# Welcome

To our presentation

# -: MINI PROJECT :-

# IMPLEMENTATION OF ABC ON HARDWARE TROJANS

**BY –**

SUPRITI DAS(510519047)

SHANTONU DEBNATH(510519085)

TARUN CHAUHAN(510519100)

# HARDWARE TROJANS

- Malicious redundant circuitry embedded inside a larger circuit
- Once activated results in
  - data leakage
  - harm to the normal functionality of the circuit
- After activation, a HT can deliver its payload either through
  - standard I/O channels
  - side channels

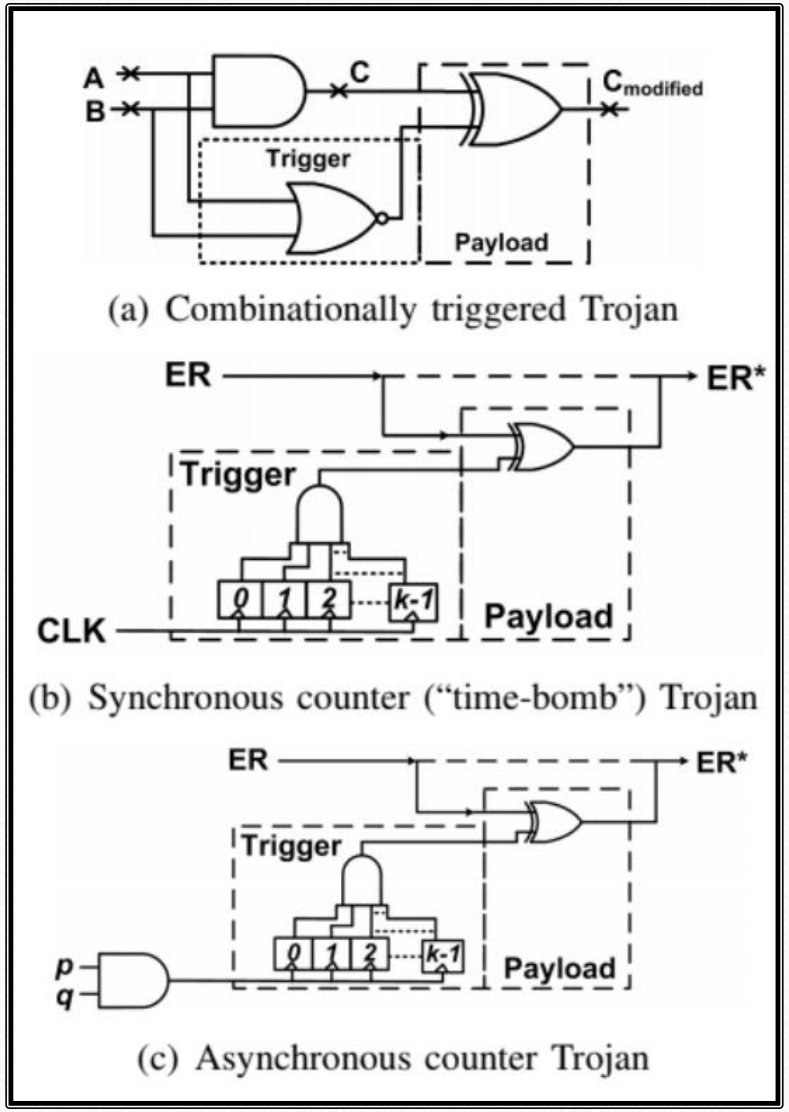
# TYPES

1.**Combinationally triggered Trojans** - Its activation depends on the occurrence of a particular condition at certain internal nodes of the circuit.

2. **Sequentially triggered Trojans** - Its activation depends on the occurrence of a specific sequence of rare logic values at internal nodes. There are two types:

(i) **Synchronous counter Trojans** - It triggers a malfunction on reaching a particular count. Fig.(b) shows a synchronous k-bit counter which activates when the count reaches 2 k−1, by modifying the node ER to an incorrect value at node ER*.

(ii) **Asynchronous counter Trojans** - As shown in fig.(c) the count is increased not by the clock, but by a rising transition at the output of an AND gate with inputs p and q.




(a) Combinationally triggered Trojan

(b) Synchronous counter ("time-bomb") Trojan

(c) Asynchronous counter Trojan

# HT PROPERTIES BASED ON HATCH ALGORITHM

1.**Trigger Signal Dimension d(T )** represents the number of wires used by HT trigger circuitry to activate the payload circuitry in order to exhibit malicious behavior. A large d shows a complicated trigger signal, hence it is harder to detect.

2.**Payload Propagation Delay t(T )** is the number of cycles required to propagate malicious behavior to the output port after the HT is triggered. A large t means it takes a long time after triggering until the malicious behavior is seen, hence less likely to be detected during testing.

**3.Implicit Behavior Factor α(T )** represents the probability that given a HT gets triggered, it will not (explicitly) manifest malicious behavior; this behavior is termed as implicit malicious behavior. Higher probability of implicit malicious behavior means higher stealthiness during testing phase.

4.**Trigger Signal Locality l(T )** shows the spread of trigger signal wires of the HT across the IP core. Small l shows that these wires are in the close vicinity of each other. Large l means that these wires are spread out in the circuit and therefore it is harder to figure out exactly which wires form the trigger signal, hence the HT becomes harder to detect.

Here we are designing FSM based HT and for this reason we require the tool ABC for sequential synthesis.

# BERKELEY ABC

- A public-domain system for logic synthesis and formal verification of binary logic circuits appearing in synchronous hardware designs
  - Fast
  - Scalable
  - High quality results (industrial quality)
  - Exploits synergy between synthesis and verification
- A programming environment
  - Open-source
  - Evolving and improving over time

# COMMANDS

**Input:**

1.**read** – Parses an input file using one of the available file readers. The file extension is used to determine what file parser to invoke. The recognized file extensions are: *aig, baf, bench, blif, eqn, pla, verilog.*

2.**read_bench** – Parses the input file in BENCH (ISCAS) format.

3.**read_blif** – Parses the input file in BLIF. This command can also read hierarchical BLIF.

## Output:

1.**write** – Writes the output file using one of the available file writers. The file extension is used to determine what file writer to invoke. The recognized file extensions are: *aig, baf, bench, blif, cnf, dot, eqn, gml, pla, verilog.*

2.**write_bench** – Outputs the current network into a BENCH file.

3.**write_blif** – Outputs the current network into a BLIF file. If the current network is mapped using a standard cell library, outputs the current network into a BLIF file.The current mapper does not map the registers. As a result, the mapped BLIF files generated for sequential circuits contain unmapped latches. Additionally, command *write_blif* with command-line switch *–l* writes out a part of the current network containing a combinational logic without latches.

## Printing:

1.**print_factor** – Prints the factored forms of the nodes in the current network.

2.**print_fanio** – Prints the distribution of nodes by the number of fanins and fanouts.

3.**print_gates** – Prints statistics about the gates used after technology mapping. For a technology-independent networks, prints how many nodes have a given type of logic function.

4.**print_io** – Prints the lists of primary inputs (PI), primary outputs (POs), and latches of the network. When called for a node (given by a name on the command line), prints its fanouts and fanouts.

5.**print_kmap** – Prints Karnaugh map of the logic function of a node.

6.**print_latch** – Prints the information about latches of the current networks.

7.**print_level** – Prints the distribution of the COs by the number of levels in their logic cones. If the network is mapped, prints a delay profile of the COs..

8.**print_sharing** – Prints the number of nodes shared by each pair of the COs in the current network.

9.**print_stats** – Prints the vital stats of the current networks. The statistics printed depend on the current network representation.

# EXAMPLE OF USING ABC FOR SYNTHESIS AND MAPPING

```
supriti@supriti-Inspiron-5570:~$ cd Downloads
supriti@supriti-Inspiron-5570:~/Downloads$ cd ABC
supriti@supriti-Inspiron-5570:~/Downloads/ABC$ cd abc-master
supriti@supriti-Inspiron-5570:~/Downloads/ABC/abc-master$ ./abc
UC Berkeley, ABC 1.01 (compiled Nov  4 2020 13:08:14)
abc 01> read a.blif
abc 02> print_factor
  v5.4 = !new_n26_
    n6 = !new_n22_
   n11 = !new_n23_
   n16 = !new_n24_
 n21_1 = !new_n25_
new_n19_ = ((v0v1)(v2(!v3!v4)))
new_n20_1_ = ((!v0v2)(!v3v4))
new_n21_ = ((!v0v1)(v2v3))
new_n22_ = Constant 1
new_n23_ = Constant 1
new_n24_ = Constant 1
new_n25_ = Constant 1
new_n26_ = (!new_n19_(!new_n20_1_!new_n21_))
```

```
abc 02> print_gates
Const         =         4     30.77 %
Buffer        =         0      0.00 %
Inverter      =         5     38.46 %
And           =         4     30.77 %
Or            =         0      0.00 %
Other         =         0      0.00 %
TOTAL         =        13    100.00 %
abc 02> print_io
Primary inputs (1):  0=v0
Primary outputs (1): 0=v5.4
Latches (4):   v1L(v1=n6_1) v2L(v2=n11_1) v3L(v3=n16_1) v4L(v4=n21)
abc 02> print_kmap
Truth table: 1-var function

 \ new_n26_
   0   1
 +---+---+
 | 1 |   |
 +---+---+
abc 02> print_latch
Total latches =     4. Init0 = 4. Init1 = 0. InitDC = 0. Const data = 0.
abc 02> print_level
Level =    1.  COs =    4.    80.0 %
Level =    3.  COs =    1.   100.0 %
```

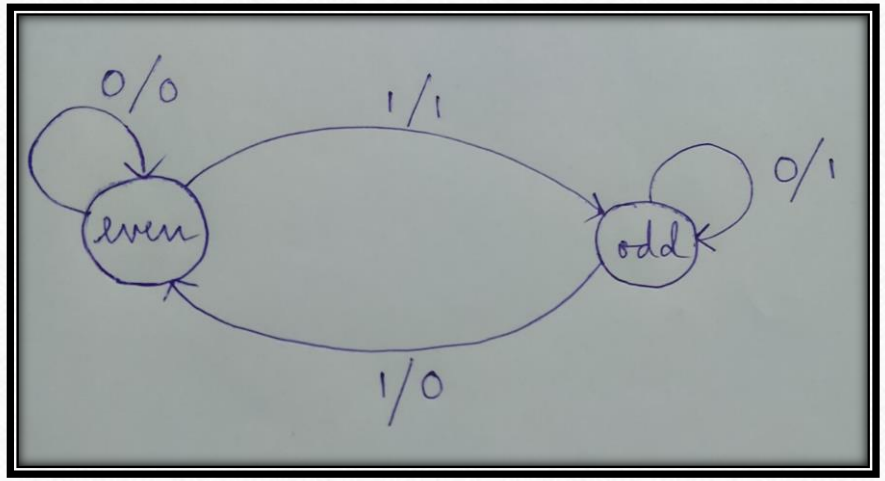
```
abc 02> print_sharing
Statistics about sharing of logic nodes among the CO pairs.
(CO1,CO2)=NumShared : (0,1)=0 (0,2)=0 (0,3)=0 (0,4)=0 (1,2)=0 (1,3)=0 (1,4)=0 (2,3)=0 (2,4)=0 (3,4)=0
abc 02> print_stats
a.kiss2                   : i/o =    1/    1  lat =    4  nd =    13  edge =     21  bdd  =    16  lev = 3
abc 02> print_supp
Structural support info:
    0                 v5.4 :  Cone =     5.  Supp =     5. (PIs =     1. FFs =     4.)
    1                 n6_1 :  Cone =     2.  Supp =     0. (PIs =     0. FFs =     0.)
    2                n11_1 :  Cone =     2.  Supp =     0. (PIs =     0. FFs =     0.)
    3                n16_1 :  Cone =     2.  Supp =     0. (PIs =     0. FFs =     0.)
    4                  n21 :  Cone =     2.  Supp =     0. (PIs =     0. FFs =     0.)
```

# EXAMPLE OF FSM-SERIAL PARITY DETECTOR

## State table:

| PS | NS | | OP | |
|---|---|---|---|---|
| | IN=0 | IN=1 | IN=0 | IN=1 |
| EVEN | EVEN | ODD | 0 | 1 |
| ODD | ODD | EVEN | 1 | 0 |

## State Diagram:



# CODE FOR THE GIVEN FSM

```
#include<stdio.h>
#include<stdlib.h>
enum {even,odd}state;

int main()
{
    char in[50],op[50];
    int i=0;
    printf("Enter the input:");
    gets(in);
    state=even;
```

```
while(in[i]!='\0')
    {
        switch(state)
        {
        case even:
            if(in[i]=='0')
            {
                state=even;
                op[i]='0';
            }
            else
            {
                state=odd;
                op[i]='1';
            }
            break;
```

```
case odd:
        if(in[i]=='0')
        {
            state=odd;
            op[i]='1';
        }
        else
        {
            state=even;
            op[i]='0';
        }
        break;
    }
    i++;
  }
  op[i]='\0';
  printf("The output is:");
  puts(op);
  return 0;
}
```

# REFERENCES

1.Source code:

https://github.com/berkeley-abc/abc

2.Visit ABC webpage:

http://www.eecs.berkeley.edu/~alanmi/abc

3.Read recent papers:

http://www.eecs.berkeley.edu/~alanmi/publications

# The End

Thank you