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

Course staff
- Instructor: Po Yee
- Teaching Assistant: Steve Ting, Mark Yang, Nena Kuo, Mike Lin, Samy, Laura Todd, Jeff Tsoi
- Readers: Yen Hsu, Brian

Course web
www.cs.berkeley.edu/~po/ta/CS150S05.html

This week
- What is logic design?
- What is digital hardware
- What will we be doing in this class?
- Class administration, overview of course web, and logistics

Why are we here?
- Obvious reasons
  - Course is required (LE/CE), prerequisite for CS 152
  - Impractical to implement complex systems in microprocessors
  - Provide another view of what computers are
- More important reasons
  - Inherent parallelism in hardware
  - First exposure to parallel computers
  - Offers interesting counterpart to software design
  - Useful in generally furthering our understanding of computers

What will we learn in CS 150?
- Language of logic design
  - Boolean algebra, logic minimization, state, timing, CAD tools
- Concept of state in digital systems
  - A sequence of variables and program counters in software systems
- How to specify/simulate/compile our designs
  - Hardware description languages
  - Tools to simulate the workings of our designs
  - Logic compilers to synthesize the hardware blocks of our designs
  - Mapping to programmable hardware (code generators)
- Contrast with software design
  - Both map well to problems of physical devices
  - Both must be flawless; the price we pay for using discrete math

Applications of logic design
- Conventional computer design
  - CPUs, buses, peripherals
- Networking and communications
  - Fiber, modems, routers
- Embedded products
  - Cars, toys, appliances, entertainment devices
- Scientific equipment
  - Testing, sensing, reporting
- World of computing much bigger than just PCs

A quick history lesson
- 1850: George Boole invents Boolean algebra
  - Maps logical propositions to symbols
  - Forms the basis of logic statements using mathematics
- 1938: Claude Shannon links Boolean algebra to switches
  - His Master's thesis
- 1945: John von Neumann develops first stored program computer
  - Its switching elements are vacuum tubes (by analogy from relays)
- 1946: ENIAC — world's first all electronic computer
  - 18,000 vacuum tubes
  - Several hundred multiplications per minute
- 1947: Shockley, Britton, and Bardeen invent the transistor
  - Replaces vacuum tubes
  - Allows integration of multiple devices into a package
  - Gateway to modern electronics

What is logic design?
- What is design?
  - Given a specification of a problem, come up with a way of solving it choosing appropriately from a collection of available components
  - While meeting some criteria for size, cost, power, beauty, elegance, etc.
- What is logic design?
  - Determines the collection of digital logic components to perform a specified control or data manipulation and/or communication function and the interactions between them
  - Which logic components to choose?
  - There are many implementation technologies (e.g., off-the-shelf fixed function components, programmable devices, transistors on a chip, etc.)
  - The design may need to be optimized or transformed to meet design constraints
What is digital hardware?

- Collection of devices that sense and/or control wires carrying a digital value (i.e., a physical quantity interpreted as a "0" or "1")
  - e.g., digital logic where 0V or 5V is "0" and 12V or 20V is "1"
  - e.g., pair of transmission wires where a "0" or "1" is distinguished by which wire has a higher voltage (differential)
- Primitive digital hardware devices
  - Logic computation devices (sense and drive)
    - "new wires both "0", then make either be "1" (AND)
    - at least one of two wires "1" - make either be "1" (OR)
  - Memory devices (store)
    - store a value
    - recall a value previously stored

What is happening now in digital design?

- Big change in how industry does hardware design
  - Larger and larger designs
  - Shorter and shorter time to market
  - Cheaper and cheaper products
- Scale
  - Frequent 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

- Understanding the basics of logic design (concepts)
- Understanding sound design methodologies (concepts)
- Modern specification methods (concepts)
- Familiarity with a full set of CAD tools (skills)
- Appreciation for the 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 problem descriptions into or implementations with programmable logic devices after validation via simulation and understanding of the advantages/disadvantages as compared to a software implementation.

Computation: abstract vs. implementation

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

Basic units of computation

- representation: "0", "1" or a wire set of wires (e.g., for binary integers)
- assignment: \( x = y \)
- data operations: \( x + y \)
- control: sequential statements: if \( x = 1 \) then \( y \)
- loops: \( x^{(k)} \) for \( (k = 1) \) to \( n \)
- procedures: \( \text{proc}(x, y, z) \)

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")

Switches (cont'd)

- Compose switches into more complex ones (Boolean functions):
Switching networks

- **Switch settings**
  - Determine whether or not a conducting path exists to light the light bulb.

- **To build larger computations**
  - Use light bulbs (output of the network) to set other switches (inputs to another network).

- **Connect together switching networks**
  - Construct larger switching networks, e.g., there is a way to connect outputs of one network to the inputs of the next.

 Relay networks

- **A simple way to convert between conducting paths and switch settings is to use (electro-mechanical) relays.**
- **What is a relay?**
  - When a current flows through coil, armature moves and contacts are turned on.
  - When no current flows, the spring forces the contacts back to their original position.

Transistor networks

- **Relays aren’t used much anymore.**
  - Some traffic lights are still electro-mechanical.

- **Modern digital systems are designed in CMOS technology.**
  - MOS stands for Metal-Oxide-Semiconductor.
  - CMOS is further divided into N-type and P-type devices.

- **MOS transistors act as voltage-controlled switches.**
  - Similar to relays, easier to work with.

MOS transistors

- **MOS transistors have three terminals: drain, gate, and source.**
  - They act as switches as follows:
    - if voltage of gate terminal is (some amount) higher/lower than source terminal, then a conducting path is established between drain and source terminals:

    ![MOS transistor diagram]

Two input networks

- **What is the relationship between X, Y, and Z?**
  - For the X-Y input:
    - If X = 3V and Y = 0V, Z = 3V.
    - If X = 3V and Y = 3V, Z = 0V.
  - For the X-Y-Z input:
    - If X = 3V, Y = 3V, and Z = 3V, Z remains 3V.
    - If X = 3V, Y = 3V, and Z = 0V, Z remains 0V.
Speed of MOS networks

- What influences the speed of MOS networks?
  - charging and discharging of voltages on wires and gates of transistors

Digital vs. analog

- It is convenient to think of digital systems as having only discrete, digital, input/output values
- In reality, real electronic components exhibit continuous, analog, behavior
- Why do we make this abstraction?
  - 1
  - 1
- Why does it work?
  - 1

Mapping from physical world to binary world

<table>
<thead>
<tr>
<th>Technology</th>
<th>State 0</th>
<th>State 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relay logic</td>
<td>Circuit Open</td>
<td>Circuit Closed</td>
</tr>
<tr>
<td>CMOS logic</td>
<td>0.0-1.0 volts</td>
<td>2.0-3.0 volts</td>
</tr>
<tr>
<td>Transistor transfer logic (TTL)</td>
<td>0.0-0.6 volts</td>
<td>2.0-3.0 volts</td>
</tr>
<tr>
<td>Fiber Optics</td>
<td>Light off</td>
<td>Light on</td>
</tr>
<tr>
<td>System RAM</td>
<td>Discharged capacitor</td>
<td>Charged capacitor</td>
</tr>
<tr>
<td>Nonvolatile memory (amenable)</td>
<td>Trapped electrons</td>
<td>No trapped electrons</td>
</tr>
<tr>
<td>Programmable ROM</td>
<td>Fuse blown</td>
<td>Fuse intact</td>
</tr>
<tr>
<td>Bubble memory</td>
<td>No magnetic bubble</td>
<td>Bubble present</td>
</tr>
<tr>
<td>Magnetic disk</td>
<td>No flux reversal</td>
<td>Flux reversal</td>
</tr>
<tr>
<td>Compact disk</td>
<td>No pin</td>
<td>Pin</td>
</tr>
</tbody>
</table>

Combinational vs. sequential digital circuits

- A simple model of a digital system is a unit with inputs and outputs:

  ![Inputs](system) ![Outputs](system)

- Combinational means 'memory-less'
  - a digital circuit is combinational if its output values only depend on its input values

Combinational logic symbols

- Common combinational logic systems have standard symbols called logic gates:

  ![Buffer, NOT](logic)  ![AND, NAND](logic)
  ![OR, NOR](logic)

  easy to implement with CMOS transistors (the switches we have available at work)
**Sequential logic**

- **Sequential systems**
  - Exhibit behaviors (output values) that depend not only on the current input values, but also on previous input values.

- **In reality, all real circuits are sequential**
  - The outputs do not change instantaneously after an input change.
  - Why not, and why is it then sequential?

- **A fundamental abstraction of digital design is to reason (mostly) about steady-state behaviors**
  - Look at outputs only after sufficient time has elapsed for the system to make its required changes and settle down.

**Synchronous sequential digital systems**

- **Outputs of a combinational circuit depend only on current inputs**
  - After sufficient time has elapsed.

- **Sequential circuits have memory**
  - Even after waiting for the transient activity to finish.

- **The steady-state abstraction is so useful that most designers use a form of it when constructing sequential circuits**
  - Memory of a system is represented as its state.
  - Changes in system state are allowed to occur at specific times controlled by an external periodic clock.
  - Clock periods are the times that elapse between state changes. It must be sufficiently long so that the system reaches the steady state before the next state change at the end of the 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
  - **A**
  - **B**
  - **C**
  - **Clock**

- **Sequential:**
  - Input A, B
  - Wait for clock edge
  - Observe C
  - Wait for another clock edge
  - Observe C again may be different

---

**Abstractions**

- Some we've seen already:
  - Digital interpretation of logical values
  - Transistors as switches
  - Switches as logic gates
  - Use of a clock to realize a synchronous sequential circuit

- Some others we will see:
  - Truth tables and boolean algebra to represent combinational logic
  - Encoding of signals with more than two logical values into binary form
  - State diagrams to represent sequential logic
  - Hardware description languages to represent digital logic
  - Waveforms to represent temporal behavior

---

**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 ) return (29);  
                else return (28);  
        case 3: return (31);  
        ...  
        case 12: return (31);  
        default: return (0);  
    }  
}
```

---

**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

---

**Case 12: return (31);**

- Case 1: return (31);
- Case 2: if (leap_year_flag == 1) return (29);
- Case 3: return (31);
- ...
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

Combinational example (cont'd)

Truth-table to logic to switches to gates
- d28 = m8'm4'm2'm1'·leap'
- d29 = m8'm4'm2'm1'·leap
- d30 = (m8'm4'm2'm1') · (m8'm4'm2'm1') · (m8'm4'm2'm1') · (m8'm4'm2'm1')
- d31 = (m8'm4'm2'm1') · (m8'm4'm2'm1') · (m8'm4'm2'm1') · (m8'm4'm2'm1')
- d31 = cot we simplify more

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[0] ) then error = 1;
    }
    while ( !new_value( ) ) {
        v2 = read_value( );
        if ( v2 == c[1] ) then error = 1;
    }
    if ( error == 1 ) then return(0); also return (1);
}

Another example

Door combination lock:
- Each of 3 values is sequence, and the door opens if there isn't 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 combinatorially or always have it available as an input

Implementation in software
Implementation as a sequential digital system

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

Behavior:
- Clicks tell us when its 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 specific actions

Sequential example (cont'd): abstract control

Finite-state diagram
- States: 5 states
- Represent part in execution of machine
- Each state has outputs
- Transitions: 6 from state to state. 5 self transitions. Logical changes of state occur when clock says its ok
- Based on value of inputs
- Inputs: reset, new results from comparator
- Outputs: specific closed

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

Internal structure
- Data-path
- Storage for comparators
- 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
- Finite state diagram to include internal structure

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

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

Sequential example (cont’d): encoding

Encode state table
- States can be S1, S2, S3, OPEN, or ERR
- Needs at least 3 bits to encode: 000, 001, 010, 011, 100
- And as many as 5.0000, 00000, 00000, 00000, 00000
- Choose 4 bits: 0000, 0000, 0000, 0000, 0000
- Output can be C, S2, or S3
- Mode 2 or 3 bits to encode
- Mode 1 or 2 bits to encode
- Encode 3 bits: 000, 010, 000
- Output open/load can be: open, load
- Mode 1 or 2 bits to encode
- Choose bits: 0
**Sequential example (cont'd): encoding**

- **Encode state table**
  - State can be S1, S2, S3, OPEN, or ISR
  - Choose 4 bits: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111
  - Output max can be C1, C2, or C3
  - Choose 3 bits: 000, 001, 010, 011, 100, 101, 110, 111

<table>
<thead>
<tr>
<th>Next state</th>
<th>Input state</th>
<th>Output max</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
<td>0 0 1 0</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
<td>0 1 1 0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 1 1 0</td>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
<td>1 0 1 0</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1 0 1 0</td>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
<td>1 1 1 0</td>
<td>1 1 1 0</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 0</td>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

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

- **Implementation of the controller**
  - Special circuit element called register for memorizing inputs when hit by clock

![Controller Implementation Diagram]

**Design hierarchy**

- System
  - Digital
    - Data path
    - Logic
    - Control
- Control
  - Registers
  - Multiplier
  - Comparator
  - State
- Logic
  - Switching networks
  - Combinational logic

**Summary**

- That was what the entire course is about
  - Converting solutions to problems into combinatorial and sequential networks effectively organizing the design hierarchically
  - Designing with a modern set of design tools that let us handle large designs effectively
  - Taking advantage of optimization opportunities

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