

# COURSE SUMMARY

---

- **Verification of Digital Systems**

- Systematic approaches to checking the functional correctness of synchronous digital systems

- **Design Representation**

- Netlists
  - \* Syntax
  - \* Various Semantics
  - \* Compositionality
- Intuitively, trying to model cycle-accurate designs (not necessarily Boolean)
- + Simple model, quite powerful
- – Cycle accuracy precludes abstraction

- **FSMs**

- strictly higher level of abstraction than netlists
- formally in terms of tuples, intuitively graphs

- **Design Equivalence**

Key: equivalence with respect to input-output behavior

- DIS assumption
- Notion of safe replaceability

Closely related to automata/regular language theory

- **BDDs**

- Fundamental data structure for representing and manipulating logic functions
- Based on Shannon expansion  $f = f_x + f_{x'}$
- Reduced Ordered Graph
- Key operations: all based on Shannon expansion and caching
  - \* ITE, compose, etc.
- Implementation: strong canonical form
  - \* hash nodes based on id, left/right children
  - \* invariant: no two nodes in the unique table represent same function
  - \* common cache for operations

- **CTL: Properties**

- formal way of specifying properties of reactive systems
    - \* atomic propositions  $\approx$  sets of states
    - \* Boolean connectives
    - \* Temporal operators—next state, globally, until
  - highly expressive: sequencing, responsiveness, etc.
  - too expressive?!
- Use in conjunction with assume-guarantee reasoning

- **CTL: Model Checking**

- Several ways possible
  - \* recursively compute all states satisfying a formula
  - \* use “fixed point” operators for EG, EU
- “state explosion” problem

- **Symbolic Model Checking**

- Key: represent/manipulate sets and relations using BDDs
  - ⇒ overcome state explosion
- Image:  $\exists X. \exists U. [T(X, U, Y) \cdot A(X)]$
- Many BDD optimizations
  - \* Variable ordering: static, dynamic
  - \* Don't cares
  - \* Partitioned transition relations

- **Fairness**

- “except out” some behaviors by specifying fairness conditions
- makes most sense for environment models
- many formalizations: weak, strong, conditional, etc.
- small modifications to standard algorithms

- **Testing**

- Most widespread form of functional verification
- Simple, scales
- incomplete, manual definitions of tests

- **SymGen**

- Represent environment using constraints
  - \* Easier to write constraints
  - \* Less likely to leave out possible cases
  - \* Can check constraints at higher levels
- Generate vectors under constraints and biases
  - \* BDD-based algorithm for test-generation
  - \* Many optimizations for performance: partitioning, hold-constraints, prioritized constraints
  - \* Can compute circuit from constraints

- **Testing: Methodology**

- Tools: lint, simulators, viewers, languages
- Processes: source code control, issue tracking
- Metrics: coverage, decision making
- Planning: documentation, mapping specs to features to tests
- Tips: behavioral modelling, harnesses, etc.

- **Software Testing**

- Guiding principles: simplicity, clarity, generality, automation
- Test systematically, automate
- Program defensively, test incrementally, know what to expect
- Many tips (array sizes, initialization, etc.)

- **Foundations**

- Automata theory
  - \* Finite automata, regular languages, closure properties, constructions
- Logic
  - \* Syntax, Semantics, Deduction, Relationships

# COURSE ANALYSIS

---

## STRENGTHS

- Theory—rigorous, automated way of design verification
  - Formal models
  - Algorithms
- Application
  - VIS software
  - Examples

## SHORTCOMINGS

- Formal methods limited in practice because of capacity constraints
- Have more testing projects
- Many specialized topics left out (e.g., processor verification, theorem proving)

## FEEDBACK

- Lectures, Homework—usefulness, interesting, understanding
- Material—add, remove