



# Verification of Special Hardware: FPGAs

Justin Hsu, Arsalan Ahmed, Daniel Narevich



**UC DAVIS**  
COLLEGE OF ENGINEERING



# Background

# Importance of Verifying FPGA Designs

- FPGAs are widely used in digital system design due to their reconfigurable nature.
- FPGA users depend on their unit being fully reliable, as it may need to emulate hundreds of different logic circuits

# FPGA Structure

- FPGAs are made up of many Configurable Logic Blocks (CLBs)
- These blocks can emulate any logic equation within a limited number of inputs and outputs
- These blocks are connected, allowing the FPGA to simulate larger devices
- The FPGA also has accessible flash memory and DFFs, allowing the user to simulate sequential logic

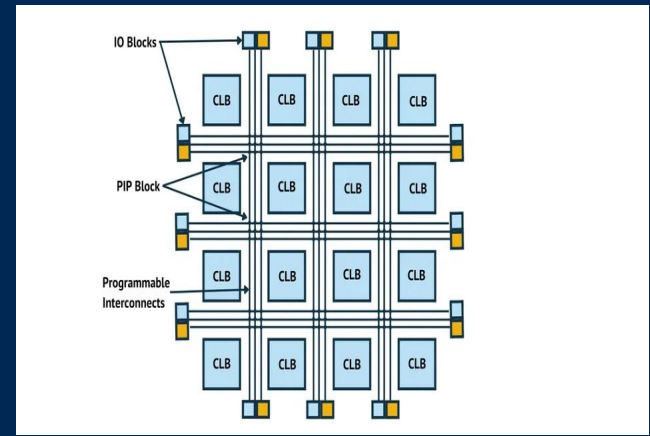
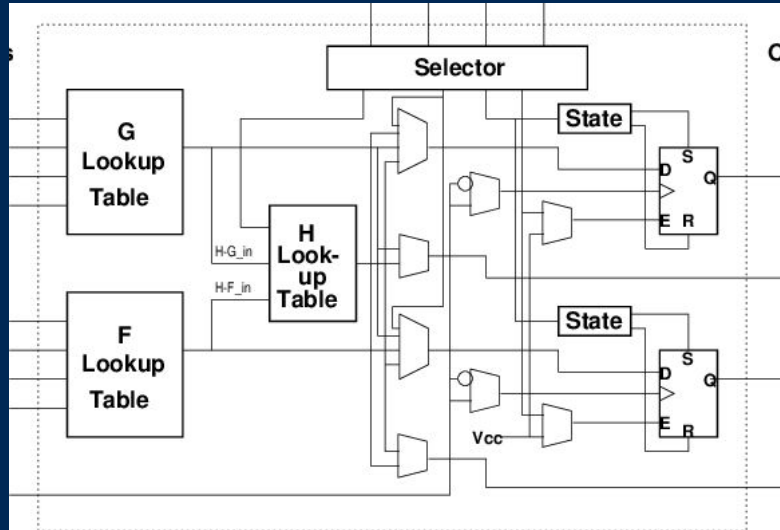


Image credit: Ravi Rao

# CLB Structure

- In our project we focus on the verification of the Look Up Table
- The CLB is framed from SRAM inputs, a 16:1 MUX Look Up Table, DFFs, and a 2:1 MUX



The architecture of XC4000 FPGA CLB  
Image credit: Eugene Shragowitz

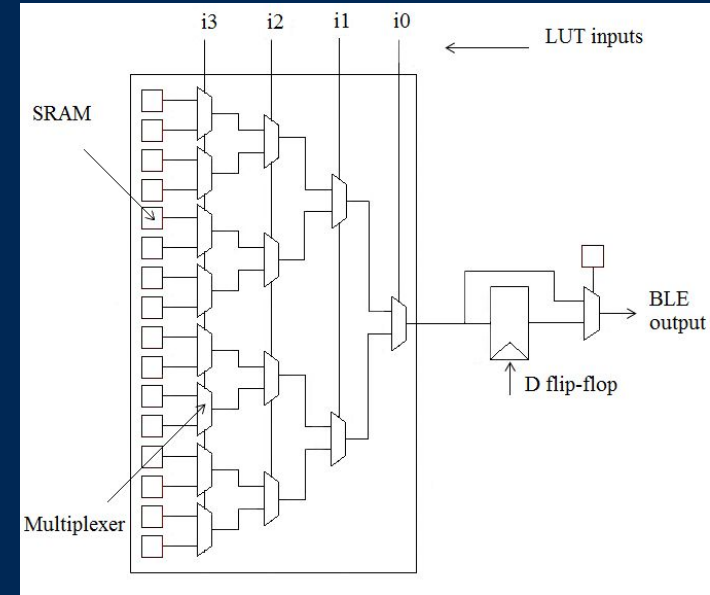


Image credit: Bijan Alizadeh



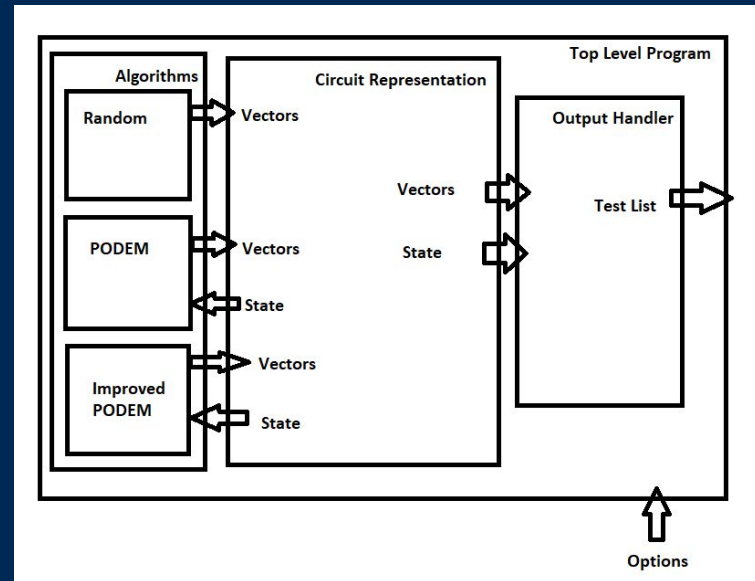
# Our Project



**UC DAVIS**  
COLLEGE OF ENGINEERING

# Project Overview

- We created a Python program that can simulate an FPGA component using 5-valued Logic
- We aim to see how 3 algorithms perform in generating a test list that can find all non-redundant faults in our circuit
  - Random
  - PODEM
  - Improved PODEM



# Structure

- Created a Gate class that can identify location and states
- Subdesign and top level design classes for our circuit

```
def my_subdesign(inputs, faulty_gate, fault_location, fault_value):  
    G1_inputs = [inputs[0], inputs[1]]  
    G1 = gate("G1", "NAND", G1_inputs, 0, 1)  
    if faulty_gate == "G1":  
        G1.fault_location = fault_location  
        G1.fault_value = fault_value  
    G2_inputs = [inputs[1], inputs[1]]  
    G2 = gate("G2", "NAND", G2_inputs, 0, 1)  
    if faulty_gate == "G2":  
        G2.fault_location = fault_location  
        G2.fault_value = fault_value  
    result(G1)  
    result(G2)  
    G3_inputs = [G2.state, inputs[2]]  
    G3 = gate("G3", "NAND", G3_inputs, 0, 1)  
    if faulty_gate == "G3":  
        G3.fault_location = fault_location  
        G3.fault_value = fault_value  
    result(G3)  
    G4_inputs = [G1.state, G3.state]  
    G4 = gate("G4", "NAND", G4_inputs, 0, 1)  
    if faulty_gate == "G4":  
        G4.fault_location = fault_location  
        G4.fault_value = fault_value  
    result(G4)  
    out = G4.state  
    return out
```





# Random

- This Algorithm is able to run very quickly, as it does not need to use the state of the gates within the circuit
- Used python random() function
- Uses a “give up” value to set run time

```
import random
from Logic_Blocks import d_values_convert

def random_vector(num_inputs):
    result = []
    for j in range(num_inputs):
        k = random.randint(0,1)
        l = d_values_convert(k)
        result.append(l)
    return result
```

# PODEM

- Each call to the circuit takes a very long time
  - This occurs frequently in PODEM, especially when backtracking
- However, this algorithm is consistent in detecting vectors
- Additionally provides vectors that contain “Don’t Cares”
- Efficiency increased by first changing primary inputs with more controllability
  - Selection Lines

```
def podem(faulty_MUX, faulty_gate, fault_location, fault_value):
    j = 0
    max_level = 20
    test_vector = [D_values.X, D_values.X, D_values.X, D_values.X, D_values.X, D_values.X, D_values.X, D_values.X,
                  track_output = D_values.X

    while (track_output != D_values.D_bar) | (track_output != D_values.D):
        temp_output = my_design(test_vector, faulty_MUX, faulty_gate, fault_location, fault_value)
        track_output = temp_output[1]
        if track_output == D_values.X:
            if j == max_level:
                if test_vector[j-1] == D_values.zero:
                    test_vector[j-1] = D_values.one
                    j = j
                else:
                    while test_vector[j-1] == D_values.one:
                        test_vector[j-1] = D_values.X
                        j = j - 1
                    test_vector[j-1] = D_values.one
            else:
                j = j + 1
                test_vector[j-1] = D_values.zero

        elif track_output in [D_values.one, D_values.zero]:
            if test_vector[j-1] == D_values.zero:
                j = j
```

# Improved PODEM

- To reduce the number of calls to our circuit, the selection lines are pre-selected based on fault location in circuit
  - Reduces PODEM search time by  $2^{\text{Inputs Pre Selected}}$  per test vector

```
j = 0
if faulty_MUX in [1, 2, 3, 4, 9, 10, 13]:
    test_vector[0] = 0
    j = 1
elif faulty_MUX in [5, 6, 7, 8, 11, 12, 14]:
    test_vector[0] = 1
    j = 1

if faulty_MUX in [1, 2, 5, 6, 9, 11]:
    test_vector[1] = 0
    j = 2
elif faulty_MUX in [3, 4, 7, 8, 10, 12]:
    test_vector[1] = 1
    j = 2

if faulty_MUX in [1, 3, 5, 7]:
    test_vector[2] = 0
    j = 3
elif faulty_MUX in [2, 4, 6, 8]:
    test_vector[2] = 1
    j = 3

max_level = 20

track_output = D_values.X
```

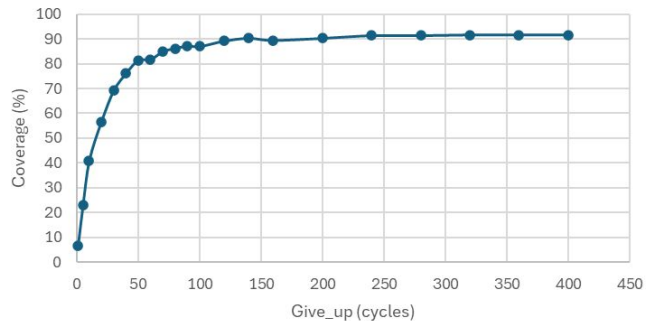


# Conclusions

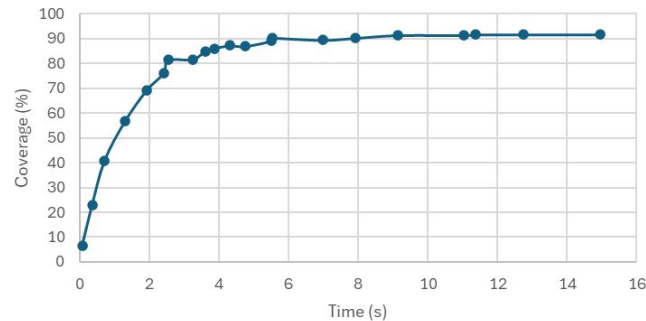
# Results

- Random
  - Performance degrades as coverage increases
  - 91.6% coverage includes redundant faults
- PODEM
  - Achieves full coverage
  - Time to run is much worse than random (583 s)

Give\_up value vs Coverage (Random Algorithm)

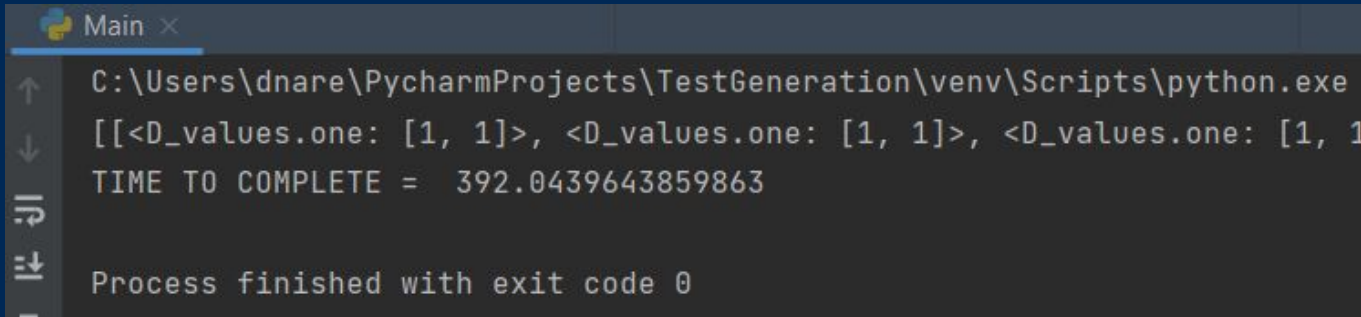


Time vs Coverage (Random Algorithm)



# Results

- PODEM
  - Achieves full coverage
  - Time to run is much worse than random (583 s)
  - 13s per vector
- Improved PODEM
  - Operates in 46% of the time of PODEM (392 s)
  - 9s per vector
  - Still slower than Random
  - Still detects all test vectors



```
Main x
C:\Users\dnare\PycharmProjects\TestGeneration\venv\Scripts\python.exe
[(<D_values.one: [1, 1]>, <D_values.one: [1, 1]>, <D_values.one: [1, 1]
TIME TO COMPLETE = 392.0439643859863

Process finished with exit code 0
```

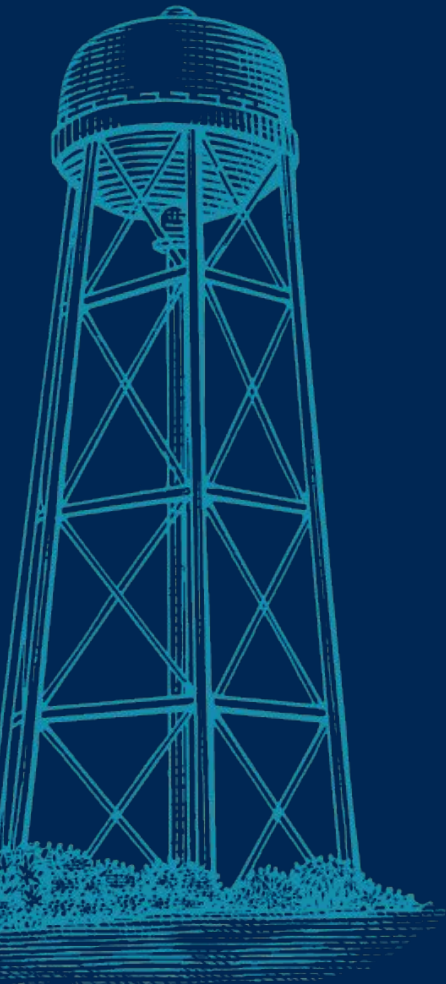
# Conclusions

- When using this tool in a larger design, random would likely struggle
- Using random initially, followed by a targeted use of our Improved PODEM algorithm would likely yield the best results

# Future Work

- Improve efficiency of our circuit representation
  - Faster Language, direct connection to a HDL Representation
- Support verification of sequential logic
- Increase adaptability of our design to different FPGAs
  - Different LUT sizes and arrangements





# Thank You!