Logic Design (Part 2)
Combinational Logic Circuits (Chapter 3)

Digital Logic Circuits
- We saw how we can build the simple logic gates using transistors
- Use these gates as building blocks to build more complex combinational circuits
  - Decoder: based on value of n-bit input control signal, select one of \(2^n\) outputs
  - Multiplexer: based on value of N-bit input control signal, select one of \(2^N\) inputs.
  - Adder: add two binary numbers
  - ...any boolean function

Digital Logic Design – Current Summary
- MOS transistors used as switches to implement logic functions.
  - n-type: connect to GND, turn on (with 1) to pull down to 0
  - p-type: connect to +2.9V, turn on (with 0) to pull up to 1
- Basic gates: NOT, NAND
- Logic functions are usually expressed with AND, OR, and NOT
- Review: a little theory behind combinational logic design and some basic combinational devices
  - DeMorgan’s Law
  - Combinational logic devices:
    - Decoder, Multiplexer, Adder, PLA
  - Boolean Algebra – review from CS 1311 (Discrete 1)

Boolean Algebra
- George Boole – Famous Mathematician/Logician
  - Boolean Algebra – branch of Algebra, where variables can only have values of true (1) or false (0)
  - Instead of +, -, \(\times\), \(/\), Boolean operators: AND(x), OR(+), NOT(!)
    - NOT is simply an Inverter
- With Boolean Algebra:
  - We create “functions” using boolean variables and operators
  - Any logical function can be expressed in terms of the three elementary operations: AND, OR and NOT
  - Boolean functions can be rearranged and sometimes simplified by applying algebraic identities
- Big idea – you can write a logical function as a boolean algebraic expression and then use various identities to rewrite that function in an equivalent (usually simpler) form.
Boolean Functions

- A function can be thought of as a mapping from inputs to outputs.
  - Think of a black box with n binary inputs and 1 binary output
- We can express the action of this function in terms of a truth table which says what the output should be for every input pattern.
  - This function implements a binary adder!

Truth table (describes behavior)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Completeness: Very Important Concept

- It can be shown that any truth table (i.e. any binary function of binary variables) can be reduced to combinations of the AND & NOT functions, or of the OR & NOT functions.
  - This result extends also to functions of more than two variables
  - Methodology: Karnaugh Maps
- In fact, it turns out to be convenient to use a basic set of three logic gates:
  - AND, OR & NOT or NAND, NOR & NOT
  - In fact, can implement all logic functions using just NAND!

Boolean Functions

- Function on boolean variables
  - F(x,y)
  - x, y are boolean variables (0 or 1 values)
  - F(x,y) is a boolean output
- If numbers are represented using binary, then all functions are boolean functions

Methodology: Karnaugh Maps

- In fact, it turns out to be convenient to use a basic set of three logic gates:
  - AND, OR & NOT or NAND, NOR & NOT
  - In fact, can implement all logic functions using just NAND!
Representation of Logic Functions

- A logic function can be represented as
  - a truth table
  - a logic expression
  - a logic circuit

**Example**

\[ f = a.(b.c + d) + \overline{a}.c = a.b.c + a.d + \overline{a}.c \]

Truth Table to Boolean Function

- Definition: literal is a boolean variable \( x \) or its complement \( x' \)
  - \( x' \) means \( x=0 \) in truth table
- Definition: minterm is product (AND) of literals where each variable/literal appears once in the term
- Disjunctive normal form (DNF): an OR of minterms
  - DNF gives a two level circuit implementation

Truth Table to DNF:

- Simple boolean function for output \( F \)
  - For each row where \( F=1 \), find minterm
  - Ex: if \( F=1 \) when \( x_1=0, x_2=1, x_3=1 \) then minterm= \((x_1' \times x_2 \times x_3)\)
  - Boolean function in DNF is sum (OR) of all minterms where \( F=1 \)

How to design combinational circuit

- Analyze the problem
  - Determine inputs and outputs (they must be binary)
- Determine boolean variables
  - inputs \( x_1, x_2, \ldots \)
  - Outputs \( y_1, y_2, \ldots \)
- Derive truth table
  - Value of each \( y_i \) for each combination of inputs \( x_1, x_2, \ldots \)
- For simple circuit, find DNF from truth table
- To find ‘optimal’ (minimum size) 2-level circuit, derive Karnaugh map and find terms

Combinational and Sequential Circuits

- A circuit is a collection of devices that are physically connected by wires
  - Combinational circuit
  - Sequential circuit
- In Combinational circuit the input determines output
- In sequential circuit, the input and the previous ‘state’ (previous values) determine output and next ‘state’
  - Need circuit to implement concept of storage
Combinational Devices

- Use basic gates to build more complex combinational logic functions
- Adder: add two binary numbers
- Decoder: enable one of many outputs
- Multiplexer: select one of many inputs

Problem

- No one will buy your new computer design unless it can do at least some math, say, like adding!
- How to build hardware for adding 2 binary numbers using what we have learnt so far?
- First look at the function performed by addition – we saw this last week
  - Bit by bit addition, right to left, propagate carry
  - Inputs: A, B and Carry-in
  - Output: sum bit and carry-out (to next bit position)

Binary Arithmetic: Half Adder

- Logical Function: \( \text{Half Adder} \), implement Carry Out:

  Position | A | B | Sum | C\text{out} |
  --- | --- | --- | --- | --- |
  0 | 0 | 0 | 0 | 0 |
  0 | 1 | 1 | 0 | 0 |
  1 | 0 | 1 | 0 | 0 |
  1 | 1 | 0 | 1 | 1 |

  \[ \text{Half Adder’s Logic Function:} \]
  \[ \text{SUM} = (A' \text{ AND } B) \text{ OR } (A \text{ AND } B') \]
  \[ C\text{out} = (A' \text{ AND } B') \text{ OR } (A \text{ AND } B) \text{ OR } (A \text{ AND } B')' \]

  \( A \) \( B \) \( \text{Sum} \) \( C\text{out} \)

  \[ A \rightarrow \text{HA} \rightarrow \text{Sum} \rightarrow \text{C\text{out}} \]

  8-bit incrementer (just adds 1)

Chaining Basic Components Together:

- Let’s create an incrementer
  - Input: A
  - Output: \( S = A+1 \)
  - Why? Recall how to create 2C number?
  - We “flip bits” then add 1

- Approach #1 (impractical)
  - Use PLA-like techniques to implement circuit
  - Problem: \( 2^8 \) or 256 rows, 8 output columns
  - In theory, possible; in practice, intractable
    - Imagine a 16-bit incrementer!

- Approach #2 (pragmatic)
  - Create a 1-bit incrementer circuit
  - Replicate it 16 times
  - We already have! A half adder can be used to just add 1
One-bit Incrementer

- Implement a single-column of an incrementer using a half adder

\[
\begin{array}{c}
00001011 \\
+00000000 \\
\hline
00001100
\end{array}
\]

This is the same operation as a half adder

We can call it a half adder or a 1-bit incrementer

N-bit Incrementer

- Chain N 1-bit incrementers together

\[
\begin{array}{c}
A_0 \\
\hline
S_0 \\
C_{in} \\
S_0 \\
C_{out}
\end{array}
\]

...but how do we start off the least-significant bit?

N-bit Incrementer, continued

- How do we handle the least-significant bit?

\[
\begin{array}{c}
\text{C}_{in} = 1 \\
\hline
\end{array}
\]

We “carry in” a 1

Addition: Full Adders

- There is a limit with the half adder
  - It can’t implement multiple-bit addition

\[
\begin{array}{c}
A \\
\hline
\text{Sum} \\
\text{C}_{out}
\end{array}
\]

- It works for “least significant bit,” but won’t work for the next

\[
\begin{array}{c}
A \\
\hline
\text{Sum} \\
\text{C}_{out}
\end{array}
\]

- We need an adder that has 3 inputs and 2 outputs
Truth Table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Carry In</th>
<th>Out</th>
<th>Carry Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Truth Table for Binary Addition

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Carry In</th>
<th>Out</th>
<th>Carry Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Full Adder

Carry OUT

SUM-OUT

1-bit Full Adder

✿ Add two bits and carry-in, produce one-bit sum and carry-out.
N-bit Adder

- Use the building block of the full-adder to build N-bit adder
  - Need to connect carry-out to carry-in of next significant bit

Four-bit Adder

Truth Table
How about a “subtracter”?

- Build a subtracter from our multi-bit adder
  - Calculate \( A - B = A + -B \)
  - Negate \( B \)
  - Recall \( -B = \text{NOT}(B) + 1 \)

**Approach #1**

```
+1
A 16
B 16
```

Now, let’s create an adder/subtracter

**Approach #2**

```
A 16
B 16
```

Can we put this together?

- In a CPU, we’d like to do BOTH addition and subtraction
  - Can we give the CPU the ability to choose between two pieces of hardware?
    - Yes!
      - We need another small piece of logic to do it: MUX

The Multiplexer (MUX) – “The selector”

- Selector/Chooser of signals – Imagine Switching Railroad Tracks
  - Multi-way switch
    - 2-to-1 Mux
    - 4-to-1 Mux

Problem: Selecting one of many

- You have \( m \) input signals and you want to use the logical value on one of them determined by a set control signals/wires – \( n \) control signals
  - Each student sends a signal (0 or 1)
  - I want to select Tim’s signal – so I can process his answer
    - Need to give Tim’s code of 10 to select his answer
In general, a MUX has $2^n$ data inputs, $n$ select (or control) lines, and 1 output. It behaves like a channel selector.

The Multiplexer (MUX)

- In general
  - $N$ select bits chooses from $2^N$ inputs
  - An incredibly useful building block

- Multi-bit muxes
  - Can switch an entire “bus” or group of signals
  - Switch $n$-bits with $n$ muxes with the same select bits

Adder/Subtractor - Approach #1

Adder/Subtractor - Approach #2
Another useful device – the Decoder

✧ You have an \( n \) bit binary number assigned as unique ID to each student. How do we select & physically connect to a specific student with ID \( y \)?
✧ In S/W, a “case”/switch statement:
  • One of the cases will be evaluated depending on value of ‘input’
✧ Scenario: 4 light bulbs, switch one of them ON depending on 2 bit input

Boolean function for decoder

✧ Need to select one of four:
  • 2 bits needed to encode the four outcomes
  • \( a_1a_0 \)
✧ 4 outputs – 1 associated with each signal
  • \( x_3x_2x_1x_0 \)
✧ What is the boolean function?
✧ When is each \( x_i \) set to 1:
  • \( x_0 = a_1',a_0' \) (NOT \( a_1 \) AND NOT \( a_0 \))
  • \( x_1 = a_1',a_0 \)
  • \( x_2 = a_1,a_0' \)
  • \( x_3 = a_1,a_0 \)

Truth table

<table>
<thead>
<tr>
<th>( a_1 )</th>
<th>( a_0 )</th>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Decoder

An \( n \) input decoder has \( 2^n \) outputs.

Output is 1 iff the binary value of the \( n \)-bit input is \( i \).

At any time, exactly one output is 1, all others are 0.

2-bit decoder
(4 input decoder)
Combinational vs. Sequential

Combinational Circuit
- always gives the same output for a given set of inputs
  - ex: adder always generates sum and carry, regardless of previous inputs

Sequential Circuit
- stores information
- output depends on stored information (state) plus input
  - so a given input might produce different outputs, depending on the stored information
- example: vending machine
  - Current total increases when you insert coins
  - output depends on previous state
- useful for building "memory" elements and "state machines"

Next . . Sequential Circuits

First we need to build a device that can store a bit
- Building memory follows

How to model sequential circuits/machines
- Finite state machine

How to build a sequential circuit?
- Limitations of sequential machines...more in Foundations course

Can we use a sequential circuit to "control" how computations take place in a processor?