What will we learn in ECE 4242?

• Language of logic design
  – Boolean algebra, logic minimization, state, timing, CAD tools
• Concept of state in digital systems
  – Analogous to 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 onto programmable hardware (code generation)
• Contrast with software design
  – Both map well-posed problems to 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
  – Phones, 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
  – Permits manipulation of logic statements using mathematics
• 1938: Claude Shannon links Boolean algebra to switches
  – His Masters’ thesis
• 1945: John von Neumann develops first stored program computer
  – Its switching elements are vacuum tubes (a big advance from relays)
• 1946: ENIAC -- world’s first all 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 carrying a digital value (i.e., a physical quantity interpreted as a “0” or “1”)
  - e.g., digital logic where voltage < 0.8v is a “0” and > 2.0v is a “1”
  - e.g., pair of transmission wires where a “0” or “1” is distinguished by which wire has a higher voltage (differential)
  - e.g., orientation of magnetization signifies a “0” or a “1”
- Primitive digital hardware devices
  - Logic computation devices (sense and drive)
    - two wires both “1” - make another be “1” (AND)
    - at least one of two wires “1” - make another be “1” (OR)
    - a wire “1” - then make another be “0” (NOT)
  - 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
  - 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

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” on a wire
  - assignment: x = y
  - data operations: x + y – 5
  - control:
    - sequential statements: A; B; C
    - conditionals: if x == 1 then y
    - loops: for ( i = 1 ; i <= 10, i++)
    - procedures: A; proc(...); B;
- 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 “1” 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

\[ Z = A \text{ and } B \]

OR

\[ Z = A \text{ or } B \]

MOS transistors

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

n-channel
open when voltage at G is low
closed 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?

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

Two input networks

what is the relationship between x, y and z?

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

• What influences the speed of CMOS networks?
  – charging and discharging of voltages on wires and gates of transistors

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>

Representation of digital designs

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

Combinational vs. sequential digital circuits

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

  ![Combinational model](image)

  • 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
    \[
    \begin{array}{c}
    A \\
    \hline
    \end{array} \quad \rightarrow \quad Z
    \]
  - AND, NAND
    \[
    \begin{array}{c}
    A \\
    \hline
    B \\
    \hline
    \end{array} \quad \rightarrow \quad Z
    \]
  - OR, NOR
    \[
    \begin{array}{c}
    A \\
    \hline
    B \\
    \hline
    \end{array} \quad \rightarrow \quad Z
    \]

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 only allowed to occur at specific times controlled by an external periodic clock
  - 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

Implementation in software

```cpp
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);
    }
}
```

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

```

<table>
<thead>
<tr>
<th>month</th>
<th>leap</th>
<th>d28</th>
<th>d29</th>
<th>d30</th>
<th>d31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0110</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1110</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

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

Combinational example (cont’d)

- Truth-table to logic to switches to gates
  - \( d_{28} = 1 \) when month=0010 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 month=0011 or \( \ldots \) 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?

<table>
<thead>
<tr>
<th>month</th>
<th>leap</th>
<th>( d_{28} )</th>
<th>( d_{29} )</th>
<th>( d_{30} )</th>
<th>( d_{31} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>…</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1111</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

---

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):
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):
finite-state machine

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

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

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 told to by clock

Design hierarchy

Design hierarchy

system

control

data-path

code

registers

multiplexer

comparator

state

registers

combinational

logic

switching

networks

logic

register

 registers