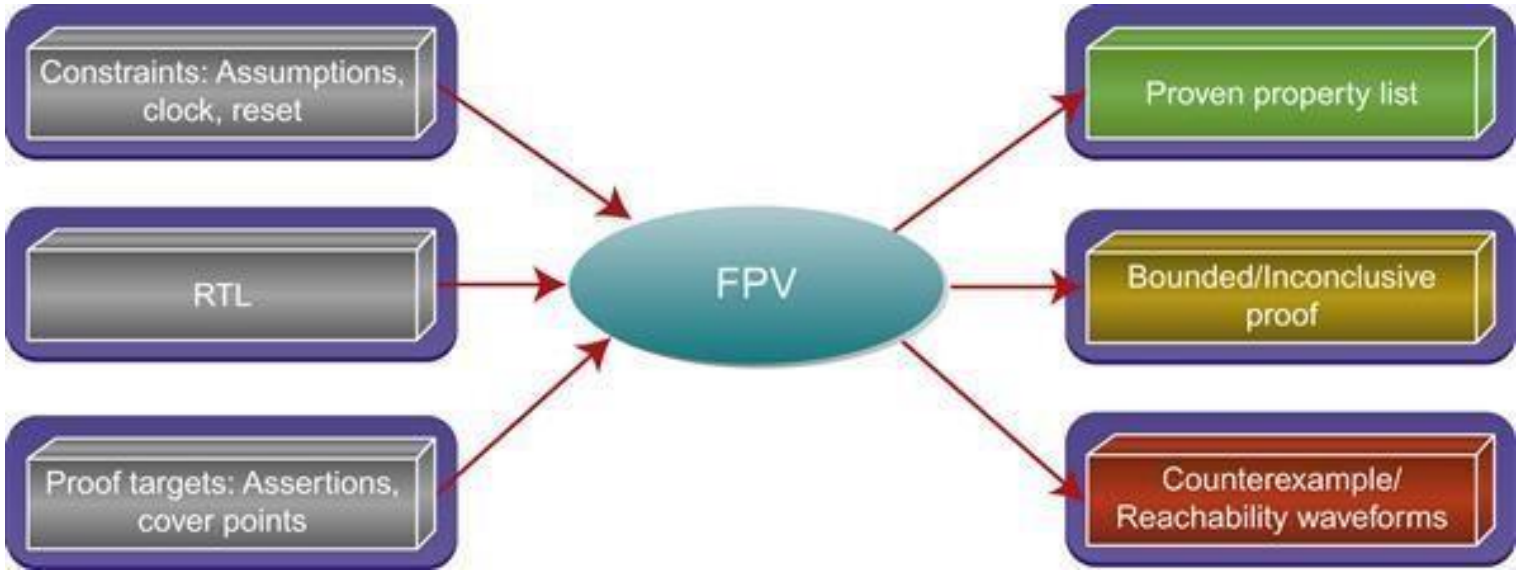
Formal Verification: An Introductory Lab

Sheena Goel
Venky Kottapalli
Prof. Flemming Andersen
Prof. Michael Quinn

Spring 2018

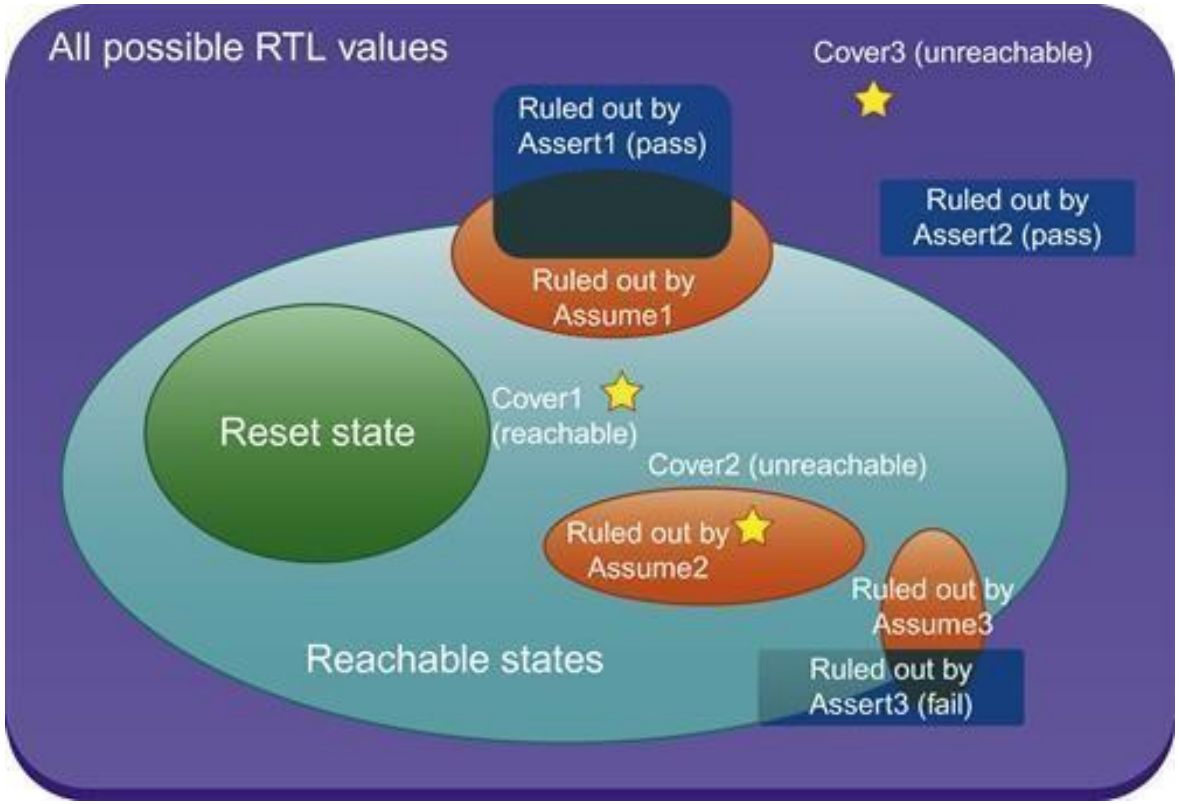
Formal Property Verification

- Prove a set of properties (SVA)
- Analyze space of all possible logical execution paths



Execution Space

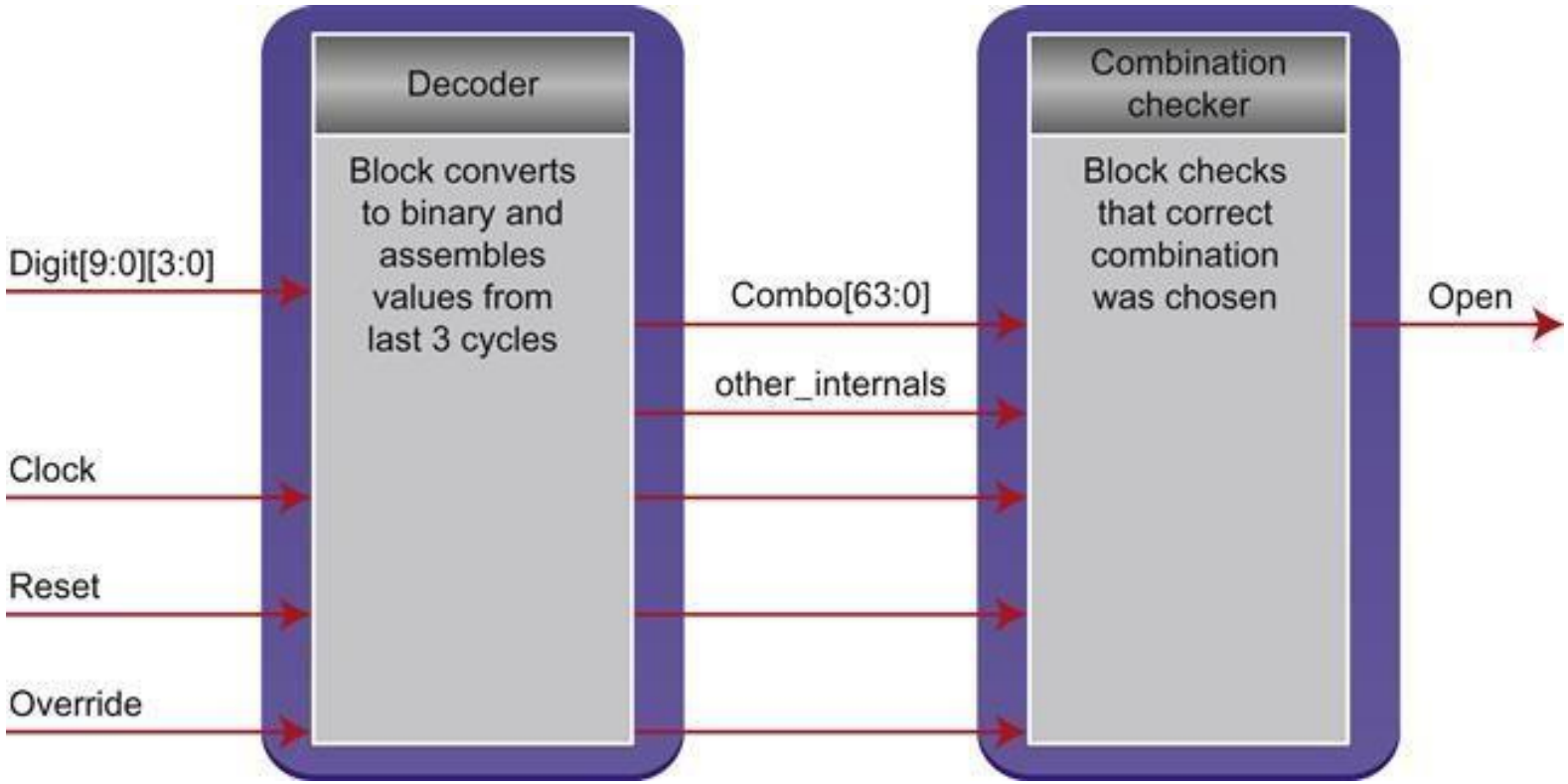
- Each point = set of values for all state elements (FF/latch)



Usage Models

- Design Exercise FPV
- Bug Hunting FPV
- Traditional Full Proof FPV
- Application Specific FPV
  - Debug top level bus connectivity
  - Coverage holes

Example: Combinational Lock



Goals (Verify):

- Open lock with correct combination
- No incorrect combination can open lock

Steps:

- Compile RTL
  - Synthesizable RTL only
- Create cover points
- Create assumptions
- Create assertions
- Specify clock and reset
- Run the verification

RTL

```
module combination_lock (
    input bit [9:0] digits [3:0],
    input bit override,
    input bit clk, rst,
    output bit open
    );

    bit [63:0] combo;
    decoder d1(clk, rst, digits, combo);
    combination_checker c1(clk, rst, combo, override,
    open);
endmodule
```

- What is the correct combination to open the lock?

Cover Points

- Wrongly treated as afterthought
- Include:
  - Each documented flow
  - Interesting input and output values
  - Each type of transaction/operation
  - Each state of FSM
- It's possible for our lock to be locked or unlocked.

```
c1: cover property (open == 0);
c2: cover property (open == 1);
```

- Each input on the dial can legally take on each digit

```
generate for (i=0; i<3; i++) begin
 for (j=0; j<9; j++) begin
  // We use a one-hot encoding
  c3: cover property (digit[i][j] == 1);
 end
end
```

Assumptions

- Basic behavior of design inputs
- Impossible to list all assumptions in advance
- Describes environment around DUV
- One hot encoding:

```
generate for (i = 0; i<4; i++) begin
  a1: assume property ($onehot(digits[i]));
end
```

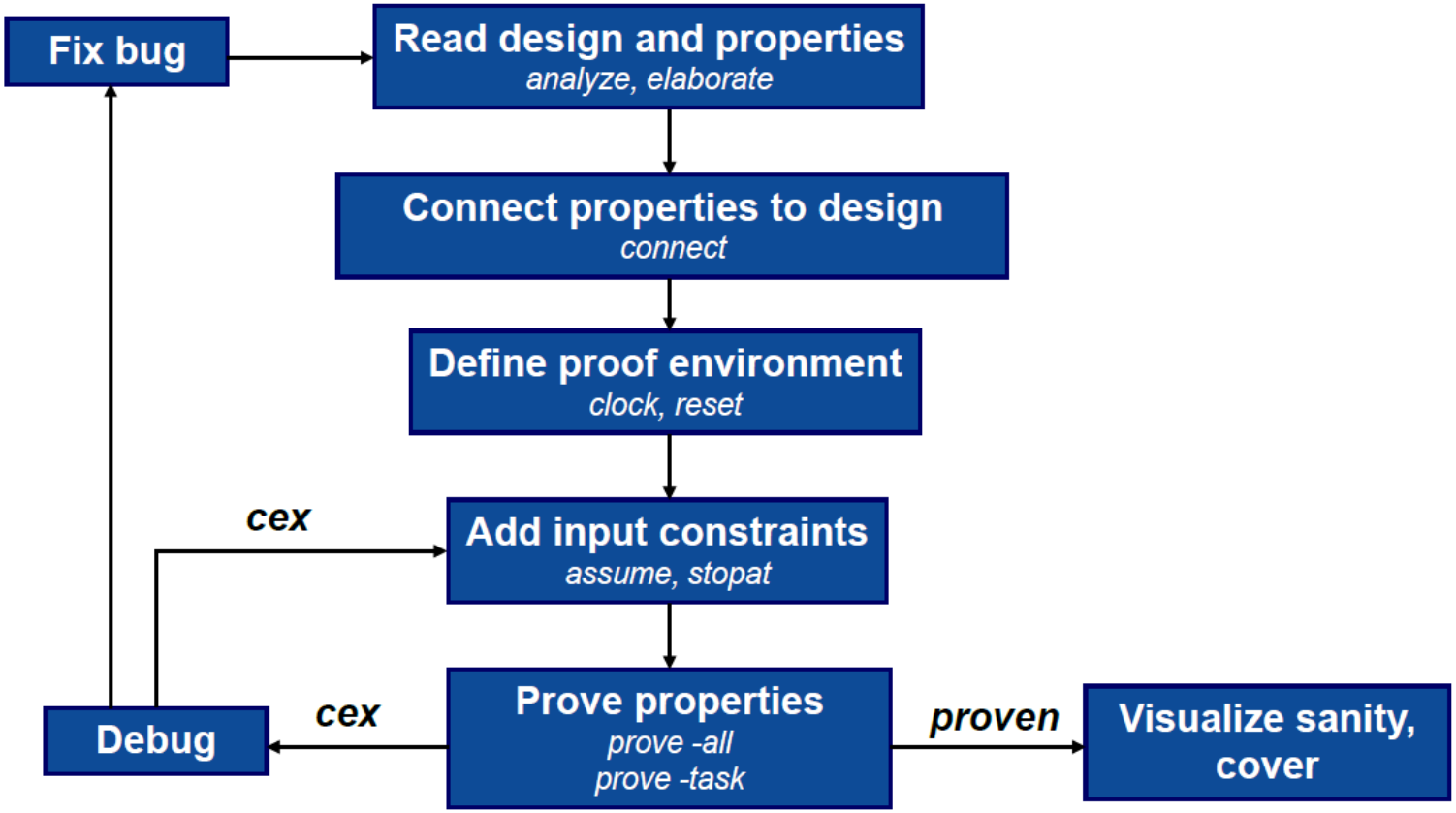
Assertions

```
sequence correct_combo_entered;
 (digits == COMBO_FIRST_PART) ##1
 (digits == COMBO_SECOND_PART) ##1
 (digits == COMBO_THIRD_PART);
endsequence

open_good: assert property (correct_combo_entered |=> open);

open_bad2: assert property (
  open |-> $past(digits,3)==COMBO_FIRST_PART));
open_bad1: assert property (
  open|-> $past(digits,2)==COMBO_SECOND_PART));
open_bad0: assert property (
  open |-> $past(digits,1)==COMBO_THIRD_PART));
```

JasperGold Use Model



Run the Tool

Expected Results:

- Passing cover point = example trace
- Failing assertion = counter example trace
- Passing assertion/ failing cover point
- Assertion with bounded proofs

First Debug

Property	Status
combination_lock.c1	covered: 1 cycle
combination_lock.c2	covered: 2 cycles
combination_lock[*][*].c3	covered: 1 cycle
combination_lock.open_good	fail: 4 cycles
combination_lock.open_ok2	fail: 4 cycles
combination_lock.open_ok1	fail: 3 cycles
combination_lock.open_ok0	fail: 2 cycles

- 'override' input
- Solutions:
  - Revise cover points
    - `c2: cover property (open == 1);` with
      `c2: cover property ((open==1) && !$past(override));`
  - Create assumption
    - `fix1: assume property (override == 0);`

Second Debug

Property	Status
combination_lock.c1	covered: 1 cycle
combination_lock.c2	covered: 4 cycles
combination_lock[*][*].c3	covered: 1 cycle
combination_lock.open_good	fail: 4 cycles
combination_lock.open_ok2	fail: 4 cycles
combination_lock.open_ok1	fail: 4 cycles
combination_lock.open_ok0	fail: 4 cycles

- Solution: Add the right combination

Third Debug

Property	Status
combination_lock.c1	covered: 1 cycle
combination_lock.c2	covered: 4 cycles
combination_lock[*][*].c3	covered: 1 cycle
combination_lock.open_good	Pass
combination_lock.open_ok2	fail: 4 cycles
combination_lock.open_ok1	fail: 4 cycles
combination_lock.open_ok0	fail: 4 cycles

- Solution: Fix RTL Bug

Final Result

Property	Status
combination_lock.c1	covered: 1 cycle
combination_lock.c2	covered: 4 cycles
combination_lock[*][*].c3	covered: 1 cycle
combination_lock.open_good	pass
combination_lock.open_ok2	pass
combination_lock.open_ok1	pass
combination_lock.open_ok0	pass

Simulation vs FPV

Key area of difference	Simulation	FPV
What types and sizes of models can be run?	**Up to full-chip level**, and both synthesizable and behavioral code.	**Unit or cluster level**, or full-chip with lots of removed logic, and synthesizable code only.
How to reach targeted behaviors?	Describe **the journey**: generate input values to create desired behaviors.	Describe **the destination**: specify target property or state, tool finds if it's reachable.
What values are checked?	**Specific values**: simulator checks design based on user-specified inputs.	**Universe of all possible values**: tool analyzes space of legal values under current constraints.
How do we constrain the model?	**Active constraints**: we implicitly constrain the model by limiting the values passed in during simulation.	**Passive constraints**: use assumptions to rule out all illegal cases.
How are constraints on internal nodes handled?	**Forced values**: a live constraint on an internal signal affects only downstream logic.	**Back-propagation + Forward-propagation**: a constraint affects the universe of possible values, in its fanin and fanout logic.
What do we use for debug?	**Single trace**: when a simulation fails, you need to debug what was simulated.	**Example from FPV space**: the tool presents one example from a family of possible failures.
How long are typical traces?	**Thousands of cycles**: typically need to represent full machine startup and do many random things before hitting complex bug.	**Tens of cycles**: FPV usually generates a minimal path to any target, resulting in much smaller traces with little extra activity.

References

- Chapter 4 of ‘Formal Verification’ by Erik Seligman et al.
- Verification Academy
- Cadence ABV, JasperGold Tutorials

Git Commands

git clone https://github.tamu.edu/sheenagoel/CSCE-689-Lab8.git

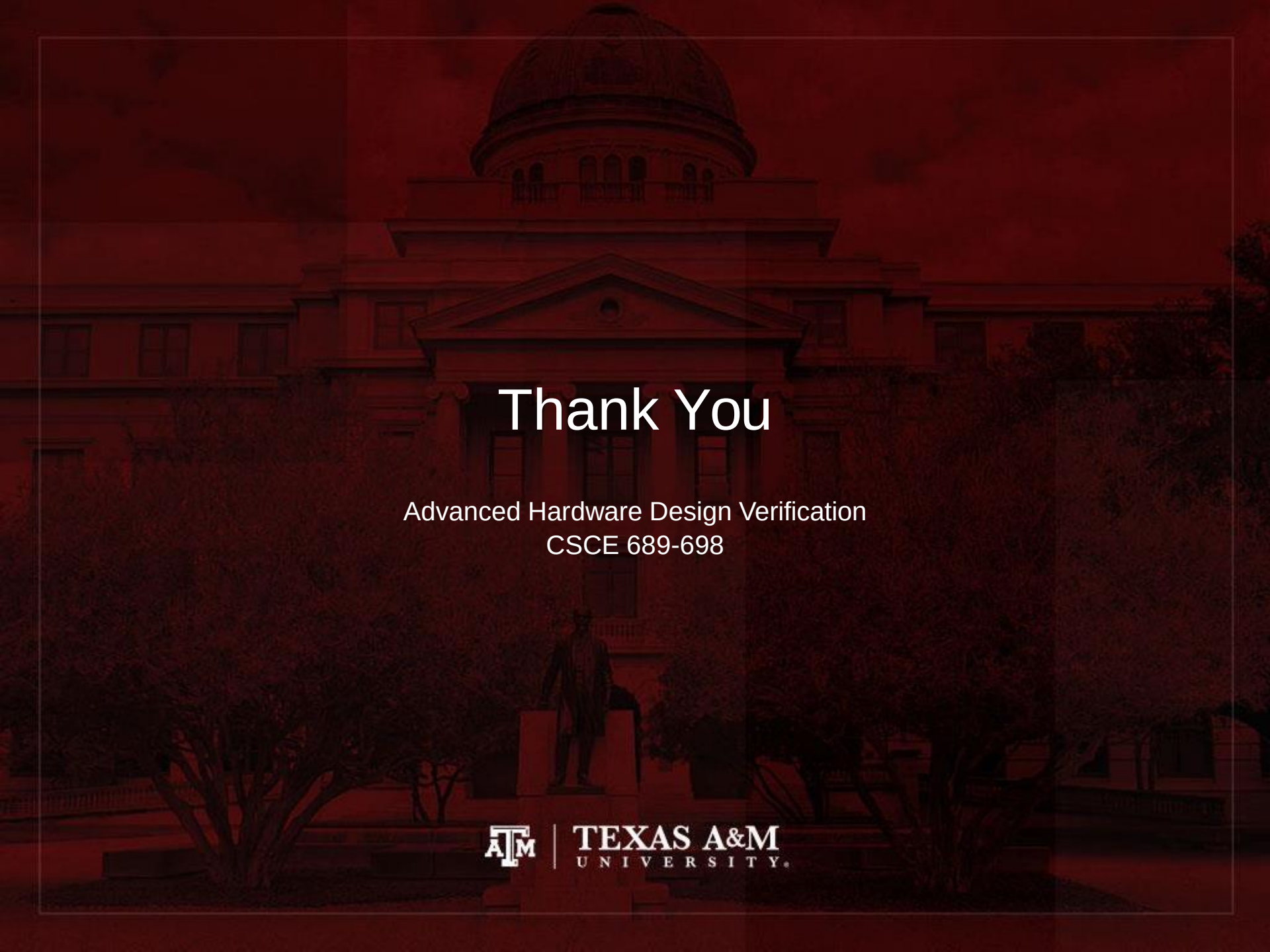
Demo Part 1:
git checkout 4186693

Demo Part 2:
git checkout 727d53f

Demo Part 3:
git checkout a5e34af

Demo Part 4:
git checkout f9703ba

Demo Part 5:
git checkout fd665aa


Thank You
Advanced Hardware Design Verification
CSCE 689-698
TEXAS A&M
UNIVERSITY