Gates

- Gates realize logic functions, e.g., electronically.
- Gates can be combined using what is essentially functional programming.
- We give symbolic versions of common logic functions.

### and

**Form 1 Table:**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>(\text{and}(x, y))</th>
<th>Gate Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td><img src="image1" alt="Gate Diagram" /></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td><img src="image2" alt="Gate Diagram" /></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td><img src="image3" alt="Gate Diagram" /></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td><img src="image4" alt="Gate Diagram" /></td>
</tr>
</tbody>
</table>

**Arguments:**

- x
- y

**Results:**

- \(\text{and}(x, y)\)

- Because \(\text{and}\) is associative and commutative, the inputs can be regarded as a set with no particular order.

- 3 inputs
- 7 inputs

### or

**Form 1 Table:**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>(\text{or}(x, y))</th>
<th>Gate Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td><img src="image5" alt="Gate Diagram" /></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td><img src="image6" alt="Gate Diagram" /></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td><img src="image7" alt="Gate Diagram" /></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td><img src="image8" alt="Gate Diagram" /></td>
</tr>
</tbody>
</table>

**Arguments:**

- x
- y

**Results:**

- \(\text{or}(x, y)\)

- Because \(\text{or}\) is associative and commutative, the inputs can be regarded as a set with no particular order.

- 3 inputs
- 7 inputs
**not (inverter)**

<table>
<thead>
<tr>
<th>x</th>
<th>not(x)</th>
<th>gate symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>inverter</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

*“bubble” required*

**and, or, implies**

- 0 and x is
- 1 and x is
- 0 or x is
- 1 or x is
- 0 implies x is
- 1 implies x is

**half-adder (HA) circuit**

**1-bit multiplexer**

- The gate equivalent of if-then-else
- \( z = \text{select?} \ x : y \)

**1-bit demultiplexer**

- The gate equivalent of if-then-else
- \( y = \text{select?} \ z : 0 \)
- \( x = !\text{select?} \ z : 0 \)

**implies circuit**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>implies(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\( z = \text{implies}(x, y) \)
**Gate Repertoire**

- By “repertoire” we mean a set of gate types, such as (2-input and, 2-input or, inverter).

**Universal Repertoire**

- A repertoire of n-input and and or gates, and not, is adequate to realize any switching function.

---

**Proof that (and, or, not) are adequate**

- Suppose \( f \) is an n-ary switching function.
- Basis: \( n = 1 \):
  - There are four 1-input functions.
  - \( f_{00}(x) = 0 = \text{and}(x, \text{not}(x)) \)
  - \( f_{01}(x) = x \)
  - \( f_{10}(x) = \text{not}(x) \)
  - \( f_{11}(x) = 1 = \text{or}(x, \text{not}(x)) \)
- (The subscripts of these functions are the result column of the truth table.)

<table>
<thead>
<tr>
<th>( f_{00} )</th>
<th>( f_{01} )</th>
<th>( f_{10} )</th>
<th>( f_{11} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Proof Continued: Induction Step**

- Suppose that we can realize any n-ary function with \( \text{and}, \text{or}, \text{not} \).
- Let \( f \) be an arbitrary \((n+1)\)-ary function.
- Then \( f(x_0, x_1, \ldots, x_n) = \text{or}(\text{and}(x_0, f(1, x_1, \ldots, x_n)), \text{and}(\text{not}(x_0), f(0, x_1, \ldots, x_n))) \).
- But \( f(0, x_1, \ldots, x_n) \) and \( f(1, x_1, \ldots, x_n) \) are realizable, as they are n-ary functions.

---

**Proof of the equality**

- If \( x_0 = 1 \), the equality is:
  - \( f(1, x_1, \ldots, x_n) = \text{or}(\text{and}(1, f(1, x_1, \ldots, x_n)), \text{and}(\text{not}(1), f(0, x_1, \ldots, x_n))) \).
- But the RHS is \( \text{or}(\text{and}(1, f(1, x_1, \ldots, x_n)), \text{and}(0, \ldots)) \).
  - So the equality checks for the case \( x_0 = 1 \).

- If \( x_0 = 0 \), the equality is:
  - \( f(0, x_1, \ldots, x_n) = \text{or}(\text{and}(0, f(1, x_1, \ldots, x_n)), \text{and}(\text{not}(0), f(0, x_1, \ldots, x_n))) \).
  - But the RHS is \( \text{or}(\text{and}(0, \ldots)), \text{and}(1, f(0, x_1, \ldots, x_n))) \).
  - So the equality checks for the case \( x_0 = 0 \).
Diagrammatic Proof

This is an application of the Boolean-Shannon principle.

1-bit multiplexer

Proof as Synthesis

- The preceding proof determines one synthesis method for an arbitrary switching function:
  - If the function is of 1-ary, it is one of the four basis functions.
  - If the function is n+1-ary, then decompose it into a composition of n-ary functions, and, or, and not.

Synthesis Example

- Synthesis the function \( f \) represented by this truth-table:

<table>
<thead>
<tr>
<th>( w )</th>
<th>( x )</th>
<th>( y )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Decompose on \( w \):

<table>
<thead>
<tr>
<th>( w )</th>
<th>( x )</th>
<th>( y )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Decompose each on \( x \):

<table>
<thead>
<tr>
<th>( x )</th>
<th>( w )</th>
<th>( y )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Another Synthesis Method**

- A problem with the previous method is that it tends to produce circuits with long paths (= delay).
- The next method reduces this problem.

**Minterm Synthesis**

- For a given set of variables, a minterm is a product (and) of those variables and their negations.
- Example: 3 variables: x, y, z
  - Some minterms: xyz', xy'z', ...
- Each minterm represents a function that is "on" (1) for exactly one combination of variables
  - e.g. xyz' is on for 110, x'y'z is on for 010.

**Minterm Representation**

- Any function can be represented as the sum (or) of its minterms.
- These correspond to the rows of the truth table for which the function is 1.

**Minterm Example**

<table>
<thead>
<tr>
<th>minterm representation</th>
<th>corresponding to rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>w'xy' + w'xy + wx'y + wxy'</td>
<td>010, 011, 101, 110</td>
</tr>
</tbody>
</table>

**Minterm Synthesis**

- Use "rails" for clarity

```
\[ w'xy' + w'xy + wx'y + wxy' \]
```
**A Universal Repertoire of One Gate Type**

- **nand** = "not and"

**nand**

<table>
<thead>
<tr>
<th>x ( \bar{y} )</th>
<th>( \text{nand}(x, y) )</th>
<th>gate symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Universality of Nand**

- Consider any minterm realization, e.g., \( m_1 + m_2 + m_3 \).
- To realize this with a final nand, use a version of deMorgan's law:

\[
(m_1 \land m_2 \land m_3)' = \text{nand}(m_1', m_2', m_3')
\]

But each minterm \( m_i \) is an **and** of variables or their negations. So each \( m_i' \) is a **nand** of the same variables and their negations.
Nand Minterm Example

and/or = nand

Another universal set: \{ \text{nor} \}

- \text{nor} is the negation of \text{or}
- Show that any minterm form can be re-expressed using only \text{nor}.

Number of Functions

- For \( n \) variables, how many minterms are there?
- How many distinct \( n \)-ary functions are there?

Building Blocks

- For functions of large arity, it is sometime impractical to synthesize by previous methods: Tables may be too large.
- In such cases, it is common to use standard building blocks to construct circuits.

xor (exclusive or) building block

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( x \oplus y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\text{minterm realization}

\text{ xor symbols: }
The full adder is a building block sometimes used to construct adders of arbitrary length.

Unlike the Half-Adder, an FA has:
- 3 inputs: x, y, carryIn
- 2 outputs: sum, carryOut

Each output can be synthesized as a separate function. sum is a 3-input xor. CarryOut is a 2/3 majority.

This adder is easy to make, but slow for large n.

The longest path in # of gates through the circuit determines the delay. This is called the "gate depth".

An n-bit ripple carry adder has gate depth $O(n)$.

As an example of a faster adder, the carry-select adder has a gate depth of $O(\sqrt{n})$.

It works by computing outputs for both carryIn = 0 and carryIn = 1, then selecting the appropriate result when the carry is known.
Carry-Select Adder Construction

In general, \( \sqrt{n} \) blocks of size \( \sqrt{n} \) can be used.

Note that this is a Divide & Conquer approach.


Even Faster Addition

- A lower-bound for \( n \)-bit addition is \( \Omega(\log n) \). (\( \Omega \), rather than \( O \), is used for lower bounds).
- The Kogge-Stone adder achieves this bound.
- This is an example of a "digital" (radix-based) approach.

http://en.wikipedia.org/wiki/Kogge%E2%80%93Stone_adder

Multiplier Problem

- In the current assignment, you are asked to construct a multiplier.
- Consider using a "digital" method, wherein the multiplier's bits are used to decide whether or not to include the multiplicand at various points in the product computation.
- Essentially use the algorithm learned in school, but in binary.
- One approach is to use a "controlled" version of the ripple-carry adder as a building block.

A Partial Multiplier Tester

Programmable Logic Arrays (PLAs)
Elements with Memory

- So far we have discussed "combinational" elements. The output, after all signals have fully propagated, is a function of the input, without regard to history.

- More capabilities are available through elements with memory. These allow computations to be sequential, rather than just combinational.

Flip-Flop: A basic memory unit

- The flip-flop remembers the value (0 or 1) it was last assigned.

- Assignment normally is done at the tick of a clock.

- Circuits constructed this way are called clocked sequential circuits, or synchronous circuits.

D Flip-Flop with Clock

- The flip-flop remembers the value (0 or 1) it was last assigned.

- Assignment normally is done at the tick of a clock.

- Circuits constructed this way are called clocked sequential circuits, or synchronous circuits.

D Flip-Flop Behavior

- Circuits constructed this way are called clocked sequential circuits, or synchronous circuits.

Sequential Circuit

- remembers whether Data was ever 1.

State Diagram

- Input = Value of Data at Tick

- State = Value of Q
Sequential Adder

- Using a D Flip-Flop, the circuit can remember the carry from one tick to the next.
- Use two D Flip-Flops for input, one for output, one for carry.

State Diagram for Adder

Input/Output = XY/Z

State = Value of Carry Stored

Flip-Flop Construction

- Flip-Flops can be constructed from basic gates.
- Broad-categories, of increasing sophistication, are:
  - Latch
  - Gated Latch, or Level-Triggered Flip-Flop
  - Edge-triggered Flip-Flop

Latches

- A latch is the basic memory structure for a flip-flop.
- Here is one possible latch, called set-reset (SR), constructed from nor gates:

  Shown is the reset state: Q = 0, Q' = 1.

Latch Operation

- Only one of S, R should be 1 at any given time.
- When S is raised, the latch goes to the set state (Q = 1, Q' = 0).
- When S is lowered, the set state is maintained.
- When R is raised, the latch goes to the reset state (Q = 0, Q' = 1).
- When R is lowered, the set state is maintained.
Latch Operation

Gated Latch
- This enables or disables the latch set and reset.
- Enable line is also called “Strobe”.
- D is for “Data”

Level-Triggered Flip-Flop
- Adds clock input to gated latch
- Enable also called “Strobe”

Edge-Triggered Flip-Flop

Registers
- A flip-flop is a 1-bit register.
- Flip-flops can be combined in parallel to produce multiple-bit registers.
- Contents of one register can be transferred to another on clock tick

Straight Register Transfer

Register Transfer with Combinational Functions

Extra In-Register Functions
- Shift (Left or Right)
- Increment or Decrement
- Zero
- Complement

Sequential Multiplier

Implementing General Finite-State Machines
- Finite-State machines have many uses in computers, including registers and sequencers.
- Here we deal with the general problem of implementing an FSM starting with a state-transition diagram.

Memory Considerations
- The state of the machine has to be stored in some kind of memory, usually flip-flops.
- The flip-flop values represent an encoding of the current state.

State-Machine Example
**Target FSM Architecture**

**Input and State Encodings**
- Encodings may be pre-specified, or at the discretion of the designer.
- Example: Suppose the input is pre-specified to be (0, 1), while the state encodings are discretionary.

**Encoding State Set**
- State set is {0, 1, 2}.
- What are the possible ways to encode it in binary?
- What is the fewest number of bits?

**Binary Encoding**
- 0 → 00
- 1 → 01
- 2 → 10

**Gray-Code Encoding**
- 0 → 00
- 1 → 01
- 2 → 11

**General Gray-Code Pattern: Reflection**

<table>
<thead>
<tr>
<th>1-bit</th>
<th>3-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2-bit</th>
<th>3-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>000</td>
</tr>
<tr>
<td>0 1</td>
<td>001</td>
</tr>
<tr>
<td>1 0</td>
<td>011</td>
</tr>
<tr>
<td>1 1</td>
<td>101</td>
</tr>
<tr>
<td>1 0</td>
<td>111</td>
</tr>
<tr>
<td>0 1</td>
<td>100</td>
</tr>
<tr>
<td>0 0</td>
<td>010</td>
</tr>
</tbody>
</table>
One-Hot Encoding

- 0 → 100
- 1 → 010
- 2 → 001

Simplest Logic

- One-Hot Encoding often yields simplest design.
- However, it is expensive if the number of states is large (n flip-flops for n states, vs. ceil(log(n)) possible).

For One-Hot, logic functions can be read off from the state diagram

- Let the input variable be x.
- Let the state variables be y1y2y3 (initially 100).
- Then in combinational logic:
  - next y1 = y1' + y2x
  - next y2 = y1x + y3x
  - next y3 = y2' + y3x
- Note that in a one-hot design, the negations of flip-flop values are not needed.

One-Hot Version

```
initial state
0 1 1 0 2 1
encode states
```

"Thermometer" Encoding

- 0 → 100
- 1 → 110
- 2 → 111

First Equation

\[ \text{next } y_1 = y_1' + y_2x \]
Using a Different State Encoding

To demonstrate, let’s use the Gray-code encoding:
- \(0 \rightarrow 00\)
- \(1 \rightarrow 01\)
- \(2 \rightarrow 11\)

Transition Table

Here is the state-transition function in tabular form:

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Edit the States

Carefully replace each state with its corresponding encoding:
- \(0 \rightarrow 00\)
- \(1 \rightarrow 01\)
- \(2 \rightarrow 11\)

Split the Table into Separate Tables for Each Flip-Flop
Read Off Equations from Each Table

Design for Gray-Code Encoding

Check for Gray-Code Version Operation

Adding Output

Output Models

Acceptor

- The preceding discussion only addresses the state-transition aspect of finite-state machines.
- To do useful work, our machine may need output as well.

- Each state of an acceptor is either
  - Accepting, or
  - Rejecting
- The convention is that accepting states are doubly circled.
- There can be multiple accepting states.

- There are two general models for output:
  - Attach output to transitions (called the *Mealy model*, after George H. Mealy)
  - Attach output to states (called the *Moore model*, after E.F. Moore
    - A special case of just two possible output values is called an acceptor.
Acceptor Example

This acceptor accepts sequence that are multiples of 3 in binary, msb-first, e.g. 0, 11, 110, 1001, 1100, ...

Initial state 0
Accepting state (since doubly circled)

Acceptor Example 2

This acceptor accepts sequence that are not multiples of 3 in binary, msb-first, e.g. 1, 10, 100, 101, ...

Initial state 0
Accepting state (since doubly circled)

Note on Acceptance

- Acceptance and Rejection apply to the amount of the input sequence seen so far. They are generally not final conditions.

- It is possible to go from accept to reject and back, by providing more input.

Acceptance → Output

- We can assign an output of 1 to acceptance, 0 to rejection using combinational logic. For multiple-of-3, one-hot:

Mealy Model

- The output is attached to transitions, but a transition comes from a state and is based on an input.

- Therefore the output is a combination of the current state and the input.

- This output can be “captured” by flip-flops. This was indicated back on the sequential binary adder example.
Mealy Model Example

Output as a function of current state Carry and input X Y

Subsequent Discussions

- Finite-State Machines in the abstract
- Regular (type 3) languages: languages accepted by finite-state acceptors
- Regular expressions