CSE370: Introduction to Digital Design

- Course staff
  - Gaetano Borriello, Steven Balensiefer, Waylon Brunette
- Course web
  - currently: www.cs.washington.edu/education/courses/370/03au/
  - soon to be: www.cs.washington.edu/370/
- Today
  - Class administration, overview of course web, and logistics
- This week
  - What is logic design?
  - What is digital hardware?
  - What will we be doing in this class?

Why are we here?

- Obvious reasons
  - this course is part of the CS/CompE requirements
  - it is the implementation basis for all modern computing devices
    - building large things from small components
    - provide a model of how a computer works
- More important reasons
  - the inherent parallelism in hardware is often our first exposure to parallel computation
  - it offers an interesting counterpoint to software design and is therefore useful in furthering our understanding of computation, in general
What will we learn in CSE370?

- The language of logic design
  - Boolean algebra, logic minimization, state, timing, CAD tools
- The concept of state in digital systems
  - analogous to variables and program counters in software systems
- How to specify/simulate/compile/realize our designs
  - hardware description languages
  - tools to simulate the workings of our designs
  - logic compilers to synthesize the hardware blocks of our designs
  - mapping onto programmable hardware
- Contrast with software design
  - sequential and parallel implementations
  - specify algorithm as well as computing/storage resources it will use

Applications of logic design

- Conventional computer design
  - CPUs, busses, peripherals
- Networking and communications
  - phones, modems, routers
- Embedded products
  - in cars, toys, appliances, entertainment devices
- Scientific equipment
  - testing, sensing, reporting
- The world of computing is much much bigger than just PCs!
A quick history lesson

- 1850: George Boole invents Boolean algebra
  - maps logical propositions to symbols
  - permits manipulation of logic statements using mathematics
- 1938: Claude Shannon links Boolean algebra to switches
  - his Masters' thesis
- 1945: John von Neumann develops the first stored program computer
  - its switching elements are vacuum tubes (a big advance from relays)
- 1946: ENIAC . . . The world's first completely electronic computer
  - 18,000 vacuum tubes
  - several hundred multiplications per minute
- 1947: Shockley, Brittain, and Bardeen invent the transistor
  - replaces vacuum tubes
  - enable integration of multiple devices into one 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?
  - determining the collection of digital logic components to perform a specified control and/or data manipulation and/or communication function and the interconnections 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 and/or transformed to meet design constraints
What is digital hardware?

- Collection of devices that sense and/or control wires that carry a digital value (i.e., a physical quantity that can be interpreted as a “0” or “1”)
  - example: digital logic where voltage < 0.8v is a “0” and > 2.0v is a “1”
  - example: pair of transmission wires where a “0” or “1” is distinguished by which wire has a higher voltage (differential)
  - example: orientation of magnetization signifies a “0” or a “1”
- Primitive digital hardware devices
  - logic computation devices (sense and drive)
    - are two wires both “1” - make another be “1” (AND)
    - is at least one of two wires “1” - make another be “1” (OR)
    - is a wire “1” - then make another be “0” (NOT)
  - memory devices (store)
    - store a value
    - recall a previously stored value

What is happening now in digital design?

- Important trends in how industry does hardware design
  - larger and larger designs
  - shorter and shorter time to market
  - cheaper and 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
  - simulate and verify before you build
CSE 370: 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)
- Realize digital designs in an implementation technology (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 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

Computation: abstract vs. implementation

- Up to now, computation has been a mental exercise (paper,
  programs)
- This class is about physically implementing computation using
  physical devices that use voltages to represent logical values
- Basic units of computation are:
  - representation: "0", "1" on a wire
  - assignment: \( x = y \)
  - data operations: \( x + y - 5 \)
  - control:
    - sequential statements: \( A; B; C \)
    - conditionals: \( \text{if } x == 1 \text{ then } y \)
    - loops: \( \text{for } (i = 1; i == 10, i++) \)
    - procedures: \( A; \text{proc}(...) ; B \)
- We will study how each of 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”):

  ![Diagram of a simple circuit](image)

  close switch (if A is "1" or asserted)
  and turn on light bulb (Z)

  open switch (if A is "0" or unasserted)
  and turn off light bulb (Z)

  \[ Z = A \]

Switches (cont’d)

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

  ![Diagram of AND and OR circuits](image)

  Z = A \text{ and } B

  Z = A \text{ or } B
Switching networks

- **Switch settings**
  - determine whether or not a conducting path exists to light the light bulb
- **To build larger computations**
  - use a light bulb (output of the network) to set other switches (inputs to another network).
- **Connect together switching networks**
  - to construct larger switching networks, i.e., 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?**

  - conducting path composed of switches closes circuit
  - current flowing through coil magnetizes core and causes normally closed (nc) contact to be pulled open
  - when no current flows, the spring of the contact returns it to its normal position

  What determines the switching speed of a relay network?
Transistor networks

- Relays aren’t used much anymore
  - some traffic light controllers are still electro-mechanical
- Modern digital systems are designed in CMOS technology
  - MOS stands for Metal-Oxide on Semiconductor
  - C is for complementary because there are both normally-open and normally-closed switches
- MOS transistors act as voltage-controlled switches
  - similar, though easier to work with than relays.

MOS transistors

- MOS transistors have three terminals: drain, gate, and source
  - they act as switches in the following way:
    if the voltage on the gate terminal is (some amount) higher/lower than the source terminal then a conducting path will be established between the drain and source terminals

\[
\begin{align*}
\text{n-channel} & : & \text{open when voltage at G is low} \\
& & \text{closes when:} \\
& & \text{voltage(G) > voltage (S) + } \varepsilon \\
\end{align*}
\]

\[
\begin{align*}
\text{p-channel} & : & \text{closed when voltage at G is low} \\
& & \text{opens when:} \\
& & \text{voltage(G) < voltage (S) – } \varepsilon \\
\end{align*}
\]
MOS networks

What is the relationship between \( x \) and \( y \)?

\[
\begin{array}{c|c}
 x & y \\
 \hline
 0 \text{ volts} & \\
 3 \text{ volts} & \\
\end{array}
\]

Two input networks

What is the relationship between \( x \), \( y \) and \( z \)?

\[
\begin{array}{c|c|c|c}
 x & y & z_1 & z_2 \\
 \hline
 0 \text{ volts} & 0 \text{ volts} & \\
 0 \text{ volts} & 3 \text{ volts} & \\
 3 \text{ volts} & 0 \text{ volts} & \\
 3 \text{ volts} & 3 \text{ volts} & \\
\end{array}
\]
Speed of MOS networks

- What influences the speed of CMOS networks?
  - charging and discharging of voltages on wires and gates of transistors
- Capacitors hold charge
  - capacitance is at gates of transistors and wire material
- Resistors slow movement of electrons
  - resistance mostly due to transistors

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 vs. analog

- 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 the digital abstraction anyway?
- switches operate this way
- easier to think about a small number of discrete values

Why does it work?
- does not propagate small errors in values
- always resets to 0 or 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 transistor logic (TTL)</td>
<td>0.0-0.8 volts</td>
<td>2.0-5.0 volts</td>
</tr>
<tr>
<td>Fiber Optics</td>
<td>Light off</td>
<td>Light on</td>
</tr>
<tr>
<td>Dynamic RAM</td>
<td>Discharged capacitor</td>
<td>Charged capacitor</td>
</tr>
<tr>
<td>Nonvolatile memory (erasable)</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 disc</td>
<td>No pit</td>
<td>Pit</td>
</tr>
</tbody>
</table>
Combinational vs. sequential digital circuits

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

![Combinational circuit diagram]

- 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**
    
    ![Buffer NOT diagram]
  
  - **AND, NAND**
    
    ![AND NAND diagram]

  - **OR, NOR**
    
    ![OR NOR diagram]

- easy to implement with CMOS transistors (the switches we have available and use most)
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
  - because 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 the 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:
  - the memory of a system is represented as its state
  - changes in system state are only allowed to occur at specific times controlled by an external periodic clock
  - the clock period is the time that elapses between state changes it must be sufficiently long so that the system reaches a 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

- **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 analog 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
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

```c
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 specification

```
+-----+-----+-----+-----+-----+-----+-----+
| month | leap | d28 | d29 | d30 | d31 |
+-----+-----+-----+-----+-----+-----+
| 0000 |   -  |   - |   - |   - |   - |
| 0001 |   -  |  0  |  0  |  0  |  1  |
| 0010 |   0  |  1  |  0  |  0  |  0  |
| 0010 |   1  |  0  |  1  |  0  |  0  |
| 0011 |   -  |  0  |  0  |  0  |  1  |
| 0100 |   -  |  0  |  0  |  1  |  0  |
| 0101 |   -  |  0  |  0  |  0  |  1  |
| 0110 |   -  |  0  |  0  |  1  |  0  |
| 0111 |   -  |  0  |  0  |  0  |  0  |
| 1000 |   -  |  0  |  0  |  0  |  1  |
| 1001 |   -  |  0  |  0  |  1  |  0  |
| 1010 |   -  |  0  |  0  |  0  |  1  |
| 1011 |   -  |  0  |  0  |  1  |  0  |
| 1100 |   -  |  0  |  0  |  0  |  1  |
| 1101 |   -  |   - |   - |   - |   - |
| 1111 |   -  |   - |   - |   - |   - |
```

Combinational example (cont’d)

- Truth-table to logic to switches to gates
  - \( d_{28} = 1 \) when \( \text{month}=0010 \) and \( \text{leap}=0 \)
  - \( d_{28} = m_8\cdot m_4\cdot m_2\cdot m_1\cdot \text{leap}' \)
  - \( d_{31} = 1 \) when \( \text{month}=0001 \) or \( \text{month}=0011 \) or \( \ldots \) \( \text{month}=1100 \)
  - \( d_{31} = (m_8\cdot m_4\cdot m_2\cdot m_1) + (m_8\cdot m_4\cdot m_2\cdot m_1) + \ldots + (m_8\cdot m_4\cdot m_2\cdot m_1') \)
  - \( d_{31} = \) can we simplify more?

```
+-----+-----+-----+-----+-----+-----+-----+
| month | leap | d28 | d29 | d30 | d31 |
+-----+-----+-----+-----+-----+-----+
| 0001 |   -  |  0  |  0  |  0  |  1  |
| 0010 |   0  |  1  |  0  |  0  |  0  |
| 0010 |   1  |  0  |  1  |  0  |  0  |
| 0011 |   -  |  0  |  0  |  0  |  1  |
| 0100 |   -  |  0  |  0  |  1  |  0  |
| 0101 |   -  |  0  |  0  |  0  |  1  |
| 0110 |   -  |  0  |  0  |  1  |  0  |
| 0111 |   -  |  0  |  0  |  0  |  0  |
| 1000 |   -  |  0  |  0  |  0  |  1  |
| 1001 |   -  |  0  |  0  |  1  |  0  |
| 1010 |   -  |  0  |  0  |  0  |  1  |
| 1011 |   -  |  0  |  0  |  1  |  0  |
| 1100 |   -  |  0  |  0  |  0  |  1  |
| 1101 |   -  |   - |   - |   - |   - |
| 1111 |   -  |   - |   - |   - |   - |
```

symbol for \( \text{and} \)
symbol for \( \text{or} \)
symbol for \( \text{not} \)
Combinational example (cont’d)

- 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)
  = (m8'•m4•m1') + (m8•m4'•m1)
- d31 = (m8'•m4'•m2'•m1) + (m8'•m4'•m2•m1) + (m8'•m4•m2'•m1) + (m8'•m4•m2•m1) + (m8•m4'•m2'•m1') + (m8•m4'•m2•m1') + (m8•m4•m2'•m1')

Activity

- How much can we simplify d31?
- What if we started the months with 0 instead of 1? (i.e., January is 0000 and December is 1011)
Combinational example (cont’d)

- 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’) + (m8•m4’•m2’•m4’) + (m8•m4’•m2•m1’) + (m8•m4•m2’•m1’)

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
  - 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

```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

![State diagram]
## 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)

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S2</td>
<td>ERR</td>
<td>closed</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S3</td>
<td>C3</td>
<td>closed</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S3</td>
<td>ERR</td>
<td>closed</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S3</td>
<td>OPEN</td>
<td>open</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>OPEN</td>
<td>OPEN</td>
<td>open</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>ERR</td>
<td>ERR</td>
<td>closed</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>–</td>
<td>0001</td>
<td>0010</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>0000</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>–</td>
<td>0010</td>
<td>0100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>0000</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0010</td>
<td>0100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>–</td>
<td>0100</td>
<td>0100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>–</td>
<td>0100</td>
<td>0000</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0100</td>
<td>1000</td>
<td>–</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>1000</td>
<td>–</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>–</td>
<td>0000</td>
<td>0000</td>
<td>–</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
Activity

- Have lock always wait for 3 key presses exactly before making a decision

Sequential example (cont’d): controller implementation

- Implementation of the controller
Design hierarchy

Summary

- That was what the entire course is about
  - converting solutions to problems into combinational and sequential networks effectively organizing the design hierarchically
  - doing so with a modern set of design tools that lets us handle large designs effectively
  - taking advantage of optimization opportunities

- Now lets do it again
  - this time we'll take nine weeks instead of one