# Ver2Smv - A Tool for Automatic Verilog to nuXmv Language Translation for Verification of Digital Circuits

**Mishal Minhas, Osman Hasan, Kashif Saghar**

System Analysis and Verification (SAVe) Lab

**National University of Sciences and Technology**

**Islamabad, Pakistan**

**ICEET-2018**

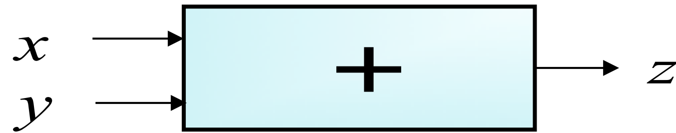
**Lahore, Pakistan**

# Outline

- Introduction and Motivation
- Proposed Approach
- Formal Analysis of Sequential RTL Verilog
- Conclusions

# Hardware Verification

- 8 Bit Adder




- Model
  - VHDL/Verilog
- Test Cases

| Test vectors (x,y) | System output (z) | z=x+y |
|---|---|---|
| (1,1) | 2 | True |
| (4,0) | 4 | True |
| (100,100) | 200 | True |
| (127,127) | 254 | True |

- Conclusion: The property is true as it is found to be true for all the test vectors used

# Safety-Critical Systems

- Accuracy is Extremely Important
  - Failure can cause loss of life or severe injury

**Hardware Systems**

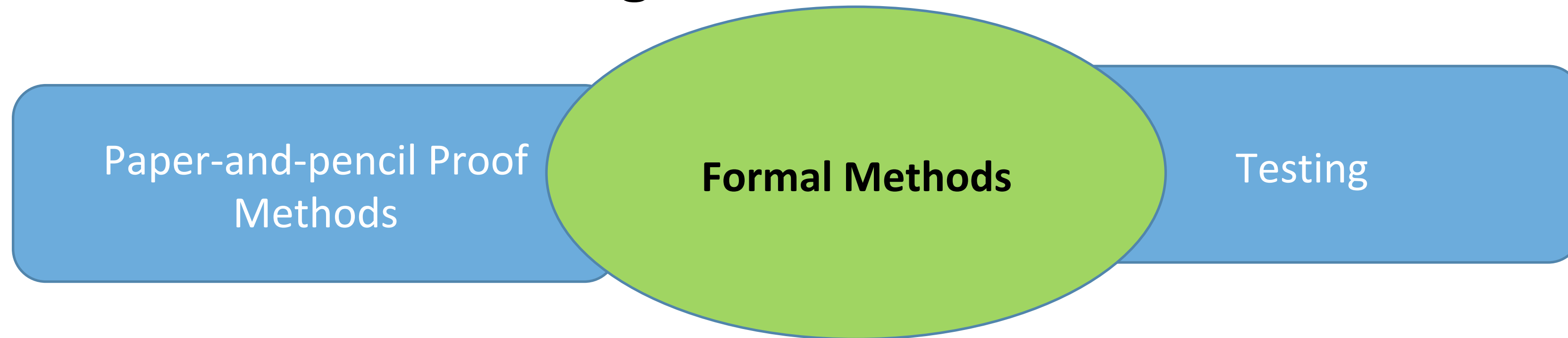
# Faulty Systems can be disastrous

- FDIV bug in Intel Pentium (60 Mhz, 90Mhz)
  - Hardware error in the floating point division unit
  - expected precision up to 18 positions
  - in practice, only 4 positions
  - Example:
    - 5505001 / 294911
    - wrong answer: 18.66600093
    - expected answer: 18.6665197

  - Resulted in net loss of US $500M to the company in 1994

# Solution: Formal Methods

- System Validation technique that bridges the gap between Paper-and-pencil proof methods and testing

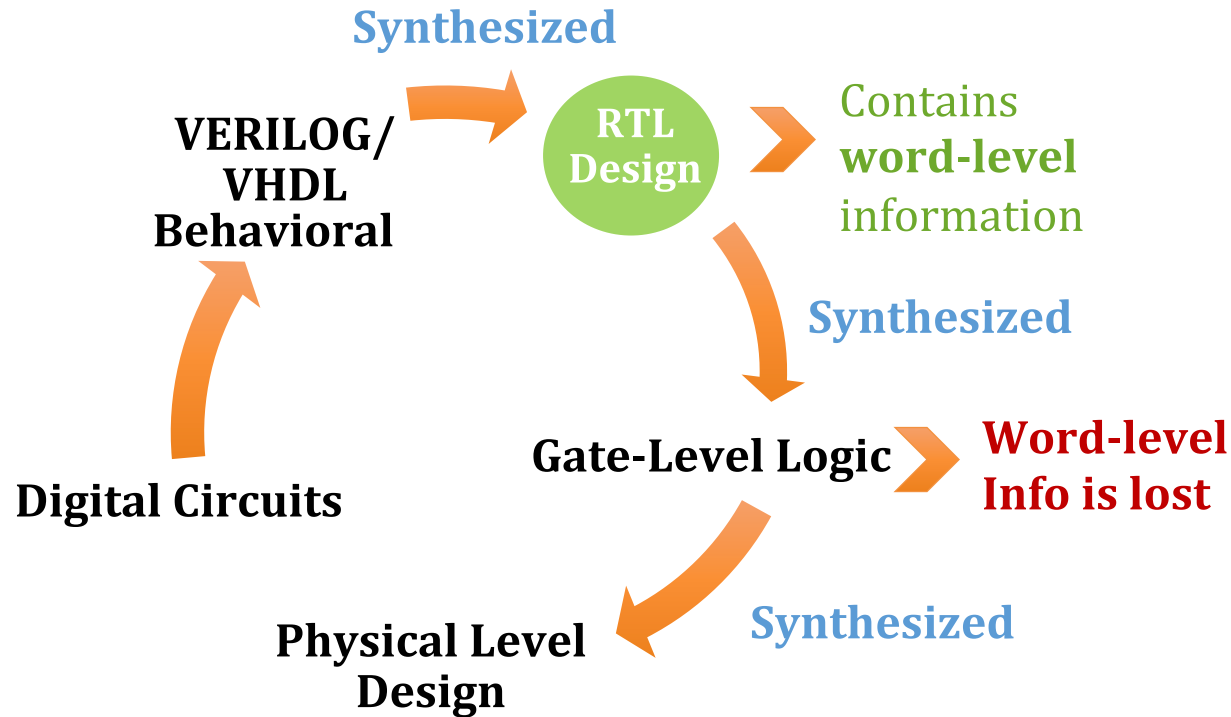



- Shares their advantages
  - As precise as a mathematical proof can be
  - Computers are used for bookkeeping
- Not as straightforward to use as testing

# Formal Verification Methods

- Based on Mathematical techniques
  - Construct a computer-based mathematical model of the VHDL/Verilog model (*implementation*)
  - Use mathematical reasoning to check if the implementation satisfies the properties of interest (*specifications or assertions*) in a computerized environment

# Problem Description




```
MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value)  := FALSE;
  next(value)  := value xor carry_in;
DEFINE
  carry_out  := value & carry_in;

MODULE main
VAR
  bit0 : counter_cell(TRUE);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter cell(bit1.carry out);

   LTLSPEC
     G F bit2.carry_out
```

- Assertions describe how the circuit should behave

Formal Modeling at the RTL level and writing quality assertions manually requires a lot of effort and time – not effective for industrial applications

# Related Work

- EBMC – Enhanced Bounded Model Checker

- Reads Verilog designs with properties in LTL or System Verilog Assertions
- Outputs boolean level MC problem in SMV
- Uses BMC and/or k-induction

**Limitation:** Slow performance

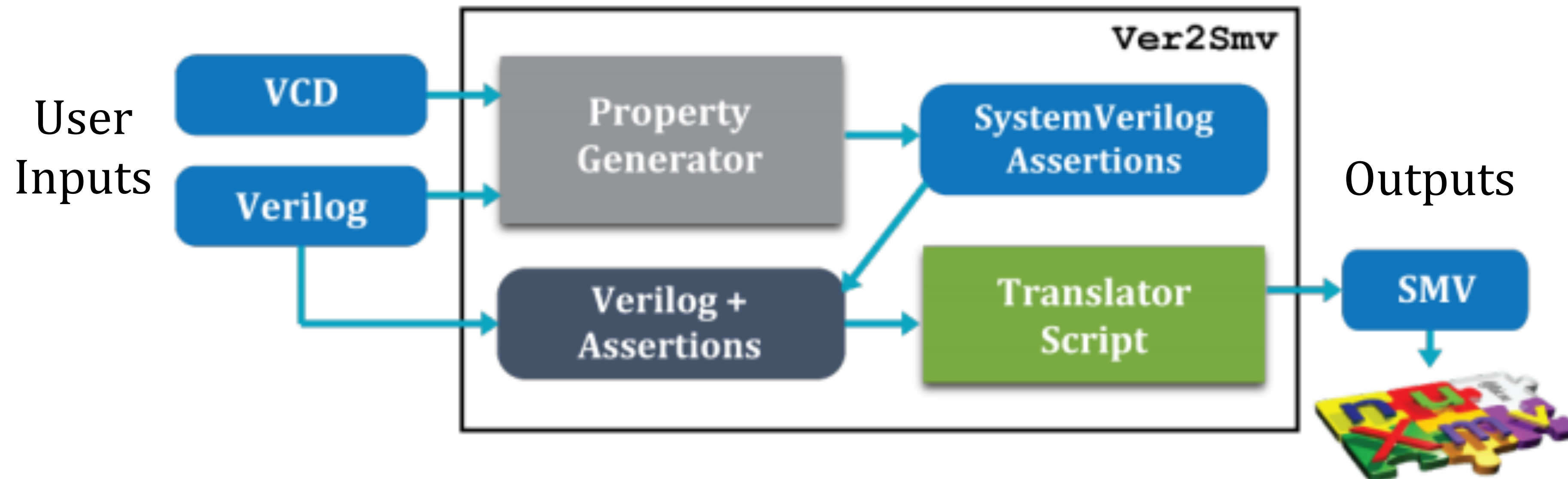
- Verilog2smv - Translator for Verilog Designs

- Open source
- Reads Verilog with Assertions
- Outputs a word-level MC problem in SMV

**Limitation:** Assertions are written manually by user

# Proposed Approach

*Generates the SystemVerilog Assertions and then translates the circuit's implementation and assertions to SMV model for automatic formal verification*



# Formal Analysis of Sequential RTL Verilog

**Sequential Circuit**

```
always @ (posedge clk, posedge rst)
  if (rst)
        state <= 0;
        state <= gnt1;  else
always @ (*)
  if (state)  begin
        gnt1 = req1 & ~req2;
        gnt2 = req2;    end
  else
  begin
        gnt1 = req1;
        gnt2 = req2 & ~req1;    end
  assert property ( !( req1 == 0 ) || ( gnt1 == 0 ) );
  assert property ( !( req1 == 1 && req2 == 0 ) || ( gnt1 == 1 ));
  assert property ( !( req2 == 0 ) || ( gnt2 == 0 ) );
  assert property ( !( req1 == 0 && req2 == 1 ) || ( gnt2 == 1 ));
endmodule
```

**RTL Verilog**
**Input by user**

**Assertions generated automatically**

**Translated SMV Model**

```
MODULE main
IVAR
   "clk" : boolean;
   "req1" : boolean;
   "req2" : boolean;
   "rst" : boolean;
VAR
   "state" : boolean;
DEFINE
   __expr0 := word1("req1");
       .
       .
       .
   __expr85 := bool(0ub1_1);
   __expr86 := (__expr85 -> __expr84);
TRANS next("state") =   expr8
INVARSPEC   __expr22;
INVARSPEC   __expr44;
INVARSPEC   __expr64;
INVARSPEC   __expr86;
```

***Reg and mem*** **assignments into** ***TRANS*** **constraint**

***Assertions*** **into** ***Invar specs***

# Conclusions

- Extensive **Circuit Simulations** require heavy computations and can still result in faulty hardware and uncertainties
- **Formal analysis** of digital circuits require a formal model with desired properties
  - Writing a circuit's formal model **manually is ineffective**
  - Property specification requires effort and time
- Proposed methodology presents a tool to **automate** this process that ensures **minimum user involvement** with **complete verification**

# Thanks!

SCHOOL OF ELECTRICAL ENGINEERING & COMPUTER SCIENCE (SEECS)