Logic Circuits

Chapter 2

Overview

- Many important functions computed with *straight-line programs*
  - No loops nor branches
  - Conveniently described with *circuits*
- Circuits are *directed acyclic graphs*
  - Characterized by *size* and *depth*
- Circuits where operations involved are Boolean are called *logic circuits*
Overview (cont.)

- Circuits where operations involved are algebraic are called **algebraic circuits**
- Circuits where operations involved are comparisons are called **comparator circuits**
- **Logic circuits** are basic building blocks of real-world computers
- All machines with bounded memory can be built from logic circuits and binary memory units

Overview (cont.)

- Machines that perform finite-step computation can be simulated by logic circuits
- Chapter discusses:
  - Circuits vis-à-vis straight-line programs
  - Class of functions computed by logic circuits
  - Circuit designs for a number of important functions
  - Important results about Boolean functions and logic circuits
  - **Problem reduction** as a powerful tool of analysis
Designing Circuits

- Logic circuit
  - A directed, acyclic graph (DAG) whose vertices are labeled with Boolean functions (i.e., logic gates) or variables (i.e., inputs)
  - Computes a binary function
    \[ f : \mathcal{B}^n \rightarrow \mathcal{B}^m \]
    where \( n \) is the number of input variables and \( m \) is the number of outputs in the circuit

Designing Circuits (cont.)

- A binary function
  \[ f : \mathcal{B}^n \rightarrow \mathcal{B}^m \]

- Goal is to design efficient circuits
  - Small size (i.e., number of gates)
  - Small depth (i.e., length of longest path)
Designing Circuits (cont.)

- Logic circuits help provide framework for problem classification based on computational complexity
  - Used to identify hard computational problems such as \( \mathbf{P} \)-complete languages and \( \mathbf{NP} \)-complete languages (cf. Ch. 3)
- Show number of Boolean functions much greater than number of possible logic circuits of a given maximum size \( \Rightarrow \) most Boolean functions must be complex

Straight-line Programs and Circuits

- Example from text (cf. pp. 36-38):
  
  Functional description:
  
  \[
  \begin{align*}
  g_1 & := x; \\
  g_2 & := y; \\
  g_3 & := \neg g_1; \\
  g_4 & := \neg g_2; \\
  g_5 & := g_1 \land g_4; \\
  g_6 & := g_2 \land g_3; \\
  g_7 & := g_5 \lor g_6;
  \end{align*}
  \]

  Straight-line program:
  
  \[
  \begin{align*}
  (1 & \text{ READ } x) \\
  (2 & \text{ READ } y) \\
  (3 & \text{ NOT } 1) \\
  (4 & \text{ NOT } 2) \\
  (5 & \text{ AND } 1 4) \\
  (6 & \text{ AND } 3 2) \\
  (7 & \text{ OR } 5 6) \\
  (8 & \text{ OUTPUT } 5) \\
  (9 & \text{ OUTPUT } 7)
  \end{align*}
  \]
Straight-line Programs and Circuits (cont.)

- Formal definitions
  - **Definition 2.2.1**: A *straight-line program* is a set of steps each of which is an **input step**, denoted by $(s \text{ READ } x)$, or an **output step**, denoted by $(s \text{ OUTPUT } i)$, or a **computation step**, denoted by $(s \text{ OP } i \ldots k)$.
    
    Here $s$ is the (ordinal) number of a step (allowing us to see the sequence by which the steps are to be executed).
    
    $x$ denotes an input variable.
    
    The arguments $i \ldots k$ for an OP step must be less than $s$ the step number of that OP step, *i.e.*, $s > i, \ldots, k$

Straight-line Programs and Circuits (cont.)

- Formal definitions (cont.)
  - **Definition 2.2.1**: A *circuit* is the graph of a straight-line program. The **fan-in** of a circuit is the maximum in-degree of any vertex. The **fan-out** is the maximum out-degree of any vertex. A **gate** is any vertex associated with a computation step (*i.e.*, an OP step).
    
    In our example, both **fan-in** and **fan-out** are equal to 2.
    
    The gates are those vertices representing the NOT, AND, and OR operations.
Definition 2.2.1: The \textit{basis} $\Omega$ of a circuit and its corresponding straight-line program is the set of operations that they use.

The bases of Boolean straight-line programs and logic circuits contain only Boolean functions.

The \textit{standard basis} $\Omega_0$ is the set \{NOT, AND, OR\}.

Our example uses the standard basis.

---

Definition 2.2.2: Let $g_s$ be the \textit{function computed by the $s$-th step of a straight-line program}.

If the $s$-th step is the input step ($s$ \textsc{READ} $x$), then $g_s = x$.

If the $s$-th step is the computation step ($s$ \textsc{OP} $i \ldots k$), then $g_s = \text{OP}(g_i, \ldots, g_k)$, where $g_i, \ldots, g_k$ are the functions computed by steps $i \ldots k$.

If a straight-line program has $n$ inputs and $m$ outputs, it computes a function $f : \mathcal{B}^n \rightarrow \mathcal{B}^m$. If $s_1, s_2, \ldots, s_m$ are the output steps, then $f = (g_1, g_2, \ldots, g_m)$.

The function computed by a circuit is the function computed by the corresponding straight-line program.
Functions Computed by Circuits (cont.)

- Example from text (cf. pp. 36-38):

<table>
<thead>
<tr>
<th>Straight-line program:</th>
<th>Functions computed by circuit:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 READ x)</td>
<td>$g_1 := x$;</td>
</tr>
<tr>
<td>(2 READ y)</td>
<td>$g_2 := y$;</td>
</tr>
<tr>
<td>(3 NOT 1)</td>
<td>$g_3 := \neg x$;</td>
</tr>
<tr>
<td>(4 NOT 2)</td>
<td>$g_4 := \neg y$;</td>
</tr>
<tr>
<td>(5 AND 1 4)</td>
<td>$g_5 := x \land \neg y$;</td>
</tr>
<tr>
<td>(6 AND 3 2)</td>
<td>$g_6 := y \land \neg x$;</td>
</tr>
<tr>
<td>(7 OR 5 6)</td>
<td>$g_7 := (x \land \neg y) \lor (y \land \neg x)$;</td>
</tr>
<tr>
<td>(8 OUTPUT 5)</td>
<td></td>
</tr>
<tr>
<td>(9 OUTPUT 7)</td>
<td>$f(x,y) = (g_5, g_7)$</td>
</tr>
</tbody>
</table>

Fig. 2.1

Circuits That Compute Functions

- Given a circuit, we know how to determine the function it computes.
- Given a function, how do we construct a circuit (and straight-line program) that computes it?
  - Method involves following steps:
    - Construct functional table
    - Express in normal form, can be transformed directly into a circuit
    - Simplify to reduce circuit complexity
Example: $f: B^3 \rightarrow B^2$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y_1$</th>
<th>$y_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</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>1</td>
</tr>
</tbody>
</table>

Express output in disjunctive normal form (DNF):

$y_1 = (\neg x_1 \land \neg x_2 \land \neg x_3) \lor (\neg x_1 \land x_2 \land \neg x_3) \lor (x_1 \land \neg x_2 \land \neg x_3) \lor (x_1 \land x_2 \land x_3)$

$y_2 = (\neg x_1 \land \neg x_2 \land \neg x_3) \lor (\neg x_1 \land \neg x_2 \land x_3) \lor (\neg x_1 \land x_2 \land x_3) \lor (x_1 \land \neg x_2 \land \neg x_3) \lor (x_1 \land x_2 \land x_3)$

Each of the terms can be realized by a simple circuit:

- Circuit that computes the function is made up of the OR’s of these components.
Circuits That Compute Functions (cont.)

- Circuit that computes the function is made up of the or’s of these simpler components:

- **THEOREM 2.3.1** Every function \( f : \mathcal{X}^n \rightarrow \mathcal{X}^m \) can be realized by a logic circuit.

Circuit Complexity Measures

- **Definition 2.2.3**: The *size* of a logic circuit is the number of gates it contains. The *depth* is the number of gates on the longest path through the circuit.

  The *circuit size*, \( C_\Omega(f) \), and *circuit depth*, \( D_\Omega(f) \), of a Boolean function \( f \) are defined as the smallest size and smallest depth of any circuit, respectively, over the basis \( \Omega \) of \( f \).

- Clearly, it is desirable to be able to construct the smallest or most shallow circuit for a function –
  - If the circuit is small in size, the complexity of the function computed must also be modest
  - If the circuit is shallow in depth, the speed of computation tends to be faster when the circuit is physically realized
Other Normal Forms

- **Conjunctive Normal Form (CNF)**
  \[(\neg x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (x_1 \lor x_2 \lor \neg x_3)\]

- **Sum-Of-Products Expansion (SOPE)**
  \[x_1 \cdot x_2 \cdot x_3 \lor x_1 \cdot x_2 \cdot x_3 \lor x_1 \cdot x_2 \cdot x_3 \lor x_1 \cdot x_2 \cdot x_3 \lor x_1 \cdot x_2 \cdot x_3\]

- **Product-Of-Sums Expansion (POSE)**
  \[(x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor x_3)\]

Due to the definitions of each, it can be shown that although the DNF and CNF for a Boolean function are unique, there can be more than one equivalent SOPE or POSE for the same function.

Other Normal Forms (cont.)

- **Ring-Sum Expansion**
  - EXCLUSIVE OR (\(\oplus\)) of a constant and products (\(\land\)) of **unnegated** variables of the function
  - Can be constructed from the DNF; since only one of the possible input combinations can present itself at any single time, the \(\lor\) can be replaced by the \(\oplus\)
  - Example:
    \[(\neg x_1 \land x_2 \land x_3) \lor (\neg x_1 \land \neg x_2 \land x_3) \lor (x_1 \land x_2 \land x_3) \Rightarrow\]
    \[(x_1 \cdot x_2 \cdot x_3) \oplus (x_1 \cdot x_2 \cdot x_3) \oplus (x_1 \cdot x_2 \cdot x_3) =\]
    \[(x_1 \oplus 1) \cdot x_2 \cdot x_3 \oplus (x_2 \oplus 1) \cdot x_1 \cdot x_2 \cdot x_3 =\]
    \[x_1 \cdot x_2 \cdot x_3 \oplus x_2 \cdot x_3 \oplus x_1 \cdot x_2 \cdot x_3 \oplus x_2 \cdot x_3 \oplus x_1 \cdot x_3 \oplus x_1 \cdot x_2 \cdot x_3 =\]
    \[x_1 \cdot x_2 \cdot x_3 \oplus x_1 \cdot x_3 \oplus x_3\]
Other Normal Forms (cont.)

- **Comparison of Normal Forms**
  - RSE for a Boolean function is unique but is not necessarily the most compact representation
  - Some functions can be represented compactly in one form but not as compact in another
  - Example: Parity function \( f_P(x_1, \ldots, x_n) = 1 \) when an odd number of the \( n \) inputs is 1 and 0 otherwise
    \[
    \begin{align*}
    f_P(x_1, \ldots, x_n) &= (x_1 \land \neg x_2 \land \ldots \land \neg x_n) \lor (\neg x_1 \land x_2 \land \ldots \land x_n) \lor \ldots \lor \\
    &\ldots (\neg x_1 \land \neg x_2 \land \ldots x_n) \lor \ldots
    \end{align*}
    \]
    (DNF has \( 2^{n-1} \) terms)
  - \( f_P(x_1, \ldots, x_n) = x_1 \oplus x_2 \oplus \ldots \oplus x_n \)
    (RSE has \( n \) terms)

Reduction Between Functions

- Common method to solve a new problem:
  - See if an existing solution can be applied to it
  - Called a “reduction” in complexity theory
- **Definition 2.4.1** A function \( f: \mathcal{A}^n \rightarrow \mathcal{A}^m \) is a reduction to the function \( g: \mathcal{A}^r \rightarrow \mathcal{A}^s \) through application of a function \( p: \mathcal{A}^s \rightarrow \mathcal{A}^m \) and \( q: \mathcal{A}^n \rightarrow \mathcal{A}^r \) if for all \( x \in \mathcal{A}^n \):
  \[
  f(x) = p(g(q(x)))
  \]
A simple reduction is the subfunction

**Definition 2.4.2** Let $g$ be a function $g : A^n \rightarrow A^m$. A subfunction $f$ of $g$ can be obtained by assigning values to some of the input variables of $g$, assigning variable names (not necessarily unique) to the rest of the input, deleting and/or permuting some of its output variables. We say the $f$ is a reduction to $g$ via the subfunction relationship.

**Lemma 2.4.1** If $f$ is a subfunction of $g$, a straight-line program for $f$ can be created from a straight-line program for $g$ without increasing the size nor depth of its circuit.
Example: Reduction of \textit{logical shift} function to \textit{cyclic shift} function

\[ f^{(n)}_{\text{shift}} \rightarrow f^{(2n)}_{\text{cyclic}} \]

Example: Reduction of \textit{cyclic shift} function to \textit{logical shift} function

\[ f^{(n)}_{\text{cyclic}} \rightarrow f^{(2n)}_{\text{shift}} \]
Reduction Between Functions (cont.)

- **Lemma 2.5.2** The *cyclic shift* function is a subfunction of the *logical shift* function and vice-versa.

- Circuits for important Boolean functions are presented in detail in text (not to be discussed)
  - Encoders and decoders
  - Multiplexers and demultiplexers
  - Arithmetic operations
  - Symmetric functions
    - Binary sorting function
    - Modulus functions

Most Boolean Functions Are Complex

- A Boolean function on $n$ variables can be represented by a table with $2^n$ rows
  - Each entry can be filled with either a 1 or 0
  - There can be $2^n$ Boolean functions on $n$ variables

- The number of logic circuits bounded by some limit on size is not as “explosive”.
  - Most Boolean functions cannot be realized by small circuits
Most Boolean Functions Are Complex

- Our first main result is given here with proof sketch:

**THEOREM 2.12.1** Let \( 0 < \epsilon < 1 \). The fraction of the Boolean functions \( f : B^n \rightarrow B \) that have size complexity \( C_{\omega_0}(f) \) satisfying the following lower bound is at least \( 1 - 2^{-[\log_2(1 - \epsilon) / \epsilon]} \) when \( n \geq 2[(1 - \epsilon) / \epsilon] \log_2[(3\epsilon)^3(1 - \epsilon) / 2] \). (Here \( e \approx 2.71828 \ldots \) is Euler's constant.)

\[
C_{\omega_0}(f) \geq \frac{2^n}{n} (1 - \epsilon) - 2n^2
\]

- Count number of circuits realizable with \( g \) gates using the standard basis \( \Omega_0 \)
- Approximate upper bound to this count and use it to give an upper bound to number of circuits realizable with up to \( G \) gates
- Form ratio of this to total number of functions in \( n \) variables

Most Boolean Functions Are Complex (cont.)

- Our second main result is given here with yet another proof sketch:

**THEOREM 2.12.2** For each \( 0 < \delta < 1 \) a fraction of at least \( 1 - 2^{-6\delta n} \) of the Boolean functions \( f^{(n)} : B^n \rightarrow B \) have depth complexity \( D_{\omega_0}(f) \) that satisfies the following bound when \( n \geq 5 

\[
D_{\omega_0}(f) \geq n - \log \log n - O(1)
\]

- Simulate any circuit of depth \( d \) with an equivalent tree circuit with the same depth
- Count the number of distinct labeled tree circuits with depth \( d \) up to a given \( D \)
- Form ratio of this to total number of functions in \( n \) variables
Most Boolean Functions
Are Complex (cont.)

Most Boolean functions on \( n \) variables require circuits whose size and depth are approximately \( 2^n/n \) and \( n \), respectively.

Upper Bounds on Circuit Size

- Every Boolean function on \( n \) variables can be realized with circuit size and depth close to the lower bounds derived previously
  - Every function \( f^{(n)} \) can be expressed in DNF
  - DNF can be realized by a circuit consisting of a circuit for a decoder function (to select a term in the DNF) and then a circuit for an OR tree
Upper Bounds on Circuit Size (cont.)

- From results in Sections 2.2.2 and 2.5.4:
  - Circuit size has the following upper bound:
    \[ C_\Omega(f) \leq C_\Omega(f_{\text{decode}}) + 2^{n-1} \leq 3 \times 2^{n-1} + (2n-2)2^{n/2} \]
  - Circuit depth has the following upper bound:
    \[ D_\Omega(f) \leq D_\Omega(f_{\text{decode}}) + n + 1 \leq n + \lceil \log_2 n \rceil \]

- **Theorem 2.13.1** The depth complexity of every Boolean function is \( O(n) \).

---

Upper Bounds on Circuit Size (cont.)

- By expressing an arbitrary Boolean function \( f \) on \( n \) variables \((x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_n)\) as a function of two sets of variables \((x_1, x_2, \ldots, x_k)\) and \((x_{k+1}, \ldots, x_n)\). We can express \( f \) using its **\((k,s)\)-Lupanov representation**
  - Circuit size has the following upper bound:
    \[ C_{k,s}(f) \leq O(2^n/n^2) + O(2^n/n^3) + 2^n / (n-5 \log_2 n) \]

- **Theorem 2.13.2** The size complexity of every Boolean function is \( O(2^n/n) \).