Welcome to CS 150: Components and Design Techniques for Digital Systems

- Course staff
  - Instructor: [Instructor Name]
  - Teaching Assistant: [Teaching Assistant Name]

- Course web: in.st.eecs.berkeley.edu/~cs150

This week:

- Contrast with software design
- Hardware system design
- Hardware system building
- Language of logic design
- Hardware system design methodology
- Language of logic design
- Hardware system design methodology
- Concept of state in digital systems
- Language of logic design
- Hardware system design methodology
- Language of logic design
- Hardware system design methodology
- Language of logic design
- Hardware system design methodology

- Devices that sense/control wires carrying digital values (physical quantity interpreted as "0" or "1")
- Primitive digital hardware devices

What is digital hardware?

- Devices that sense/control wires carrying digital values (physical quantity interpreted as "0" or "1")
  - Logic (voltage): a voltage of 0 < V < 2V is a 0.
  - Pair of wires where "0" is distinguished by which has higher voltage (differential)
  - Magnetic orientation signifies "0" or "1"

- Primitive digital hardware devices
  - Logic (voltage): a voltage of 0 < V < 2V is a 0.
  - Pair of wires where "0" is distinguished by which has higher voltage (differential)
  - Magnetic orientation signifies "0" or "1"

What is logic design?

- Design? (design problem spec, solve it with available components)
- While meeting criteria for size, cost, power, beauty, elegance etc.

Why are we here?

- Implementation basis for modern computing devices
- Constructing large systems from small components
- Another view of a computer: controller + datapath
- Inherent parallelism in hardware
- Parallel computations beyond logic
- Counterpoint to software design
  - Furthering our understanding of computers

What is the current state of digital design?

- Changes in industrial practice
  - Larger designs
  - Shorter time to market
  - Cheaper products

- Scale
  - Prerogative to 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 tools
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 similarities and understanding of the advantages/disadvantages as compared to a software implementation.

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

Computation: abstract vs. implementation

- Computation as a mental exercise (paper, programs)
- vs. implementation with physical devices using voltages to represent logical values

- Basic units of computation:
  - Representation: "0", "1" on a wire
  - Assignment: \( x = y \)
  - Data operators: \( x + y \)
  - Control:
    - Sequential statements: \( A; B; C \)
    - Conditionals: if \( x = 1 \) then \( y \)
    - Loops: for \( i = 1; i < 10; i++ \)
    - Procedures: \( x = proc(C, x) \)

- Study how these are implemented in hardware and composed into computational structures.

Switches: basic element of physical implementations

- Implementing a simple circuit (arrow shows action if wire changes to "1"):
  - Close switch (if \( A \) is "0" or asserted) and turn on light bulb (Z)
  - Open switch (if \( A \) is "0" or unasserted) and turn off light bulb (Z)

Switches (cont'd)

- Compose switches into more complex ones (Boolean functions):
  - AND
  - OR

Switching networks

- Switch settings
  - Determine whether conducting path exists to light the bulb
- To build larger computations
  - Use bulb (output of the network) to set other switches (inputs to another network)
- Interconnect switching networks
  - Construct larger switching networks, i.e., connect outputs of one network to the inputs of the next.

Transistor networks

- Modern digital systems designed in CMOS
  - MOS: Metal-Oxide on Semiconductor
  - C: for complementary; normal-use and normal-failure switches
  - MOS transistors act as voltage-controlled switches
- Similar, though easier to work with, than relays.
MOS transistors

Three terminals: drain, gate, and source

Switch acts as follows:
- If voltage at gate terminal is (some amount) higher/lower than source terminal, then conducting path established between drain and source terminals.

n-channel
- Open when voltage at G is low
- Closed when voltage at G is low
- Open when: $\text{voltage}(G) > \text{voltage}(S) + \varepsilon$

p-channel
- Closed when voltage at G is low
- Opens when: $\text{voltage}(G) < \text{voltage}(S) - \varepsilon$

MOS networks

What is the relationship between x and y?

Two input networks

What is the relationship between x, y, and z?

Representation of digital designs

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

Mapping from physical world to binary world

Combination vs. sequential digital circuits

Simple model of a digital system is a unit with inputs and outputs:

$\text{inputs} \rightarrow \text{system} \rightarrow \text{outputs}$

Combination means "memory-less" - digital circuit is combinational if its output values only depend on its inputs.
Combinational logic symbols

- Common combinational logic systems have standard symbols called logic gates:
  - Buffer, NOT: \( A \rightarrow \bar{Z} \)
  - AND, NAND: \( A \land B \rightarrow Z \)
  - OR, NOR: \( A \lor B \rightarrow Z \)

Sequential logic

- Sequential systems exhibit behaviors (output values) that depend on current as well as previous inputs:
- All real circuits are sequential:
  - Outputs do not change instantaneously after an input change
  - Why not, and why is it then sequential?
- Fundamental abstraction of digital design is to reason (mostly) about steady-state behaviors:
  - Examine outputs only after sufficient time has elapsed for the system to make its required changes and settle down.

Synchronous sequential digital systems

- Combinational circuit outputs depend only on current inputs:
  - After sufficient time has elapsed
- Sequential circuits have memory:
  - Even after waiting for transient activity to finish
- Steady-state abstraction: most designers use it when constructing sequential circuits:
  - Memory of system is its state
  - Changes in system state are allowed at specific times controlled by an external periodic signal (the clock)
  - Clock period is elapsed time between state changes sufficient long so that system reaches steady-state before next state change of end of period

Example of combinational and sequential logic

- Combinational:
  - input A, B
  - wait for clock edge
  - observe C
  - wait for another clock edge
  - observe C again: will stay the same

- Sequential:
  - input A, B
  - wait for clock edge
  - observe C
  - wait for another clock edge
  - observe C again: may be different

An example

- Calendar subsystem: number of days in a month (to control watch display)
  - used in controlling the display of a wrist-watch LCD screen
  - inputs: month, leap year flag
  - outputs: number of days

Implementation in software

```java
integer number_of_days ( month, leap_year_flag) {
    switch (month) {
        case 1: return (31);
        case 2: if (leap_year_flag == 1) then return (29) else return (28);
        case 3: return (31);
        ...
        case 12: return (31);
        default: return (0);
    }
}
```
Implementation as a combinational digital system

Encoding:
- How many bits for each input/output?
- Binary number for month
- Four wires for 28, 29, 30, and 31

Behavior:
- Combinational
- Truth table

Combination example (cont’d)

Truth-table to logic to switches to gates
- \( d_{28} = 1 \) when month=0000 and leap=0
- \( d_{28} = m_8' \cdot m_4'' \cdot m_2'' \cdot m_1' \cdot \text{leap}' \)
- \( d_{31} = 1 \) when month=0001 or 0011 or ... month=1100
- \( d_{31} = (m_8' \cdot m_4'' \cdot m_2' \cdot m_1') + (m_8' \cdot m_4'' \cdot m_2'' \cdot m_1') + ... + (m_8' \cdot m_4'' \cdot m_2'' \cdot m_1') \)

Another example

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

Input: sequence of input values, reset
Output: 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:
1. How many bits per input value?
2. How many values in sequence?
3. How do we know a new input value is entered?
4. How do we represent the states of the system?

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

Sequential example (cont'd):

Finite-state diagram
1. States: 5 states
2. Each state has outputs
3. Transitions: 6 from state to state, 8 self transitions, 1 global
4. Changes of state occur when clock says it's ok.
5. Based on value of inputs.

Finite-state machine
1. Refine state diagram to include internal structure.

Finite-state machine
1. Generate state table (much like a truth-table).

Finite-state machine
1. Encode state table
2. Input: at least 3 bits to encode 0000, 0001, 0010, 0011, 0100.
3. Output: can be: C, 2, or C1.
4. Choice 2 bits to encode:
5. Choice 3 bits: 000, 001, 010, 011.
6. Output open/closed can be open or closed.
7. Choice 1 or 2 bits to encode:
8. Choice 1 bit: 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 setup mux open/closed
0 0 0001 001 0
0 1 0001 0000 1 0
good choice of encoding
0 1 0001 0000 1 0
0 0 1 0000 0000 1 0
0 1 0 0000 0000 1 0
output is identical to
last 3 bits of state
0 0 1 0001 0000 1 0
0 1 0 0001 0000 1 0
open/closed is
identical to first bit
0 0 1 0001 0000 1 0
0 1 0 0000 0000 1 0
```

Sequential example (cont'd): controller implementation

- Implementation of the controller

![Controller implementation diagram]

Design hierarchy

- What the entire course is about
  - Converting solutions to problems into combinational and sequential networks effectively
  - Organizing the design hierarchically
  - Using a modern set of design tools to handle large designs effectively
  - Taking advantage of optimization opportunities

Summary

- Now let's do it again
  - This time we'll take the rest of the semester!