Machines with Memory

Chapter 3 (Part A)

Overview

- Though every finite computational task can be realized by a circuit, it is not practical – we cannot afford to design a special circuit for each one
- Machines with memory allow for reuse of equivalent circuits to realize functions of high circuit complexity
- Discuss the following machines
  - Finite State Machines (FST)
  - Random Access Machines (RAM)
  - Turing Machines (TM)
Overview (cont.)

- Show via simulation that RAMs and TMs are universal machines
  - RAMs can simulate any FSM
  - RAMs and TMs can simulate each other
  - RAMs and TMs are therefore excellent reference models of computation
  - Simulate with circuits computations performed by FSMs, RAMs, and TMs
    - Establish that all computations are constrained by available time and space resources
    - Show there exists an $O(\log ST)$-space, $O(ST)$-time program to write descriptions of circuits simulating FSMs, RAMs, and TMs
    - Provide first examples of $P$-complete and $NP$-complete problems

Finite-State Machines

**Definition 3.1.1** A finite-state machine (FSM) $M$ is a seven-tuple $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$, where $\Sigma$ is the input alphabet, $\Psi$ is the output alphabet, $Q$ is the set of states, $\delta : Q \times \Sigma \rightarrow Q$ is the next-state function, $\lambda : Q \rightarrow \Psi$ is the output function, $s$ is the initial state (which may be fixed or variable), and $F$ is the set of final states ($F \subseteq Q$). If the FSM is given input letter $\alpha$ when in state $q$, it enters state $\delta(q, \alpha)$. While in state $q$ it produces the output letter $\lambda(q)$.
Finite-State Machines (cont.)

- A FSM can be represented with a transition diagram:

- This FSM enters the final state $q_1$ when an odd number of 1’s are read at input
- A realization of the EXCLUSIVE OR function on an arbitrary number of inputs

Finite-State Machines (cont.)

- This variant is a Moore machine – output associated with each state
- Another variant, called the Mealey machine, output is generated with each state transition
- Can be shown that one can simulate the other and vice-versa
Functions Computed by FSMs

- Allow our FSMs only to compute functions that receive inputs and produce outputs at data-independent times
  - Trace computation starting at initial state
    \[
    q^{(0)} = s \quad \text{reads input symbol } w_1 \text{ enters state} \\
    q^{(1)} = \delta( q^{(0)}, w_1 ) \text{ producing output } y_1 \quad \text{reads input symbol } w_2 \text{ enters state} \\
    q^{(2)} = \delta( q^{(1)}, w_2 ) \text{ producing output } y_2 \quad \text{reads input symbol } w_3 \text{ enters state} \\
    \vdots
    \]
    \[
    q^{(T)} = \delta( q^{(T-1)}, w_T ) \text{ producing output } y_T
    \]
  - Function thus computed can be expressed as
    \[
    f_M^{(T)}(q^{(0)}, w_1, \ldots, w_T) = (q^{(T)}, y_1, y_2, \ldots, y_T)
    \]

Functions Computed by FSMs (cont.)

- “General” function computed by an FSM in \( T \) steps:
  \[
  f_M^{(T)}(q^{(0)}, w_1, \ldots, w_T) = (q^{(T)}, y_1, y_2, \ldots, y_T)
  \]
- Memory serves to “remember” the state where computation currently is:
  - “the role of memory is to hold intermediate results on which the logical circuitry of the machine can operate in successive cycles”

Figure 3.3  A circuit computing the same function, \( f_M^{(T)} \), as a finite-state machine \( M \) in \( T \) steps.
Most FSMs used in $T$-step computations compute only subfunctions of this general function $f_M(T)$.

- Example: FSM that forms the EXCLUSIVE OR of $n$ variables.

Only the final output symbol is used (all intermediate “output” symbols are ignored).

Use this general function $f_M(T)$ in deriving space-time product inequalities for RAMs, in establishing a connection between Turing time and circuit complexity, and in the definition of certain $P$-complete and $NP$-complete problems.

Computational tasks are modeled by binary functions, for example, the function $f_M(T)$ computed by a general FSM if all components involved in the definition ($\Sigma$, $\Psi$, $Q$, $\delta$, $\lambda$, $s$, $F$) are all assumed to be encoded in binary.

First result involves time and circuit size and depth of a logical circuit realizing the function computed by an FSM.
Computational Inequalities for the FSM

Theorem 3.1.1 Let $f_M^{(T)}$ be the function computed by the FSM $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ in $T$ steps, where $\delta$ and $\lambda$ are the binary next-state and output functions of $M$. The circuit size and depth over the basis $\Omega$ of any function $f$ computed by $M$ in $T$ steps satisfy the following inequalities:

\[
C_{\Omega}(f) \leq C_{\Omega} \left( f_M^{(T)} \right) \leq TC_{\Omega}(\delta, \lambda)
\]

\[
D_{\Omega}(f) \leq C_{\Omega} \left( f_M^{(T)} \right) \leq TD_{\Omega}(\delta, \lambda)
\]

- Result follows from definitions of $C_{\Omega}(\cdot)$ and $D_{\Omega}(\cdot)$, and previous result (Lemma 2.4.1) about circuit size and depth of subfunctions vis-à-vis function of reference
- Define $T \times C_{\Omega}(\delta, \lambda)$ for an FSM $M$ as the equivalent number of logic operations performed by $M$ in $T$ steps

Computational Inequalities for the FSM (cont.)

Theorem 3.1.1 Let $f_M^{(T)}$ be the function computed by the FSM $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ in $T$ steps, where $\delta$ and $\lambda$ are the binary next-state and output functions of $M$. The circuit size and depth over the basis $\Omega$ of any function $f$ computed by $M$ in $T$ steps satisfy the following inequalities:

\[
C_{\Omega}(f) \leq C_{\Omega} \left( f_M^{(T)} \right) \leq TC_{\Omega}(\delta, \lambda)
\]

\[
D_{\Omega}(f) \leq C_{\Omega} \left( f_M^{(T)} \right) \leq TD_{\Omega}(\delta, \lambda)
\]

- First inequality above says that “the number of equivalent logic operations performed by an FSM to compute a function $f$ must be at least the minimum number of gates necessary to compose $f$ with a circuit”
- Place upper limits on the size and depth complexities of functions computable in a bounded number of steps by an FSM
Circuits are Universal for Bounded FSM Computations

- Next result identifies logical circuits with bounded-time FSMs

**THEOREM 3.1.2** Every subfunction of the function $f^{(T)}_M$, computable by an FSM on $n$ inputs, is computable by a Boolean circuit and vice versa.

- One direction ($\Rightarrow$) is implied by previous discussion (cf. Fig. 3.3)

![Image](image.png)

**Figure 3.3** A circuit computing the same function, $f^{(T)}_M$, as a finite-state machine $M$ in $T$ steps.

Circuits are Universal for Bounded FSM Computations (cont.)

**THEOREM 3.1.2** Every subfunction of the function $f^{(T)}_M$, computable by an FSM on $n$ inputs, is computable by a Boolean circuit and vice versa.

- Other direction ($\Leftarrow$) is established by considering a “binary tree” transition diagram describing an FSM with the start state as root and branching into two states at each node to represent transition in response to two possible input values. If there are $n$ inputs, then the transition diagram would have $n$ levels and at most $2^{n+1} - 1$ states.

![Image](image.png)
Circuits are Universal for Bounded FSM Computations (cont.)

**THEOREM 3.1.2** Every subfunction of the function \( f_M^n \), computable by an FSM on \( n \) inputs, is computable by a Boolean circuit and vice versa.

- Example:

![Image of FSM](image)

*Figure 3.4* A fifteen-state FSM that computes the EXCLUSIVE OR of three inputs as a subfunction of \( f_M^3 \), obtained by deleting all outputs except the third.

Interconnections of Finite-State Machines

- Synchronous FSMs read inputs, advance from state to state, and produce outputs in synchrony
- Can achieve greater computational power and flexibility by interconnecting synchronous FSMs, for example

![Image of FSM interconnection](image)

*Figure 3.5* The interconnection of two finite-state machines in which one of the three outputs of \( M_1 \) is supplied as an input to \( M_2 \) and two of the three outputs of \( M_2 \) are supplied to \( M_1 \) as inputs.
Interconnections of Finite-State Machines (cont.)

- Analogous inequality results for interconnected FSMs:

**THEOREM 3.1.3** Let $f_{M_1 \times M_2}^{(T)}$ be a function computed in $T$ steps by a pair of interconnected synchronous FSMs, $M_1 = (\Sigma_1, \Psi_1, Q_1, \delta_1, \lambda_1, s_1, F_1)$ and $M_2 = (\Sigma_2, \Psi_2, Q_2, \delta_2, \lambda_2, s_2, F_2)$. Let $\delta_1$, $\lambda_1$, $\delta_2$, and $\lambda_2$ be the size and depth of encodings of the next-state and output functions. Then, the circuit size and depth over the basis $\Omega$ of any function $f$ computed by the pair $M_1 \times M_2$ in $T$ steps (that is, a subfunction of $f_{M_1 \times M_2}^{(T)}$) satisfy the following inequalities:

$$C_{\Omega}(f) \leq T[C_{\Omega}(\delta_1, \lambda_1) + C_{\Omega}(\delta_2, \lambda_2)]$$
$$D_{\Omega}(f) \leq T[\max(D_{\Omega}(\delta_1, \lambda_1), D_{\Omega}(\delta_2, \lambda_2))]$$

- Proof is immediate from the definition of synchronous FSMs and previous inequality results

Nondeterministic Finite-State Machines

- Potentially more general FSM model obtained from “normal” FSMs (which are deterministic) by allowing state transition relations rather than strict functions

- The equivalent of being able to “guess” the correct state transition in order to arrive eventually (if at all possible) at an accepting state – nondeterministic FSM or NFSM
Nondeterministic Finite-State Machines (cont.)

- NFSMs can be thought of as deterministic FSMs with additional input – choice input – in addition to standard input.
- Choice agent supplies the choice input and disambiguates state transition whenever nondeterminism is involved.

Nondeterministic Finite-State Machines (cont.)

- NFSM that accepts strings that are binary representations of odd integers (e.g., 1, 11, 101, 111, 1001, …):

- Equivalent DFSM:
Nondeterministic Finite-State Machines (cont.)

- NFSMs and DFSMs recognize the same class of languages – non-determinism does not make an FSM more powerful.
- For Turing Machines (TMs), the difference between the power of deterministic vs. non-deterministic is not known – at the heart of the $P \neq NP$ issue.

\[ P \quad \text{NP} \quad \text{vs.} \quad P = NP \]

Designing Sequential Circuits

- **Sequential circuits** – circuits constructed from logic circuits and clocked (binary) memory devices.
  - Inputs to logic circuit may be external input and/or outputs from binary memory units.
  - Example has one EXCLUSIVE OR gate and one clocked memory unit which feeds its output back to the gate.
  - One external input provides other input to the EXCLUSIVE OR gate.

Sketch a procedure to design sequential circuits that realize any FSM – show relationship between FSMs and logic circuits.
Designing Sequential Circuits (cont.)

- **First step**: assign unique binary tuples to each FSM input symbol, output symbol, and state
- **Second step**: tables for the next-state function $\delta$ and output function $\lambda$ are produced based on FSM description and binary encoding of first step
- **Third step**: logical circuits are designed to realize the binary functions of second step
- **Fourth step**: logical circuits are connected to clocked binary memory devices to produce the desired sequential circuit realizing the FSM

---

**Example**: two-bit adder (*cf.* Fig. 2.14)

- **First step**: assign unique binary tuples to input symbols, output symbols, and states

<table>
<thead>
<tr>
<th>$q$</th>
<th>$c$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$q_2$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$q_3$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example: two-bit adder (cf. Fig. 2.14)

Second step: tables for the next-state function \( \delta \) and output function \( \lambda \) are produced

\[
\begin{align*}
\delta(q_0,00) &= q_0 \\
\delta(00,00) &= 00 \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>c</th>
<th>s</th>
<th>u</th>
<th>v</th>
<th>c'</th>
<th>s'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Designing Sequential Circuits (cont.)

**Example:** two-bit adder

Third & fourth steps: build logical and sequential circuits

<table>
<thead>
<tr>
<th>$\delta : B^4 \rightarrow B^2$</th>
<th>$\lambda : B^2 \rightarrow B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>$s$</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$1$</td>
<td>$0$</td>
</tr>
<tr>
<td>$1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

| $0$ | $0$ | $1$ | $0$ | $1$ | $0$ | $0$ | $1$ |
| $0$ | $1$ | $0$ | $1$ | $0$ | $1$ | $1$ | $0$ |
| $1$ | $0$ | $0$ | $1$ | $1$ | $0$ | $0$ | $1$ |
| $1$ | $1$ | $0$ | $1$ | $1$ | $0$ | $1$ | $0$ |
| $0$ | $0$ | $1$ | $1$ | $1$ | $0$ | $0$ | $1$ |
| $1$ | $0$ | $1$ | $1$ | $1$ | $0$ | $1$ | $1$ |

Random Access Machines

**RAMs** – model essential features of traditional serial computers

- Composed of two synchronous interconnected FSMs
  - A CPU that has a modest number of local registers (memory device); and
  - A random-access memory that has a large number of local registers

Random-Access Memory
RAM Architecture

- **CPU** implements a *fetch-and-execute cycle*, alternatively
  - Retrieving an instruction from memory
    (stored-program concept, cf. von Neumann); and
  - Executing the retrieved instruction
  - Instructions are of the following five categories:
    - Arithmetic/logical instructions
    - Memory load/store instructions
    - Jump instructions for breaking normal sequential “flow of logic”
    - I/O instructions
    - Halt instruction

- **Random-Access Memory** allows for equal-time access to any of its component registers
  - One output word (*out_wrd*)
  - Three input words: an address (*addr*), a data word (*in_wrd*), and a command (*cmd*)

The RAM as FSM

- **The CPU** is an FSM –
  - receives input from the random-access-memory and external sources
  - sends output to random-access-memory and output port
  - its state is a function of the content of its registers
The RAM as FSM (cont.)

- The **random-access memory** is an FSM –
  - receives input from the CPU
  - sends output to the CPU
  - its state is a function of the content of its registers

### RAM Programs

- Expressed as **assembly-language programs** to make them more readable (use of mnemonics and labels instead of bit patterns)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC ( R_i )</td>
<td>Increment the contents of ( R_i ) by 1.</td>
</tr>
<tr>
<td>DEC ( R_i )</td>
<td>Decrement the contents of ( R_i ) by 1.</td>
</tr>
<tr>
<td>CLR ( R_i )</td>
<td>Replace the contents of ( R_i ) with 0.</td>
</tr>
<tr>
<td>( R_i \leftarrow R_j )</td>
<td>Replace the contents of ( R_i ) with those of ( R_j ).</td>
</tr>
<tr>
<td>JMP_A ( N_i )</td>
<td>Jump to closest instruction above current one with label ( N_i ).</td>
</tr>
<tr>
<td>JMP_B ( N_i )</td>
<td>Jump to closest instruction below current one with label ( N_i ).</td>
</tr>
<tr>
<td>( R_j )</td>
<td>If ( R_j ) contains 0, jump to closest instruction above current one with label ( N_i ).</td>
</tr>
<tr>
<td>( R_j )</td>
<td>If ( R_j ) contains 0, jump to closest instruction below current one with label ( N_i ).</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Continue to next instruction; halt if none.</td>
</tr>
</tbody>
</table>
RAM Programs (cont.)

- Sample **assembly-language programs**, one for a simple addition, and the other for squaring a given value

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>N0</td>
<td>R1 JMP -</td>
<td>N1</td>
</tr>
<tr>
<td></td>
<td>INC R0</td>
<td></td>
</tr>
<tr>
<td>DEC R1</td>
<td></td>
<td>Decrement R1</td>
</tr>
<tr>
<td>JMP + N0</td>
<td></td>
<td>Repeat</td>
</tr>
<tr>
<td>N1</td>
<td>CONTINUE</td>
<td></td>
</tr>
</tbody>
</table>

- Second program makes use of first program as a “subroutine”

Universality of the RAM

- Define the notion of “universality”

**Definition 3.4.1** A machine $M$ is **universal** for a class of machines $C$ if $M$ is in $C$ and every machine in $C$ can be simulated by $M$.

- Universality theorem for RAMs vs. FSMs

**Theorem 3.4.1** Every $T$-step FSM $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ computation can be simulated by a RAM in $O(T)$ steps with constant space. Thus, the RAM is universal for finite-state machines.

- Proof sketch:
  - FSM is completely defined by its next-state and output functions $\Rightarrow$ just need to write a fixed-length RAM program implementing both functions and recording the output and state values in the RAM memory
  - RAM program is run repeatedly to simulate FSM function
  - RAM space required is fixed: storage for RAM simulation programs for $\delta$ and $\lambda$, plus register to store FSM state
Universality of the RAM (cont.)

- RAMs can simulate FSMs
- Since RAM components (CPU and bounded random-access-memory) are themselves FSMs, a RAM can simulate any other RAM
- Another “universality” result: RAMs can execute RAM programs
  - Two “flavors” of RAM to execute RAM programs:
    - RAM program is stored in registers specially allocated to the RAM program (loaded onto CPU)
    - RAM program is stored in registers of the random-access-memory (RASP model)
  - For later discussion (if time permits)

Computational Inequalities for the RAM

- Use prior inequalities for FSMs since bounded-memory RAM components are themselves FSMs
- Use results from Sec. 3.10.6 and Sec. 3.5 (not discussed) to obtain the following results:

**THEOREM 3.6.1** Let \( f \) be a subfunction of \( f_{\text{RAM}}^{(T,m,b)} \), the function computed by the \( m \)-word, \( b \)-bit RAM with storage capacity \( S = mb \) in \( T \) steps. Then the following bounds hold simultaneously over the standard basis \( \Omega_0 \) for logic circuits:

\[
\begin{align*}
C_{\Omega_0}(f) &= O(ST) \\
D_{\Omega_0}(f) &= O(T \log S)
\end{align*}
\]

- The quantity \( ST \) is again pivotal here – another incarnation of the “space-time” tradeoff