Implementing 2D cellular automata in plain hardware (FPGA)

Dear all, it has been some time.

A couple of weeks ago, I had the honor to exhibit at the 2nd Panhellenic meeting on New Technologies, Robotics and Enterpreneurship (http://robo.teiste.gr). The meeting took place in Lamia, Greece (where I currently live) and the venue was at a convenient 2 min drive from home camp :)

I want to warmheartedly thank Prof. Panayotis Papazoglou (http://papazoglou.edu.gr) for the hospitality. He made a great effort in making for the second consecutive time the ROBO meeting a success!

My first exhibition at a ROBO meeting was based on an all-digital, all-hardware demo based on 2D cellular automata. The demo was nicknamed "digital kaleidoscope" but the work was actually done by 2D automata using the so-called rug rule.

Introduction to 2D cellular automata

In our case, these automata comprise of a two-dimensional matrix composed of identical cells, the internal state of which can be visualized by assigning it to pixels of a display through a palette of 256 colors.

According to the rug rule, the following three steps are executed:

- Calculate the sum of the values for the 8 neighbors (Moore neighborhood) of a given cell C.
- Divide by 8 to get their floored average.
- Calculate the new value for the cell, C', by adding a small integer increment. This computation takes place in modulo 256 arithmetic.

Following these simple steps for every cell, the digital kaleidoscope presents an explosive, chaotic and at the same time, highly interesting behavior.

Implementation

For the implementation, I followed a number of specific steps.

Software exploration

First, a software-based, host-running implementation was examined. I coded this in plain ANSI C, and used it to produce PPM snapshots for each generation of the automaton. Then using gifsicle, I had these PPMs converted to nice-looking animated

GIFs. These would allow me very early in the development cycle to have a grasp of how the hardware demo would potentionally look.

The code makes use of my libpnmio library for PBM/PGM/PPM I/O and is given here:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include "pnmio.h"

#define XDIM_DEFAULT 128
#define YDIM_DEFAULT 64
int step=1, incr=1, delay=0, gens=1;
char imgout_file_name[96];
FILE *imgout_file;
//
int img_xdim=XDIM_DEFAULT, img_ydim=YDIM_DEFAULT;
int *img_temp, *img_work, *img_out;

/* decode:
 * Decode the RGB encoding of the specified color.
 * NOTE: This scheme can only allow for up to 256 distinct colors
 * (essentially: R3G3B2).
 */
void decode(int c, int *red, int *green, int *blue)
{
  int t = c;
  *red = ((t >> 5) & 0x7) << 5;
  *green = ((t >> 2) & 0x7) << 5;
  *blue = ((t ) & 0x3) << 6;
}

/* rugca:
 * Generic implementation of the rug rule automaton.
 */
void rugca(int xsize, int ysize, int s, int inc, int g, int d)
{
  int i, k, x, y;
  int taddr, u, uaddr;
  int red, green, blue;
  int height=ysize, width=xsize;
  int cs;
  int sum=0;
  int x_offset[8] = {-1, 0, 1, 1, 1, 0,-1,-1};
  int y_offset[8] = {-1,-1,-1, 0, 1, 1, 1, 0};

  i = 0;
  while (i < g) {
```

```
    printf("### GENERATION %09d ###\n", i);

    // Print current generation.
    if ((i % s) == 0) {
      sprintf(imgout_file_name, "rugca-%09d.ppm", i);
      imgout_file = fopen(imgout_file_name, "w");
      for (y = 0; y < height; y++) {
        for (x = 0; x < width; x++) {
          taddr = y*width+x;
          decode(img_temp[taddr], &red, &green, &blue);
          img_out[3*taddr+0] = red;
          img_out[3*taddr+1] = green;
          img_out[3*taddr+2] = blue;
        }
      }
      write_ppm_file(imgout_file, img_out, imgout_file_name,
        xsize, ysize, 1, 1, 255);
      fclose(imgout_file);
    }

    // Calculate next grid state.
    for (y = 1; y < height-1; y++) {
      for (x = 1; x < width-1; x++) {
        sum = 0;
        taddr = y*width + x;
        for (k = 0; k < 8; k++) {
          uaddr = taddr + y_offset[k]*width + x_offset[k];
          u = img_temp[uaddr];
          sum += u;
        }
        // Averaging sum.
        sum = sum >> 3;
        // Increment cs, modulo 256.
        cs = (sum + inc) & 0xFF;
        img_work[taddr] = cs;
      }
    }

    // Copy back current generation.
    for (x = 0; x < width*height; x++) {
      img_temp[x] = img_work[x];
    }

    // Advance generation.
    i++;
  }
}

/* print_usage:
```

```
 * Print usage instructions for the "rugca" program.
 */
static void print_usage()
{
  printf("\n");
  printf("* Usage:\n");
  printf("* rugca [options]\n");
  printf("* \n");
  printf("* Options:\n");
  printf("* -h: Print this help.\n");
  printf("* -xsize <num>: Image width (Default: 128).\n");
  printf("* -ysize <num>: Image height (Default: 64).\n");
  printf("* -step <num>: Generate a PPM image every step generations (Defau
  printf("* -gens <num>: Total number of CCA generations (Default: 1).\n");
  printf("* -incr <num>: Cell increment (Default: 1).\n");
  printf("* -delay <num>: Delay factor for slowing down the main loop (Defa
  printf("* \n");
  printf("* For further information, please refer to the website:\n");
  printf("* http://www.nkavvadias.com\n\n");
}

/* main:
 * The main routine.
 */
int main(int argc, char **argv)
{
  int i, x, y;

  // Read input arguments
  if (argc < 2) {
    print_usage();
    exit(1);
  }

  for (i = 1; i < argc; i++) {
    if (strcmp("-h",argv[i]) == 0) {
      print_usage();
      exit(1);
    } else if (strcmp("-xsize", argv[i]) == 0) {
      if ((i+1) < argc) {
        i++;
        img_xdim = atoi(argv[i]);
      }
    } else if (strcmp("-ysize", argv[i]) == 0) {
      if ((i+1) < argc) {
        i++;
        img_ydim = atoi(argv[i]);
      }
    } else if (strcmp("-step", argv[i]) == 0) {
      if ((i+1) < argc) {
```

```
      i++;
      step = atoi(argv[i]);
    }
  } else if (strcmp("-gens", argv[i]) == 0) {
    if ((i+1) < argc) {
      i++;
      gens = atoi(argv[i]);
    }
  } else if (strcmp("-incr", argv[i]) == 0) {
    if ((i+1) < argc) {
      i++;
      incr = atoi(argv[i]);
    }
  } else if (strcmp("-delay", argv[i]) == 0) {
    if ((i+1) < argc) {
      i++;
      delay = atoi(argv[i]);
    }
  }
}

/* Allocate space for image data. */
img_temp = malloc(img_xdim * img_ydim * sizeof(int));
img_work = malloc(img_xdim * img_ydim * sizeof(int));
img_out = malloc(3 * img_xdim * img_ydim * sizeof(int));

for (y = 0; y < img_ydim; y++) {
  for (x = 0; x < img_xdim; x++) {
    img_temp[y*img_xdim+x] = 0x00;
  }
}

/* Perform operations. */
rugca(img_xdim, img_ydim, step, incr, gens, delay);

/* Deallocate memory. */
free(img_temp);
free(img_work);
free(img_out);

return 0;
}
```

Adapting reference C for high-level synthesis

Following this, the reference C code had to be adapted for high-level synthesis. I used my own high-level synthesis technology, named HercuLeS HLS: http://www.nkavvadias.com/hercules/

A detailed manual for HercuLeS can be found here: http://www.nkavvadias.com/hercules-reference-manual/hercules-refman.pdf

So HercuLeS can generate single IP blocks (for single procedures) or entire system IP (from a given translation unit with a number of procedures). In our case, we will be generating a single block IP with two streaming outputs,

- ok: is the state of the currently addressed cell
- xy: the address of that cell (linearized from 0 to XDIM*YDIM-1)

This block will then be incorporated in a given system I have developed for image and video synthesis demonstrations. This happens naturally in a plug-and-play way. Meaning that for custom procedural image/video generation, this system needs only be updated by the specific finite-state machine with datapath (FSMD) with proper streaming outputs for the purpose.

The C code is adapted to the following snippet and then it is passed to HercuLeS for cooking:

```
#define XSIZE       80
#define YSIZE       60
#define XYSIZE      XSIZE*YSIZE

void rugca(int *ok, int *xy)
{
  unsigned int i, j, k, x, y;
  unsigned int g=100000000, d=10000000;
  unsigned int taddr, uaddr;
  unsigned char cs, u, sum, nval;
  static unsigned char img_temp[XYSIZE], img_work[XYSIZE];
  static char x_offset[8] = {-1, 0, 1, 1, 1, 0,-1,-1};
  static char y_offset[8] = {-1,-1,-1, 0, 1, 1, 1, 0};
  // Default.
  unsigned char incr=3;

  for (y = 0; y < YSIZE; y++) {
    for (x = 0; x < XSIZE; x++) {
      img_temp[y*XSIZE+x] = ((x*y) >> 8) & 0x1;
    }
  }

  i = 0;
  while (i < g) {

    // Calculate next grid state.
    for (y = 1; y < YSIZE-1; y++) {
      for (x = 1; x < XSIZE-1; x++) {
        sum = 0;
        taddr = y*XSIZE + x;
        for (k = 0; k < 8; k++) {
          uaddr = taddr + y_offset[k]*XSIZE + x_offset[k];
          u     = img_temp[uaddr];
          sum   = sum + u;
        }
```

```
        // Averaging sum.
        sum = sum >> 3;
        // Increment cs, modulo 256.
        nval = (sum + incr) & 0xFF;
        cs = img_temp[taddr];
        *ok = cs;
        *xy = taddr;
        img_work[taddr] = nval;
      }
    }

    // Copy back current generation.
    for (x = 0; x < XYSIZE; x++) {
      img_temp[x] = img_work[x];
    }
    j = 0;
    while (j < d) {
      j++;
    }

    // Advance generation.
    i++;
  }
}
```

Automatically generated VHDL from HercuLeS

HercuLeS now is ready to rumble. Within a few tens of seconds, the VHDL code for the block is generated. Remember that "humans were not involved in the process :)". So let's see what we can do with this automatically-generated code. First, let's see how does it look like:

```
library IEEE;
use WORK.operpack.all;
use WORK.rugca_cdt_pkg.all;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity rugca is
  port (
    clk : in std_logic;
    reset : in std_logic;
    start : in std_logic;
    mode : in std_logic_vector(3 downto 0);
    ok : out std_logic_vector(7 downto 0);
    xy : out std_logic_vector(12 downto 0);
    valid : out std_logic;
    done : out std_logic;
    ready : out std_logic
  );
```

```
end rugca;

architecture fsmd of rugca is
  type state_type is (S_ENTRY, S_EXIT, S_001_001, S_001_002, S_001_003, S_0
  S_003_001, S_003_002, S_003_003, S_003_004, S_003_005, S_003_006, S_004_0
  S_006_001, S_007_001, S_007_002, S_007_003, S_008_001, S_008_002, S_008_0
  S_009_001, S_009_002, S_009_003, S_010_001, S_010_002, S_010_003, S_010_0
  S_011_001, S_011_002, S_011_003, S_011_004, S_011_005, S_011_006, S_011_0
  S_012_001, S_012_002, S_012_003, S_013_001, S_013_002, S_013_003, S_014_0
  S_015_001, S_015_002, S_015_003, S_015_004, S_018_001, S_018_002, S_019_0
  S_020_001, S_021_001, S_022_001, S_023_001, S_024_001, S_025_001, S_026_0
  S_027_001, S_028_001, S_029_001, S_029_002, S_029_003, S_030_001,
  S_031_001, S_031_002, S_031_003, S_032_001, S_033_001, S_033_002, S_033_0
  S_034_001, S_034_002, S_034_003, S_034_004, S_035_001,
  S_036_001, S_036_002, S_036_003, S_037_001, S_037_002, S_037_003,
  S_038_001, S_039_001, S_039_002, S_039_003, S_040_001,
  S_041_001, S_041_002, S_042_001);
  signal current_state, next_state: state_type;
  signal img_temp_we : std_logic;
  signal img_temp_addr : std_logic_vector(12 downto 0);
  signal img_temp_din : std_logic_vector(7 downto 0);
  signal img_temp_dout : std_logic_vector(7 downto 0);
  signal img_work_we : std_logic;
  signal img_work_addr : std_logic_vector(12 downto 0);
  signal img_work_din : std_logic_vector(7 downto 0);
  signal img_work_dout : std_logic_vector(7 downto 0);
  signal x_offset_we : std_logic;
  signal x_offset_addr : std_logic_vector(2 downto 0);
  signal x_offset_din : std_logic_vector(7 downto 0);
  signal x_offset_dout : std_logic_vector(7 downto 0);
  signal y_offset_we : std_logic;
  signal y_offset_addr : std_logic_vector(2 downto 0);
  signal y_offset_din : std_logic_vector(7 downto 0);
  signal y_offset_dout : std_logic_vector(7 downto 0);
  signal x_1_next : std_logic_vector(31 downto 0);
  signal x_1_reg : std_logic_vector(31 downto 0);
  signal j_1_next : std_logic_vector(31 downto 0);
  signal j_1_reg : std_logic_vector(31 downto 0);
  signal i_1_next : std_logic_vector(31 downto 0);
  signal i_1_reg : std_logic_vector(31 downto 0);
  signal D_1408_1_next : std_logic_vector(31 downto 0);
  signal D_1408_1_reg : std_logic_vector(31 downto 0);
  signal y_1_next : std_logic_vector(31 downto 0);
  signal y_1_reg : std_logic_vector(31 downto 0);
  signal taddr_1_next : std_logic_vector(31 downto 0);
  signal taddr_1_reg : std_logic_vector(31 downto 0);
  signal D_1417_1_next : std_logic_vector(31 downto 0);
  signal D_1417_1_reg : std_logic_vector(31 downto 0);
  signal uaddr_1_next : std_logic_vector(31 downto 0);
  signal uaddr_1_reg : std_logic_vector(31 downto 0);
```

```
signal sum_1_next : std_logic_vector(15 downto 0);
signal sum_1_reg : std_logic_vector(15 downto 0);
signal k_1_next : std_logic_vector(31 downto 0);
signal k_1_reg : std_logic_vector(31 downto 0);
signal D_1429_1_next : std_logic_vector(7 downto 0);
signal D_1429_1_reg : std_logic_vector(7 downto 0);
signal D_1412_1_next : std_logic_vector(7 downto 0);
signal D_1412_1_reg : std_logic_vector(7 downto 0);
signal g_1_next : std_logic_vector(31 downto 0);
signal g_1_reg : std_logic_vector(31 downto 0);
signal d_1_next : std_logic_vector(31 downto 0);
signal d_1_reg : std_logic_vector(31 downto 0);
signal c_1_next : std_logic_vector(7 downto 0);
signal c_1_reg : std_logic_vector(7 downto 0);
signal D_1439_1_next : std_logic_vector(7 downto 0);
signal D_1439_1_reg : std_logic_vector(7 downto 0);
signal D_1413_1_next : std_logic_vector(7 downto 0);
signal D_1413_1_reg : std_logic_vector(7 downto 0);
signal D_1418_1_next : std_logic_vector(7 downto 0);
signal D_1418_1_reg : std_logic_vector(7 downto 0);
signal u_1_next : std_logic_vector(7 downto 0);
signal u_1_reg : std_logic_vector(7 downto 0);
signal u_next : std_logic_vector(15 downto 0);
signal u_reg : std_logic_vector(15 downto 0);
signal cs_1_next : std_logic_vector(7 downto 0);
signal cs_1_reg : std_logic_vector(7 downto 0);
signal x_next : std_logic_vector(31 downto 0);
signal x_reg : std_logic_vector(31 downto 0);
signal j_next : std_logic_vector(31 downto 0);
signal j_reg : std_logic_vector(31 downto 0);
signal i_next : std_logic_vector(31 downto 0);
signal i_reg : std_logic_vector(31 downto 0);
signal y_next : std_logic_vector(31 downto 0);
signal y_reg : std_logic_vector(31 downto 0);
signal k_next : std_logic_vector(31 downto 0);
signal k_reg : std_logic_vector(31 downto 0);
signal taddr_next : std_logic_vector(31 downto 0);
signal taddr_reg : std_logic_vector(31 downto 0);
signal sum_next : std_logic_vector(15 downto 0);
signal sum_reg : std_logic_vector(15 downto 0);
signal g_next : std_logic_vector(31 downto 0);
signal g_reg : std_logic_vector(31 downto 0);
signal cs_next : std_logic_vector(7 downto 0);
signal cs_reg : std_logic_vector(7 downto 0);
signal d_next : std_logic_vector(31 downto 0);
signal d_reg : std_logic_vector(31 downto 0);
signal nval_next : std_logic_vector(15 downto 0);
signal nval_reg : std_logic_vector(15 downto 0);
signal mode16_next : std_logic_vector(15 downto 0);
signal mode16_reg : std_logic_vector(15 downto 0);
```

```
signal D_1407_1_next : std_logic_vector(31 downto 0);
signal D_1407_1_reg : std_logic_vector(31 downto 0);
signal D_1409_1_next : std_logic_vector(31 downto 0);
signal D_1409_1_reg : std_logic_vector(31 downto 0);
signal D_1415_1_next : std_logic_vector(31 downto 0);
signal D_1415_1_reg : std_logic_vector(31 downto 0);
signal D_1410_1_next : std_logic_vector(31 downto 0);
signal D_1410_1_reg : std_logic_vector(31 downto 0);
signal D_1414_1_next : std_logic_vector(31 downto 0);
signal D_1414_1_reg : std_logic_vector(31 downto 0);
signal D_1422_1_next : std_logic_vector(31 downto 0);
signal D_1422_1_reg : std_logic_vector(31 downto 0);
signal taddr_0_1_next : std_logic_vector(31 downto 0);
signal taddr_0_1_reg : std_logic_vector(31 downto 0);
signal D_1411_1_next : std_logic_vector(7 downto 0);
signal D_1411_1_reg : std_logic_vector(7 downto 0);
signal D_1416_1_next : std_logic_vector(31 downto 0);
signal D_1416_1_reg : std_logic_vector(31 downto 0);
signal D_1419_1_next : std_logic_vector(31 downto 0);
signal D_1419_1_reg : std_logic_vector(31 downto 0);
signal ok_next : std_logic_vector(7 downto 0);
signal ok_reg : std_logic_vector(7 downto 0);
signal xy_next : std_logic_vector(12 downto 0);
signal xy_reg : std_logic_vector(12 downto 0);
signal serenity_next : std_logic;
signal serenity_reg : std_logic;
signal waitstate_next : std_logic;
signal waitstate_reg : std_logic;
constant CNST_0 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_1 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_500000 : std_logic_vector(63 downto 0)    := "0000000000000
constant CNST_2000000 : std_logic_vector(63 downto 0)   := "0000000000000
constant CNST_10000000 : std_logic_vector(63 downto 0)  := "0000000000000
constant CNST_25000000 : std_logic_vector(63 downto 0)  := "0000000000000
constant CNST_100000000 : std_logic_vector(63 downto 0) := "0000000000000
constant CNST_2 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_254 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_3 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_4 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_4799 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_5 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_58 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_59 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_6 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_7 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_78 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_79 : std_logic_vector(63 downto 0) := "0000000000000000000000
constant CNST_8 : std_logic_vector(63 downto 0)     := "0000000000000000000
constant CNST_80 : std_logic_vector(63 downto 0)    := "0000000000000000000
constant CNST_118 : std_logic_vector(63 downto 0)   := "0000000000000000000
```

```
  constant CNST_119 : std_logic_vector(63 downto 0)   := "0000000000000000
  constant CNST_158 : std_logic_vector(63 downto 0)   := "0000000000000000
  constant CNST_159 : std_logic_vector(63 downto 0)   := "0000000000000000
  constant CNST_160 : std_logic_vector(63 downto 0)   := "0000000000000000
  constant CNST_19199 : std_logic_vector(63 downto 0) := "0000000000000000
begin
  -- current state logic
  process (clk, reset)
  begin
    if (reset = '1') then
      current_state <= S_ENTRY;
      x_1_reg <= (others => '0');
      j_1_reg <= (others => '0');
      i_1_reg <= (others => '0');
      D_1408_1_reg <= (others => '0');
      y_1_reg <= (others => '0');
      taddr_1_reg <= (others => '0');
      D_1417_1_reg <= (others => '0');
      uaddr_1_reg <= (others => '0');
      sum_1_reg <= (others => '0');
      k_1_reg <= (others => '0');
      D_1429_1_reg <= (others => '0');
      D_1412_1_reg <= (others => '0');
      g_1_reg <= (others => '0');
      d_1_reg <= (others => '0');
      c_1_reg <= (others => '0');
      D_1439_1_reg <= (others => '0');
      D_1413_1_reg <= (others => '0');
      D_1418_1_reg <= (others => '0');
      u_1_reg <= (others => '0');
      u_reg <= (others => '0');
      cs_1_reg <= (others => '0');
      x_reg <= (others => '0');
      j_reg <= (others => '0');
      i_reg <= (others => '0');
      y_reg <= (others => '0');
      k_reg <= (others => '0');
      taddr_reg <= (others => '0');
      sum_reg <= (others => '0');
      g_reg <= (others => '0');
      cs_reg <= (others => '0');
      d_reg <= (others => '0');
      nval_reg <= (others => '0');
      mode16_reg <= (others => '0');
      D_1407_1_reg <= (others => '0');
      D_1409_1_reg <= (others => '0');
      D_1415_1_reg <= (others => '0');
      D_1410_1_reg <= (others => '0');
      D_1414_1_reg <= (others => '0');
      D_1422_1_reg <= (others => '0');
```

```
    taddr_0_1_reg <= (others => '0');
    D_1411_1_reg <= (others => '0');
    D_1416_1_reg <= (others => '0');
    D_1419_1_reg <= (others => '0');
    ok_reg <= (others => '0');
    xy_reg <= (others => '0');
    serenity_reg <= '0';
    waitstate_reg <= '0';
  elsif (clk = '1' and clk'EVENT) then
    current_state <= next_state;
    x_1_reg <= x_1_next;
    j_1_reg <= j_1_next;
    i_1_reg <= i_1_next;
    D_1408_1_reg <= D_1408_1_next;
    y_1_reg <= y_1_next;
    taddr_1_reg <= taddr_1_next;
    D_1417_1_reg <= D_1417_1_next;
    uaddr_1_reg <= uaddr_1_next;
    sum_1_reg <= sum_1_next;
    k_1_reg <= k_1_next;
    D_1429_1_reg <= D_1429_1_next;
    D_1412_1_reg <= D_1412_1_next;
    g_1_reg <= g_1_next;
    d_1_reg <= d_1_next;
    c_1_reg <= c_1_next;
    D_1439_1_reg <= D_1439_1_next;
    D_1413_1_reg <= D_1413_1_next;
    D_1418_1_reg <= D_1418_1_next;
    u_1_reg <= u_1_next;
    u_reg <= u_next;
    cs_1_reg <= cs_1_next;
    x_reg <= x_next;
    j_reg <= j_next;
    i_reg <= i_next;
    y_reg <= y_next;
    k_reg <= k_next;
    taddr_reg <= taddr_next;
    sum_reg <= sum_next;
    g_reg <= g_next;
    cs_reg <= cs_next;
    d_reg <= d_next;
    nval_reg <= nval_next;
    mode16_reg <= mode16_next;
    D_1407_1_reg <= D_1407_1_next;
    D_1409_1_reg <= D_1409_1_next;
    D_1415_1_reg <= D_1415_1_next;
    D_1410_1_reg <= D_1410_1_next;
    D_1414_1_reg <= D_1414_1_next;
    D_1422_1_reg <= D_1422_1_next;
    taddr_0_1_reg <= taddr_0_1_next;
```

```
    D_1411_1_reg <= D_1411_1_next;
    D_1416_1_reg <= D_1416_1_next;
    D_1419_1_reg <= D_1419_1_next;
    ok_reg <= ok_next;
    xy_reg <= xy_next;
    serenity_reg <= serenity_next;
    waitstate_reg <= waitstate_next;
  end if;
end process;

-- next state and output logic
process (current_state, start, mode,
  ok_reg,
  xy_reg,
  serenity_reg, serenity_next,
  waitstate_reg, waitstate_next,
  img_temp_dout,
  img_work_dout,
  x_offset_dout,
  y_offset_dout,
  x_1_reg, x_1_next,
  j_1_reg, j_1_next,
  i_1_reg, i_1_next,
  D_1408_1_reg, D_1408_1_next,
  y_1_reg, y_1_next,
  taddr_1_reg, taddr_1_next,
  D_1417_1_reg, D_1417_1_next,
  uaddr_1_reg, uaddr_1_next,
  sum_1_reg, sum_1_next,
  k_1_reg, k_1_next,
  D_1429_1_reg, D_1429_1_next,
  D_1412_1_reg, D_1412_1_next,
  g_1_reg, g_1_next,
  d_1_reg, d_1_next,
  c_1_reg, c_1_next,
  D_1439_1_reg, D_1439_1_next,
  D_1413_1_reg, D_1413_1_next,
  D_1418_1_reg, D_1418_1_next,
  u_1_reg, u_1_next,
  u_reg, u_next,
  cs_1_reg, cs_1_next,
  x_reg, x_next,
  j_reg, j_next,
  i_reg, i_next,
  y_reg, y_next,
  k_reg, k_next,
  taddr_reg, taddr_next,
  sum_reg, sum_next,
  g_reg, g_next,
  cs_reg, cs_next,
```

```
  d_reg, d_next,
  nval_reg, nval_next,
  mode16_reg, mode16_next,
  D_1407_1_reg, D_1407_1_next,
  D_1409_1_reg, D_1409_1_next,
  D_1415_1_reg, D_1415_1_next,
  D_1410_1_reg, D_1410_1_next,
  D_1414_1_reg, D_1414_1_next,
  D_1422_1_reg, D_1422_1_next,
  taddr_0_1_reg, taddr_0_1_next,
  D_1411_1_reg, D_1411_1_next,
  D_1416_1_reg, D_1416_1_next,
  D_1419_1_reg, D_1419_1_next
)
begin
  valid <= '0';
  done <= '0';
  ready <= '0';
  x_1_next <= x_1_reg;
  j_1_next <= j_1_reg;
  i_1_next <= i_1_reg;
  D_1408_1_next <= D_1408_1_reg;
  y_1_next <= y_1_reg;
  taddr_1_next <= taddr_1_reg;
  D_1417_1_next <= D_1417_1_reg;
  uaddr_1_next <= uaddr_1_reg;
  sum_1_next <= sum_1_reg;
  k_1_next <= k_1_reg;
  D_1429_1_next <= D_1429_1_reg;
  D_1412_1_next <= D_1412_1_reg;
  g_1_next <= g_1_reg;
  d_1_next <= d_1_reg;
  c_1_next <= c_1_reg;
  D_1439_1_next <= D_1439_1_reg;
  D_1413_1_next <= D_1413_1_reg;
  D_1418_1_next <= D_1418_1_reg;
  u_1_next <= u_1_reg;
  u_next <= u_reg;
  cs_1_next <= cs_1_reg;
  x_next <= x_reg;
  j_next <= j_reg;
  i_next <= i_reg;
  y_next <= y_reg;
  k_next <= k_reg;
  taddr_next <= taddr_reg;
  sum_next <= sum_reg;
  g_next <= g_reg;
  cs_next <= cs_reg;
  d_next <= d_reg;
  nval_next <= nval_reg;
```

```
mode16_next <= mode16_reg;
D_1407_1_next <= D_1407_1_reg;
D_1409_1_next <= D_1409_1_reg;
D_1415_1_next <= D_1415_1_reg;
D_1410_1_next <= D_1410_1_reg;
D_1414_1_next <= D_1414_1_reg;
D_1422_1_next <= D_1422_1_reg;
taddr_0_1_next <= taddr_0_1_reg;
D_1411_1_next <= D_1411_1_reg;
D_1416_1_next <= D_1416_1_reg;
D_1419_1_next <= D_1419_1_reg;
ok_next <= ok_reg;
xy_next <= xy_reg;
serenity_next <= serenity_reg;
waitstate_next <= waitstate_reg;
img_temp_we <= '0';
img_temp_addr <= (others => '0');
img_temp_din <= (others => '0');
img_work_we <= '0';
img_work_addr <= (others => '0');
img_work_din <= (others => '0');
x_offset_we <= '0';
x_offset_addr <= (others => '0');
x_offset_din <= (others => '0');
y_offset_we <= '0';
y_offset_addr <= (others => '0');
y_offset_din <= (others => '0');
case current_state is
  when S_ENTRY =>
    ready <= '1';
    if (start = '1') then
      next_state <= S_001_001;
    else
      next_state <= S_ENTRY;
    end if;
  when S_001_001 =>
    g_1_next <= CNST_100000000(31 downto 0);
    d_1_next <= CNST_500000(31 downto 0);
    y_1_next <= CNST_0(31 downto 0);
    next_state <= S_001_002;
  when S_001_002 =>
    y_next <= y_1_reg(31 downto 0);
    g_next <= g_1_reg(31 downto 0);
    d_next <= d_1_reg(31 downto 0);
    next_state <= S_001_003;
  when S_001_003 =>
    next_state <= S_006_001;
  when S_002_001 =>
    x_1_next <= CNST_0(31 downto 0);
    next_state <= S_002_002;
```

```
when S_002_002 =>
  x_next <= x_1_reg(31 downto 0);
  next_state <= S_002_003;
when S_002_003 =>
  next_state <= S_004_001;
when S_003_001 =>
  x_1_next <= std_logic_vector(unsigned(x_reg) + unsigned(CNST_1(31 d
  D_1407_1_next <= mul(y_reg, CNST_80(31 downto 0), '0', 32);
  next_state <= S_003_002;
when S_003_002 =>
  D_1408_1_next <= std_logic_vector(unsigned(D_1407_1_reg) + unsigned
  next_state <= S_003_003;
when S_003_003 =>
  x_next <= x_1_reg(31 downto 0);
  next_state <= S_003_004;
when S_003_004 =>
  D_1412_1_next <= (others => '0');
  next_state <= S_003_005;
when S_003_005 =>
  img_temp_we <= '1';
  img_temp_addr <= D_1408_1_reg(12 downto 0);
  img_temp_din <= D_1412_1_reg(7 downto 0);
  next_state <= S_003_006;
when S_003_006 =>
  next_state <= S_004_001;
when S_004_001 =>
  if (x_reg <= CNST_79(31 downto 0)) then
    next_state <= S_003_001;
  else
    next_state <= S_005_001;
  end if;
when S_005_001 =>
  y_1_next <= std_logic_vector(unsigned(y_reg) + unsigned(CNST_1(31 d
  next_state <= S_005_002;
when S_005_002 =>
  y_next <= y_1_reg(31 downto 0);
  next_state <= S_005_003;
when S_005_003 =>
  next_state <= S_006_001;
when S_006_001 =>
  if (y_reg <= CNST_59(31 downto 0)) then
    next_state <= S_002_001;
  else
    next_state <= S_007_001;
  end if;
when S_007_001 =>
  i_1_next <= CNST_0(31 downto 0);
  next_state <= S_007_002;
when S_007_002 =>
  i_next <= i_1_reg(31 downto 0);
```

```
  next_state <= S_007_003;
when S_007_003 =>
  next_state <= S_040_001;
when S_008_001 =>
  y_1_next(31 downto 8) <= (others => '0');
  y_1_next(7 downto 0) <= CNST_1(7 downto 0);
  next_state <= S_008_002;
when S_008_002 =>
  y_next <= y_1_reg(31 downto 0);
  next_state <= S_008_003;
when S_008_003 =>
  next_state <= S_032_001;
when S_009_001 =>
  x_1_next(31 downto 8) <= (others => '0');
  x_1_next(7 downto 0) <= CNST_1(7 downto 0);
  next_state <= S_009_002;
when S_009_002 =>
  x_next <= x_1_reg(31 downto 0);
  next_state <= S_009_003;
when S_009_003 =>
  next_state <= S_030_001;
when S_010_001 =>
  sum_1_next <= CNST_0(15 downto 0);
  k_1_next <= CNST_0(31 downto 0);
  D_1407_1_next <= mul(y_reg, CNST_80(31 downto 0), '0', 32);
  next_state <= S_010_002;
when S_010_002 =>
  taddr_1_next <= std_logic_vector(unsigned(D_1407_1_reg) + unsigned(
  k_next <= k_1_reg(31 downto 0);
  sum_next <= sum_1_reg(15 downto 0);
  next_state <= S_010_003;
when S_010_003 =>
  taddr_next <= taddr_1_reg(31 downto 0);
  next_state <= S_010_004;
when S_010_004 =>
  next_state <= S_014_001;
when S_011_001 =>
  y_offset_addr <= k_reg(2 downto 0);
  x_offset_addr <= k_reg(2 downto 0);
  waitstate_next <= not (waitstate_reg);
  if (waitstate_reg = '1') then
    D_1413_1_next <= y_offset_dout;
    D_1418_1_next <= x_offset_dout;
    next_state <= S_011_002;
  else
    next_state <= S_011_001;
  end if;
when S_011_002 =>
  D_1414_1_next(31 downto 8) <= (others => D_1413_1_reg(7));
  D_1414_1_next(7 downto 0) <= D_1413_1_reg;
```

```
    D_1419_1_next(31 downto 8) <= (others => D_1418_1_reg(7));
    D_1419_1_next(7 downto 0) <= D_1418_1_reg;
    next_state <= S_011_003;
  when S_011_003 =>
    D_1415_1_next <= mul(D_1414_1_reg, CNST_80(31 downto 0), '1', 32);
    next_state <= S_011_004;
  when S_011_004 =>
    D_1416_1_next(31 downto 0) <= D_1415_1_reg;
    next_state <= S_011_005;
  when S_011_005 =>
    D_1417_1_next <= std_logic_vector(signed(D_1416_1_reg) + signed(tac
    next_state <= S_011_006;
  when S_011_006 =>
    uaddr_1_next <= std_logic_vector(signed(D_1417_1_reg) + signed(D_14
    next_state <= S_011_007;
  when S_011_007 =>
    img_temp_addr <= uaddr_1_reg(12 downto 0);
    waitstate_next <= not (waitstate_reg);
    if (waitstate_reg = '1') then
      u_1_next <= img_temp_dout;
      next_state <= S_011_008;
    else
      next_state <= S_011_007;
    end if;
  when S_011_008 =>
  u_next <= X"00" & u_1_reg(7 downto 0);
    next_state <= S_012_001;
  when S_012_001 =>
    sum_1_next <= std_logic_vector(unsigned(sum_1_reg) + unsigned(u_reg
    next_state <= S_012_002;
  when S_012_002 =>
    sum_next <= sum_1_reg(15 downto 0);
    next_state <= S_013_001;
  when S_013_001 =>
    k_1_next <= std_logic_vector(unsigned(k_reg) + unsigned(CNST_1(31 d
    next_state <= S_013_002;
  when S_013_002 =>
    k_next <= k_1_reg(31 downto 0);
    next_state <= S_013_003;
  when S_013_003 =>
  mode16_next <= X"000" & mode(3 downto 0);
sum_next <= "000" & sum_reg(15 downto 3);
    next_state <= S_014_001;
  when S_014_001 =>
    if (k_reg <= CNST_7(31 downto 0)) then
      next_state <= S_011_001;
    else
      next_state <= S_014a_001;
    end if;
  when S_014a_001 =>
```

```
  nval_next <= std_logic_vector(unsigned(sum_reg) + unsigned(mode16_r
  next_state <= S_015_001;
when S_015_001 =>
  taddr_0_1_next(31 downto 0) <= taddr_reg;
  img_temp_addr <= taddr_reg(12 downto 0);
  waitstate_next <= not (waitstate_reg);
  if (waitstate_reg = '1') then
    cs_1_next <= img_temp_dout;
    next_state <= S_015_002;
  else
    next_state <= S_015_001;
  end if;
when S_015_002 =>
  xy_next <= taddr_0_1_reg(12 downto 0);
  cs_next <= cs_1_reg(7 downto 0);
  D_1422_1_next(31 downto 8) <= (others => '0');
  D_1422_1_next(7 downto 0) <= cs_1_reg;
  serenity_next <= not (serenity_reg);
  if (serenity_reg = '1') then
    valid <= '1';
    next_state <= S_015_003;
  else
    next_state <= S_015_002;
  end if;
when S_015_003 =>
  ok_next <= D_1422_1_reg(7 downto 0);
  serenity_next <= not (serenity_reg);
  if (serenity_reg = '1') then
    valid <= '1';
    next_state <= S_015_004;
  else
    next_state <= S_015_003;
  end if;
when S_015_004 =>
  next_state <= S_018_001;
when S_018_001 =>
  img_work_we <= '1';
  img_work_addr <= taddr_reg(12 downto 0);
  img_work_din <= nval_reg(7 downto 0);
  next_state <= S_019_001;
when S_019_001 =>
  next_state <= S_020_001;
when S_020_001 =>
  next_state <= S_021_001;
when S_021_001 =>
  next_state <= S_022_001;
when S_022_001 =>
  next_state <= S_023_001;
when S_023_001 =>
  next_state <= S_024_001;
```

```
when S_024_001 =>
  next_state <= S_025_001;
when S_025_001 =>
  next_state <= S_026_001;
when S_026_001 =>
  next_state <= S_027_001;
when S_027_001 =>
  next_state <= S_028_001;
when S_028_001 =>
  next_state <= S_029_001;
when S_029_001 =>
  x_1_next <= std_logic_vector(unsigned(x_reg) + unsigned(CNST_1(31 d
  next_state <= S_029_002;
when S_029_002 =>
  x_next <= x_1_reg(31 downto 0);
  next_state <= S_029_003;
when S_029_003 =>
  next_state <= S_030_001;
when S_030_001 =>
  if (x_reg <= CNST_78(31 downto 0)) then
    next_state <= S_010_001;
  else
    next_state <= S_031_001;
  end if;
when S_031_001 =>
  y_1_next <= std_logic_vector(unsigned(y_reg) + unsigned(CNST_1(31 d
  next_state <= S_031_002;
when S_031_002 =>
  y_next <= y_1_reg(31 downto 0);
  next_state <= S_031_003;
when S_031_003 =>
  next_state <= S_032_001;
when S_032_001 =>
  if (y_reg <= CNST_58(31 downto 0)) then
    next_state <= S_009_001;
  else
    next_state <= S_033_001;
  end if;
when S_033_001 =>
  x_1_next <= CNST_0(31 downto 0);
  next_state <= S_033_002;
when S_033_002 =>
  x_next <= x_1_reg(31 downto 0);
  next_state <= S_033_003;
when S_033_003 =>
  next_state <= S_035_001;
when S_034_001 =>
  x_1_next <= std_logic_vector(unsigned(x_reg) + unsigned(CNST_1(31 d
  img_work_addr <= x_reg(12 downto 0);
  waitstate_next <= not (waitstate_reg);
```

```
  if (waitstate_reg = '1') then
    D_1439_1_next <= img_work_dout;
    next_state <= S_034_002;
  else
    next_state <= S_034_001;
  end if;
when S_034_002 =>
  img_temp_we <= '1';
  img_temp_addr <= x_reg(12 downto 0);
  img_temp_din <= D_1439_1_reg(7 downto 0);
  next_state <= S_034_003;
when S_034_003 =>
  x_next <= x_1_reg(31 downto 0);
  next_state <= S_034_004;
when S_034_004 =>
  next_state <= S_035_001;
when S_035_001 =>
  if (x_reg <= CNST_4799(31 downto 0)) then
    next_state <= S_034_001;
  else
    next_state <= S_036_001;
  end if;
when S_036_001 =>
  j_1_next <= CNST_0(31 downto 0);
  next_state <= S_036_002;
when S_036_002 =>
  j_next <= j_1_reg(31 downto 0);
  next_state <= S_036_003;
when S_036_003 =>
  next_state <= S_038_001;
when S_037_001 =>
  j_1_next <= std_logic_vector(unsigned(j_reg) + unsigned(CNST_1(31 d
  next_state <= S_037_002;
when S_037_002 =>
  j_next <= j_1_reg(31 downto 0);
  next_state <= S_037_003;
when S_037_003 =>
  next_state <= S_038_001;
when S_038_001 =>
  if (j_reg < d_reg(31 downto 0)) then
    next_state <= S_037_001;
  else
    next_state <= S_039_001;
  end if;
when S_039_001 =>
  i_1_next <= std_logic_vector(unsigned(i_reg) + unsigned(CNST_1(31 d
  next_state <= S_039_002;
when S_039_002 =>
  i_next <= i_1_reg(31 downto 0);
  next_state <= S_039_003;
```

```
    when S_039_003 =>
      next_state <= S_040_001;
    when S_040_001 =>
      if (i_reg < g_reg(31 downto 0)) then
        next_state <= S_008_001;
      else
        next_state <= S_041_001;
      end if;
    when S_041_001 =>
      next_state <= S_042_001;
    when S_042_001 =>
      next_state <= S_EXIT;
    when S_EXIT =>
      done <= '1';
      next_state <= S_ENTRY;
    when others =>
      next_state <= S_ENTRY;
  end case;
end process;

ok <= ok_reg;
xy <= xy_reg;

img_temp_instance : entity WORK.ram(img_temp)
  generic map (
    AW     => 13,
    DW     => 8,
    NR     => 4800
  )
  port map (
    clk    => clk,
    we     => img_temp_we,
    en     => '1',
    rwaddr => img_temp_addr,
    din    => img_temp_din,
    dout   => img_temp_dout
  );

img_work_instance : entity WORK.ram(img_work)
  generic map (
    AW     => 13,
    DW     => 8,
    NR     => 4800
  )
  port map (
    clk    => clk,
    we     => img_work_we,
    en     => '1',
    rwaddr => img_work_addr,
    din    => img_work_din,
```

```
      dout    => img_work_dout
    );

  x_offset_instance : entity WORK.ram(x_offset)
    generic map (
      AW      => 3,
      DW      => 8,
      NR      => 8
    )
    port map (
      clk     => clk,
      we      => x_offset_we,
      en      => '1',
      rwaddr  => x_offset_addr,
      din     => x_offset_din,
      dout    => x_offset_dout
    );

  y_offset_instance : entity WORK.ram(y_offset)
    generic map (
      AW      => 3,
      DW      => 8,
      NR      => 8
    )
    port map (
      clk     => clk,
      we      => y_offset_we,
      en      => '1',
      rwaddr  => y_offset_addr,
      din     => y_offset_din,
      dout    => y_offset_dout
    );

end fsmd;
```

Wowa! That's a lot of stuff that went on in HercuLeS. It seems that it did the work. A self-checking testbench was also automatically generated by HercuLeS but we will not focus on that in this particular blog post.

Technically, this is a single FSMD with separate processes for current state logic and next-state/output logic. Datapath actions are embedded within the next-state/output logic process, no messy code with concurrent assignments (has its pros and cons). Overall, the code closely follows the FSMD paradigm as presented in Prof. D. Gajski works and how this scheme was presented in Prof. Pong P. Chu's books (I own two of them).

The automatically-generated implementation uses a kind of triple-buffering. We need a working and a temporary memory for the automaton world, representing generations n and n+1. In the hardware-oriented version, it is of size 80×60, using 8×8 upscaling, due to the limits of the available internal RAM of the FPGA device (block RAM), which is around 360 kbits (we will use around 70% of this). For better visual output, and since we will run computations within the video on timings, we use a sep-

arate, third memory, as a video frame buffer. Of course, improvements are possible against this scheme, e.g. by using line buffers are doing all computations within the blanking interval durations. It will also be interesting to port this demo to another board using fast, zero-cycle turnaround SRAM.

About the exhibition

I had a great time with the exhibition, moving away from remote customer interaction (and their virtual whiplashes :) and meeting a lot of people in person, including school children, parents, technology afficionados, higher education students, hobbyists, local industry, teachers and professors.

This is what my demo looked like (so it is true hardware, no hidden computers running the show, I had to point this out a lot). It appears I was a little tired, but hey this was towards the end of the day (and I needed refueling).


PIRELLI

And another shot of the demo:


FLATRON
LG
ΨΗΦΙΑΚΟ ΚΑΛΕΙΔΟΣΚΟΠΙΟ
ΜΕ ΧΡΗΣΗ ΤΕΧΝΟΛΟΓΙΑΣ FPGA
ΝΙΚΟΛΑΟΣ ΚΑΒΒΑΔΙΑΣ
http://www.nkavvadias.com

I have uploaded two short videos showcasing the digital kaleidoscope demo at my YouTube channel:

- Overview of the demo: http://www.youtube.com/watch?v=ahyBUAFcXHw
- Starting sequence: http://www.youtube.com/watch?v=-sxB8DSznGU

The hardware is using a delay loop in order to let humans visualize the process. The increment parameter of the automaton is controlled by the four slide switches available on the specific Digilent board and we can set any value from 0 to 15.

Technology used for the demo and summary

The digital circuit was designed in the VHDL hardware description language.

To dramatically reduce design time, the behavior of the circuit was first described in the C programming language. The C program was automatically translated to VHDL using the HercuLeS high-level synthesis tool.

The resulting description was then synthesized on an FPGA integrated circuit (Xilinx XC3S700AN) using the Xilinx ISE/XST logic synthesis environment.

The development board which has been used is the Xilinx Spartan-3AN Starter Kit by Digilent.

Wrap-up

Folks, I hope you have enjoyed this short (or long) walkthrough through the lost art-land of Kaveirian (that's me) high-level synthesis. My next steps would involve pretty much everything, after all HercuLeS is used for day-by-day, real-life, commercial-grade work; most frequently for work intended for clients (that most times cannot be disclosed).

So I am thinking of a more impressive set of demos, like an algorithmically-generated 3D world which you can explore via a simple keyboard interface, 3D graphics demos (all done in plain hardware), chess engines, obscure IOCCC entries, etc. I am always collecting ideas across the web, especially "mini-codes" or "tiny-codes" that could be turned into interesting hardware demos.