VLSI Sudoku Report

Gregory Chekler, Matthew Karazincir, Hong-Ye Wang, Natalia Perey

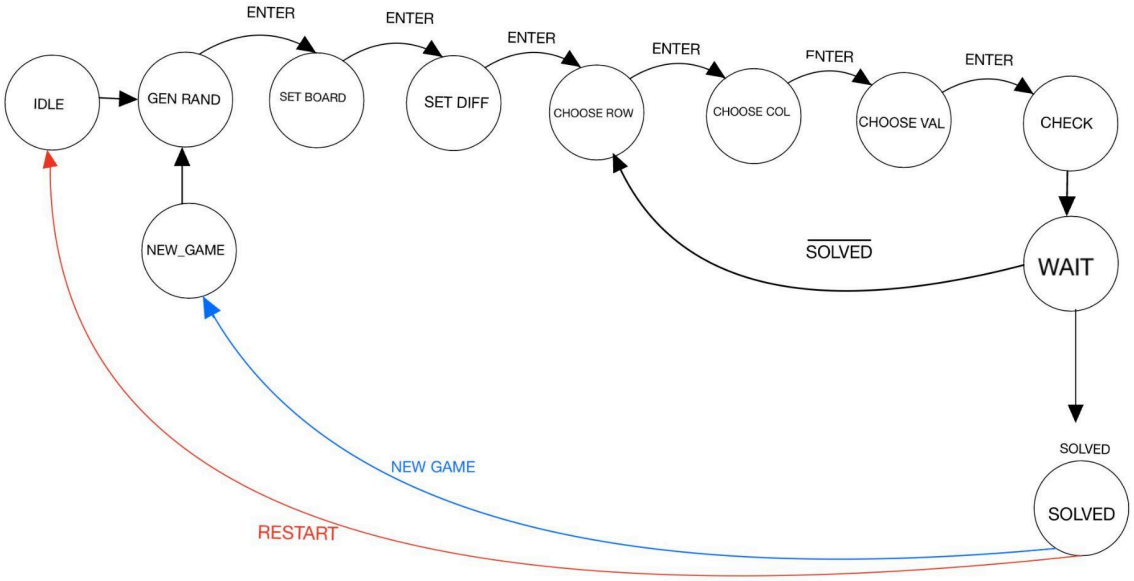
Abstract

Sudoku is a board game where a user fills in a N x N grid of numbers such that every row, column, and subsquare have only one instance of each number from 1 to N. Since playing Sudoku is formulaic, a 4 x 4 gameboard chip was created using an LFSR algorithm to randomly generate solvable preset combinations and then randomly generate filled and unfilled squares for the user based on easy, medium, and hard difficulties. The user's inputs are verified and checked against the actual solution and outputs a high solved signal once the board is correctly filled in.

Motivation

Sudoku is an interesting example of a game that is formulaic, but also dynamic with many different possibilities. Always randomly generating solvable games of Sudoku is a difficult algorithmic problem, but there are ways to slightly cheat the system in order to make seemingly random results. Furthermore, when the amount of physical space for creating such a game is minimal, shortcuts must be taken carefully and deliberately. As such, developing a simple game of Sudoku on a VLSI chip becomes an intellectually stimulating challenge.

System Overview



Sudoku FSM State Diagram

The FSM has 11 states in total which run the game through the entire generation, inserting, and checking process. The states are described as follows:

State Description Table

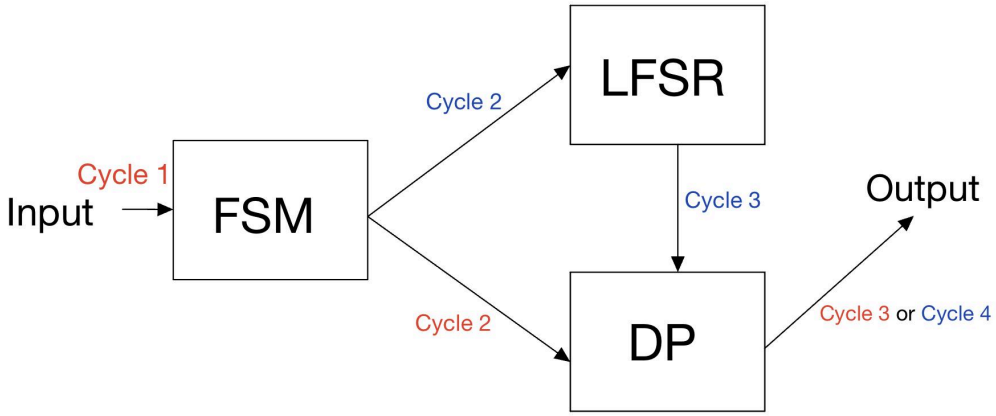
STATE	DESCRIPTION
IDLE	The state is asserted when starting the game for the first time through the restart signal. It will reset all registers and flags to their initial conditions. On the next clock cycle, the state will automatically move to GEN RAND.
NEW GAME	The state is asserted when a new game is started by raising the new game signal. It will reset all registers and flags to their initial conditions except for the LFSR which allows for the LFSR to continue churning numbers from the previous game. The restart signal must be asserted at some point prior to the new game signal, as otherwise the output of the LFSR will be incorrect. On the next clock cycle, the state will automatically move to GEN RAND.
GEN RAND	The state raises gen_rand_flag high which will let the LFSR start churning numbers. Since the user may be in this state for many clock cycles, as the

	clock will be fast, the result of the LFSR will not always be the same. Once enter is asserted in this state, the FSM will move to SET BOARD.
SET BOARD	The state lowers the gen_rand_flag which will stop the LFSR churning random numbers and set_board_flag will be raised which will signal to the Data Path that the solution for the board can be generated. Once enter is asserted in this state, the FSM will move to SET DIFF.
SET DIFF	The state lowers set_board_flag which will lock in the solution for the board. Furthermore, the set_diff_flag will be raised so that when the user inputs a difficulty from 1 to 3 corresponding to easy, medium, and hard. Furthermore, at this stage certain squares, depending on the difficulty level and the random number generator, will be filled in as hints for the user. The number 0 represents an unfilled square while a number from 1 to 4 represents a filled square. Once enter is asserted in this state, the FSM will move to SET DIFF. Once enter is asserted in this state, the FSM will move to CHOOSE ROW.
CHOOSE ROW	The state lowers the set_diff_flag which will let the user choose which row (0 to 3 which corresponds to rows 1 to 4) on the board they would like to select. It will raise the row_flag and once enter is asserted in this state, the FSM will move to CHOOSE COL.
CHOOSE COL	The state lowers the row_flag which will let the user choose which column (0 to 3 which corresponds to columns 1 to 4) on the board they would like to select. It will raise the col_flag and once enter is asserted in this state, the FSM will move to CHOOSE VAL.
CHOOSE VAL	The state lowers the col_flag which will let the user choose which value (0 to 3 which corresponds to values 1 to 4) they would like to insert. It will raise the val_flag and once enter is asserted in this state, the FSM will move to CHECK.
CHECK	The state lowers the val_flag and compares the board the user sees (after inserting the new value) with the solution board by sending a check_flag to the datapath. The FSM will automatically move to the WAIT state, allowing for the datapath to process and send a solved signal to the FSM.
WAIT	The state lowers the check_flag and then the FSM moves to SOLVED if the solved_flag is high and moves back to CHOOSE ROW if it is low.

SOLVED	The FSM sits in the SOLVED state until a new_game or restart signal is asserted.

The datapath works in conjunction with the FSM such that if the FSM receives a signal on clock cycle 1, then on clock cycle 2 the FSM outputs a signal. On that same clock cycle, the datapath receives the signal as does the LFSR module. On the third clock cycle, the datapath and the LFSR output their result. The LFSR would then give its output to the datapath module on clock cycle 3, while the datapath would update the FSM and the output on clock cycle 3. The output of the datapath from the LFSR would then happen on clock cycle 4.

Modules Interconnection Diagram from a Clock Cycle Perspective



The datapath takes in 6 bits of internal input: clka, clkb, restart, new_game, and two bits for the user input. The datapath outputs 3 bits for each square on the board, and since there are 16 squares, it outputs 48 bits for the board. It also outputs a solved flag meaning that the game has been completed.

The board is preset to certain variables such that setting any number from 0 to 3 for A (which can be represented with two bits in binary), and B is equal to a number that is not A or its complement, but still in the range 0-3. Taking the complement of both A and B will result in a viable solution for a 4 x 4 board. Ex: If A = 00 then ~A = 11. B = 01 and ~B = 10. These values, mapping 0-3 to 1-4 by adding 1 will result in a solution to the boards below.

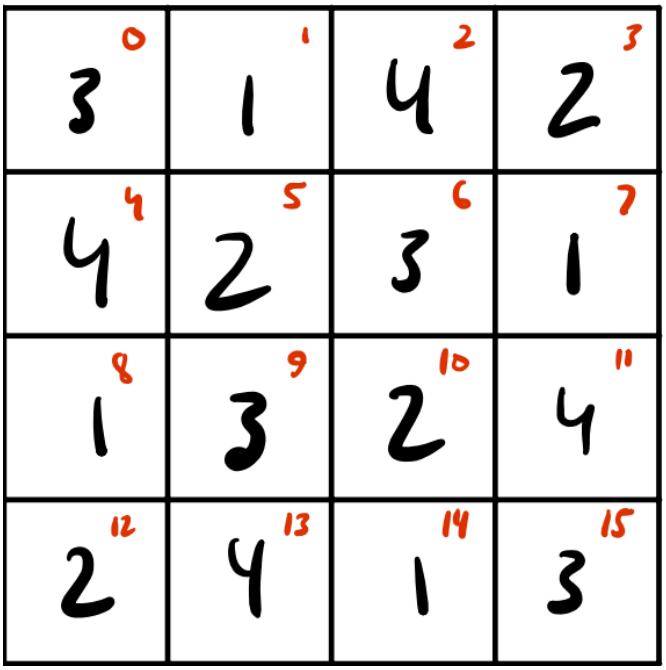
Sudoku Board Presets

Preset 1

A	~B	B	~A
B	~A	A	~B
~B	A	~A	B
~A	B	~B	A

Preset 2

~A	A	~B	B
~B	B	~A	A
A	~A	B	~B
B	~B	A	~A



Example filled board with indices

At the beginning of the game, the amount of hints given to the user is based on the difficulty of the game. For easy mode, the number of filled squares at the beginning of the game is 6, medium is 5, and hard is 4. The user cannot overwrite the hints to make an incorrect board, and if they try to do so, the board will not update.

The datapath also takes internal flags from the FSM so that the outputs can be correctly updated as needed.

Datapath Internal and External I/O

restart (external)	Sets all of the registers in the datapath module to the default of 0
new_game (external)	Sets all of the registers in the datapath module to the default of 0
rand_setup (internal)	This is a four bit internal input coming from the LFSR module where the most significant bit chooses the board preset, the middle bit chooses the value of A, and the least significant bit chooses the value of B.
rand_A (internal)	This is a four bit internal input that determines the place of the hints for the top two rows of the boards

rand_B (internal)	This is a four bit internal input that determines the place of the hints for the bottom two rows of the boards
set_board_flag (internal)	Tells the DP to generate the solutions board with the values generated from the rand_setup
set_diff_flag (internal)	Tells the DP to update the user board with the hints generated from the rand_A and rand_B inputs
diff_cell_val (internal)	This is a two bit external input that determines the game difficulty, row, column, and value being input by the user. The values range from 0 to 3, but map to 1 through 4
row_flag (internal)	Tells the DP to select the row on the board that the user wants to input the value in from diff_cell_val
col_flag (internal)	Tells the DP to select the column on the board that the user wants to input the value in from diff_cell_val
val_flag (internal)	Tells the DP to input the value from diff_cell_val into the corresponding square
check_flag (internal)	Tells the DP to check the user board against the real board to see if it is correct
solved (external)	DP outputs 1 if the real and user board match and 0 if they do not
fill_flag (internal)	Stores which values hold hints that were given at the beginning of the game
user_board_# (external)	A three bit external output that represents a square in the board that the user sees. 0 means that the board is empty on that square, while values 1 to 4 represent an inserted value or a hint
real_board_# (internal)	A three bit internal output that represents a square in the solution board for the game
clk_a (external)	Part of the two phase clock used for clocking the combinational logic
clk_b (external)	Part of the two phase clock used for clocking into different states

The datapath interconnects with both the FSM and LFSR module. The LFSR module on restart initializes the LFSR at a 24 bit seed that is predetermined. When the gen_rand_flag is raised this will cause the LFSR to begin shifting bits in accordance with the feedback polynomial required for 24 bits, which is x^24 + x^23 + x^22 + x^17 + 1. This means that the 24th, 23rd,

22nd, and 17th bits are the ones that will be XORd to create a new bit. This new bit is then added to the end of the LFSR which shifts all the other bits over by 1. The process keeps occurring until the gen_rand_flag is low. To ensure that the LFSR does not start over at the initial seed when a new game is started, the new_game input is used to move the FSM to the NEW GAME state. This input is used in the LFSR module to reset all registers to 0, except for the LFSR, which will retain its previous value. This prevents each new game from having the same random numbers and ensures a different experience for each game.

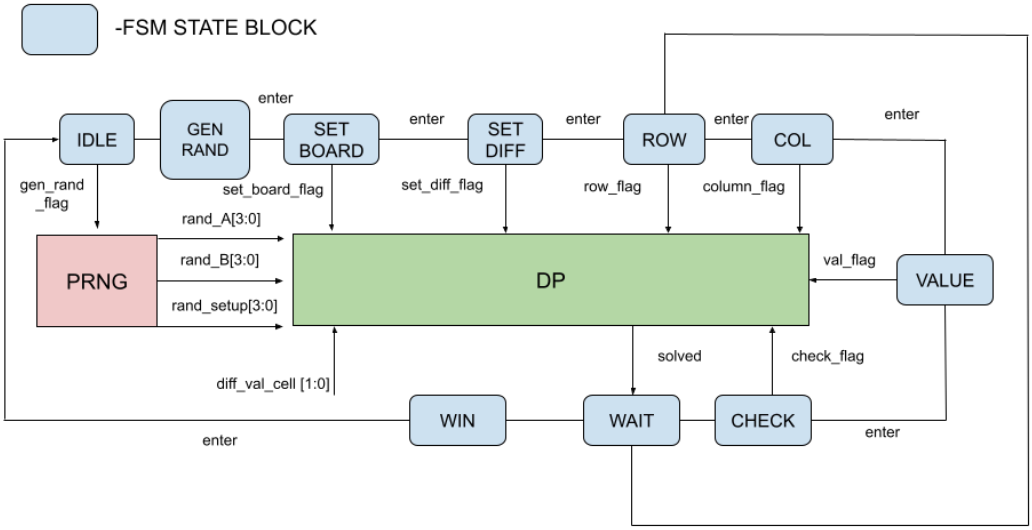
In order to ensure the outputs appear as random as possible, the 12 least significant bits of the 24 bit number are then used for the three 4 bit outputs: rand_setup, rand_A, and rand_B. rand_setup takes the 4 least significant bits, rand_A takes the next 4, and rand_B takes the next 4.

LFSR Internal and External I/O

clka (external)	Part of the two phase clock used for clocking the combinational logic
clkb (external)	Part of the two phase clock used for clocking into different states
restart (external)	Sets all of the registers in the LFSR module to the default of 0
new_game (external)	Sets all of the registers in the module to the default of 0 except for the LFSR which retains its value
gen_rand_flag (internal)	When high, tells the LFSR to generate new random numbers
rand_setup (internal)	Comes from bits [3:0] of the current LFSR number. Will be used by the datapath to generate the board
rand_A (internal)	Comes from bits [7:4] of the current LFSR number. Will be used by the datapath to determine the place of the hints for the top two rows of the boards
rand_B (internal)	Comes from bits [11:8] of the current LFSR number. Will be used by the datapath to determine the place of the hints for the bottom two rows of the boards

The entire system is run by an external two phase clock. In all three modules, clka takes the inputs, does the necessary calculations and stores them into temporary registers. On clkb, the output registers are assigned the values in the temporary registers. One assumption that is made by the designers is that the clocks run very fast such that the user is not able to, or is under any stipulation, to time their inputs. Thus the LFSR module is random and the game can be played in real time.

FSM, DP, and PseudoRNG (LFSR) interconnection

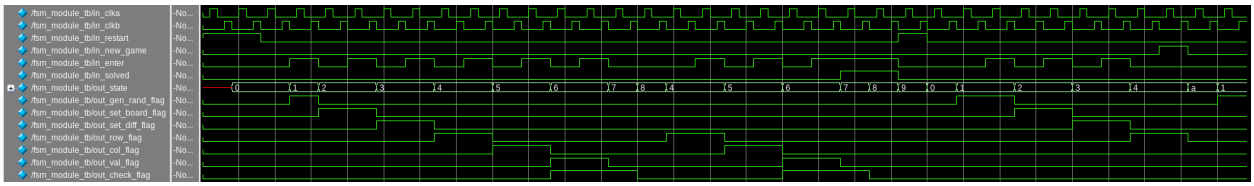


Testing

The Sudoku core and its smaller modules FSM, LFSR, and Datapath were tested extensively prior to and post synthesis as well as through Irsim. Below are the testing results with explanations.

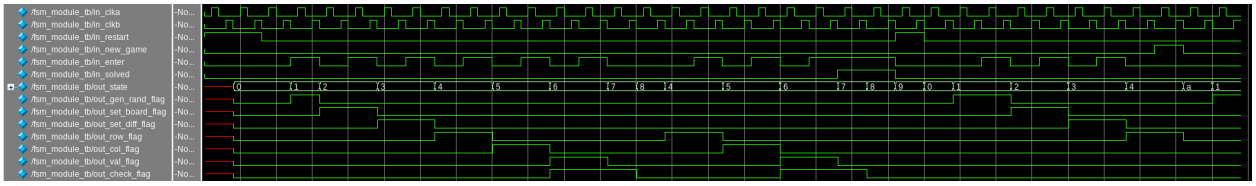
FSM

Questa



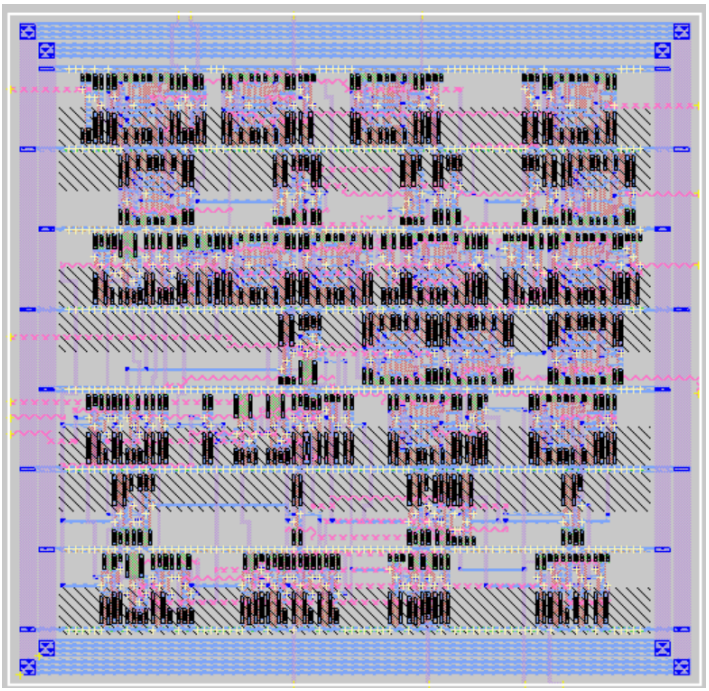
When *in_restart* is inserted, the FSM begins at IDLE (0) and moves to GEN RAND (1) immediately after. Consecutive *in_enter* signals propel the states through SET BOARD (2), SET DIFF (3). The user begins to insert values on the board by inserting *in_enter* and transitioning through CHOOSE ROW (4), CHOSE COL (5), and CHOOSE VAL (6). The FSM then moves to CHECK (7) and WAIT (8) automatically after CHOOSE VAL. When *solved* is high after CHECK and WAIT, the FSM moves to WIN (9), indicating that the board has been solved correctly. Towards the end when *in_new_game* is high, the FSM moves to the NEW GAME (a) state where the LFSR should keep running without its initial state being restored. Overall, the FSM behaves as expected with the input signals.

Questa Post Design Compiler

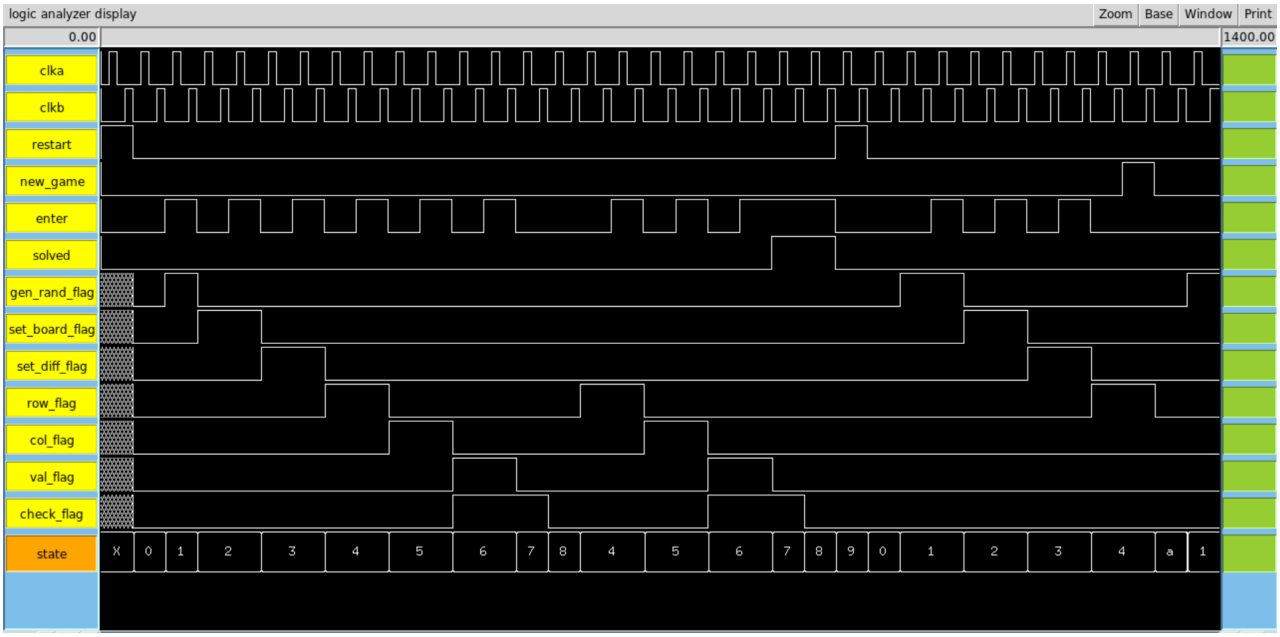


Post DC Questa behaves identical to Questa before synthesis, but with initial delays.

Magic



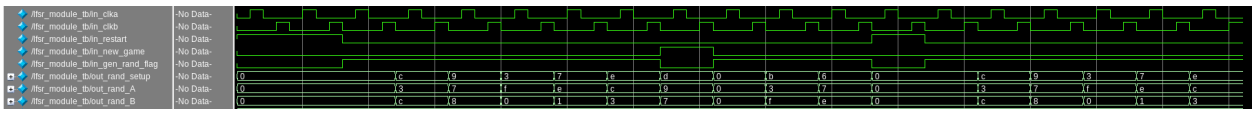
Irsim



Irsim behaves identical to Questa simulations from above.

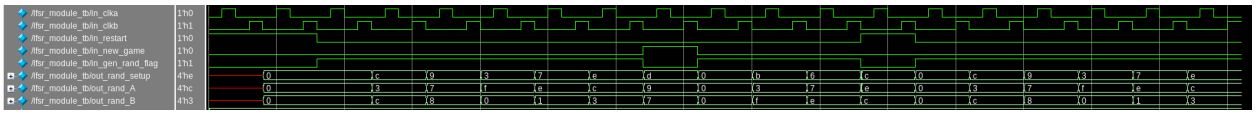
LFSR

Questa



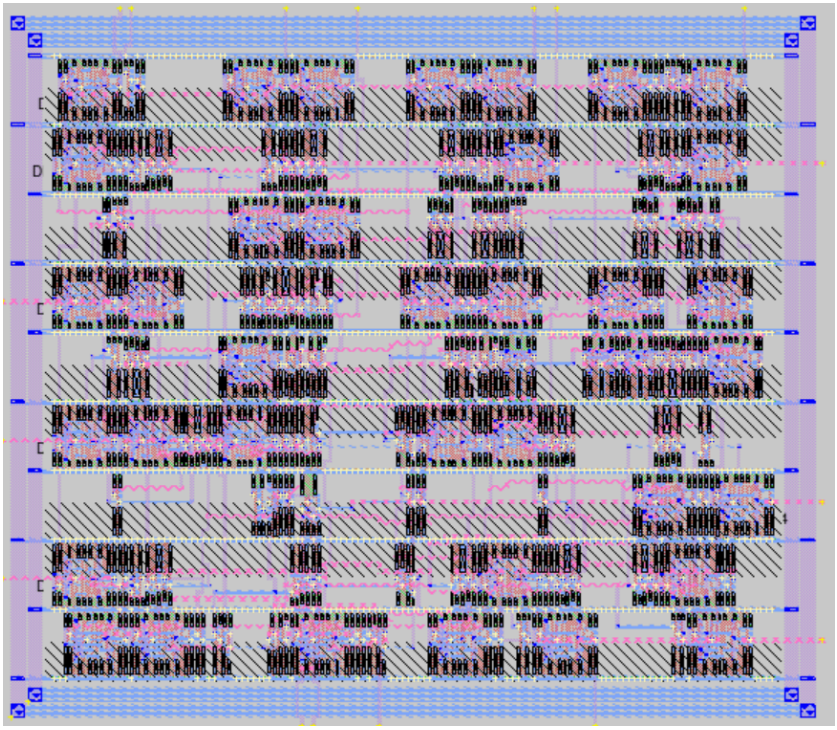
When *in_restart* is high at the beginning, the LFSR outputs are initialized to zeroes. Then, *in_gen_rand_flag* is on so the LFSR begins to run as shown in outputs of *out_rand_set_up, out_rand_A, out_rand_B* from “c3c” to “ec3” before *in_gen_rand_flag* is low and the outputs clear to “000”. *New_game* is then asserted high, so the LFSR continues to run without resetting from outputs “b3c” to “67e”. Then, *in_ restart* is high again for the second time. Notice that the LFSR output sequence is identical to that after the first *in_restart* since the initial state is resetted, which is not the case for when *in_new_game* is high where the output sequence is different. Overall, the LFSR behaves as expected with the input signals.

Questa Post Design Compiler

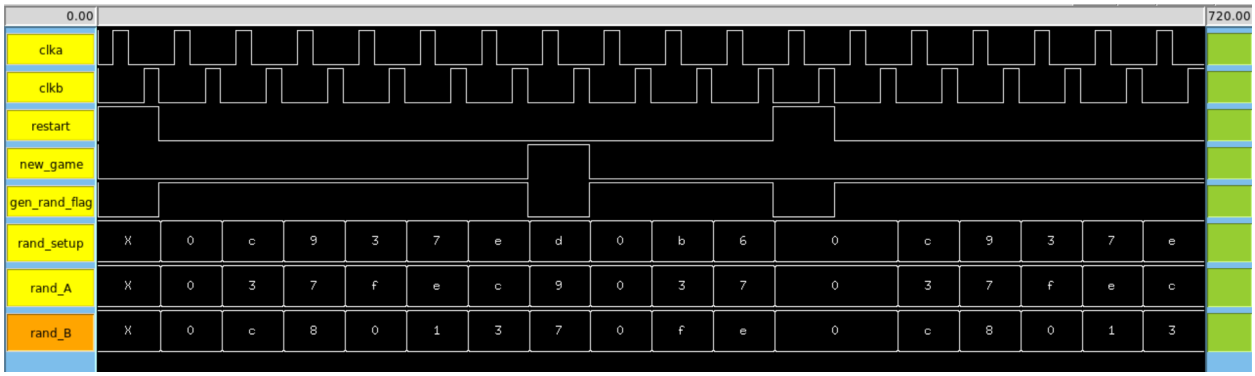


Post DC Questa behaves identical to Questa before synthesis, but with initial delays.

Magic



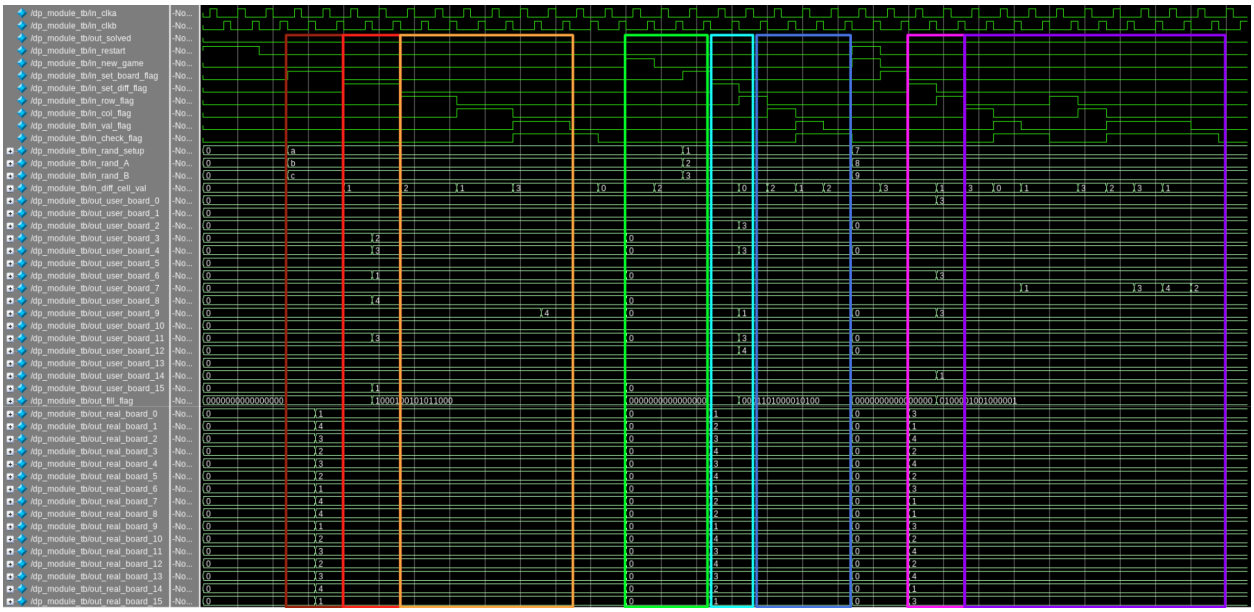
Irsim



Irsim behaves identical to Questa simulations from above.

Datapath

Questa



The testbench tests all functionalities of the Datapath (DP).

Brown: After *restart* is high the *in_set_board_flag* is turned on, the DP successfully checks the three random number inputs “abc” and sets the solution board as shown from *out_real_board_0* to *out_real_board_15* following the preset method discussed before.

Red: When *in_set_diff_flag* is high, the DP reads the value of *in_diff_cell_val* which is 1, corresponding to easy mode. Then, 6 hints are displayed on the *user_board* signals successfully at the next clock cycle.

Orange: The DP reads *in_diff_cell_val* to obtain the row (2) when *in_set_row_flag* is high, the column (1) when *in_set_col_flag* is high, and the value (3) when *in_set_diff_flag* is high. This corresponds to inserting 4 at cell 9, which is updated as seen in *out_user_board_9*, indicating that the DP can read a user input and update the user board correctly.

Green: When *in_new_game* is high, we see that both the user board and the real board resets to 0 as planned.

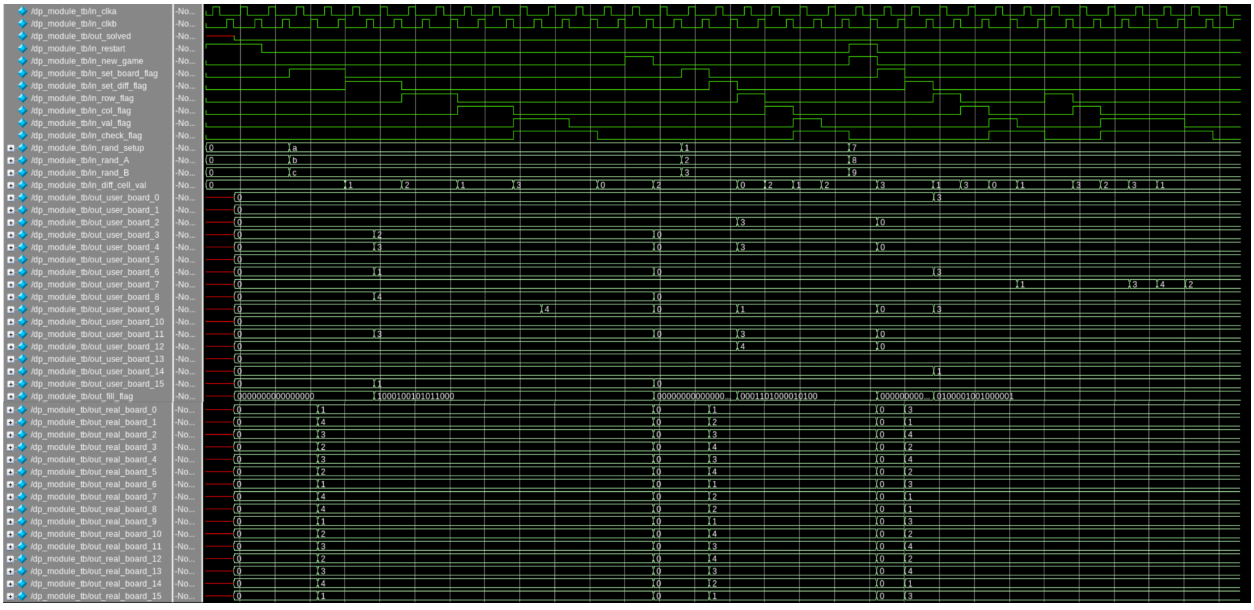
Cyan: When *in_set_diff_flag* is high, the DP reads the value of *in_diff_cell_val* which is now 2, corresponding to easy mode. Then, 5 hints are displayed on the *user_board* signals successfully at the next clock cycle.

Blue: The DP reads *in_diff_cell_val* to obtain the row (0) when *in_set_row_flag* is high, the column (2) when *in_set_col_flag* is high, and the value (1) when *in_set_diff_flag* is high. This corresponds to inserting 2 at cell 2, but in this case *out_user_board_2* is a hint which means the user should not be able to overwrite it, as verified with *out_fill_flag*. We see that at the next cycle *out_user_board_2* is not updated, which is the intended behavior to test if the hints are not writable.

Pink: For the last game, when *in_set_diff_flag* is high, the DP reads the value of *in_diff_cell_val* which is now 3, corresponding to hard mode. Then, 4 hints are displayed on the *user_board* signals successfully at the next clock cycle.

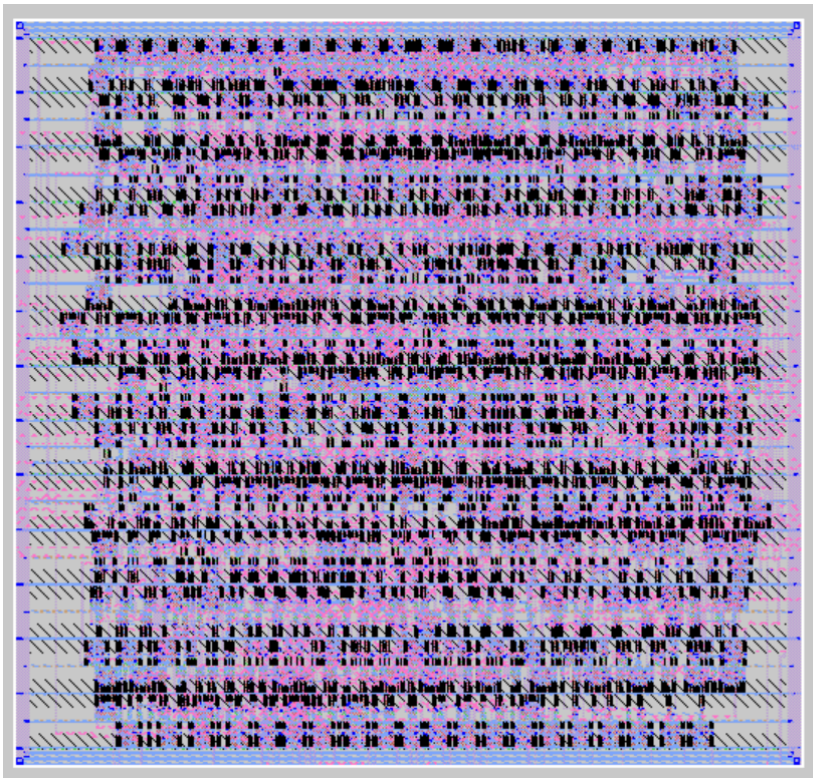
Purple: We see that the user is able to overwrite values to the same cell, in this case *out_user_board_7* is first updated with value of 1, and the user reselects the same row and column to overwrite it successfully. This part also tests that when *in_val_flag* is high and the value of *in_diff_cell* changes from 2 to 3 to 1, the corresponding values 3, 4, and 2 are updated to the user_board, meaning that the user is able to change their input value as long as the *in_val_flag* is high.

Questa Post Design Compiler

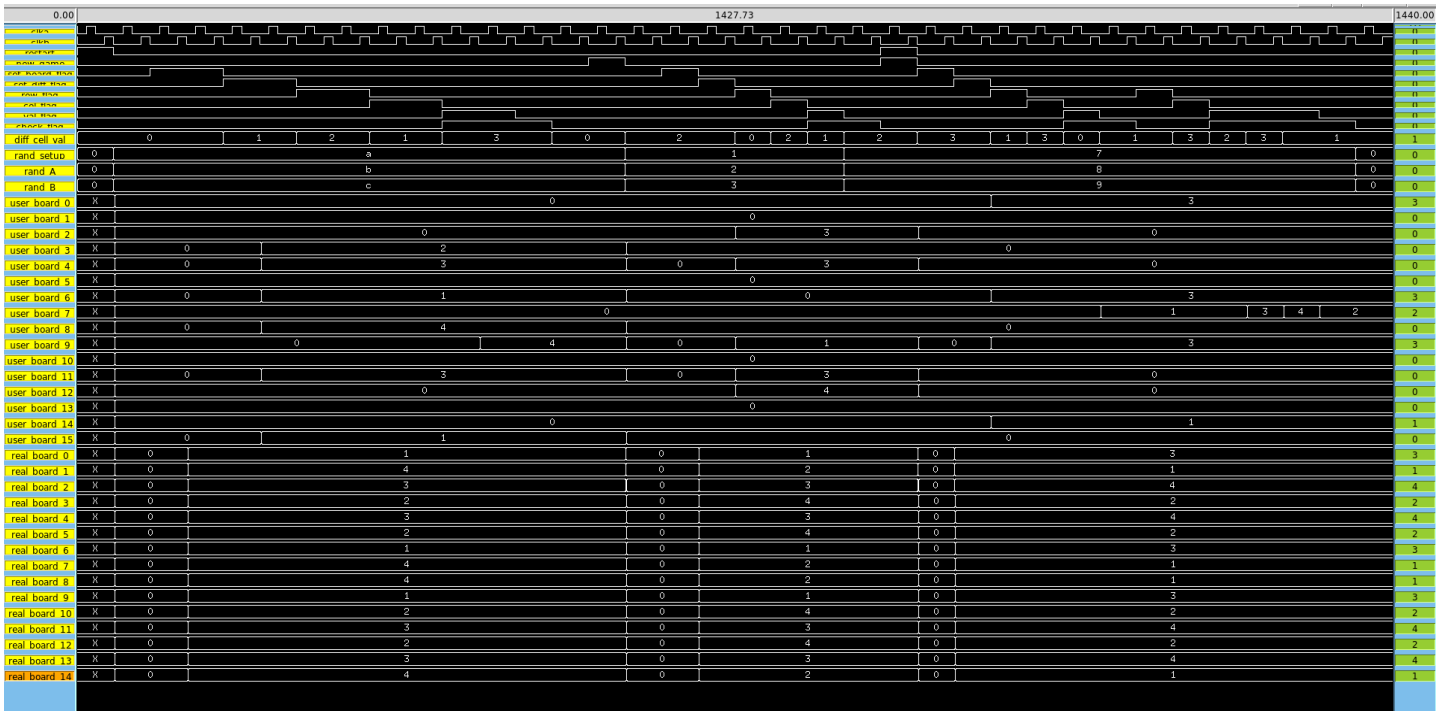


Post DC Questa behaves identical to Questa before synthesis, but with initial delays.

Magic



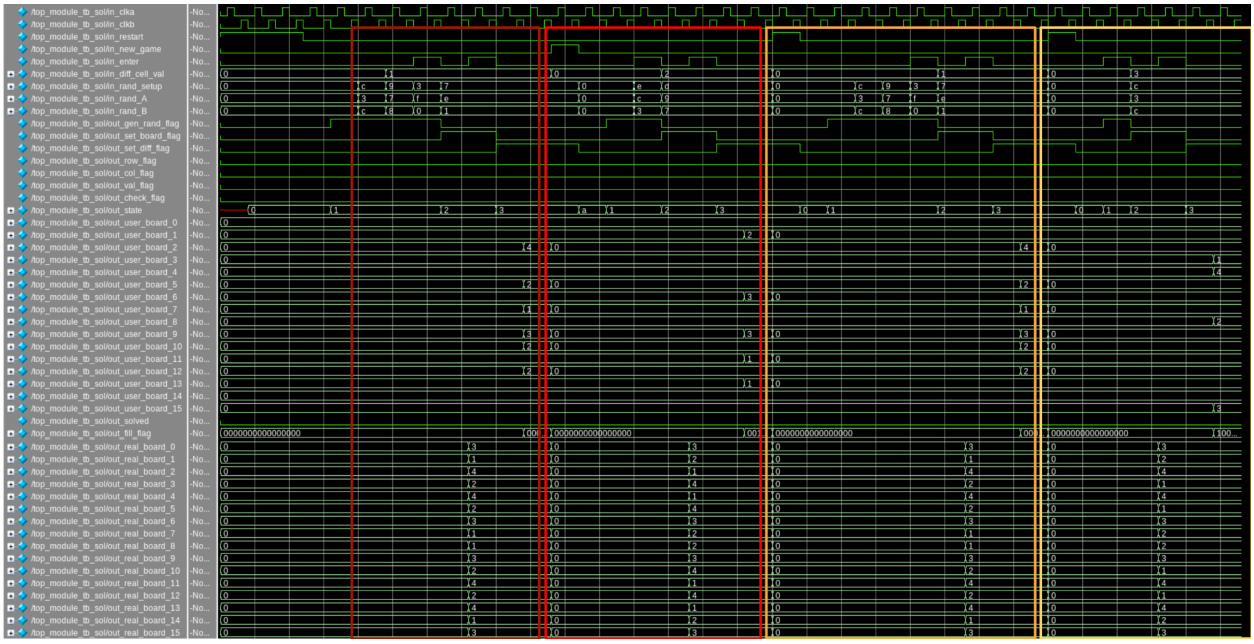
Irsim



Irsim behaves identical to Questa simulations from above.

Top Module Test: Generate Solutions

Questa



This testbench ensures that the sudoku core generates different solutions depending on the 3 random number inputs.

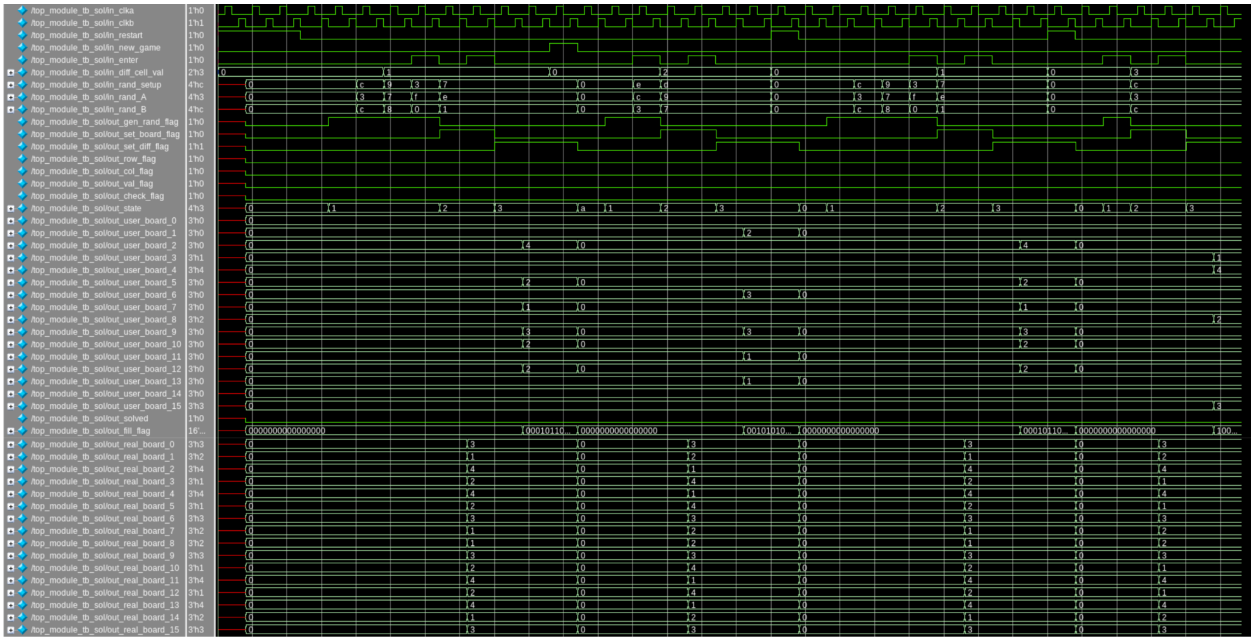
Brown: After *in_restart*, *out_real_board_0* to *out_real_board_15* are populated with the solution based on the preset for *in_rand_setup* = 7, *in_rand_A* = e, and *in_rand_B* = 1. Inserting *in_enter* then transitions to SET DIFF where easy mode is selected as *in_diff_cell_val* = 1, and 6 hints appear on the next clock cycle.

Red: After *in_new_game*, both the user_board and real_board are not cleared to 0, and the new set of solutions is generated for *in_rand_setup* = d, *in_rand_A* = 9, and *in_rand_B* = 7. Inserting *in_enter* then transitions to SET DIFF where easy mode is selected as *in_diff_cell_val* = 1, and 6 hints appear on the next clock cycle. Inserting *in_enter* then transitions to SET DIFF where medium mode is selected as *in_diff_cell_val* = 2, and 5 hints appear on the next clock cycle as expected.

Orange: Now *in_restart* is inserted for a second time, clearing the user_board and real_board as expected. We see that the random number sequence is the same as that in the brown box because the LFSR now starts from its predetermined initial state. We insert *in_enter* after waiting for the same time as in the brown box, which yields the same solution board as expected since the random number inputs are the same at *in_rand_setup* = 7, *in_rand_A* = e, and *in_rand_B* = 1.

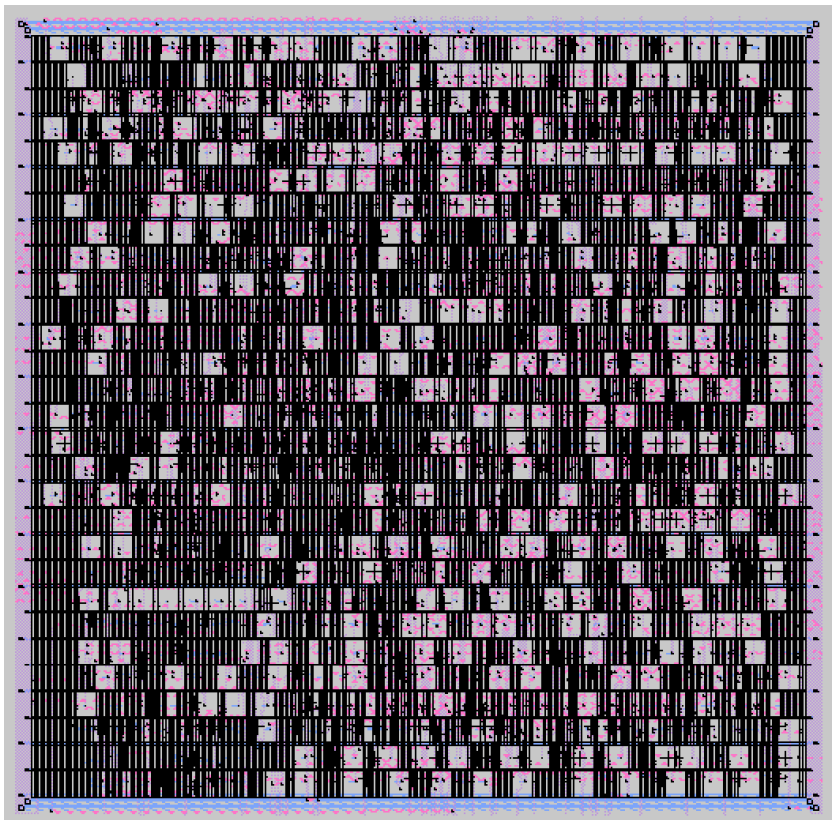
Yellow: We insert *in_restart* for the third time but wait for a shorter time than in the brown and yellow box to hit *in_enter*. This case shows that although *in_restart* yields the same number sequence from the LFSR, different board solutions can still be generated depending on when the user inputs *in_enter* to start the game.

Questa Post Design Compiler

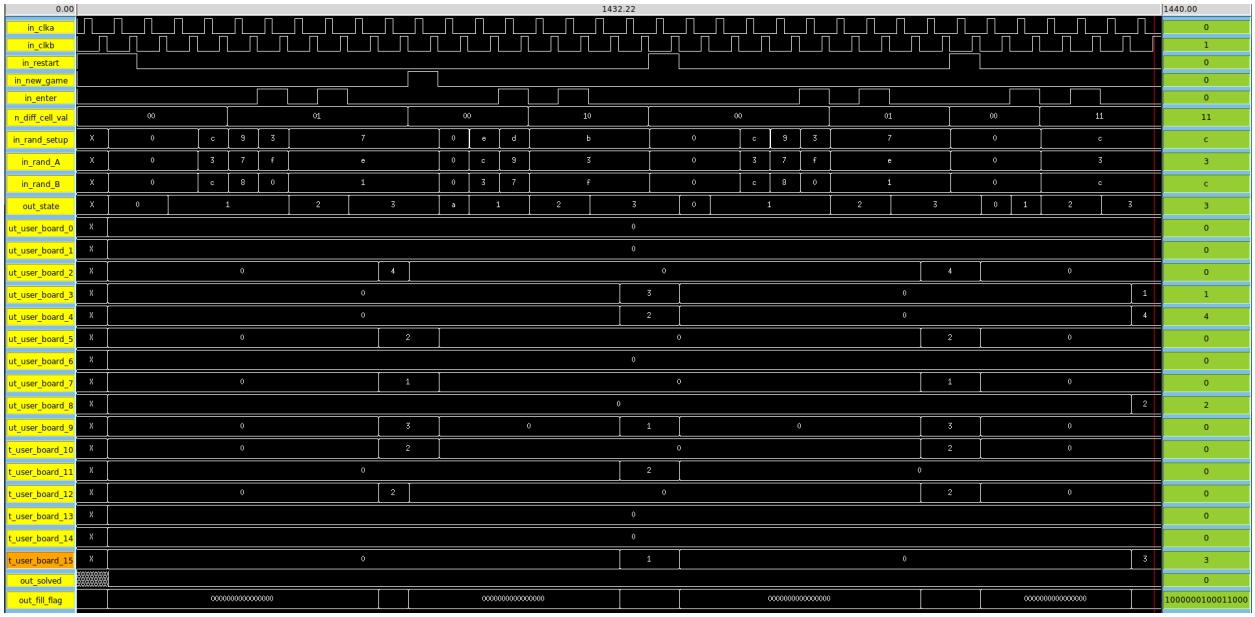


Post DC Questa behaves identical to Questa before synthesis, but with initial delays.

Magic



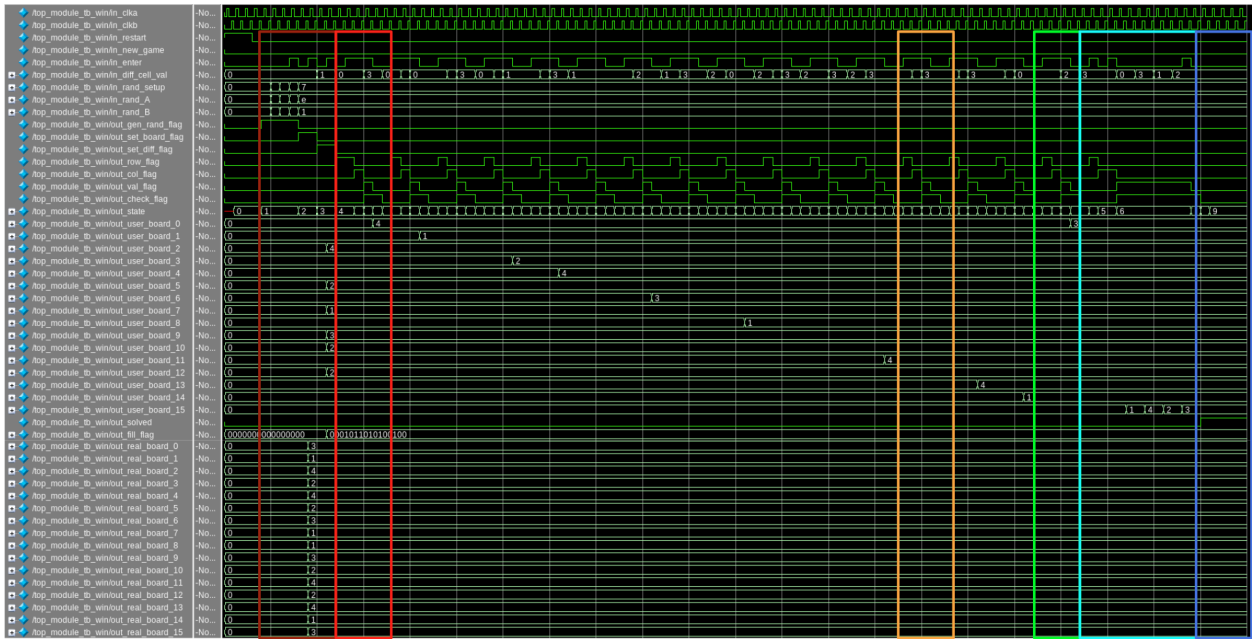
Irsim



Irsim behaves identical to Questa simulations from above.

Top Module Test: Win

Questa



This testbench tests that the user can reach the WIN state and *out_solved = 1* by filling out the board correctly.

Brown: We see that after *in_restart* all *out_user_board* and *out_real_board* initialize to 0, and we can start the LFSR by going to the GEN RAND (1) state. Subsequent *in_enter* moves to

SET BOARD (2), where *out_real_board* is updated to a preset solution based on *in_rand_setup* = 7, *in_rand_A* = e, and *in_rand_B* = 1. The next *in_enter* moves to SET DIFF (3) state, where easy mode is selected because *in_diff_cell* is last read as 1 before another *in_enter* transitions to CHOOSE ROW (4). We successfully see 6 hints appear on the user_board.

Red: With three consecutive *in_enter* signals, the user inputs *in_diff_cell_val = 0, in_diff_cell_val = 0, in_diff_cell_val = 3* for CHOOSE ROW, CHOOSE COL, CHOOSE VAL, which means filling *out_user_board_0* with 4. In the next clock cycle we see that *out_user_board_0* is indeed updated to 4.

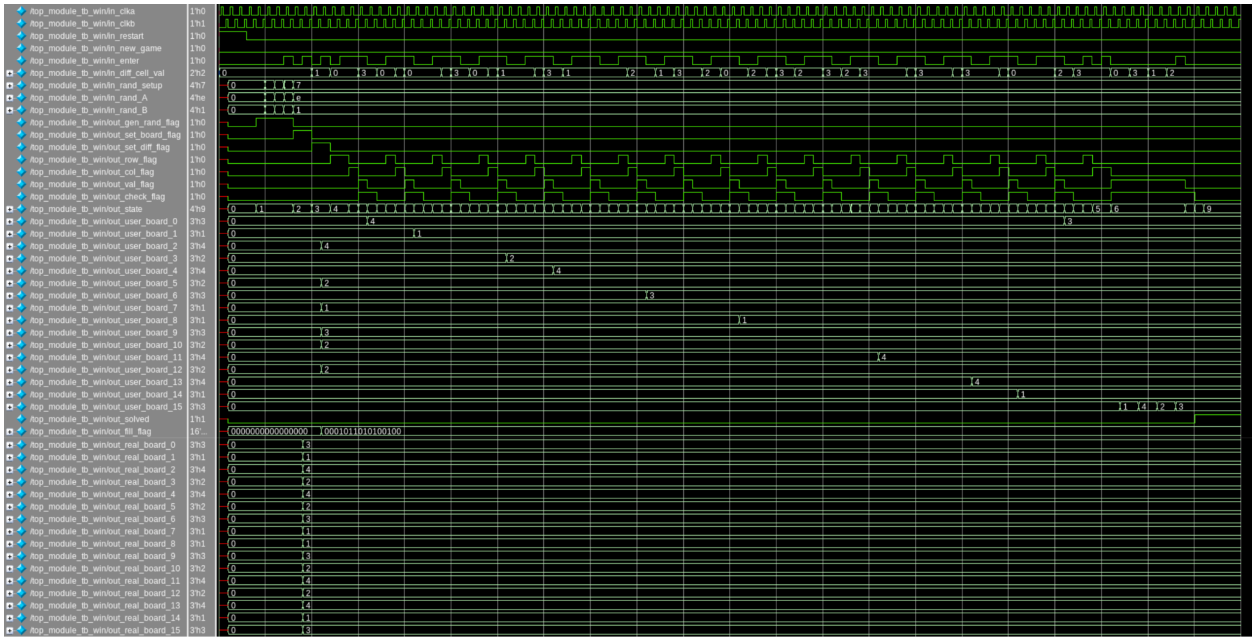
Orange: Again, with three consecutive *in_enter* signals, the user inputs *in_diff_cell_val = 3, in_diff_cell_val = 0, in_diff_cell_val = 3* for CHOOSE ROW, CHOOSE COL, CHOOSE VAL to try to fill *out_user_board_12* with 4. However, *out_user_board_12* is set as a hint as shown in the brown box, and we see that the users attempt to overwrite it fails as *out_user_board_12* remains to be 2.

Green: The user successfully rewrites *out_user_board_0* with a new value than that in the red box, showing it is possible to reselect a cell and overwrite if one intends to. However, this is only if the box filled is not a given hint.

Cyan: When the user selects *out_user_board_15*, we see that the different *in_diff_cell_val* values are updated to the board when in the CHOOSE VAL state, yet only the value that remains when *in_enter* is high is finalized. This tests that the user is able to change their input value of the cell before they confirm with *in_enter*.

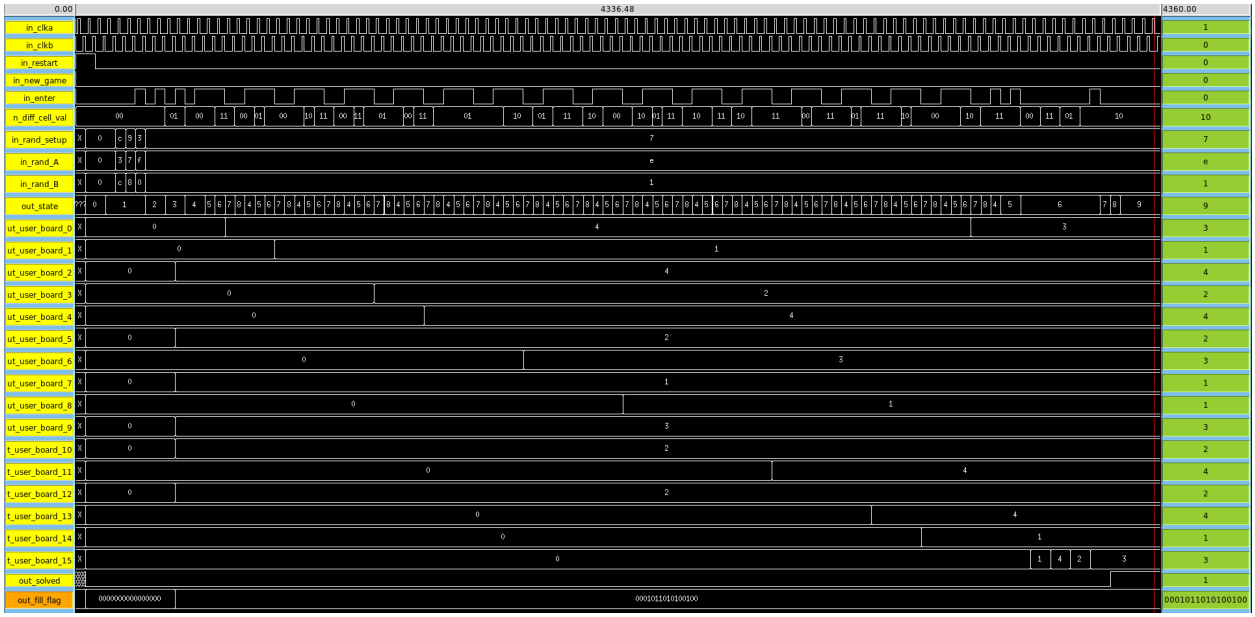
Blue: Once all the *out_user_board* values have been filled correctly, *out_solved* successfully becomes 1 and the user ends in the WIN (9) state.

Questa Post Design Compiler



Post DC Questa behaves identical to Questa before synthesis, but with initial delays.

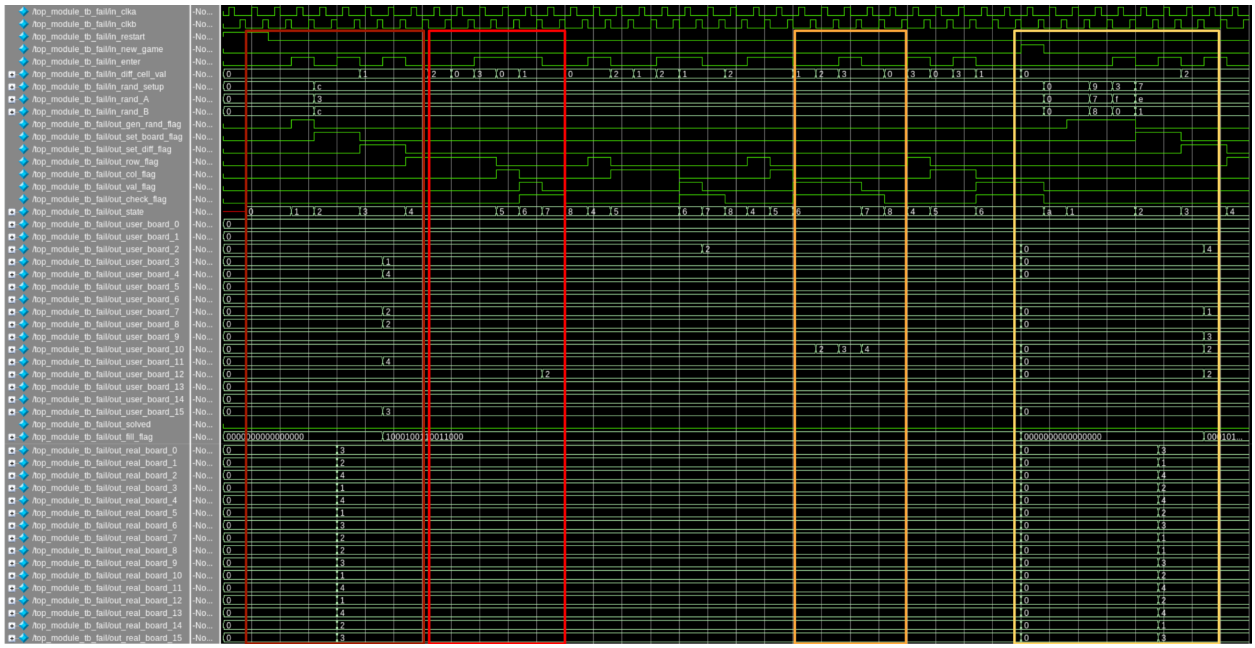
Irsim



Irsim behaves identical to Questa simulations from above.

Top Module: Fail and New Game

Questa



This testbench tests that gameboard can be generated and reset multiple times if the user decided to quit the current game.

Brown: We see that after *in_restart* all *out_user_board* and *out_real_board* initialize to 0, and we can start the LFSR by going to the GEN RAND (1) state. Subsequent *in_enter* moves to

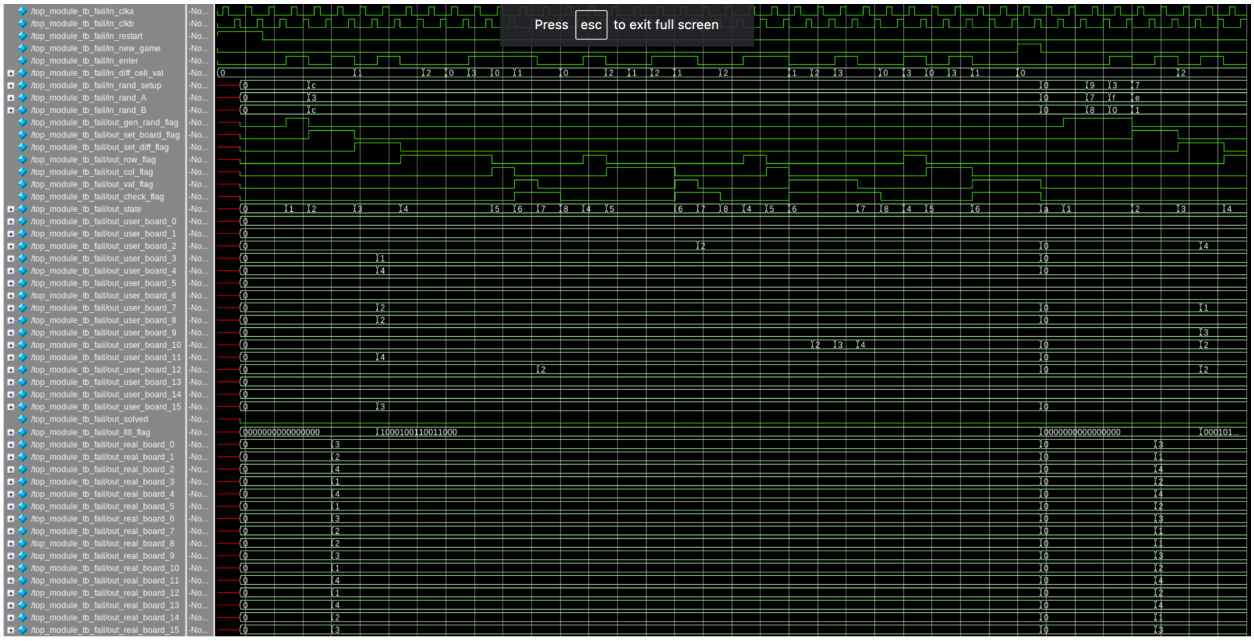
SET BOARD (2), where *out_real_board* is updated to a preset solution based on *in_rand_setup* = c, *in_rand_A* = 3, and *in_rand_B* = c. The next *in_enter* moves to SET DIFF (3) state, where easy mode is selected because *in_diff_cell* is last read as 1 before another *in_enter* transitions to CHOOSE ROW (4). We successfully see 6 hints appear on the user_board.

Red: The user is able to write values to a cell similar to the previous testbench.

Orange: The user is able to write values to a cell similar to the previous testbench.

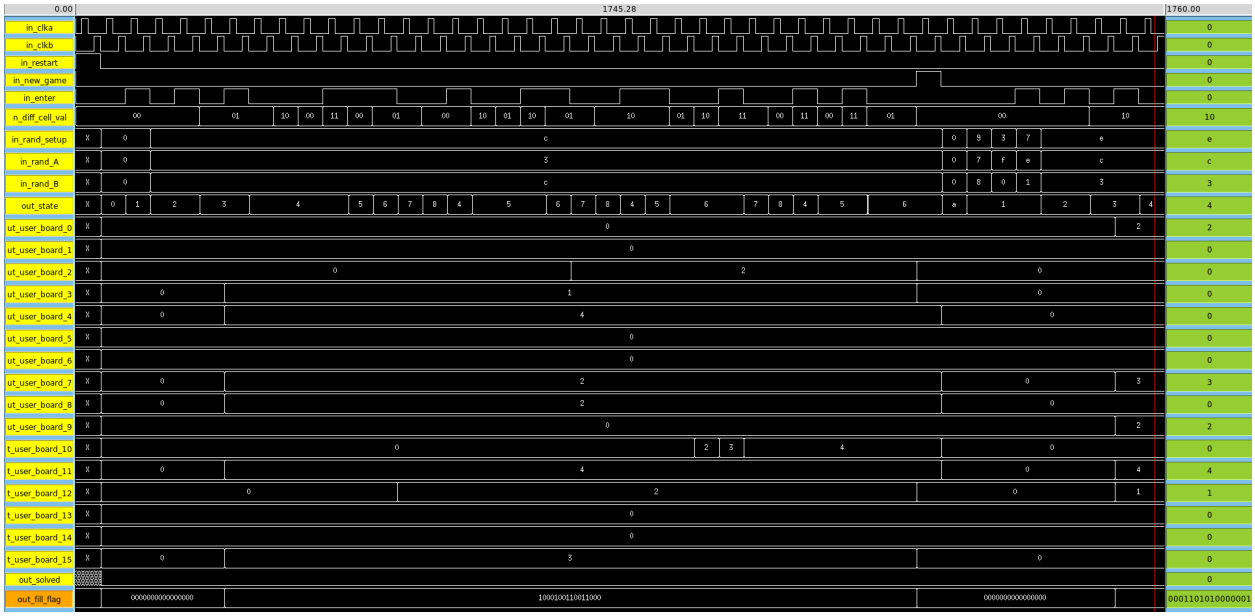
Yellow: During the game in CHOOSE COL, the user is able to quit the current game by inserting *in_new_game* which successfully clears *out_user_board* and *out_real_board*. Subsequently, they proceed to a new game with a new set of solutions and a different difficulty mode as shown.

Questa Post Design Compiler



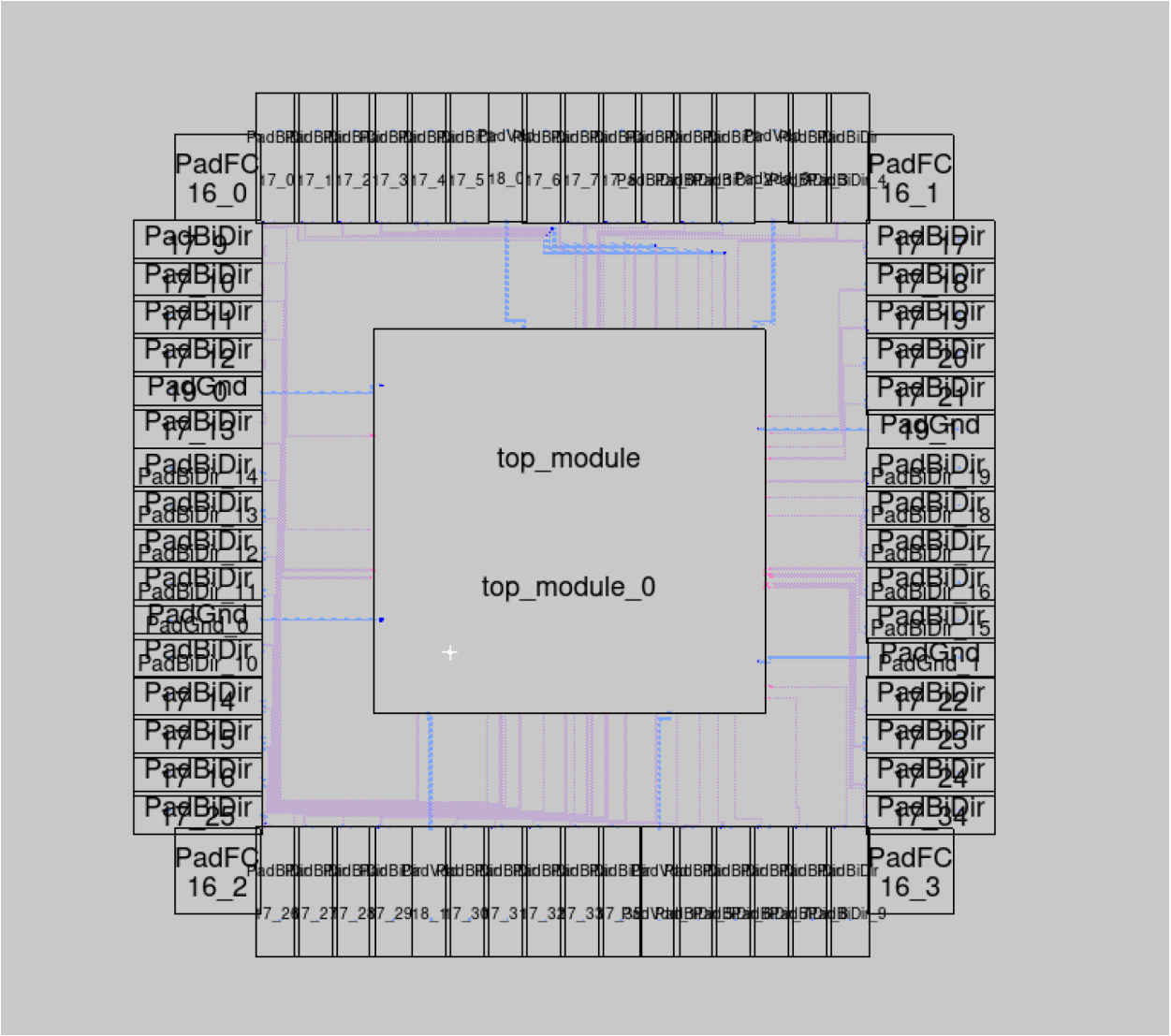
Post DC Questa behaves identical to Questa before synthesis, but with initial delays.

Irsim



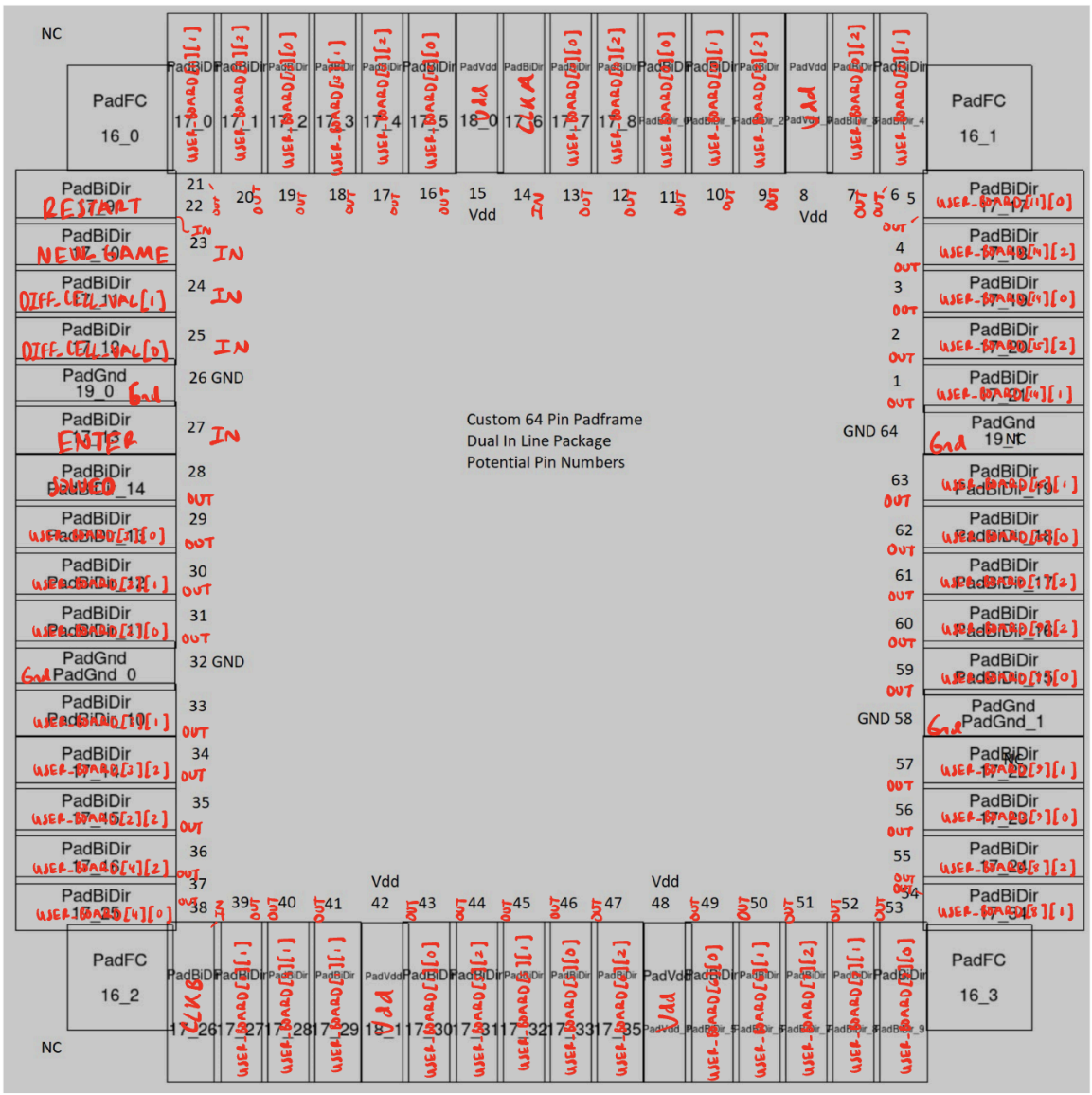
Irsim behaves identical to Questa simulations from above.

Pad Frame Integration



top_module is the Sudoku game core. All 56 I/O pads are used. Additionally, all 8 Vdd and Gnd pads are wired to the core. Note that this figure does not include extra polysilicon, metal1, and metal2 added to satisfy density requirements, but this *is* included in submission.

Pin Map



Pins are:

User_board[i][j] (OUT) - where i is the index of the cell on the board; where 0 is the top left, 3 is the top right, 12 is the bottom left, 15 is the bottom right, numbers ascending as if reading a book. J is the bit position within each 3 bit value that represents each cell in the board - 2 being the MSB and 0 being the LSB.

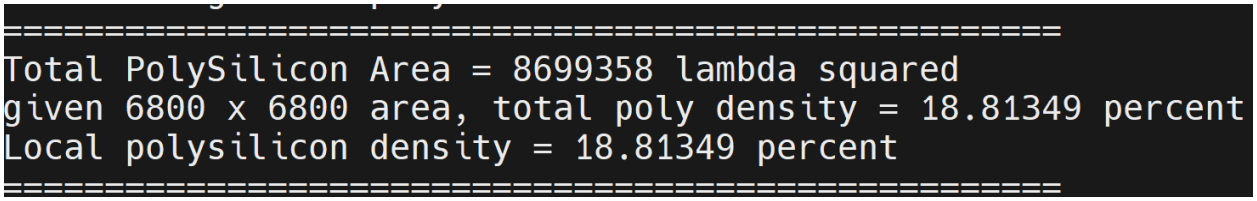
CLKA (IN) - Used for FSM combinational logic clocking on its negative edge.

CLKB (IN) - Used for FSM state switching on its negative edge.

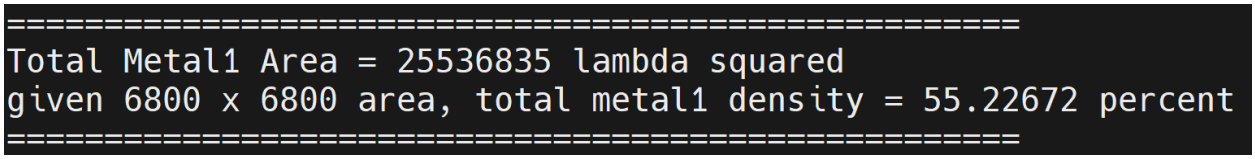
SOLVED (OUT) - Tells the user whether their current board matches the solution board generated by the game. High if matching, low otherwise. Therefore, no need for a “lose” output - you have either a matching solution or not at all times.
RESTART (IN) - Sets all of the registers to the default of 0.
NEW_GAME (IN) - Sets all of the registers to the default of 0. However, the LFSR churning process is not reset, guaranteeing a new unique board in the new game.
DIFF_CELL_VAL[i], i = 0 and 1 - Multi-purpose input channel for gameplay. In difficulty state, allows user to input a 2-bit binary number for selecting the difficulty level. Then is used to select row and column for number that will be inputted to the board. Finally, number is input with this channel with the already selected row and column values. i = 1 is the MSB, and i = 0 is the LSB.

Density Calculations

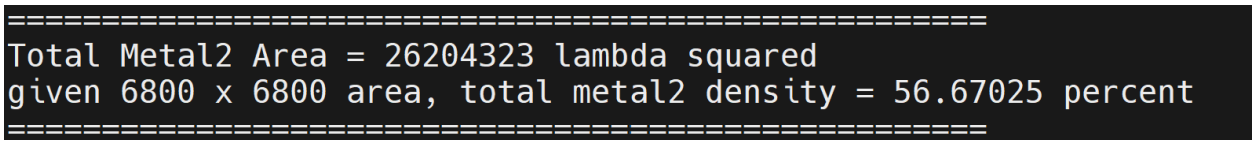
From the density calculation scripts given for calculating density of polysilicon, metal1, and metal2, density percentages are shown (in the order stated) below.



Polysilicon Density



Metal1 Density



Metal2 Density

All densities exceed the ideal minimum densities in the design of 15%, 30%, and 30% for Polysilicon, Metal1, and Metal2, respectively.

Core Power, Cell Area, and Timing Stats

From Synthesis reports, the following screen captures show the core power, cell area, and timing statistics of the chip.

```
--------------------------------------------------------------------------------
Total 1132 cells                                   449577.000000
```

The reported total cell area is approximately 449577μm^2.

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000            0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000            0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000            0.0000             0.0000  (   0.00%)
clock_network      4.3691           0.0000            0.0000             4.3691  (  73.88%)  i
register       7.9517e-02        2.1427e-02          36.3334             0.1010  (   1.71%)
sequential         0.6846        3.9421e-03          33.6146             0.6886  (  11.64%)
combinational      0.1925           0.5625           58.1497             0.7551  (  12.77%)
--------------------------------------------------------------------------------------------------
Total              5.3257 mW        0.5879 mW       128.0976 nW         5.9137 mW
```

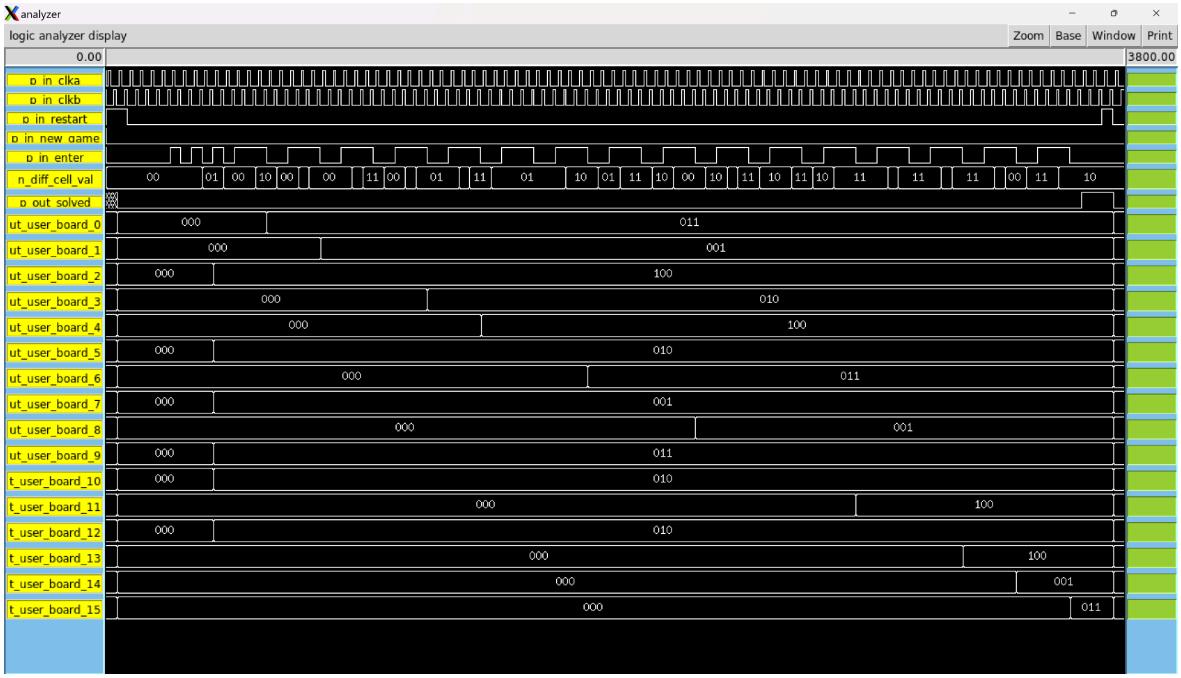
The reported total power consumption during the chip's operation is 5.9137mW.

```
Point                                                   Incr       Path
--------------------------------------------------------------------------
clock in_clka (rise edge)                               0.00       0.00
clock network delay (ideal)                             0.00       0.00
input external delay                                    1.00       1.00 r
in_restart (in)                                         0.06       1.06 r
U41/Y (INVX2)                                           0.15       1.21 f
U20/Y (INVX2)                                           0.37       1.58 r
Sudoku_DP/U678/Y (NOR2X1)                               0.26       1.84 f
U21/Y (BUFX2)                                           0.23       2.07 f
U10/Y (BUFX2)                                           0.26       2.34 f
Sudoku_DP/U550/Y (NAND3X1)                              0.54       2.88 r
U42/Y (INVX2)                                           0.55       3.43 f
U22/Y (INVX2)                                           0.50       3.93 r
U2/Y (INVX2)                                            0.63       4.56 f
Sudoku_DP/U287/Y (AOI22X1)                              0.25       4.81 r
Sudoku_DP/U286/Y (NAND2X1)                              0.36       5.17 f
Sudoku_DP/U5/Y (AND2X2)                                 0.41       5.58 f
Sudoku_DP/U281/Y (AOI22X1)                              0.14       5.72 r
Sudoku_DP/U280/Y (OAI21X1)                              0.08       5.80 f
Sudoku_DP/temp_user_board_6_reg[0]/D (DFFNEGX1)         0.00       5.80 f
data arrival time                                                  5.80

clock in_clka (fall edge)                              25.00      25.00
clock network delay (ideal)                             0.00      25.00
Sudoku_DP/temp_user_board_6_reg[0]/CLK (DFFNEGX1)       0.00      25.00 f
library setup time                                     -0.37      24.63
data required time                                                24.63
--------------------------------------------------------------------------
data required time                                                24.63
data arrival time                                                 -5.80
--------------------------------------------------------------------------
slack (MET)                                                       18.83
```

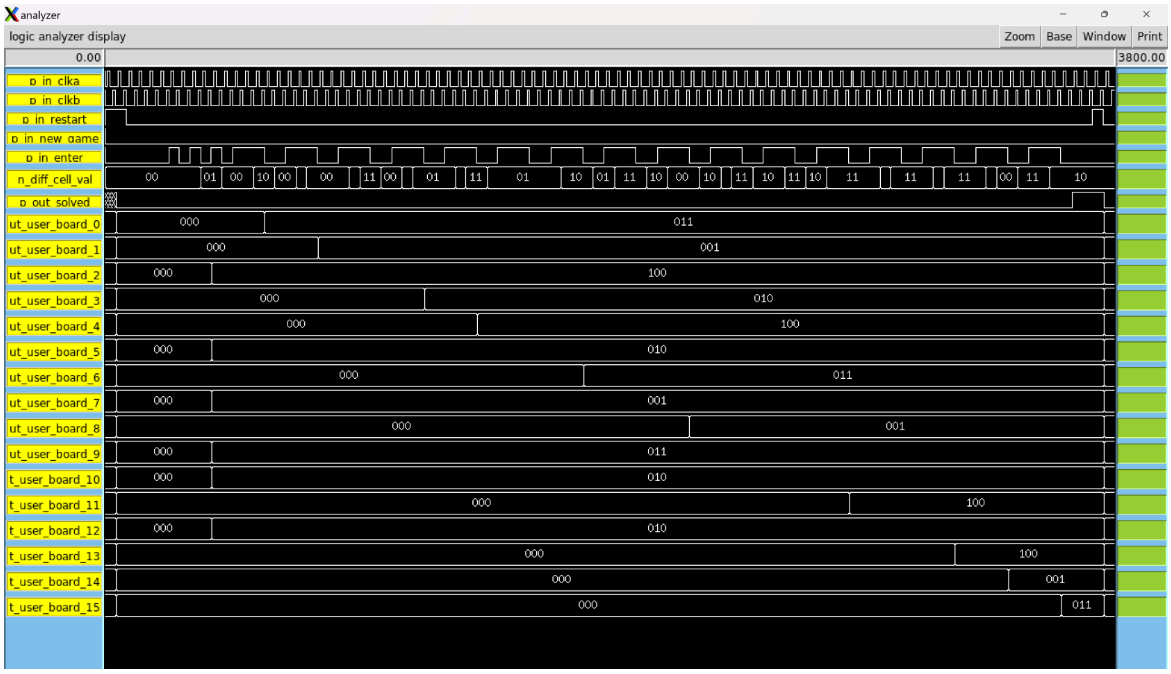
As can be seen, timing constraints for the chip are met after its synthesis. Total signal propagation time is 5.80ns-1.06ns = 4.74ns.

Testing



IRSIM on Hierarchical 64-pin PadFrame-integrated core

note: "..user_board_x" is "p_out_user_board_x", and "..._diff_cell_val" is "p_in_diff_cell_val".
Shows a normal game played with SOLVED asserted at the end.



IRSIM on flattened version of core integrated into 64-pin PadFrame (same as hierarchical)

Shows the same game played as in the hierarchical design IRSIM (with identical results).

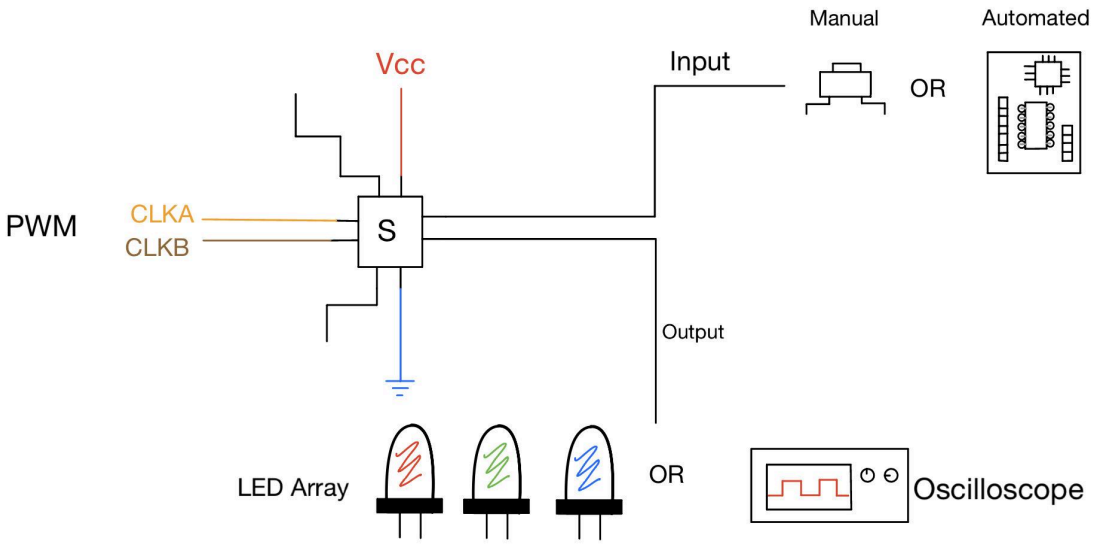
Unresolved Issues

There are no design rule errors or unresolved issues with the design.

Testing Strategy Post-Verification

Once the chip is fabricated, the first thing that needs to be done is to place the chip inside a chip socket such that all of the pads could be connected to external devices through wires. Then, using the pin out diagram as a guide on which pads correspond to which inputs and outputs, the external wiring could be done. The CLKA and CLKB signals could be generated either with a function generator from an oscilloscope or through a microcontroller such that the duty cycle of clocks matches that of the simulations. Furthermore, initially, it would be advisable to slow the clocks such that the change of inputs could be easily observed. Then, once comfortable with the clock speed, it could be increased to the frequency used in simulations. The inputs to the chip could either be generated with buttons such that the user would have to hold down a button to hit enter or to insert a value. A debounce feature may need to be implemented externally of the chip for testing and for actual gameplay. Another option would be to automate the inputs such that the input changes at the same rate of the PWM on a microcontroller such as an MSP430 Launchpad. Each input could be generated by a different pin on the MSP. The outputs to the chip could either be analyzed through an LED array where a diode lighting up represents a logical high. Another option for testing the chip is through an oscilloscope. This way, the rise and fall times of the chip can be tested as well as the general logical output. Through this design verification, post-fabrication testing is robust.

Wiring Diagram



Parts List

- 49 LEDs (for the outputs)
- 49 4.7k Ohm Resistors
- Wires
- Chip Socket (for Sudoku chip)
- 7 Buttons (for the inputs)
- MSP430 Launchpad
- NI VirtualBench Oscilloscope