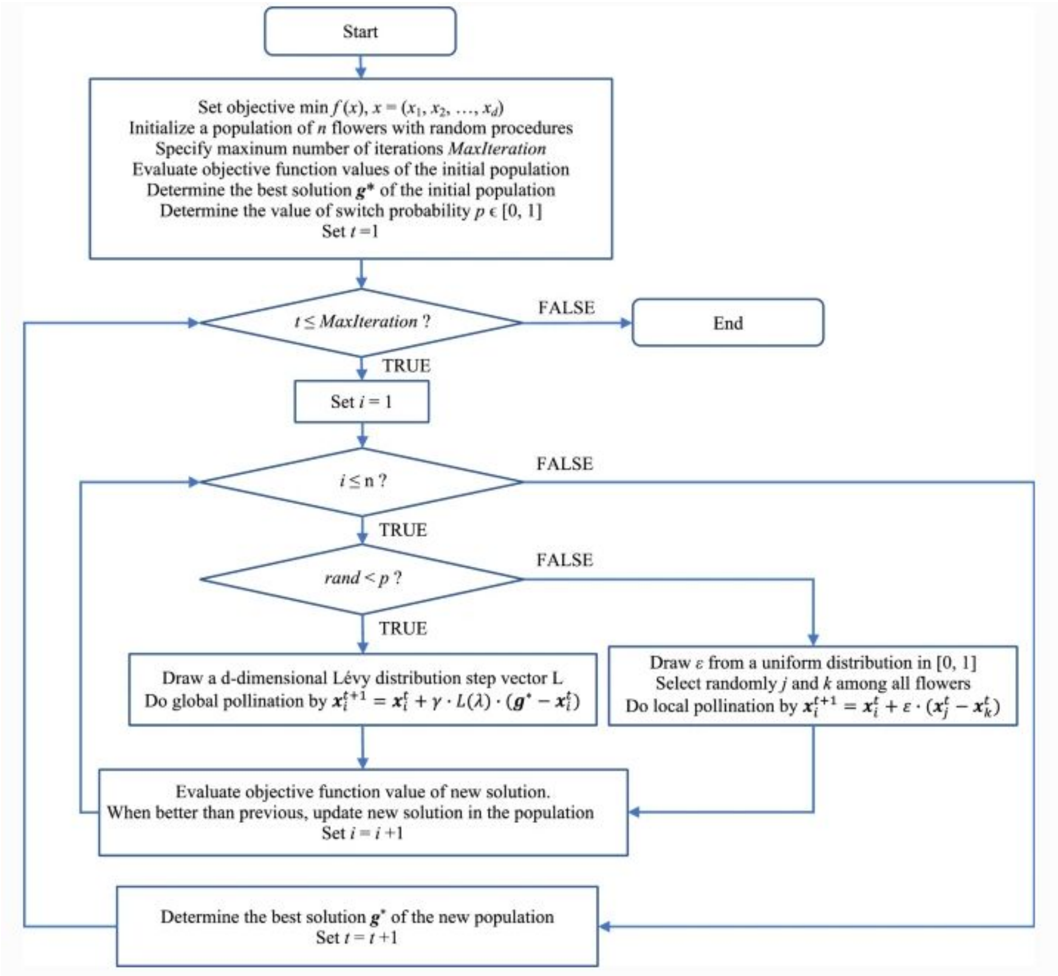
Flower Pollination Algorithm

Piper Ellsworth, Jaehyun Lee, Shawn Hagler

Methodology

Data Flow Graph



Python Code

```
In [3]: import random
        import math
        import os

        def target_function():
            return

        def initial_position(flowers = 3, min_values = [-5, -5], max_values = [5, 5], target_function = target_function):
            position = [[0] * (len(min_values) + 1) for _ in range(flowers)]

            for i in range(0, flowers):
                for j in range(0, len(min_values)):
                    position[i][j] = random.uniform(min_values[j], max_values[j])
                position[i][-1] = target_function(position[i][0:len(min_values)])
            print(position)
            return position

        def levy_flight(beta):
            beta    = beta
            r1      = int.from_bytes(os.urandom(8), byteorder = "big") / ((1 << 64) - 1)
            r2      = int.from_bytes(os.urandom(8), byteorder = "big") / ((1 << 64) - 1)
            sig_num = math.gamma(1 + beta) * math.sin((math.pi * beta) / 2.0)
            sig_den = math.gamma((1 + beta) / 2) * beta * 2**((beta - 1) / 2)
            sigma   = (sig_num / sig_den)**(1 / beta)
            levy    = (0.01 * r1 * sigma) / (abs(r2)**(1 / beta))
            return levy

        def clip(num, min_value, max_value):
            return max(min(num, max_value), min_value)
```

Python Code

```
def pollination_global(position, best_global, flower = 0, gama = 0.5, lamb = 1.4, min_values = [-5, -5],
                                                                  max_values = [5, 5], target_function = target_function):
    x = list(best_global)
    for j in range(0, len(min_values)):
        x[j] = clip((position[flower][j]  + gama * levy_flight(lamb) * (position[flower][j] - best_global[j])),
                                                                                 min_values[j], max_values[j])
    x[-1]  = target_function(x[0:len(min_values)])
    return x

def pollination_local(position, best_global, flower = 0, nb_flower_1 = 0, nb_flower_2 = 1, min_values = [-5,-5],
                                                                         max_values = [5,5], target_function = target_function):
    x = list(best_global)
    for j in range(0, len(min_values)):
        r = int.from_bytes(os.urandom(8), byteorder = "big") / ((1 << 64) - 1)
        x[j] = clip((position[flower][j]  + r * (position[nb_flower_1][j] - position[nb_flower_2][j])),
                    min_values[j], max_values[j])
    x[-1] = target_function(x[0:len(min_values)])
    return x
```

Python Code

```
def flower_pollination_algorithm(flowers = 3, min_values = [-5, -5], max_values = [5, 5], iterations = 50,
                                 gama = 0.5, lamb = 1.4, p = 0.8, target_function = target_function):
    count = 0
    score_list=[]
    position = initial_position(flowers = flowers, min_values = min_values, max_values = max_values,
                                target_function = target_function)
    best_global = sorted(position, key=lambda x: x[-1])[0]
    x = list(best_global)
    while (count <= iterations):
        print("Iteration = ", count, " f(x) = ", best_global[-1])
        score_list.append(best_global[-1])
        for i in range(0, len(position)):
            nb_flower_1 = int(random.random() * len(position))
            nb_flower_2 = int(random.random() * len(position))
            while nb_flower_1 == nb_flower_2:
                nb_flower_1 = int(random.random() * len(position))
            r = int.from_bytes(os.urandom(8), byteorder = "big") / ((1 << 64) - 1)
            if (r < p):
                x = pollination_global(position, best_global, flower = i, gama = gama, lamb = lamb,
                                       min_values = min_values, max_values = max_values, target_function = target_function)
            else:
                x = pollination_local(position, best_global, flower = i, nb_flower_1 = nb_flower_1,
                                      nb_flower_2 = nb_flower_2, min_values = min_values, max_values = max_values,
                                      target_function = target_function)
            if (x[-1] <= position[i][-1]):
                for j in range(0, len(x)):
                    position[i][j] = x[j]
            value = sorted(position, key=lambda x: x[-1])[0]
            if (best_global[-1] > value[-1]):
                best_global = list(value)
        count = count + 1
    print(best_global)
    return best_global, score_list
```

Python Code

```
def six_hump_camel_back(variables_values = [0, 0]):
    return 4 * variables_values[0]**2 - 2.1 * variables_values[0]**4 + (1/3) * variables_values[0]**6
               + variables_values[0] * variables_values[1] - 4 * variables_values[1]**2 + 4 * variables_values[1]**4

flower_best_solution, score_list = flower_pollination_algorithm(flowers = 175, min_values = [-5,-5],
                                                                max_values = [5,5], iterations = 300,
                                                                gama = 0.1, lamb = 1.5, p = 0.8,
                                                                target_function = six_hump_camel_back)
```

Coding Alternatives

```
package flower_pollination_package is
    type real_vector is array(natural range <>) of real;
    type real_vector_vector is array(natural range <>) of real_vector(1 to 3);
end flower_pollination_package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.flower_pollination_package.all;

entity flower_pollination is
    port (
        clk: in std_logic;
        reset: in std_logic;
        flower_count: in integer;
        min: in integer;
        max: in integer;
        gamma: in real;
        lamb: in real;
        p: in real;
        iterations: in integer;
        best_solution: out real_vector(1 to 3)
    );
end flower_pollination;

architecture beh of flower_pollination is
    function random_real return real_vector is
            variable seed1: integer := 123456789;
            variable seed2: integer := 987654321;
            variable seed3: integer := 239438458;
            variable rand_real1: real;
            variable rand_real2: real;
            variable random: real_vector(1 to 2);
    begin
        uniform(seed1, seed2, rand_real1);
        uniform(seed2, seed3, rand_real2);
        random(1) := rand_real1;
        random(2) := rand_real2;
        return random;
    end random_real;
```

Coding Alternatives

```
function random_integer(min_value: integer; max_value: integer) return integer is
    variable real_random: real;
    variable seed1, seed2: positive := 987654321;
begin
    uniform(seed1, seed2, real_random);
    return integer(real(min_value) + real_random * real(max_value - min_value + 1));
end random_integer;

function six_hump_camel_back(variables: real_vector) return real is
        variable x: real := variables(1);
        variable y: real := variables(2);
begin
    return 4.0 * (x**2) - 2.1 * (x**4) + ((1.0/3.0) * (x**6)) + x * y - 4.0 * (y**2) + 4.0 * (y**4);
end six_hump_camel_back;

function fitness_compare(a: real_vector; b: real_vector) return boolean is
begin
    return (six_hump_camel_back(a) <= six_hump_camel_back(b));
end fitness_compare;

function initial_positions(flower_count: integer; min: integer; max: integer) return real_vector_vector is
    variable positions: real_vector_vector(1 to flower_count);
begin
    for i in 1 to flower_count - 1 loop
        for j in 1 to 2 loop
            positions(i)(j) := real(min) + random_real(j) * real(max - min + 1);
        end loop;
        positions(i)(3) := six_hump_camel_back(positions(i));
    end loop;
    return positions;
end initial_positions;
```

Coding Alternatives

```
function pollination_global(positions: real_vector_vector; best_position: real_vector; min: integer; max: integer; flower: integer; gamma: real; lamb: real) return real_vector is
    variable x: real_vector(1 to 3);
    variable delta: real;
begin
    for i in 1 to 2 loop
        x(i) := positions(flower)(i) + gamma * levy_flight(lamb) * (best_position(i) - positions(flower)(i));
        if x(i) < real(min) then
            x(i) := real(min);
        elsif x(i) > real(max) then
            x(i) := real(max);
        end if;
    end loop;
    x(3) := six_hump_camel_back(x);
    return x;
end pollination_global;

function pollination_local(flower_count: integer; flower: integer; positions: real_vector_vector; min: integer; max: integer) return real_vector is
    variable x: real_vector(1 to 3);
    variable delta: real;
    variable r: real;
    variable nb_flower_1: integer := random_integer(1, flower_count);
    variable nb_flower_2: integer := random_integer(1, flower_count);
 begin
    r := random_real(1);
    for i in 1 to 2 loop
        delta := r * (positions(nb_flower_1)(i) - positions(nb_flower_2)(i));
        if (positions(flower)(i) + delta) > real(max) then
            x(i) := real(max);
        elsif (positions(flower)(i) + delta) < real(min) then
            x(i) := real(min);
        else
            x(i) := positions(flower)(i) + delta;
        end if;
    end loop;
    x(3) := six_hump_camel_back(x);
    return x;
end pollination_local;
```

Coding Alternatives

```

    signal count: integer := 0;
begin
    process (clk, reset)
        variable positions: real_vector_vector(1 to 175);
        variable best_position: real_vector(1 to 3);
        variable x: real_vector(1 to 3);
    begin
        if (reset = '1') then
            count <= 0;
            positions := initial_positions(flower_count, min, max);
            best_position := positions(1);
        elsif (rising_edge(clk)) then
            if (count < iterations) then
                for i in 1 to flower_count loop
                    if (random_real(1) < p) then
                        x := pollination_global(positions, best_position, min, max, i, gamma, lamb);
                    else
                        x := pollination_local(flower_count, i, positions, min, max);
                    end if;
                    if (fitness_compare(positions(i), best_position)) then
                        best_position := positions(i);
                    end if;
                end loop;
                count <= count + 1;
            end if;
        else
            best_solution <= best_position;
        end if;
    end process;
end beh;
```

Limitations

1. Complex algorithm design: The FPA involves complex mathematical operations. Translating these intricate operations into VHDL code may require significant effort and a thorough understanding of both the algorithm and the hardware design.

2. Limited resources: FPGA devices have limited resources, including logic elements, memory, and DSP blocks.

3. Performance constraints: FPGA based implementations of the FPA may face performance limitations due to the inherent parallelism and pipeline capabilities of FPGAs.

4. Debugging and testing complexities: Debugging and testing FPGA designs is inherently more challenging than software-based implementations.

Division of VHDL code

Lead Software Developer: Shawn

Assistant Software Developer: Jason

Tester/Debugger: Piper

Results

VHDL Code- Main

```
begin
    process (clk, rst)
        variable xx : real_vector(1 to 2);
        variable temp, minimum : real := 0.0;
        variable best_vector : real_vector(1 to 2) := (others => 0.0);
        variable count : integer;
    begin
        if (rst = '1') then
            best_vector := (others => 0.0);
            minimum := 0.0;
        elsif (rising_edge(clk)) then
            if (xxx < p) then
                xx := pollination_global(x_vector, levy_vector, best_vector, min, max, gamma, lamb);
            else
                xx := pollination_local(local_vector, x_vector, xxx, min, max);
            end if;
           temp := ((4.0 - 2.1*(xx(1)**2) + (xx(1)**4)/3.0))*(xx(1)**2)
                                  + (xx(1)*xx(2)) + (-4.0 + 4.0*(xx(2)**2))*(xx(2)**2);
                    if (temp <= minimum) then
                        minimum := temp;
                        best_vector(1) := xx(1);
                        best_vector(2) := xx(2);
                    end if;

        best_x(1) <= best_vector(1);
        best_x(2) <= best_vector(2);
        fx <= minimum;
        end if;
    end process;
end Behavioral;
```

VHDL Code- Global

```
architecture Behavioral of final2 is

    function levy_flight(beta: real; levy_vector: real_vector) return real is
        constant sig_num: real := 0.9399856029866254;
        constant sig_den: real := 1.6168504121556964;
        variable sigma: real;
        variable levy: real;
        variable r1, r2: real;
    begin
        r1 := levy_vector(1);
        r2 := levy_vector(2);
        sigma := (sig_num / sig_den) ** (1.0 / beta);
        levy := (0.01 * r1 * sigma) / (real(abs(r2)) ** (1.0 / beta));
        return levy;
    end levy_flight;

    function pollination_global(position: real_vector; levy_vector: real_vector; best_position: real_vector;
                                min: integer; max: integer; gamma: real; lamb: real) return real_vector is
        variable x: real_vector(1 to 2);
        variable delta: real;
    begin
        for i in 1 to 2 loop
            x(i) := position(i) + gamma * levy_flight(lamb, levy_vector) * (best_position(i) - position(i));
            if x(i) < real(min) then
                x(i) := real(min);
            elsif x(i) > real(max) then
                x(i) := real(max);
            end if;
        end loop;
        return x;
    end pollination_global;
```

$$L \sim \frac{(\beta + 1) \times sin\left(\frac{\beta\pi}{2}\right)}{\pi} \times \frac{1}{s^{\beta+1}}, \quad (s \gg s_o > 0)$$

$$x_i^{t+1} = x_i^t + L(x_i^t - g\ *)$$

VHDL Code- Local

```
function pollination_local(local_vector: real_vector; position: real_vector; xxx: real; min: integer;
                                                                max: integer) return real_vector is
    variable x: real_vector(1 to 2);
    variable delta: real;
    variable r: real;
 begin
    r := xxx;
    for i in 1 to 2 loop
        delta := r * (position(i) - local_vector(i));
        if (position(i) + delta) > real(max) then
            x(i) := real(max);
        elsif (position(i) + delta) < real(min) then
            x(i) := real(min);
        else
            x(i) := position(i) + delta;
        end if;
    end loop;
    return x;
end pollination_local;
```

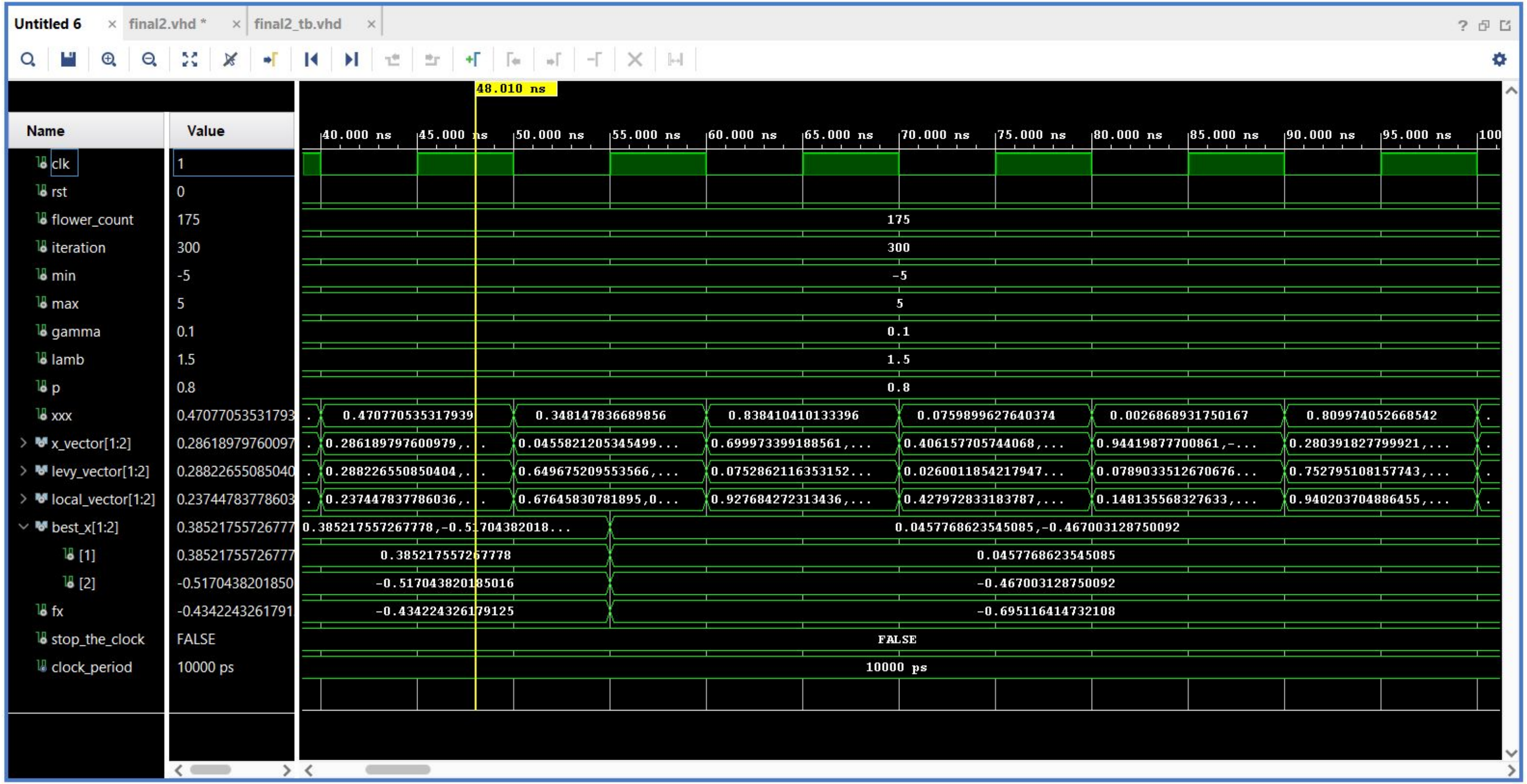
$$x_i^{t+1} = x_i^t + \epsilon \left(x_j^t - x_k^t \right)$$

Testbench

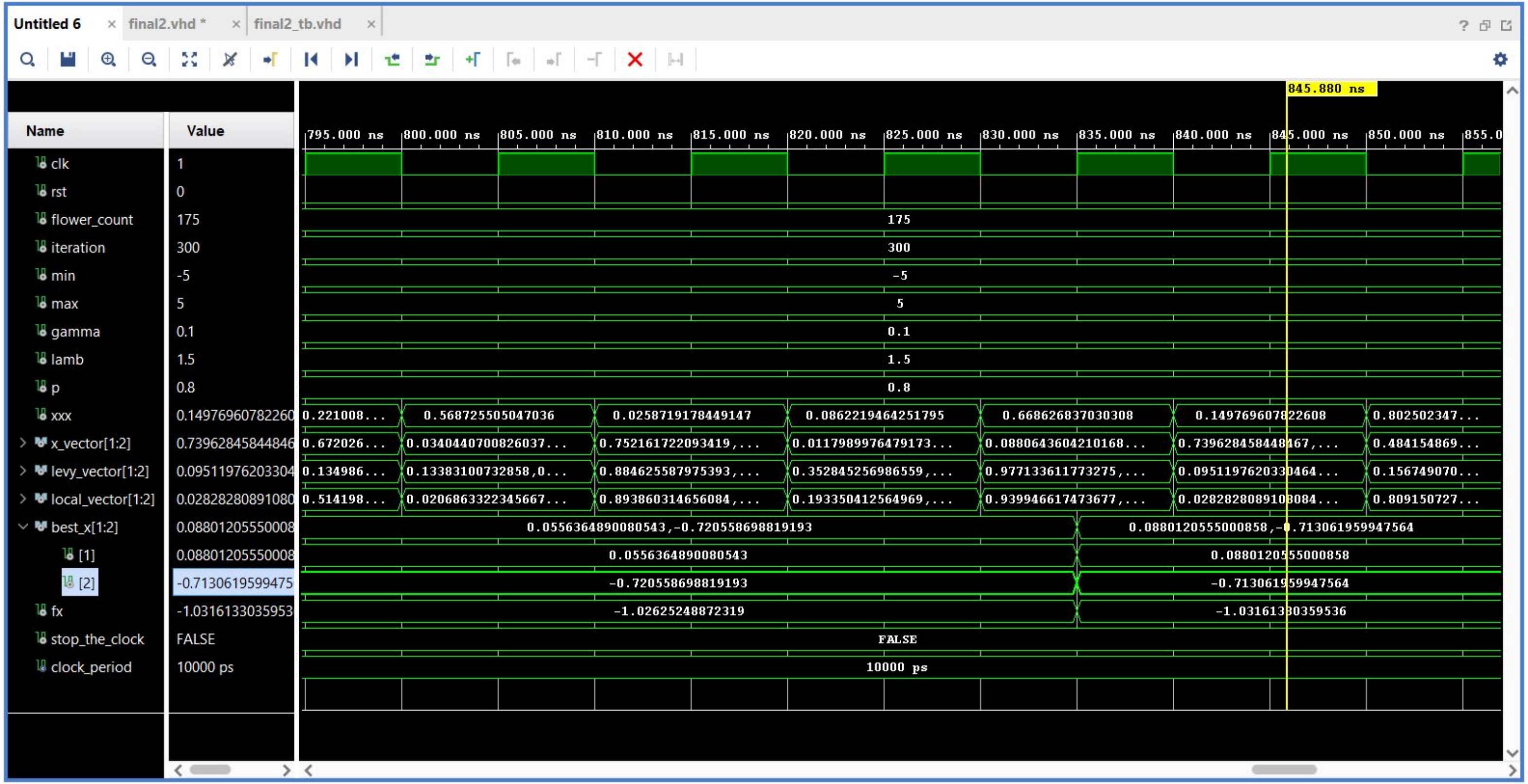
```
clocking: process
begin
  while not stop_the_clock loop
    clk <= '0', '1' after clock_period / 2;
    wait for clock_period;
  end loop;
  wait;
end process;
```

```
stimulus: process
variable seed1, seed2, seed3, seed4, seed5 : positive;
variable xx, y, z, levy1, levy2, local1, local2 : real;
begin
  rst <= '1';
      wait for 10 ns;
  rst <= '0';
  flower_count  <= 175;
  iteration <= 300;
  min           <= -5;
  max           <= 5;
  gamma         <= 0.1;
  lamb          <= 1.5;
  p             <= 0.8;

  seed1 := 1;
  seed2 := 2;
  seed3 := 3;
  seed4 := 4;
  seed5 := 5;
  for n in 1 to 300 loop
      uniform(seed1, seed2, xx);
      uniform(seed1, seed3, y);
      uniform(seed2, seed3, z);
      uniform(seed1, seed4, levy1);
      uniform(seed1, seed5, levy2);
      uniform(seed2, seed4, local1);
      uniform(seed3, seed5, local2);
      xxx <= xx;
      x_vector(1) <= y-0.002;
      x_vector(2) <= z-0.984;
      levy_vector(1) <= levy1;
      levy_vector(2) <= levy2;
      local_vector(1) <= local1;
      local_vector(2) <= local2;
          wait for 10 ns;
  end loop;

  wait;
end process;
```

Simulation Waveform



Simulation Waveform



Calculations Python

```
def six_hump_camel_back(variables_values = [0, 0]):
    return 4 * variables_values[0]**2 - 2.1 * variables_values[0]**4 + (1/3) * variables_values[0]**6
               + variables_values[0] * variables_values[1] - 4 * variables_values[1]**2 + 4 * variables_values[1]**4

flower_best_solution, score_list = flower_pollination_algorithm(flowers = 175, min_values = [-5,-5],
                                                                max_values = [5,5], iterations = 300,
                                                                gama = 0.1, lamb = 1.5, p = 0.8,
                                                                target_function = six_hump_camel_back)
```

```
Iteration =  283  f(x) =  -1.0316275783783373
Iteration =  284  f(x) =  -1.0316275783783373
Iteration =  285  f(x) =  -1.0316275783783373
Iteration =  286  f(x) =  -1.0316275783783373
Iteration =  287  f(x) =  -1.0316275783783373
Iteration =  288  f(x) =  -1.0316275783783373
Iteration =  289  f(x) =  -1.0316275783783373
Iteration =  290  f(x) =  -1.0316275783783373
Iteration =  291  f(x) =  -1.0316275783783373
Iteration =  292  f(x) =  -1.0316275783783373
Iteration =  293  f(x) =  -1.0316275783783373
Iteration =  294  f(x) =  -1.0316275783783373
Iteration =  295  f(x) =  -1.0316275783783373
Iteration =  296  f(x) =  -1.0316275783783373
Iteration =  297  f(x) =  -1.0316275783783373
Iteration =  298  f(x) =  -1.0316275783783373
Iteration =  299  f(x) =  -1.0316275783783373
Iteration =  300  f(x) =  -1.0316275783783373
[0.08945476939947151, -0.7128225465183462, -1.0316275783783373]
```

Calculations MATLAB

untitled.m

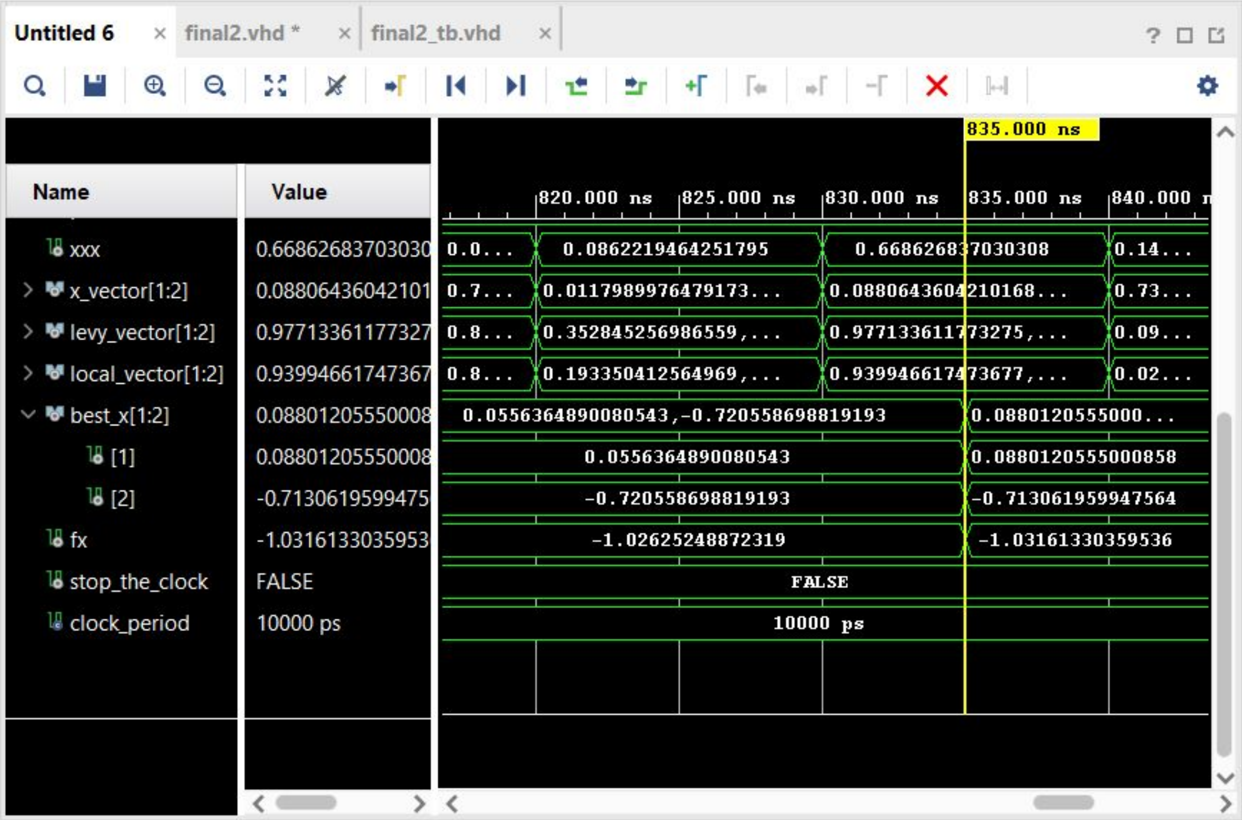
/MATLAB Drive/untitled.m

```matlab
% INPUTS:
%
 xx = [0.088012, -0.71306];
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x1 = xx(1);
x2 = xx(2);

term1 = (4-2.1*x1^2+(x1^4)/3) * x1^2;
term2 = x1*x2;
term3 = (-4+4*x2^2) * x2^2;

y = term1 + term2 + term3;

fprintf ("%d", y);
```

Command Window

```
>> untitled
-1.031613e+00
>>
```

Timing Information of Software Solution

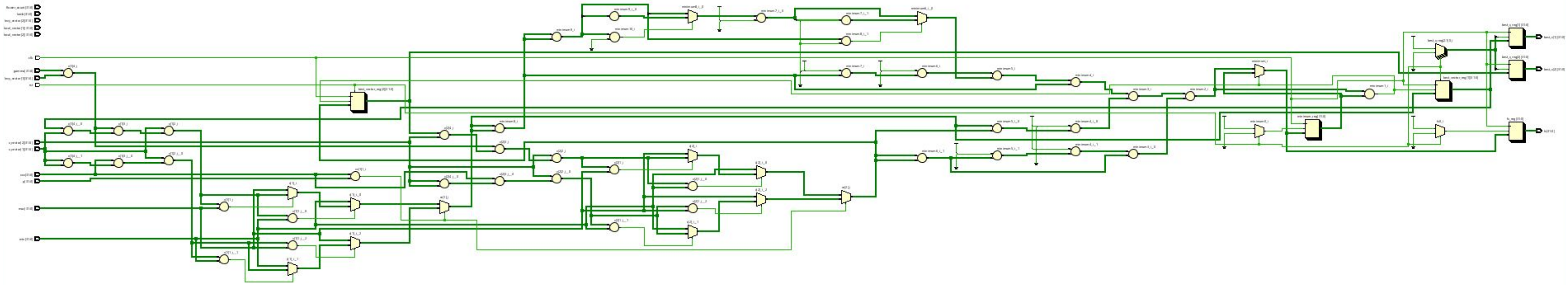


Timing Information of Software Solution Python

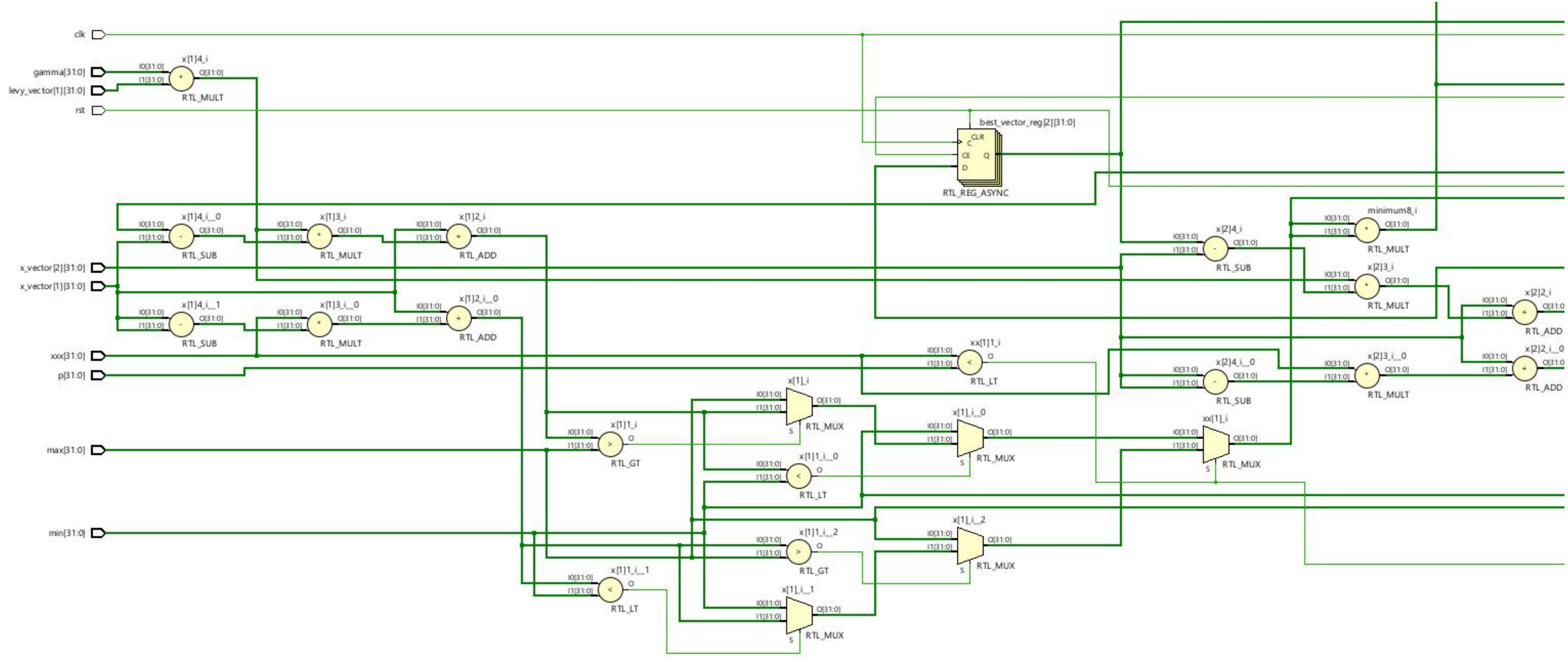
```
start = time.time()
...
end = time.time()
print(end - start)
```

```
Iteration =  181  f(x) =  -1.0315827026212248
Iteration =  182  f(x) =  -1.0315827026212248
Iteration =  183  f(x) =  -1.0315827026212248
Iteration =  184  f(x) =  -1.0315827026212248
time elapsed:  1.0451202392578125
Iteration =  185  f(x) =  -1.0315853648975715
time elapsed:  1.05312180519104

Iteration =  186  f(x) =  -1.0316095330641744
Iteration =  187  f(x) =  -1.0316095330641744
Iteration =  188  f(x) =  -1.0316095330641744
Iteration =  189  f(x) =  -1.0316095330641744
Iteration =  190  f(x) =  -1.0316095330641744
Iteration =  191  f(x) =  -1.0316095330641744
time elapsed:  1.0831239223480225
Iteration =  192  f(x) =  -1.031617853236341
Iteration =  193  f(x) =  -1.031617853236341
Iteration =  194  f(x) =  -1.031617853236341
Iteration =  195  f(x) =  -1.031617853236341
```

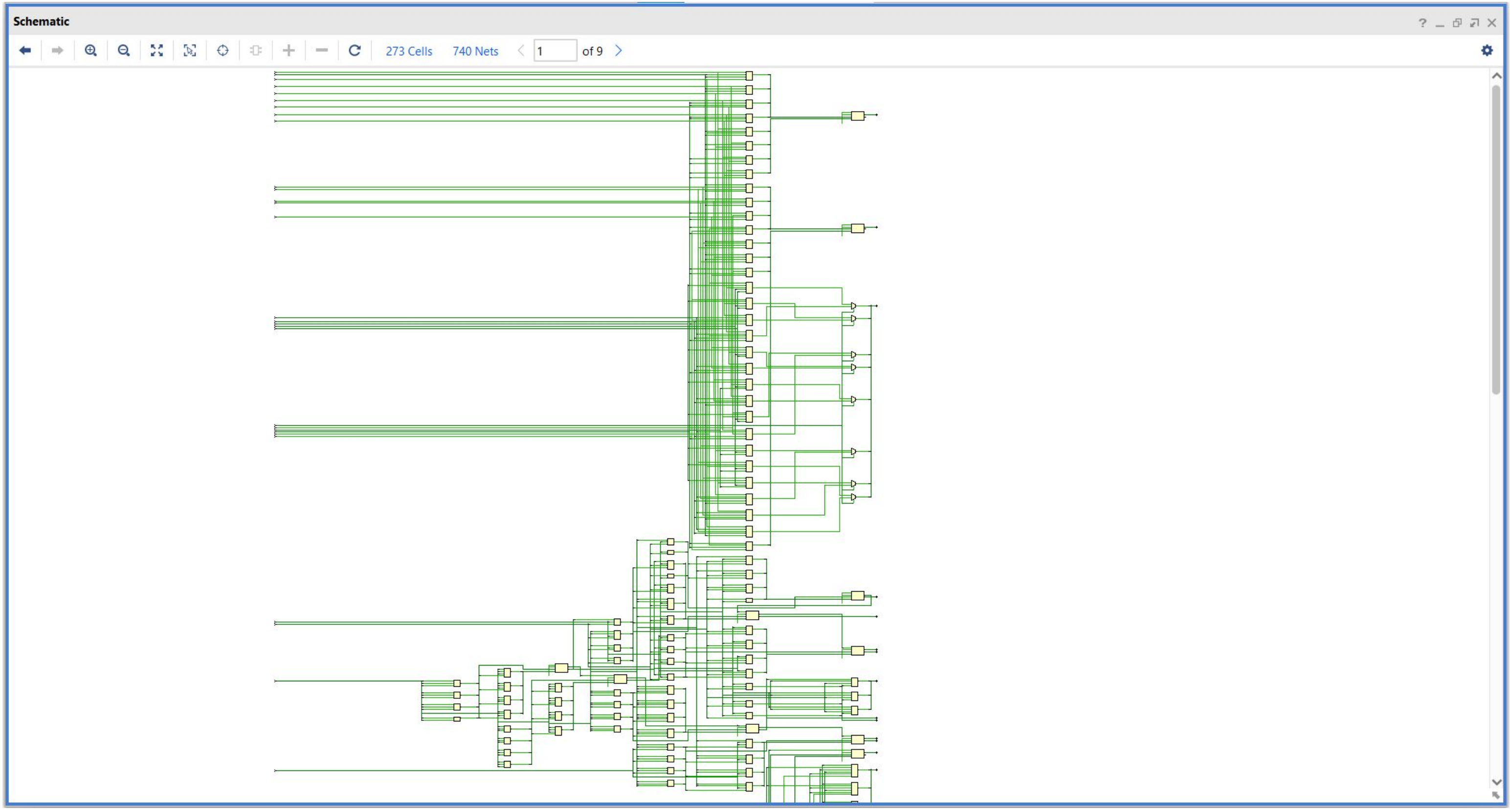
RTL Analysis



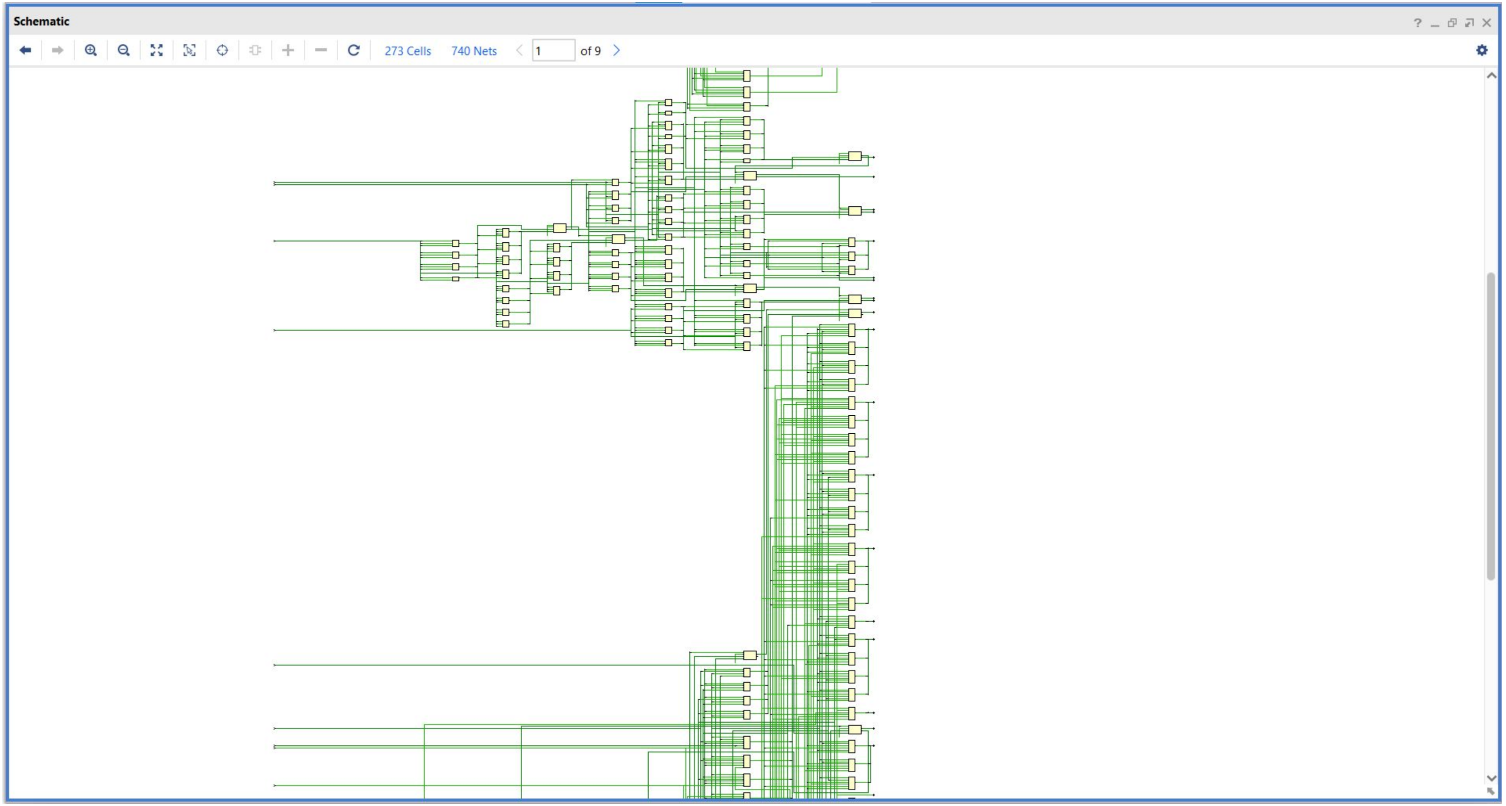
RTL Analysis



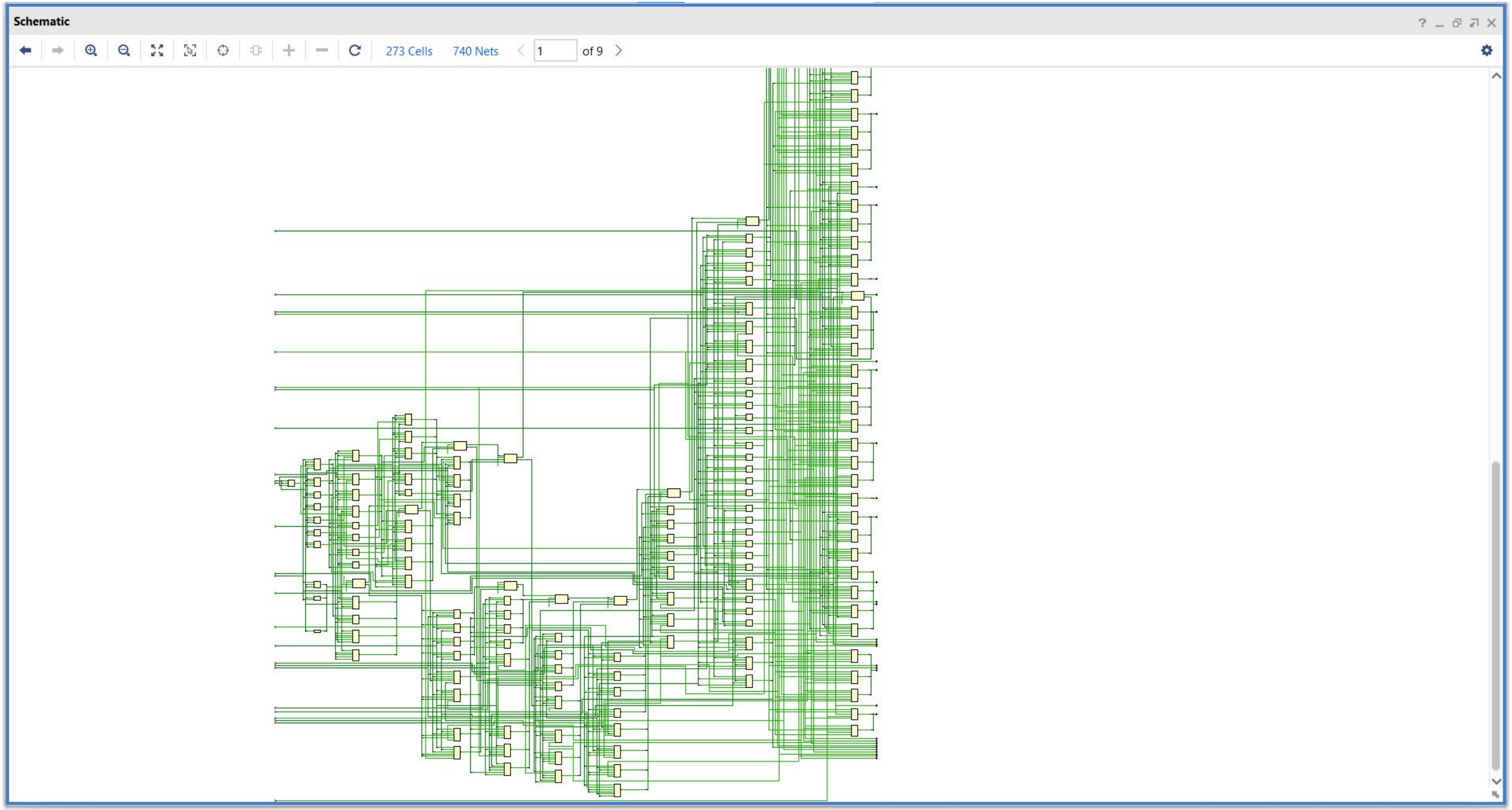
Synthesized Circuit



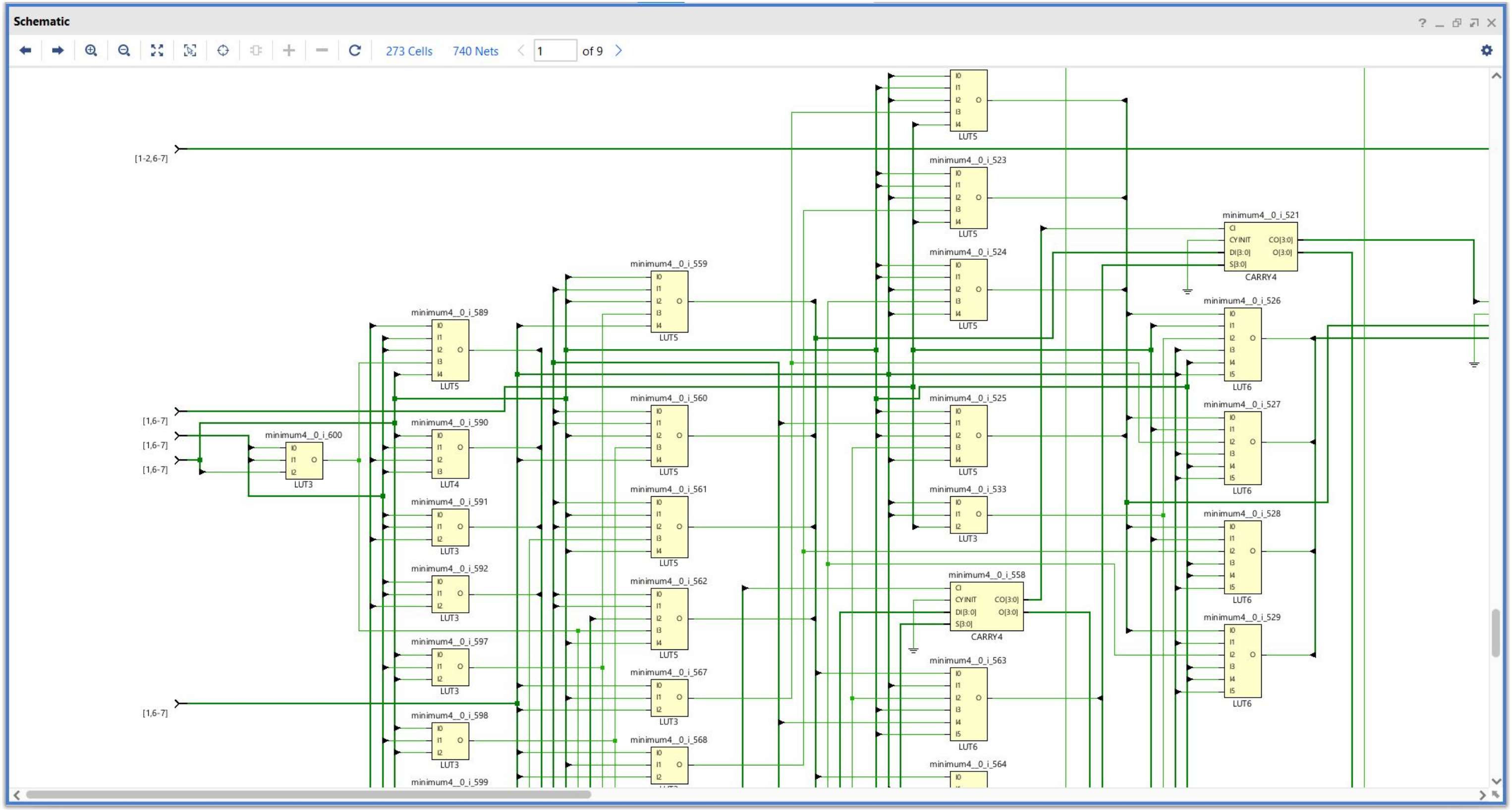
Synthesized Circuit



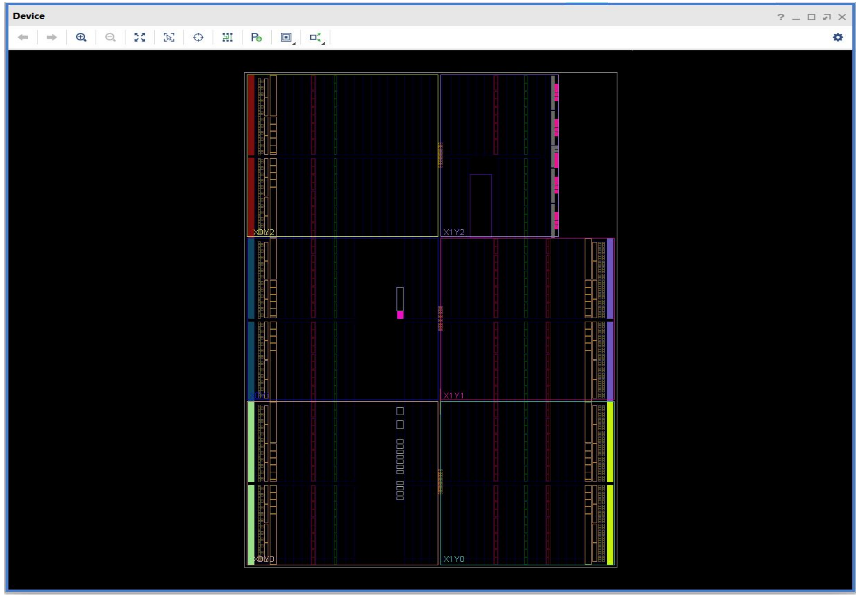
Synthesized Circuit



Synthesized Circuit



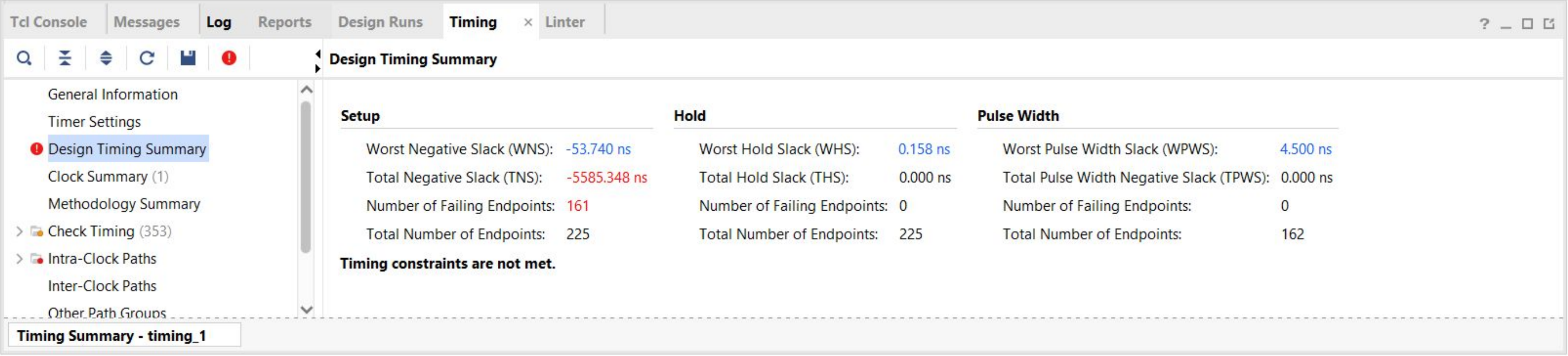
Device Implementation



Place Design (103 errors)

[Place 30-415] IO Placement failed due to overutilization. This design contains 354 I/O ports
while the target device: 7a35t package: cpg236, contains only 106 available user I/O. The target device has 106 usable I/O pins of which 0 are already occupied by user-locked I/Os.
To rectify this issue:
1. Ensure you are targeting the correct device and package. Select a larger device or different package if necessary.
2. Check the top-level ports of the design to ensure the correct number of ports are specified.
3. Consider design changes to reduce the number of I/Os necessary.

Timing Analysis



Timing Analysis

Timing

- General Information
- Settings
- Timing Checks (20)
  - Setup (10)
  - Hold (10)

Timing Checks - Setup

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Logic %	Net %	Requirement	Source Clock	Destination Clock	Exception
Constrained Paths (1)															
clk (10)															
Path 1	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][0]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	
Path 2	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][10]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	
Path 3	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][11]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	
Path 4	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][12]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	
Path 5	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][13]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	
Path 6	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][14]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	
Path 7	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][15]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	
Path 8	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][16]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	
Path 9	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][17]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	
Path 10	-53.740	68	37	96	best_vector_reg[1][1]/C	best_vector_reg[1][18]/CE	63.358	40.651	22.707	64.2	35.8	10.000	clk	clk	

Power Report