What we HOPE you learned in CS 150 ...

- Language of logic design
- Logic optimization, state, timing, CAD tools
- Concept of state in digital systems
- Analogous to variables and program counters in software systems
- Hardware system building
- Data path + control = digital systems
- Tools to simulate design behavior: output = function (inputs)
- Logic compilers synthesize hardware blocks of our designs
- Mapping onto programmable hardware (code generation)
- Contrast with software design
  - Both map specifications to physical devices
  - Both must be flawless... the price we pay for using discrete math

Current state of digital design

- Changes in industrial practice
  - Larger designs
  - Shorter time to market
  - Cheaper products
- Scale
  - Pervasive use of computer-aided design tools over hand methods
  - Multiple levels of design representation
- Time
  - Emphasis on abstract design representations
  - Programmable rather than fixed function components
  - Automatic synthesis techniques
  - Importance of sound design methodologies
- Cost
  - Higher level of integration
  - Use of simulation to debug designs

CS 150: concepts/skills/abilities

- Basics of logic design (concepts)
- Sound design methodologies (concepts)
- Modern specification methods (concepts)
- Familiarity with full set of CAD tools (skills)
- Appreciation for differences and similarities (abilities) in hardware and software design

New ability: to accomplish the logic design task with the aid of computer-aided design tools and map a problem design onto an implementation with programmable logic devices after validation via simulation and understanding of the advantages/disadvantages as compared to a software implementation

Representation of digital designs

- Physical devices (transistors, relays)
- Switches
- Truth tables
- Boolean algebra
- Gates
- Waveforms
- Finite state behavior
- Register-transfer behavior
- Concurrent abstract specifications

Digital System Design

- Door combination lock:
  - Punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - Inputs: sequence of input values, reset
  - Outputs: door open/close
  - Memory: must remember combinations or always have it available as an input

Implementation in software

```c
integer combination_lock ( ) {  
  integer v1, v2, v3;
  integer error = 0;
  static integer c[3] = 3, 4, 2;
  while ( !new_value ( ));
  v1 = read_value ( );
  if ( v1 != c[1] ) then error = 1;
  while ( !new_value ( ));
  v2 = read_value ( );
  if ( v2 != c[2] ) then error = 1;
  while ( !new_value ( ));
  v3 = read_value ( );
  if ( v3 != c[3] ) then error = 1;
  if ( error == 1 ) then return(0); else return (1);
}  
```
Implementation as a sequential digital system

Encoding:
- How many bits per input value?
- How many values in sequence?
- How do we know a new input value is entered?
- How do we represent the states of the system?

Behavior:
- Clock wire tells us when it's ok to look at inputs (i.e., they have settled after change)
- Sequential: sequence of values must be entered
- Sequential: remember if an error occurred
- Finite-state specification

Sequential example (cont'd):
- Abstract control
  - Finite-state diagram
    - States: 5 states
      - Represents part in execution of machine
      - Each state has outputs
      - Transitions: 6 from state to state, 3 self transitions, 1 global
        - Changes of state occur when clock says it's ok
        - Based on value of inputs
    - Inputs: reset, new, results of comparisons
    - Output: specific closed

Sequential example (cont'd):
- Data-path vs. control
  - Internal structure
    - Data-path
      - Storage for combination
      - Comparators
    - Control
      - Finite-state machine controller
      - Control for data-path
      - State changes are controlled by clock
      - Value
  - Output mux
    - Mux control
    - Clock
    - Value
    - New
    - Equal
    - Reset
    - Open/Closed

Sequential example (cont'd):
- Finite-state machine
  - Finite-state machine
    - Refine state diagram to include internal structure

Sequential example (cont'd):
- Finite-state machine
  - Generate state table (much like a truth-table)

Sequential example (cont'd):
- Encoding
  - Encode state table
    - State changes: S1, S2, S3, OPEN, or ERR
    - Needs at least 3 bits to encode: 000, 001, 010, 011, 100
    - And many as 5: 00001, 00010, 00100, 01000, 10000
    - Choose 4 bits: 0001, 0010, 0100, 1000
  - Output mux can be C1, C2, or C3
    - Needs 2 to 3 bits to encode
    - Choose 2: 00, 01, 10
  - Output open/closed can be open or closed
    - Needs 1 or 2 bits to encode
    - Choose 1: 0, 1
### Sequential example (cont'd): encoding

**Encode state table**
- **input:** can be: S1, S2, S3, OPEN or ERR
  - choose 4 bits: 0000, 0000, 0000, 0000
- **output max:** can be: C, C1, or C2
  - choose 3 bits: 000, 010, 100
- **output open/closed can be:** open or closed
  - choose 1 bit: 0

<table>
<thead>
<tr>
<th>next</th>
<th>next</th>
<th>next</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **good choice of encoding**
- **mux is identical to last 3 bits of state**
- **open/closed is identical to first bit of state**

### Sequential example (cont'd): controller implementation

**Implementation of the controller**
- **special circuit element called a register for remembering inputs when to be by click**
- **controller**

### Design hierarchy

- **digital system**
  - **data-path**
    - **control**
      - **register**
      - **shifting networks**

### Design methodology

- **Structure and function (behavior) of a design**
  - **HDL specification**
  - **Simulation**
  - **Synthesis**
  - **Verification:** Design Behaviors as Required
  - **Functional:** Input/output Register and Hardware
  - **Behavioral level:** Logic Level (Directed)
  - **Implementation level:** Timing, Waveform Behavior

### Combinational Logic Implementation

- **K-map method to map truth tables into minimized gate level descriptions**
- **Alternative implementation approaches:**
  1. Two-level logic, multi-level logic, logic implementation with multiplexers
  2. Programmable logic in the form of FAs, PLAs, and ROs
  3. Field-programmable logic in the form of devices like Xilinx
- **Combinational logic building blocks**
  1. Arithmetic and logical blocks, including adders/subtractors and other arithmetic functions (e.g., combinatorial multipliers)

### Sequential Logic Implementation

- **Models for representing sequential circuits**
  - Finite state machines and their state diagrams
  - Inputs/outputs
  - Mealy, Moore, and synchronous Mealy machines
- **Finite state machine design procedure**
  - Deriving state diagram
  - Deriving state transition table
  - Determining next state and output functions
  - Implementing combinational logic
- **Sequential logic building blocks**
  - Registers, Register files (with multiple read and write ports), Shifters, Counters, RAMs
  - Arbitrators
State Machine Implementation

- Partitioned State Machines
  - Ways to organize a complex monolithic state machine into simpler, interacting state machines based on functional partitioning
  - Time state approach offers one conceptual method
  - Much more relevant is what you likely did in your course project

- Issues of synchronization across independently clocked subsystems
  - Synchronization of signals
  - Four cycle handshake

Final Exam

- Exam Group 2
- May 14, 12:30-3:30 PM
- Room 10 Evans

- (Long) Design Specification in English for an “interesting” digital subsystem
  - Function described in terms of desired input/output behavior
  - You will need to be able to hand generate waveform diagrams to demonstrate that you understand the design specification
  - You will have to partition the subsystem into control and datapath
    - Design the control parts as one or more interacting Finite State Machines
    - State diagrams as well as Verilog for control
    - Design the datapath blocks
    - Behavioral Verilog mostly, but gate level hand-drawn schematics for some selected parts
  - You will have to revise the design to improve its performance

Final Exam

- The Exam is conceptual and DESIGN-skills oriented
- The Exam is not about obscure details of technologies like the Xilinx internal architecture or fast arithmetic, etc.
- The best way to study for the Exam is to review your course project and to reflect on the process you went through in designing and implementing it
- The Exam design problem won’t be a network switch—it will be some kind of digital system with control and a datapath that can be specified in a couple of pages of English text
- You will need to write a lot for the Exam! Bring multiple pencils, erasers, rulers, AND AT LEAST TWO BLUE BOOKS!!! You won’t need a computer or a calculator!
- Open course textbook and open course notes. They probably won’t help you much ;-)