

9/2/26

COMP 435:



Computer Security Concepts

Lecture 7: Hardware Security & Security Properties



The University
of North Carolina
at Chapel Hill

<https://tinyurl.com/comp435-fa26>

Logistics Update



- First Written Assignment
 - One part of worksheet due Thursday (9/3) at 11:59 (70% of grade)
 - Second part due in class (30%)

Written Assignment Details: Security in Practice



- Come to class with your individual incident analysis completed
- In small groups, briefly teach your incident to your classmates
- Compare the assets, adversaries, attack paths, risk, and ATT&CK techniques across incidents
- Take notes on your classmates' incidents using the in-class worksheet
- As a group, discuss similarities, differences, and what you learned
- Turn in the completed worksheet at the end of class
- Take home worksheet due tomorrow is 70%, in-class ws is 30%

Plan for Class Today



- 10:10-10:20: Some Hardware Background
- 10:20-11:00: How to Specify a Security Property in Hardware

Guiding Question 😊

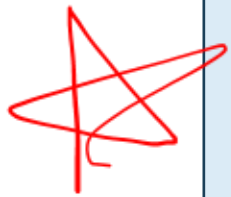
How can we determine/test if hardware deserves our trust?

Hardware Is Not Perfect!



** Side channel attack*

- Correct architectural behavior is not the whole story
- Attacks can exploit side effects of implementation details
- Timing ⌚, caches 🗄, power ⚡, and ⌞ speculation can leak information



Hardware is both a security mechanism and a potential source of vulnerabilities!

UNIT 1: TRUSTING THE MACHINE

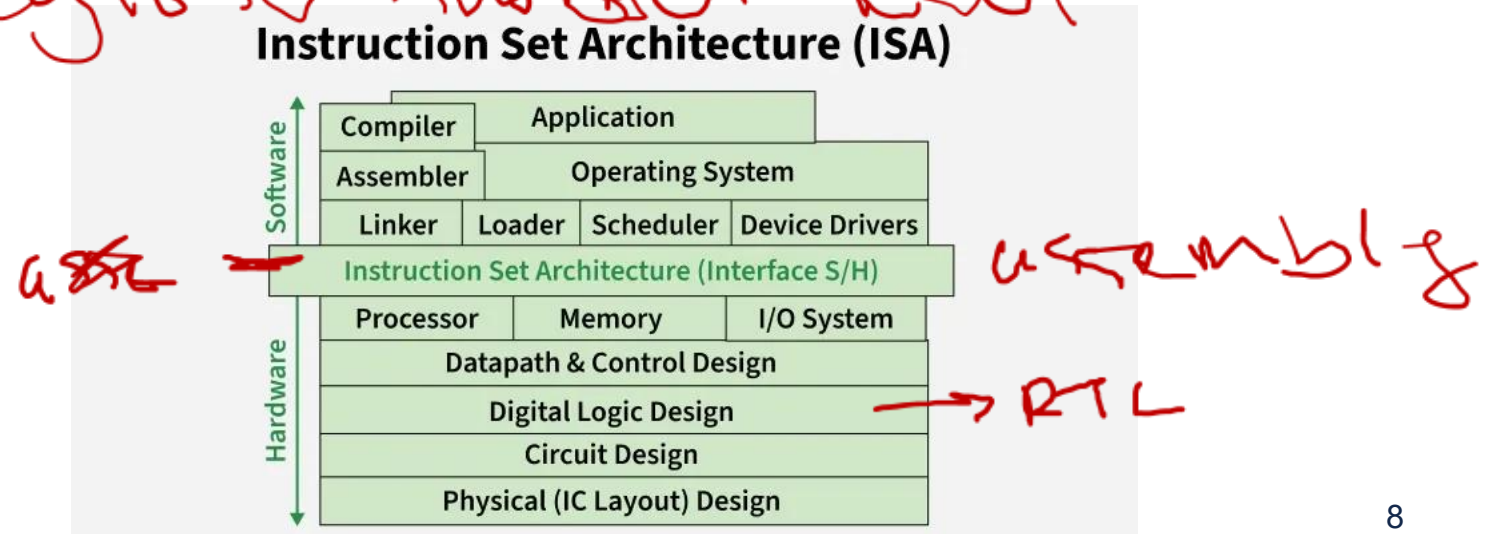
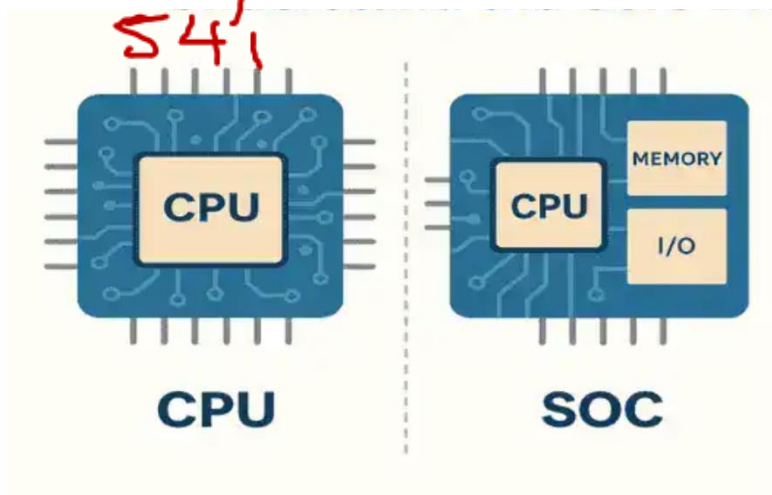
Some Hardware Background

311 is not a pre-req... lets get up to speed!

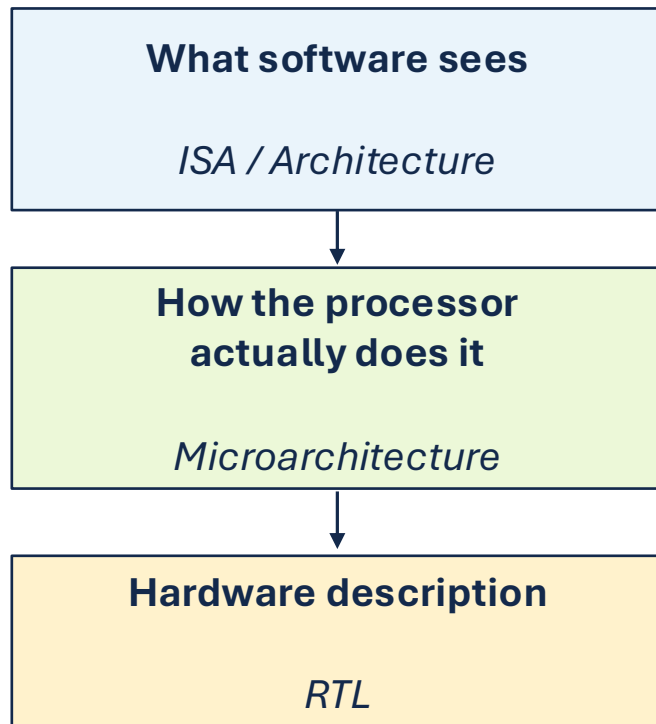


- **CPU / processor:** hardware that executes program instructions
- **SoC:** processor plus other hardware components in one system
- **ISA:** the behavior the processor exposes to software
 - Ex: MIPS, RISC-V, x86
- **RTL:** a code-like description of hardware behavior and state
 - Ex: Verilog, VHDL

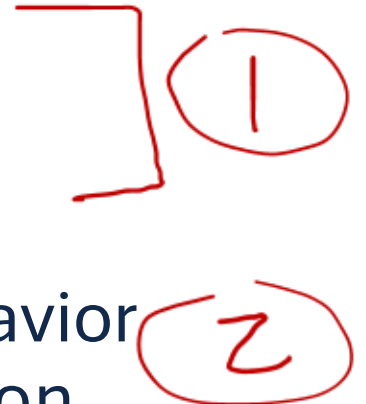
instruction set architecture
register transfer level



Architecture vs. Implementation



- A design can be functionally correct at one level and still create security problems at another.
- **Verification** asks whether an implementation satisfies stated properties.
- **Side channels** often exploit behavior below the architectural abstraction.
 - i.e.; microarchitectural timing, cache state



Hardware Has State

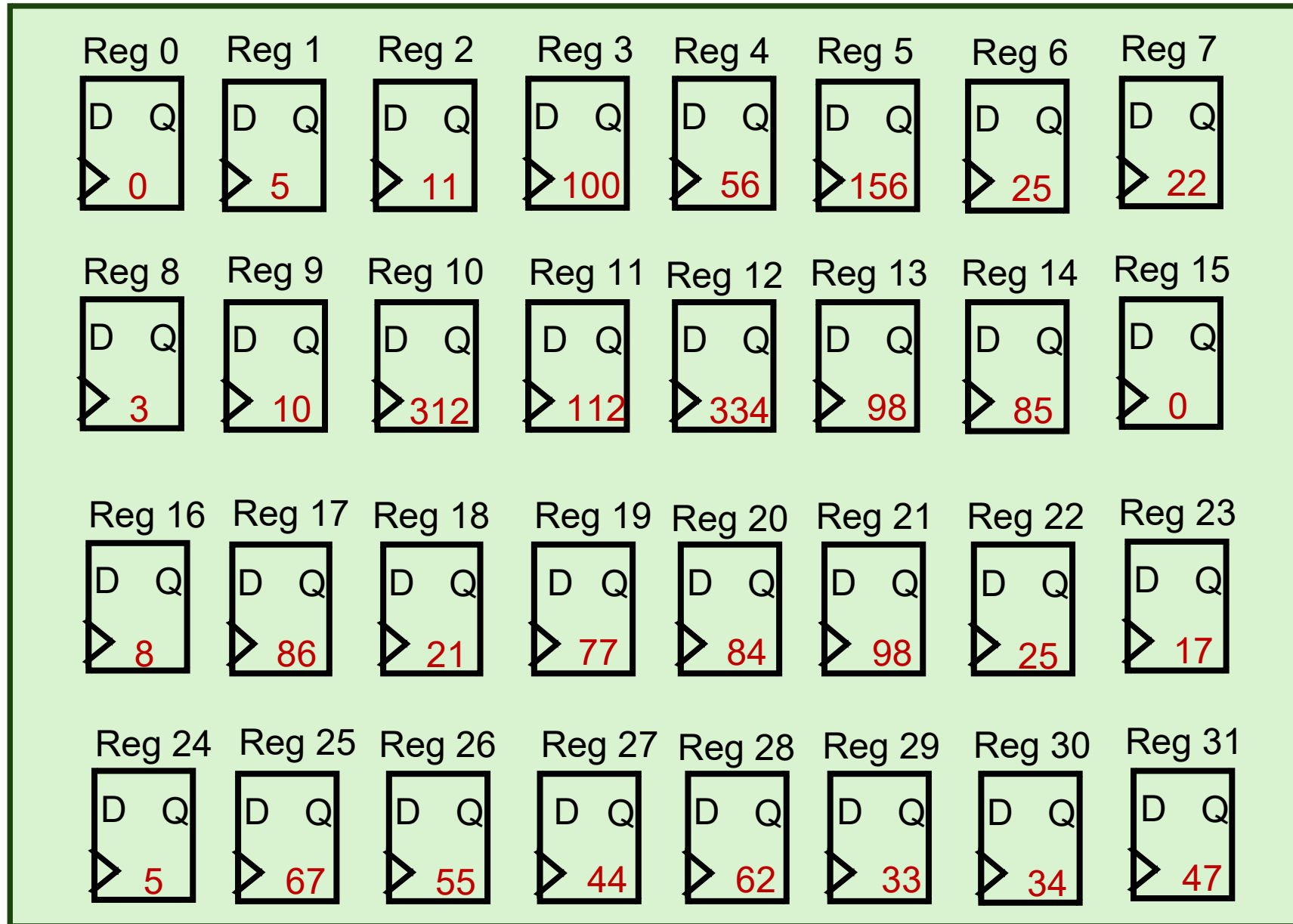


RTL

- **Signal:** a value moving through hardware
- **Register:** stores a value across clock cycles
- **Clock cycle:** one step of hardware execution
- **Module:** a hardware component with inputs, outputs, and state

```
module counter (  
    input logic clk,  
    input logic reset,  
    output logic [3:0] count  
);  
  
    always_ff @(posedge clk) begin  
        if (reset)  
            count <= 0;  
        else  
            count <= count + 1;  
        end  
    endmodule
```

CPU State: Register File



Hardware Has State



- **Signal:** a value moving through hardware
- **Register:** stores a value across clock cycles
- **Clock cycle:** one step of hardware execution
- **Module:** a component with inputs, outputs, and state

```
module counter (  
    input  logic clk,  
    input  logic reset,  
    output logic [3:0] count  
);  
  
always_ff @(posedge clk) begin  
    if (reset)  
        count <= 0;  
    else  
        count <= count + 1;  
end
```

Security properties are often specified as: “this state should never change in this situation”



UNIT 1: TRUSTING THE MACHINE

Security Properties

From Security Policy to Hardware Property



- **Recall, Security Policy:**
- **Security Property:** A precise rule describing behavior that a secure design should always satisfy

what system should /
should not allow

Security policy

Ordinary programs should not perform privileged operations.

Security property

IF privilege = USER
THEN privileged_write = FALSE

From Security Policy to Hardware Property



- **Recall, Security Policy:** A statement of what a system should and should not allow in order to protect its assets.
- **Security Property:** A precise rule describing behavior that a secure design should always satisfy

Security policy

Ordinary programs should not perform privileged operations.

Security property

IF privilege = USER
THEN privileged_write = FALSE

- **Counterexample:** A sequence of events showing the property can be violated.

Property Pattern



- IF privilege = user, THEN privileged register cannot change
- IF debug interface is locked, THEN debug write cannot occur
- IF authentication has not succeeded, THEN protected state cannot be modified

IF _____ THEN _____

Using Properties → Find Counterexamples



Property:

IF locked = TRUE
THEN secret_register cannot change



Potential Counterexample:

1. set locked = TRUE
2. send debug command
3. secret_register changes

A counterexample is not always the full real-world attack, but it can reveal a feasible violation path.

Security Property Writing



- IF $privilege == USER \Rightarrow$
 $priv_reg_sk_next_st == priv_reg_cur_state$
- Try writing a security property, given a policy.

- Just use pseudocode + math notation! Doesn't have to be perfect

- Policy

- A. User code should not modify a privileged configuration register. $asset$ $asset$
- B. Once a security lock is enabled, the protected register should not change.
- C. A debug interface should not write protected state until authentication succeeds.

Identify the asset, forbidden behavior, property, and a potential counterexample.

$IF privilege == USER THEN$
 $write_to_privileged_register == FALSE$

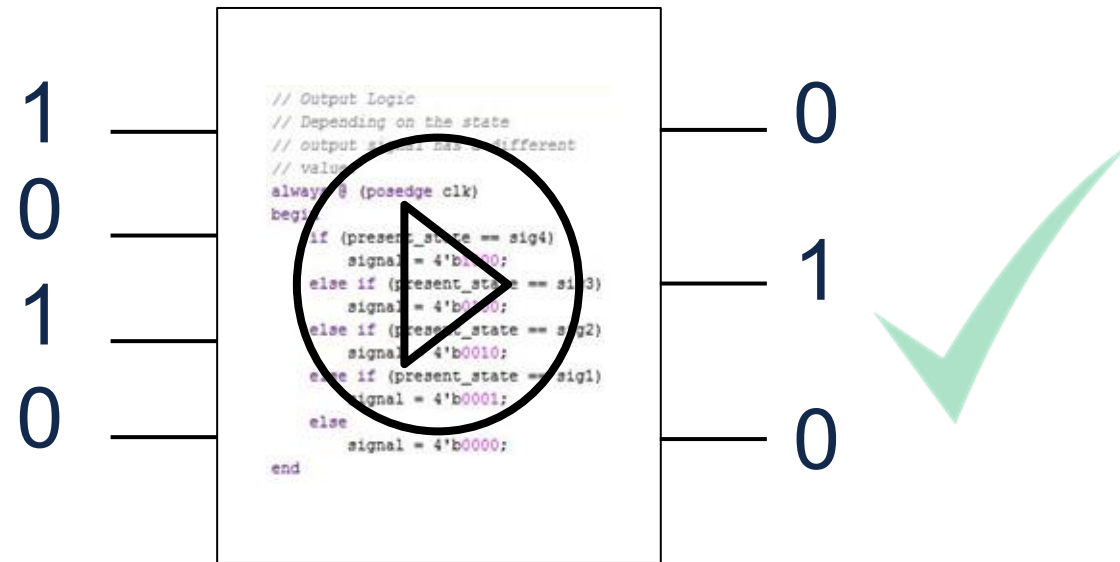
How do we check a property?



- IF privilege = user, THEN privileged register cannot change
- IF debug interface is locked, THEN debug write cannot occur
- IF authentication has not succeeded, THEN protected state cannot be modified

IF _____ THEN _____

Simulation-Based Testing



Simulation-Based Testing: Limitations



```
if (x == 13):  
    ERROR  
if (x > 0):  
    count := count + 1  
else:  
    count := count - 1
```

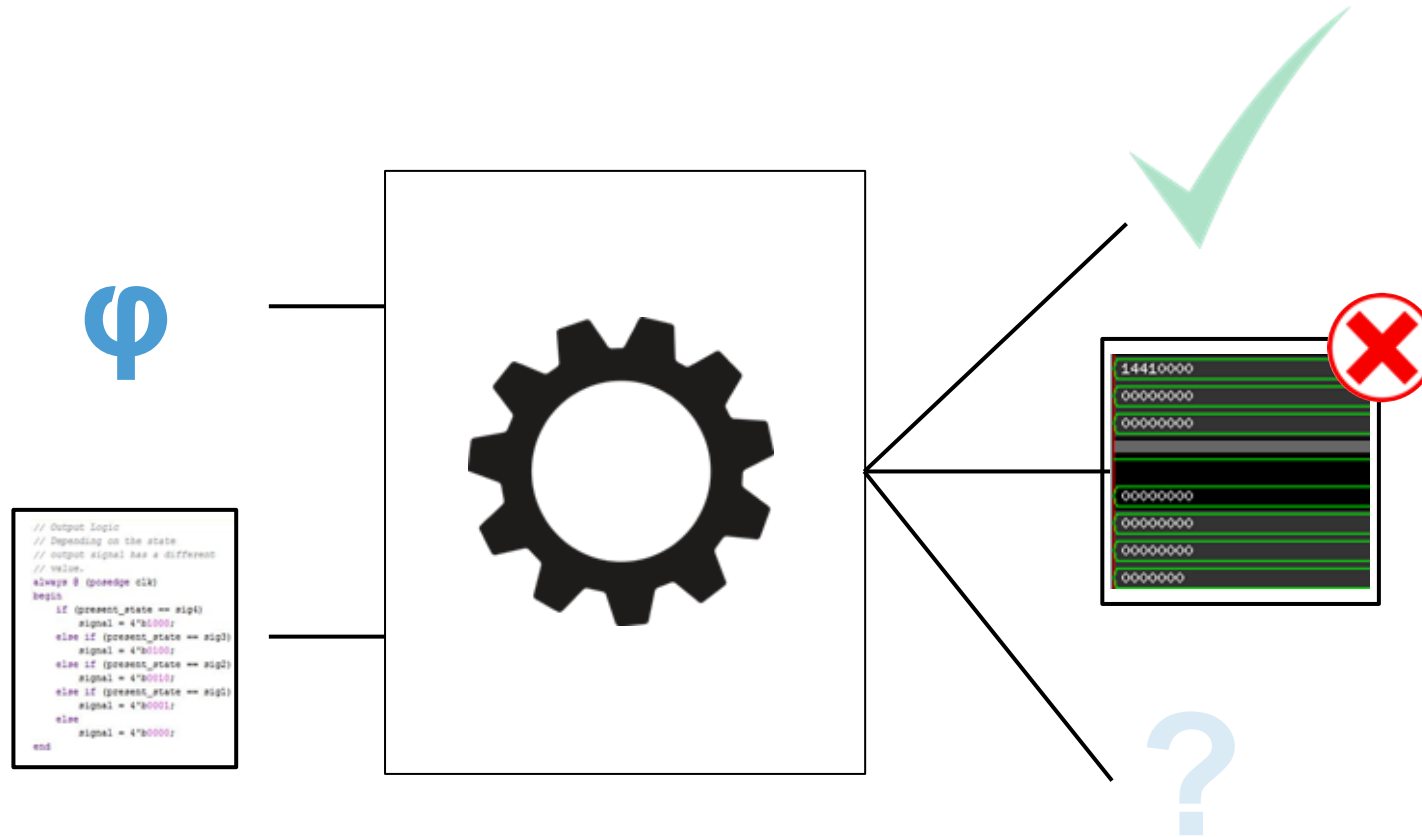
Simulation-Based Testing: Limitations



```
if (x == 13):  
    ERROR  
if (x > 0):  
    count := count + 1  
else:  
    count := count - 1
```

 Coverage

Model Checking

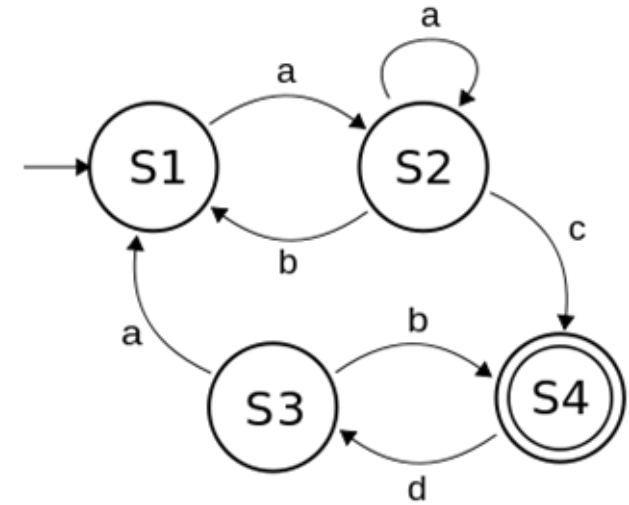


Model-Checking: Limitations



```
// Output Logic
// Depending on the state
// output signal has a different
// value.
always @ (posedge clk)
begin
    if (present_state == sig4)
        signal = 4'b1000;
    else if (present_state == sig3)
        signal = 4'b0100;
    else if (present_state == sig2)
        signal = 4'b0010;
    else if (present_state == sig1)
        signal = 4'b0001;
    else
        signal = 4'b0000;
end
```

formal model



Scalability