Announcements

• Reading assignments
  – 7th Edition, Section 13.4
  – 6th Edition, Section 12.4

• Homework 7 due today
• Homework 8 out Friday, due Friday, June 7
• Final exam, Monday June 10. Room TBA. Study materials out Friday/Monday.

Last lecture highlights

• NFAs from Regular Expressions

\[(01 \cup 1)^*0\]

• “Subset construction”: NFA to DFA
DFAs ≡ Regular Expressions

We have shown how to build an optimal DFA for every regular expression
- Build NFA
- Convert NFA to DFA using subset construction
- Minimize resulting DFA

Theorem: A language is recognized by a DFA iff it has a regular expression

Generalized NFAs

- Like NFAs but allow
  - Parallel edges
  - Regular Expressions as edge labels
    - NFAs already have edges labeled $\lambda$ or $\alpha$
  - An edge labeled by $A$ can be followed by reading a string of input chars that is in the language represented by $A$
- A string $x$ is accepted iff there is a path from start to final state labeled by a regular expression whose language contains $x$
Starting from NFA

- Add new start state and final state
- Then eliminate original states one by one, keeping the same language, until it looks like:
- Final regular expression will be A

Only two simplification rules:

- **Rule 1:** For any two states \( q_1 \) and \( q_2 \) with parallel edges (possibly \( q_1 = q_2 \)), replace

\[
q_1 \xrightarrow{A} q_2 \quad \text{by} \quad q_1 \xrightarrow{A \cup B} q_2
\]

- **Rule 2:** Eliminate non-start/final state \( q_3 \) by replacing all

\[
q_1 \xrightarrow{A} q_3 \xrightarrow{B} q_2 \quad \text{by} \quad q_1 \xrightarrow{AB^*C} q_2
\]

for every pair of states \( q_1, q_2 \) (even if \( q_1 = q_2 \))

Converting an NFA to a regular expression

- Consider the DFA for the mod 3 sum
  - Accept strings from \( \{0,1,2\}^* \) where the digits mod 3 sum of the digits is 0

Splicing out a node

- Label edges with regular expressions

\[
t_0 \rightarrow t_1 \rightarrow t_0 : \quad 10^*2 \\
t_0 \rightarrow t_1 \rightarrow t_2 : \quad 10^*1 \\
t_2 \rightarrow t_1 \rightarrow t_0 : \quad 20^*2 \\
t_2 \rightarrow t_1 \rightarrow t_2 : \quad 20^*1
\]
A = \{0^n1^n : n \geq 0\} cannot be recognized by any DFA

Consider the infinite set of strings
S = \{\lambda, 0, 00, 000, 0000, \ldots\}

Claim: No two strings in S can end at the same state of any DFA for A

Proof: Suppose n \neq m and 0^n and 0^m end at the same state p.
Since 0^n1^n is in A, following 1^n after state p must lead to a final state.
But then the DFA would accept 0^m1^n which is a contradiction to the DFA recognizing A. \(\square\)

Given claim, the # of states of any DFA for A must be \(\geq |S|\) which is not finite, which is impossible for a DFA.

The set B of binary palindromes cannot be recognized by any DFA

Consider the infinite set of strings
S = \{\lambda, 0, 00, 000, 0000, \ldots\} = \{0^n : n \geq 0\}

Claim: No two strings in S can end at the same state of any DFA for B

Proof: Suppose n \neq m and 0^n and 0^m end at the same state p.
Since 0^n1^n is in B, following 1^n after state p must lead to a final state.
But then the DFA would accept 0^m1^n which is not in B and is a contradiction since the DFA recognizes B. \(\square\)

Given claim, the # of states of any DFA for A must be \(\geq |S|\) which is not finite, which is impossible for a DFA.

What can Finite State Machines do?

- We’ve seen how we can get DFAs to recognize all regular languages
- What about some other languages we can generate with CFGs?
  - \{0^n1^n : n \geq 0\}?
  - Binary Palindromes?
  - Strings of Balanced Parentheses?
The set $P$ of strings of balanced parentheses cannot be recognized by any DFA

- What infinite set of simple strings can we choose that all must go to different states?

- For each pair of strings in this set what common extension should we choose that shows that they can’t go to the same state?

FSMs in Hardware

- Encode the states in binary: e.g. states 0,1,2,3 represented as 000,100, 010,001, or as 00,01,10,11.
- Encode the input symbols as binary signals
- Encode the outputs possible as binary signals
- Build combinational logic circuit to compute transition function:

Example: 1-bit Full Adder
FSM for binary addition

- Assume that the two integers are $a_{n-1}a_{n-2}\ldots a_1a_0$ and $b_{n-1}b_{n-2}\ldots b_1b_0$ and bits arrive together as $[a_0,b_0]$ then $[a_1,b_1]$ etc.

![ FSM for binary addition diagram ]

[1,1] Generate a carry of 1
[0,1],[1,0] Propagate a carry of 1 if it was already there

FSM for binary addition using output on edges

- Assume that the two integers are $a_{n-1}a_{n-2}\ldots a_1a_0$ and $b_{n-1}b_{n-2}\ldots b_1b_0$ and bits arrive together as $[a_0,b_0]$ then $[a_1,b_1]$ etc.

![ FSM for binary addition using output on edges diagram ]

[1,1] Generate a carry of 1
[0,1],[1,0] Propagate a carry of 1 if it was already there

FSMs without sequential logic

- What if the entire input bit-strings are available at all once at the start?
  - E.g. 64-bit binary addition
- Don’t want to wait for 64 clock cycles to compute the output!
- Suppose all input strings have length $n$
  - Can chain together $n$ copies of the state transition circuit as one big combinational logic circuit

A 2-bit ripple-carry adder

![ A 2-bit ripple-carry adder diagram ]
Problem with Chaining Transition Circuits

- Resulting Boolean circuit is “deep”
- There is a small delay at each gate in a Boolean circuit
  - The clock pulse has to be long enough so that all combinational logic circuits can be evaluated during a single pulse
  - Deep circuits mean slow clock.

Carry-Look-Ahead Adder

Compute generate $G_i = a_i \land b_i$ [1,1]
propagate $P_i = a_i \oplus b_i$ [0,1],[1,0]

These determine transition and output functions
- Carry $C_i = G_i \lor (P_i \land C_{i-1})$ also written $C_i = G_i + P_i C_{i-1}$
- Sum $S_i = P_i \oplus C_{i-1}$

Unwinding, we get

\[ C_0 = G_0 \quad C_1 = G_1 + G_0 P_1 \quad C_2 = G_2 + G_1 P_2 + G_0 P_1 P_2 \]
\[ C_3 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 \]
\[ C_4 = G_4 + G_3 P_4 + G_2 P_3 P_4 + G_1 P_2 P_3 P_4 + G_0 P_1 P_2 P_3 P_4 \]
etc.

Finally, use these to compute

\[ \text{Sum}_0 = P_0 \quad \text{Sum}_1 = P_1 \oplus C_0 \quad \text{Sum}_2 = P_2 \oplus C_1 \]
\[ \text{Sum}_3 = P_3 \oplus C_2 \quad \text{Sum}_4 = P_4 \oplus C_3 \quad \text{Sum}_5 = P_5 \oplus C_4 \quad \text{etc} \]

If all $C_i$ are computed using 2-level logic, total depth is 6.

Smaller Fast Adders?

Carry-look-ahead circuit for carry $C_{n-1}$ has $2 + 3 + \ldots + n = (n+2)(n-1)/2$ gates
- a lot more than ripple-carry adder circuit.

Can do this with roughly $2 \log_2 n$ depth and linear size using ideas from DFAs
Speed things up but stay small?
• To go faster, work on both 1\textsuperscript{st} half and 2\textsuperscript{nd} half of the input at once
  – How can you determine action of FSM on 2\textsuperscript{nd} half without knowing state reached after reading 1\textsuperscript{st} half?
  \[ b_1 b_2 \ldots b_{n/2} b_{n/2+1} \ldots b_{n-1} b_n \]
  what state?
• Idea: Figure out what happens in 2\textsuperscript{nd} half for \textit{all} possible values of the middle state at once

Transition Function Composition

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_0</td>
<td>s_0</td>
<td>s_1</td>
</tr>
<tr>
<td>s_1</td>
<td>s_0</td>
<td>s_2</td>
</tr>
<tr>
<td>s_2</td>
<td>s_0</td>
<td>s_3</td>
</tr>
<tr>
<td>s_3</td>
<td>s_1</td>
<td>s_1</td>
</tr>
</tbody>
</table>

State reached on input \( b_1 \ldots b_n \) is
\[ f_{b_n}(f_{b_{n-1}} \ldots (f_{b_2}(f_{b_1}(\text{start})))) \ldots) = f_{b_n} \circ f_{b_{n-1}} \circ \ldots \circ f_{b_2} \circ f_{b_1}(\text{start}) \]

Transition Function Composition

Constant size 2-level Boolean logic to
• convert input symbol to bits for transition function
\[ f_b \]
• compute composition of two transition functions
\[ g \circ f \]
Total depth 2 \( \log_2 n \)
and size \( \approx n \)

Computing all the values
• We need to compute all of
\[ f_{b_7} \circ f_{b_6} \circ f_{b_5} \circ f_{b_4} \circ f_{b_3} \circ f_{b_2} \circ f_{b_1} \circ f_{b_0} \]
Already computed
\[ f_{b_6} \circ f_{b_5} \circ f_{b_4} \circ f_{b_3} \circ f_{b_2} \circ f_{b_1} \circ f_{b_0} \]
\[ = f_{b_6} \circ (f_{b_5} \circ f_{b_4}) \circ (f_{b_3} \circ f_{b_2} \circ f_{b_1} \circ f_{b_0}) \]
Already computed
\[ f_{b_5} \circ f_{b_4} \circ f_{b_3} \circ f_{b_2} \circ f_{b_1} \circ f_{b_0} \]
\[ = (f_{b_5} \circ f_{b_4}) \circ (f_{b_3} \circ f_{b_2} \circ f_{b_1} \circ f_{b_0}) \]
Already computed
\[ f_{b_4} \circ f_{b_3} \circ f_{b_2} \circ f_{b_1} \circ f_{b_0} \]
\[ = f_{b_4} \circ (f_{b_3} \circ f_{b_2} \circ f_{b_1} \circ f_{b_0}) \]
Already computed
\[ f_{b_3} \circ f_{b_2} \circ f_{b_1} \circ f_{b_0} \]
\[ = f_{b_3} \circ (f_{b_2} \circ f_{b_1} \circ f_{b_0}) \]
Already computed
\[ f_{b_2} \circ f_{b_1} \circ f_{b_0} \]
\[ = f_{b_2} \circ (f_{b_1} \circ f_{b_0}) \]
Already computed
\[ f_{b_1} \circ f_{b_0} \]
\[ = f_{b_1} \circ (f_{b_0}) \]
Already computed
\[ f_{b_0} \]
Already computed
Parallel Prefix Circuit

• The general way of doing this efficiently is called a parallel prefix circuit
  – Designed and analyzed by Michael Fischer and Richard Ladner (University of Washington)

• Uses an adder composition operation that sets $G'' = G' + G'P'$ and $P'' = P'P$
  – we just show it for the part for computing $P''$ which is a Parallel Prefix AND Circuit

Parallel Prefix Adder

• Circuit depth $2 \log_2 n$
  Circuit size $4 n \log_2 n$

• Can get linear size if depth goes to $2 \log_2 n + 2$

• Actual adder circuits in hardware use combinations of these ideas and more but this gives the basics

• Nice overview of adder circuits at http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorith.html