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
  - Datapath + control = digital systems
- Hardware system design *methodology*
  - Hardware description languages: Verilog
  - 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 levels 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 description into 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** ← *Simulation & Chipscope*
- **Finite state behavior**
- **Register-transfer behavior** ← *Verilog Structural & Behaviorial Descriptions*
- Concurrent abstract specifications

Digital System Design

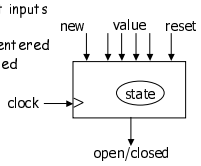
- Door combination lock:
  - punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - inputs: sequence of input values, reset
  - outputs: door open/close
  - memory: must remember combination
    or always have it available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

Implementation in software

```
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 (v2 != c[3]) then error = 1;

    if (error == 1) then return(0); else return (1);
}
```

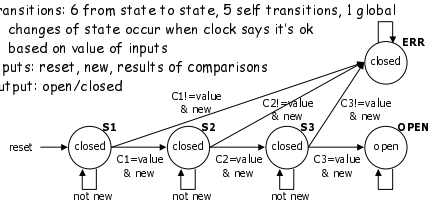
Implementation as a sequential digital system

- 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



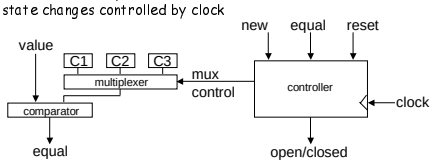
Sequential example (cont'd): abstract control

- Finite-state diagram
  - States: 5 states
    - represent point in execution of machine
    - each state has outputs
  - Transitions: 6 from state to state, 5 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: open/closed



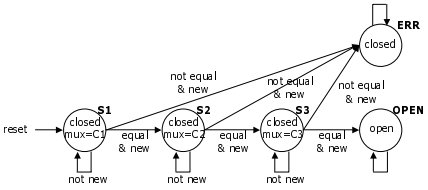
Sequential example (cont'd): data-path vs. control

- Internal structure
  - data-path
    - storage for combination
    - comparators
  - control
    - finite-state machine controller
    - control for data-path
    - state changes controlled by clock



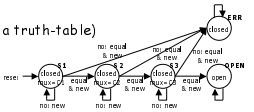
Sequential example (cont'd): finite-state machine

- Finite-state machine
  - refine state diagram to include internal structure



Sequential example (cont'd): finite-state machine

- Finite-state machine
  - generate state table (much like a truth-table)



reset	new	equal	state	next state	mux	open/closed
1	-	-	-	S1	C1	closed
0	0	-	S1	S1	C1	closed
0	1	0	S1	ERR	-	closed
0	1	1	S1	S2	C2	closed
0	0	-	S2	S2	C2	closed
0	1	0	S2	ERR	-	closed
0	1	1	S2	S3	C3	closed
0	0	-	S3	S3	C3	closed
0	1	0	S3	ERR	-	closed
0	1	1	S3	OPEN	-	open
0	-	-	OPEN	OPEN	-	open
0	-	-	ERR	ERR	-	closed

Sequential example (cont'd): encoding

- Encode state table
  - state can be: S1, S2, S3, OPEN, or ERR
    - needs at least 3 bits to encode: 000, 001, 010, 011, 100
    - and as many as 5: 00001, 00010, 00100, 01000, 10000
    - choose 4 bits: 0001, 0010, 0100, 1000, 0000
  - output mux can be: C1, C2, or C3
    - needs 2 to 3 bits to encode
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - needs 1 or 2 bits to encode
    - choose 1 bits: 1, 0

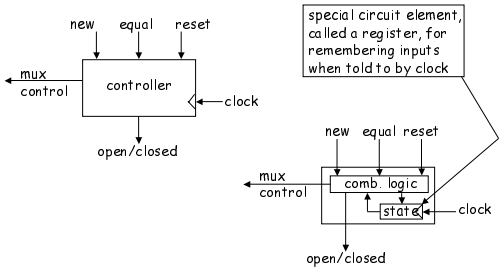
Sequential example (cont'd): encoding

- Encode state table
  - state can be: S1, S2, S3, OPEN, or ERR
    - choose 4 bits: 0001, 0010, 0100, 1000, 0000
  - output mux can be: C1, C2, or C3
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - choose 1 bits: 1, 0

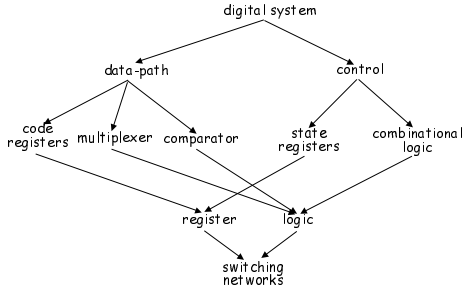
reset	new	equal	state	next state	mux	open/closed	
1	-	-	-	0001	001	0	
0	0	-	0001	0001	001	0	
0	1	0	0001	0000	-	0	good choice of encoding!
0	1	1	0001	0010	010	0	
0	0	-	0010	0010	010	0	mux is identical to last 3 bits of state
0	1	0	0010	0000	-	0	
0	1	1	0010	0100	100	0	
0	0	-	0100	0100	100	0	open/closed is identical to first bit of state
0	1	0	0100	0000	-	0	
0	1	1	0100	1000	-	1	
0	-	-	1000	1000	-	1	
0	-	-	0000	0000	-	0	

Sequential example (cont'd): controller implementation

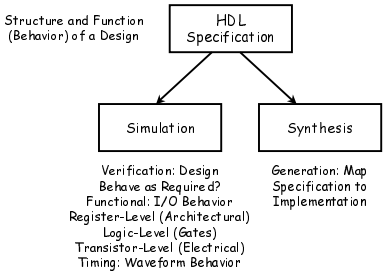
- Implementation of the controller



Design hierarchy



Design methodology



Combinational Logic Implementation

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

Sequential Logic Implementation

- Models for representing sequential circuits
  - Abstraction of sequential elements
  - 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 single 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

Final Exam

- (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 part 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 this 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 ;-)