Chapter 3
Digital Logic Structures

Transistor: Building Block of Computers
Microprocessors contain millions of transistors
• Intel Pentium 4 (2000): 48 million
• IBM PowerPC 750FX (2002): 38 million
• IBM/Apple PowerPC G5 (2003): 58 million

Logically, each transistor acts as a switch
Combined to implement logic functions
• AND, OR, NOT
Combined to build higher-level structures
• Adder, multiplexer, decoder, register, …
Combined to build processor
• LC-3

How do we represent data in a computer?
At the lowest level, a computer has electronic “plumbing”
• Operates by controlling the flow of electrons

Easy to recognize two conditions:
1. Presence of a voltage – we’ll call this state “1”
2. Absence of a voltage – we’ll call this state “0”

Computer use transistors as switches to manipulate bits
• Before transistors: tubes, electro-mechanical relays (pre 1950s)
• Mechanical adders (punch cards, gears) as far back as mid-1600s

Before describing transistors, we present an analogy…

A Transistor Analogy: Computing with Air
Use air pressure to encode values
• High pressure represents a “1” (blow)
• Low pressure represents a “0” (suck)

Valve can allow or disallow the flow of air
• Two types of valves
**Analogy Explained**

Pressure differential $\rightarrow$ electrical potential (voltage)
- Air molecules $\rightarrow$ electrons
- High pressure $\rightarrow$ high voltage
- Low pressure $\rightarrow$ low voltage

Air flow $\rightarrow$ electrical current
- Pipes $\rightarrow$ wires
- Air only flows from high to low pressure
- Electrons only flow from high to low voltage
- Flow only occurs when changing from 1 to 0 or 0 to 1

Valve $\rightarrow$ transistor
- The transistor: one of the century’s most important inventions

---

**Transistors as Switches**

Two types
- N-type
- P-type

Properties
- Solid state (no moving parts)
- Reliable (low failure rate)
- Small (90nm channel length)
- Fast (<0.1ns switch latency)

---

**MOS + FET**

MOS: three materials needed to make a transistor
- Metal (Al, W, Cu): conductor
- Oxide (SiO$_2$): insulator
- Semiconductor (doped Si): conducts under certain conditions

FET: field effect (the mechanism) transistor
- Voltage on gate: current flows source to drain (transistor on)
- No voltage on gate: no current (transistor off)

Recall, two types of MOSFET: n and p

---

**N-type MOS Transistor**

- When Gate has positive voltage, short circuit between #1 and #2 (switch closed)
- When Gate has zero voltage, open circuit between #1 and #2 (switch open)

Terminal #2 connected to GROUND (0V).
P-type MOS Transistor

P-type is complementary to n-type
- When Gate has positive voltage, open circuit between #1 and #2 (switch open)
- When Gate has zero voltage, short circuit between #1 and #2 (switch closed)

Terminal #1 connected to POWER (in this example, +2.9V)

Inverter (NOT Gate)

Inverter is an example of Complementary MOS (CMOS)
Uses both n-type and p-type MOS transistors
- p-type
  ➢ Attached to POWER (high voltage)
  ➢ Pulls output voltage UP when input is zero
- n-type
  ➢ Attached to GROUND (low voltage)
  ➢ Pulls output voltage DOWN when input is one

For all inputs, make sure that output is either connected to GROUND or to POWER, but not both! (why?)

NAND Gate (NOT-AND)

Note: Parallel structure on top, serial on bottom.
**Basic Gates**

From Now On... Gates
- Covered transistors mostly so that you know they exist
- Note: “Logic Gate” not related to “Gate” of transistors

Will study implementation in terms of gates
- Circuits that implement Boolean functions

More complicated gates from transistors possible
- XOR, Multiple-input AND-OR-Invert (AOI) gates
More than 2 Inputs?
AND/OR can take any number of inputs
• AND = 1 if all inputs are 1
• OR = 1 if any input is 1
• Similar for NAND/NOR

Implementation
• Multiple two-input gates or single CMOS circuit

Visual Shorthand for Multi-bit Gates
Use a cross-hatch mark to group wires
• Example: calculate the AND of a pair of 4-bit numbers
• A₃ is “high-order” or “most-significant” bit
• If “A” is 1000, then A₃ = 1, A₂ = 0, A₁ = 0, A₀ = 0

Shorthand for Inverting Signals
Invert a signal by adding either
• A before/after a gate
• A “bar” over letter

Logical Completeness
AND, OR, NOT can implement ANY truth table
1. AND combinations that yield a “1” in the truth table
2. OR the results of the AND gates
Logical Completeness via PLAs

Any truth table as a Programmable Logic Array (PLA)
- Traditionally a grid of AND and OR gates
- Configurable by removing wires

Single-output custom PLA (as on previous slide):
- One AND gate per row with “1” in output in truth table
- Maximum number of AND gates: $2^n$ for $n$ inputs
- One OR gate

Multiple-output custom PLA:
- Build multiple single-output PLAs
- Share AND gates “in common”
- One OR gate per output column in truth table

DeMorgan’s Law

Converting AND to OR (with some help from NOT)

Consider the following gate:

$$A \land B \quad A \lor B$$

To convert AND to OR (or vice versa), invert inputs and output.

$$A \land B \quad \overline{A} \lor \overline{B}$$

Why might this be useful?

Summary

MOS transistors are used as switches to implement logic functions
- n-type: connect to GROUND, turn on (with 1) to pull down to 0
- p-type: connect to POWER, turn on (with 0) to pull up to 1

Basic gates: NOT, NOR, NAND
- Logic functions are usually expressed with AND, OR, and NOT
- Universal: any truth table to simple gates (via a PLA)

DeMorgan’s Law
- Convert AND to OR (and vice versa) by inverting inputs/output

Next Time

Lecture
- Combinational Logic Circuits

Reading
- Chapter 3.3

Quiz
- Online (as always)

Upcoming
- HW2 due next Friday
Chapter 3
Digital Logic Structures

AND, OR, NOT Gates: What Good Are They?

Last time:
- Transistors and gates
- Can implement any logical function using gates (using PLAs)

Today:
- We’ll use gates to create some building blocks of a processor
- One goal: automate binary arithmetic from Chapter 2
- Continuing on our bottom-up journey

Next time:
- Storing bits (memory)
- Circuits with “state”

Incrementer
Let’s create a incrementer
- Input: A (as a 16-bit 2’s complement integer)
- Output: A+1 (also as a 16-bit 2’s complement integer)

Approach #1 (impractical):
- Use PLA-like techniques to implement circuit
- Problem: $2^{16}$ or 65536 rows, 16 output columns
- In theory, possible; in practice, intractable

Approach #2 (pragmatic):
- Create a 1-bit incrementer circuit
- Replicate it 16 times

One-bit Incrementer
Implement a single-column of an incrementer
Aside: XOR

<table>
<thead>
<tr>
<th>A</th>
<th>C_{in}</th>
<th>S</th>
<th>C_{out}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

N-bit Incrementer

Chain N 1-bit incrementers together

4-bit example

...but how do we correctly handle the least-significant bit?

N-bit Incrementer, continued

How do we handle the least-significant bit?

Adder

Conceptually similar to an incrementer
- Build a one-bit slice, replicate $n$ times

Add

00001011
+00110011
00111100

C_{in} = 1
No longer needed; implicitly encoded with C_{in}
**One-bit Adder**

Add two bits and carry-in produce one-bit sum and carry-out

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

**N-bit Adder**

Add two bits and carry-in produce one-bit sum and carry-out

**Aside: Efficient Adders**

Full disclosure:
- Our adder: Ripple-carry adder
- No one (sane) actually uses ripple-carry adders
- Why? way too slow
  - Latency proportional to \( n \)

We can do better
- Many ways to create adders with latency proportional to \( \log_2(n) \)
- In theory: constant latency (build a big PLA)
- In practice: too much hardware, too many high-degree gates
- “Constant factor” matters, too
- More on this topic in CSE371

**Subtractor**

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

**Approach#1:**

**Approach#2:**

Now, let's create an adder/subtractor
But First, The Multiplexer (MUX)

Selector/Chooser of signals
- Multi-way switch

2-to-1 Mux

4-to-1 Mux

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
- CarryIn
- A
- B
- CarryOut
- S

Subtractor
- CarryIn
- A
- B
- S

Adder/Subtractor
- A
- B
- 1
- S

Adder/Subtractor - Approach #2

Adder
- CarryIn
- A
- B
- CarryOut
- S

Subtractor
- CarryIn
- A
- B
- S

Adder/Subtractor
- A
- B
- 1
- S
Ok, So We Can Add and Subtract

Other arithmetic operations similar
  • Even floating point operations

We can calculate; but we can’t remember
  • Next time: storage and memory
  • After that: simple “state machines”
  • After that: a simple processor

Remember: Readings, Quizzes, and Homework
  • We’ll return Homework 1 on Wednesday
  • Homework 2 due Friday

Combinational vs. Sequential Logic

Combinational Circuit
  • Always gives the same output for a given set of inputs
    ➢ For example, adder always generates sum and carry, regardless of previous inputs

Sequential Circuit
  • Stores information
  • Output depends on stored information (state) plus input
    ➢ Given input might produce different outputs, depending on stored information
  • Example: ticket counter
    ➢ Advances when you push the button
    ➢ Output depends on previous state
  • Useful for building “memory” elements and “state machines”

Chapter 3
Digital Logic Structures

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin

Storage - Cross-Coupled Inverters
Cross-coupled inverters (INV) gates
  • Holds value $Q$ and $Q'$ ($Q'$ is the same as $\overline{Q}$)

  • Read: get value from either $Q$ or $Q'$

  Maintains its “state”, but how do we change the state?
  • Write: Option #1: put opposite values on $Q$ and $Q'$ simultaneously
    ➢ Requires “analog” overdriving of $Q$ and $Q'$
**Storage - NANDs**

Option #2: “Digital” alternative for changing state

Write: change Q to one

\[
\begin{align*}
S & \quad 0 \\
1 & \quad \quad 0 \\
\quad 1 & \quad \quad 1 \\
\quad 0 & \quad \quad 0 \\
\quad 1 & \quad \quad 1 \\
\end{align*}
\]

Maintains state even after S = 1

Write: change Q to zero

\[
\begin{align*}
R & \quad 0 \\
\quad 1 & \quad \quad 0 \\
\quad 1 & \quad \quad 1 \\
\quad 1 & \quad \quad 1 \\
\quad 0 & \quad \quad 0 \\
\quad 0 & \quad \quad 1 \\
\end{align*}
\]

Maintains state even after R = 1

---

**Storage - Cross-Coupled NANDs (R-S Latch)**

Write either a zero or one

- When S=1 and R=1, “quiescent” state; maintains value

\[
\begin{align*}
S & \quad 1 \\
\quad 0 & \quad \quad 0 \\
\quad 1 & \quad \quad 1 \\
\quad 0 & \quad \quad 0 \\
\quad 1 & \quad \quad 1 \\
\quad 0 & \quad \quad 0 \\
\end{align*}
\]

- When S=0 and R=1, state changes to one (“set”)
- When S=1 and R=0, state changes to zero (“reset” or “clear”)

**Gated D-Latch**

Add logic to an R-S latch

- Create a better interface

Two inputs: D (data) and WE (write enable)

- When \( WE = 1 \), latch is set to value of D
  - \( S = \text{NOT}(D), R = D \)
- When \( WE = 0 \), latch continues to hold previous value
  - \( S = R = 1 \)
- Does not allow S=0, R=0 case to occur

---

What happens with S=0 and R=0?

- Short answer: bad things
- Long answer: value stored will depend on timing on circuit

\[
\begin{align*}
S & \quad 0 \\
\quad 1 & \quad \quad 1 \\
\quad 0 & \quad \quad 0 \\
\end{align*}
\]

- Does S or R go to one first?
  - If they change at the same time?
  - Oscillation or meta-stability can result
- Let’s make sure this can never happen...

---

CSE 240

3-49

CSE 240

3-50

CSE 240

3-51

CSE 240

3-52
Register
A register stores a multi-bit value
• A collection of D-latches, controlled by a common WE
• When WE=1, n-bit value D is written to register

Aside: More on Representing Multi-bit Values
Number bits from right (0) to left (n-1)
• Just a convention -- could be left to right, but must be consistent
Use brackets to denote range:
D[l:r] denotes bit l to bit r, from left to right

May also see A<14:9>
• Especially in hardware block diagrams.

Let’s Try to Build a Counter
How quickly will this count?
• Timing dependent
Will it even work?
• Probably not
• D-latches are “transparent”
  ➢ Allows next input to immediately flow to output
  ➢ Outputs will never be “stable”

What’s Missing? The Clock
A clock controls when registers are “updated”
• Oscillating global signal with fixed period
• Typical clock frequencies today: a couple of gigahertz

• Corresponds to <1 nanosecond between one rise and the next
• Generated on-chip by special circuitry (for example, oscillating ring of inverters)
Let’s Try Again: a Counter

Solves half the problem
- Controls the rate of updates

Remaining problem
- When clock=1, same problem
- D-latches are still transparent

Example of Incorrect Operation

Set WE (Write Enable) to 1

D latches write new value: 011
Goal: 100
Example of Incorrect Operation
Incrementer calculates 1st bit

Example of Incorrect Operation
Incrementer calculates 2nd bit, 1st bit latched

Example of Incorrect Operation
Incrementer calculates 3rd bit, 2nd bit latched, 1st bit re-calculated

Example of Incorrect Operation
1st and 3rd bits latched value is: 101\(_2\) (not the desired 100\(_2\))
Correct Operation
Additional D-latches, WE is NOT(Clock) and Clock
Initial state: 010₂

Correct Operation
Clock switches to 1, 2nd latch WE = 1

Correct Operation
D latches write new value: 011₂
Goal: 100₂

Correct Operation
Incrementer begins calculation
Correct Operation
Incrementer calculates 2nd bit, First bit not written to latch

Correct Operation
Incrementer calculates 3rd bit, 1st, 2nd bits not written to latch

Correct Operation
Correct value ready to latch (100_2), circuit quiescent

Correct Operation
Clock changes to 0
Correct Operation
2nd set of latches write correct value, circuit quiescent

D Flip-Flop (or master-slave flip-flop)
D Flip-Flop is a pair of D latches
• Stupid name, but it stuck
• Isolate next state from current state

Two phases:
• Clock = 1, Clock = 0

D Flip-Flop - WE = 0
Latch #1
Latch #2

When WE = 0
• Latch 1: writing disabled (output is stable Q_{inter})
• Latch 2: writing enabled (Q = Q_{inter})
D Flip-Flop - Phase 1

- Latch #1: writing disabled (output is stable $Q_{\text{inter}}$)
- Latch #2: writing enabled ($Q = Q_{\text{inter}}$)

Phase 1
- Clock = 1

D Flip-Flop - Phase 2

- Latch #1: writing enabled ($Q = Q_{\text{inter}}$)
- Latch #2: writing disabled (output is stable $Q$)

Back to Phase 1
- $Q$ becomes $Q_{\text{inter}}$

Working Counter

- Use a clocked register (made of D flip-flops)
- More simply
  - If $WE = 1$ assumed if $WE$ is not present
- Use $WE$ input for conditional counter (stop watch)

Memory

- Now that we know how to store bits, we can build a memory – a logical $k$ by $m$ array of stored bits

  Address Space: number of locations ($k = 2^n$ locations) (usually a power of 2)

  Addressability: number of bits per location ($m$ bits) (e.g., byte-addressable)
Memory Interface

Memory (2^n by m-bit)

2^n by 3-bit memory

Read operation

2^2 or 4 registers

Write operation

The Decoder

n inputs, 2^n outputs

- Exactly one output is 1 for each possible input pattern

2-bit decoder
2^2 by 3-bit memory - Multiple “Ports”
Independent Read/Write

An Efficient 2^2 by 3-bit Memory - Single Port

More Memory Details
This is still not the way actual memory is implemented
• Real mem: Fewer transistors, much more dense, relies on analog properties
But the logical structure is similar
• Address decoder
• Word select line, word write enable
• Bit line
Two basic kinds of RAM (Random Access Memory)
Static RAM (SRAM)
• Fast, maintains data as long as power applied
Dynamic RAM (DRAM)
• Slower but denser, bit storage decays – must be periodically refreshed

Also, non-volatile memories: ROM, PROM, flash, …
### State Machine

Another type of sequential circuit
- Combines combinational logic with storage
- "Remembers" state, and changes output (and state) based on inputs and current state

### Combinational vs. Sequential

Two types of “combination” locks
- **Combinational**
  - Success depends only on the values, not the order in which they are set.
- **Sequential**
  - Success depends on the sequence of values (e.g., R-13, L-22, R-3).

### State

The state of system is snapshot of all relevant elements of system at moment snapshot is taken

**Examples**
- The state of a basketball game can be represented by the scoreboard
  - Number of points, time remaining, possession, etc.
- The state of a tic-tac-toe game can be represented by the placement of X's and O's on the board (and turn)

### State of Sequential Lock

Our lock example has four different states, labeled A-D:

- **A**: The lock is **not open**, and no relevant operations have been performed
- **B**: The lock is **not open**, and the user has completed the **R-13** operation
- **C**: The lock is **not open**, and the user has completed **R-13**, followed by **L-22**
- **D**: The lock is **open**
**Sequential Lock State Diagram**

Shows states and actions that cause a transition between states.

**Finite State Machine**

A description of a system with the following components:

1. A finite number of states
2. A finite number of external inputs
3. A finite number of external outputs
4. An explicit specification of all state transitions
5. An explicit specification of what determines each external output value

Often described by a state diagram
- Inputs trigger state transitions
- Outputs are associated with each state (or with each transition)

**Implementing a Finite State Machine**

**Combinational logic**
- Determine outputs and next state.

**Storage elements**
- Maintain state representation.

**Storage**

Master-slave flip-flop stores one state bit

Number of storage elements (flip-flops)
- Determined by number of states (and representation of states)

**Examples**
- Sequential lock
  - Four states – two bits
- Basketball scoreboard
  - 8 bits for each score, 5 bits for minutes, 6 bits for seconds, 1 bit for possession arrow, 1 bit for half,...
Complete Example
A blinking traffic sign
- No lights on
- 1 & 2 on
- 1, 2, 3, & 4 on
- 1, 2, 3, 4, & 5 on
  (repeat as long as switch is turned on)

Traffic Sign State Diagram

Traffic Sign State Diagram: State 00

Traffic Sign State Diagram: State 01
Traffic Sign State Diagram: State 10

Transition on each clock cycle.

Traffic Sign State Diagram: State 11

Transition on each clock cycle.

Traffic Sign State Diagram: State 00

Transition on each clock cycle.

Traffic Sign Truth Tables

Outputs (depend only on state: $S_0$, $S_1$)

Next State: $S_{1}'$, $S_{0}'$ (depend on state and input)

<table>
<thead>
<tr>
<th>$S_0$</th>
<th>$S_1$</th>
<th>$S_{0}'$</th>
<th>$S_{1}'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Whenever $In=0$, next state is 00.
Traffic Sign Logic

Programmable State Machines

What if we want to change the pattern of the sign?
- An alternative state machine implementation
- Use a memory indexed by state number

<table>
<thead>
<tr>
<th>Input</th>
<th>Output (Z, Y, X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>110</td>
<td>11</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>X/Y/Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
</tr>
</tbody>
</table>

Change to a two-state pattern:
- All off
- All on

<table>
<thead>
<tr>
<th>State: 00</th>
<th>State: 10</th>
<th>State: 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>011</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>110</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

From Logic to Data Path
The data path of a computer is all the logic used to process information.
- See the data path of the LC-3 on next slide

Combinational Logic
- Decoders -- convert instructions into control signals
- Multiplexers -- select inputs and outputs
- ALU (Arithmetic and Logic Unit) -- operations on data

Sequential Logic
- State machine -- coordinate control signals and data movement
- Registers and latches -- storage elements

Master-slave flip-flop
Looking Forward...

We've touched upon basic digital logic
- Transistors
- Gates
- Storage (latches, flip-flops, memory)
- State machines

Build some simple circuits
- Incrementer, adder, subtracter, adder/subtracter
- Counter (consisting of register and incrementer)
- Hard-coded traffic sign state machine
- Programmable traffic sign state machine

Up next: a computer as a (simple?) state machine

Next Time

Topic
- The von Neumann Model

Readings
- Chapter 4.0 - 4.2

Online quiz
- You know the drill!