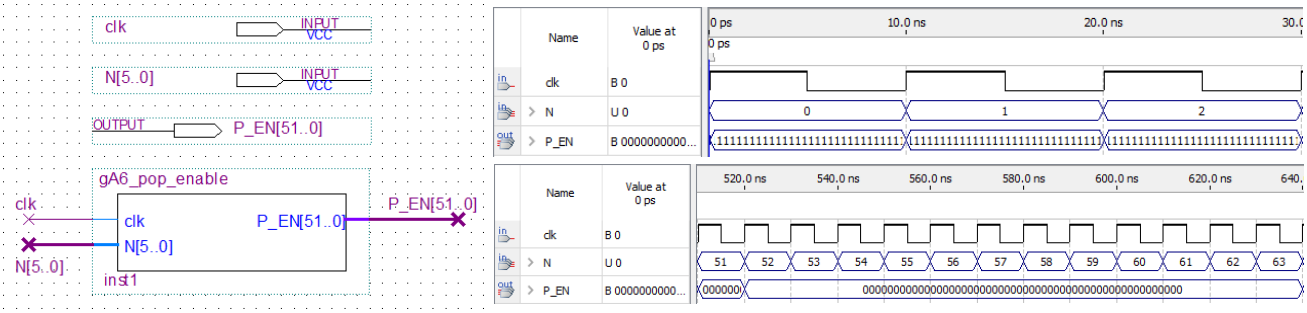
Schematics

6-Bit Pop Enable



To implement the pop enable function, we first needed to create a data table using MIF file where we could store all possible outputs. With the database created, all that was left was to create the ROM component and set the proper specifications for it. For our circuit, the database had 64 possible options, with each one being a 52-bit number and to locate the value in the table, we used a 6-bit address bus.

Simulation



We simulated our circuit to verify the outputs of lookup table. We began by running the simulation with an input of 0 and incrementing it by 1 up to 63, the maximum value for the input *N*. We notice that after each increment, an additional bit of the output switches from 1 to 0. From the moment the input *N* equals 52 to 63, every bit of the output equals 0 as expected.

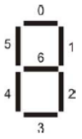
Remarks

Concerning this circuit, one possible limitation is that we need to use separate files to make the circuit work properly. Since we are using a ROM lookup table, we need to retrieve the data from another file which requires us to package the files together. The MIF file is also highly inefficient as each individual card requires 52 bits of memory.

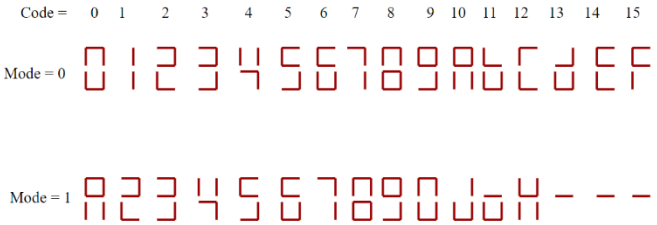
PART 3: 7-BIT SEGMENT DECODER

Objective

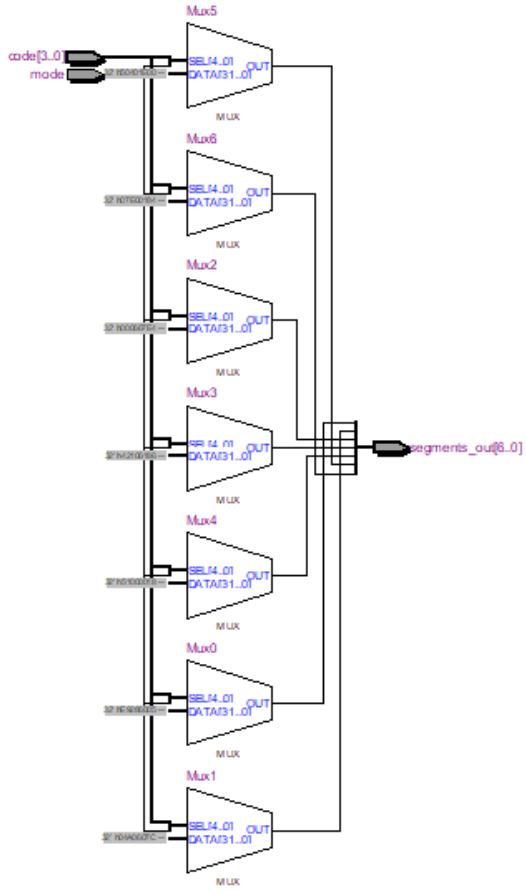
This section of the lab required us to design a 7-segment LED decoder. The LED display has 7 different segments which are numbered as on the screenshot. We are using a 4-bit input code which represents the desired number from 0 to 15, a 1-bit input mode is used to indicate which format we wish to represent and a 7-bit output segments_out. If the mode bit is 0, the LED will display the output number as a hexadecimal character from 0 to F. If the value of the mode bit is 1, the LED will display our output as one of the 13 cards in a deck from [A, 2, 3, ..., 9, 10, J, Q, K] with 10 being represented by 0 and with the final 3 inputs outputting void values. The image below indicates how the LED would display each value of each mode.



numbering of the LED segments (from Altera DE1 board manual)



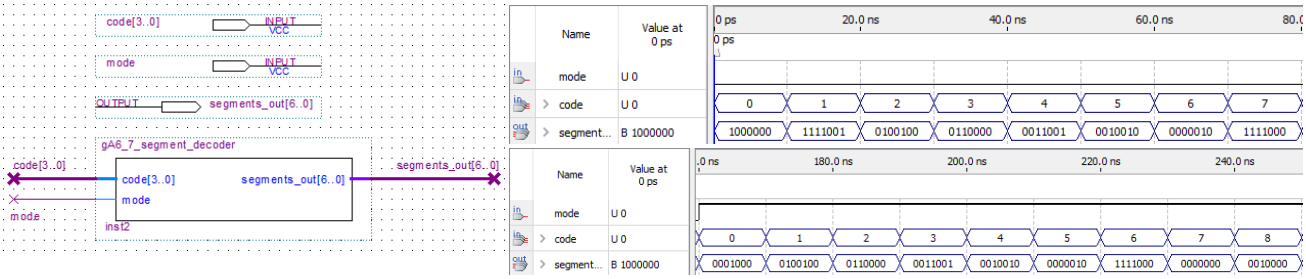
Schematics



From the 4-bit input and the mode bit, we have access to 32 different combinations. For each combination, we associate it with a 7-bit output directly linked to the 7-segment LED display. Each bit represents a different segment that should be activated or not. The circuit uses a low-active which means that when a bit is 0, its respective segment will light up. When a bit is 1, the respective segment will stay off.

Our digital circuit has a 4-bit input bus *code*, a 1-bit input *mode* along and a 7-bit output *segments_out* which would theoretically be linked to the FPGA board. The behavior of our entity component first combines our 4-bit input and 1-bit input into a 5-bit input. We then use a SELECT statement which has 32 different cases of the 5-bit input. Based on the input, we will output the corresponding 7-bit output.

Simulation



To verify that all segments were probably activated, we tested every possible scenario of the two inputs *code* and *mode*. The output of the simulation gave three distinct sections, the first being the expected values of each desired number in mode 0, the second giving the 13 possible values for a card in mode 1 and the third being the null values in mode 1.

Remarks

Concerning the limitations of this section, we did not have access to the FPGA board, so we are unable to physically test our code with the LED display. We could only test the output by trying to simulate the circuit on a waveform file which isn't as reliable as the Cyclone II boards. Another issue is that we are to accurately represent the letters using only a 7-segment LED display which doesn't for a precise display of characters. For example, the K looks like an H on the display.

References

Altera. *LPM Quick Reference Guide*, December 1996
Altera. *lpm_add_sub Megafunction User Guide*, March 2007
Altera. *lmp_rom Megafunction User Guide*, March 2005
Clark, J. *Lab #2 – Combinational Circuit Design with VHDL*, Fall 2017, McGill University