CSCI 1800 Cybersecurity and International Relations

Computer Hardware & Software

John E. Savage

Brown University
Overview

• Short history of computers
• Architecture of CPU and RAM
• Simple assembly-level program
• Assembler and compiler
• Brief intro to operating systems
• Operating system services
  – Logging, password authentication and attacks
• Application security
  – Buffer overflow and how to prevent it.
The Computer – The Big Picture

• A computer takes input, produces output, potentially storing data in the process.

• Computers can be configured to perform many tasks, i.e. they are programmable.
Binary Numbers

• Today computers store data as bits, 0s and 1s.
• Addition/subtraction done with binary numbers

• Here is a mapping of decimal to binary numbers

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary (3 bits)</th>
<th>Binary (8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>00000010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>00000011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00000100</td>
</tr>
</tbody>
</table>
Computer Architecture

- Computers have central processing units (CPUs) and random access memories (RAMs).

- Programs stored in the RAM direct the action of the CPU.
- The CPU implements the **fetch-execute cycle**
  - Fetch an instruction and execute it.
Random Access Memory (RAM)

- A **word** is a collection of bits, typically 32 or 64 bits.
- RAM is given command \{**read**, **write**, **no-op**\}, input **word**, and an **address**. It produces an **output word**.

1. **read** changes **out_wrd** to word at address **addr**.
2. **write** replaces word at address **addr** with **in_wrd**.
3. **no-op** makes no changes.
Central Processing Unit (CPU)

• **Fetch-execute cycle**: Repeat *forever*: a) read an instruction from the RAM and b) execute it.

• **Typical instructions**: ADD, SUBTRACT, SHIFT LEFT, MOVE, COMPARE, JUMP, JUMPC (conditional), READ, WRITE

• A **program** is a set of instructions.

• A **register** is a storage location

• CPU has **registers**, e.g. reg_a, that hold temporary results.
  - Permanent results stored in the RAM
Architecture of the CPU

• CPU has program counter (PC or prog_ctr) holding location of next CPU instruction.
• Instructions normally executed from sequential RAM locations (e.g. PC,PC+1,PC+2)
• To jump to new instruction, PC changed to new address.
  – JUMP and JUMPC (Jump Conditional)
Addition

• The instruction ADD A B adds the contents of registers A and B and puts the sum into A.

• E.g. Consider addition of 3 \(_{10} = 011 \_2\) and 5 \(_{10} = 101 \_2\).

\[
\begin{align*}
011 & = 0\times2^2 + 1\times2^1 + 1\times2^0 \\
101 & = 1\times2^2 + 0\times2^1 + 1\times2^0 \\
1000 & = 1\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0 \\
\end{align*}
\]

• Subtraction (SUB) implemented by adding a negative number.
COMPARE and MOVE Instructions

• **COMPARE A B** compares contents of registers A and B and sets $\text{CMP} = 1$ (0) if same (different).

• **MOVE A B** moves the contents of register (or location) A to that of register (or location) B.

• **SHIFT_LEFT A** shifts contents of register reg left one place, shifting 0 in from right. Discards bit on left.

  • $\text{SHIFT_LEFT }1011 = 0110$
  • $\text{SHIFT_LEFT }0110 = 1100$

• **SHIFT_LEFT** can be used to multiply two numbers.
JUMP Instructions

• Recall: program counter PC is a CPU register with the address of the next instruction in memory.

• Normally PC is incremented on each step.
  – That is, the next instruction in memory is executed.

• The JUMP and JUMPC (conditional) instructions change the contents of the PC.
  – The CPU jumps to a new point in the program.
  – JUMP A sets PC to A.
  – JUMPC A sets PC to A if compare bit CMP is 1.
    • JUMPC implements a conditional branch.
Multiplication

• To multiply integers \(x\) and \(y\), set \(u = 0\), \(z = 0\) then add \(y\) to \(z\) for \(x\) times.
  
  – E.g. \(x = 3\), \(y = 2\), \(u = 0\), \(z = 0\).

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>MEANING</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ADD (y) (z)</td>
<td>Add (y) to (z)</td>
<td>((z = 0) initially)</td>
</tr>
<tr>
<td>2. SUB (x) 1</td>
<td>Subtract 1 from (x)</td>
<td></td>
</tr>
<tr>
<td>3. COMPARE (x) (u)</td>
<td>Compare (x) to (u)</td>
<td>((u) remains at 0)</td>
</tr>
<tr>
<td>4. JUMPC 6</td>
<td>If (x = 0), jump to instruction 6.</td>
<td></td>
</tr>
<tr>
<td>5. JUMP 1</td>
<td>Otherwise (if (x \neq 0)), go to instr #1</td>
<td></td>
</tr>
<tr>
<td>6. DONE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multiplication Example

1. ADD $y$ $z$
2. SUB $x$ 1
3. COMPARE $x$ $u$
4. JUMPC 6
5. JUMP 1
6. DONE

<table>
<thead>
<tr>
<th>Prog Step</th>
<th>$u$</th>
<th>$y$</th>
<th>$z$</th>
<th>$u$</th>
<th>CMP</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Initial conditions</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>-</td>
<td>ADD $y$ to $z$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>-</td>
<td>Decrease $x$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>CMP set to 0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>No jump</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Jump to step 1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>ADD $y$ to $z$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>Decrease $x$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>CMP set to 0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>No jump</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>Jump to step 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>ADD $y$ to $z$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>Decrease $x$</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>CMP set to 1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>Jump to step 6</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>Done $x \times y = 6!$</td>
</tr>
</tbody>
</table>
External CPU Connections

• **READ** reads from external input, e.g. keyboard
• **WRITE** writes to external output, e.g. display
• These commands require a way to specify which device is connected to the CPU.
Assembly Language

• **CPU instructions are bit sequences**
  – Some bits represent an **opcode**
  – Others represent DATA, e.g. **addresses** and **values**.
  – E.g. \((\text{OPCODE}, \text{DATA}) = (01011, 01110010010)\)

  – In early computers, words had 16 bits
  – Modern computers operate on 32-bit words, though 64-bit words are becoming more common.
Assembly Language

- **Assembly languages** substitute mnemonics, e.g. ADD, SHIFT LEFT, MOVE, COMPARE, JUMPC for bits

- An **assembler** is a program that translates assembly language programs into binary CPU instructions.
Compilers

• High level languages are more expressive than assembly level languages.
  – Fewer statements, each having more meaning.

• Compilers are programs to translate high level languages into machine level languages.
Operating Systems and Applications
What is an Operating System (OS)?

- An OS is software that sits between computer hardware & the applications.
- The hardware consists of the CPU, small memories (caches), RAM, disks, the keyboard, displays, sound system, printers, card readers, USB, etc.
- An OS allocates system resources fairly and contains software to run hardware.
Services Offered by an OS

• **Multitasking:**
  – A CPU is shared between multiple tasks.

• **Interrupts:**
  – A task may be interrupted to handle a more urgent one.

• **File system:**
  – Programs and data are organized as files and folders.

• **Protection/Security**
  – Files have access rights (e.g., read (r), write (w), execute (x)).
    • Eg `rwxr-xr--` means **owner** has r, w, x; **group** has r, x; **world** has r only
  – OS ensures access rights are observed, tasks are separated
The Process Model

• A **process** is a running program.
  – It has instructions, data, and **temporary work space**.

• Processes typically spawn sub-processes to carry out subtasks, forming a tree.
  – Subtask returns values to parent, then halts.
  – Sub-process usually has same access rights as parent.

• Inter-process communication:
  – A variety of methods exist to communicate between processes, including from one computer to another.
Program Memory Management

- For execution a program allocates temporary space to run and communicate.
  - *Text* holds the program.
  - *Data* holds initial values.
  - *Vars* for variables without initial values.
  - *Heap* holds space allocated by program.
  - *Stack* holds *activation records* that allow a program to switch to a sub-program.
Calling Sub-Programs

- A sub-program $S$ (aka subroutine, procedure, object) has a name and parameters, e.g. $\text{square}(A,B)$.

- When program $P$ calls subprogram $S$, $P$ halts, and $S$ starts.
- When $S$ ends, $P$ is restarted using $S$’s results.

- OS must know where to restart $P$ when $S$ is done.

- When $P$ “calls” $S$, OS puts an activation record $AR$ on the stack showing where to restart $P$ with $S$’s results.
- The OS “pops” $AR$ from the stack when $S$ finished.

- If $AR$ is changed by attacker, his/her program starts
## Calling a Sub-Program

<table>
<thead>
<tr>
<th>Line No.</th>
<th>Statement</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Hypotenuse</strong> (a, b){</td>
<td>// Given a and b, compute c</td>
</tr>
<tr>
<td>2</td>
<td>asquare = a * a;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>bsquare = b * b;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>sum = asquare + bsquare;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c = sqrt(sum);</td>
<td>// Push return address 6 on stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Jump to program <strong>sqrt</strong>, put c on stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// On completion of <strong>sqrt</strong>, pop address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// and resume <strong>hypotenuse</strong> at address 6.</td>
</tr>
<tr>
<td>6</td>
<td>return c;</td>
<td></td>
</tr>
</tbody>
</table>

![Pythagorean Theorem Diagram](image)
Memory Hierarchies

- Memory organized into registers, cache memory, RAM, disk, CDs, tape, etc.
- As we move up hierarchy, memories get slower and larger.
- **Memory hierarchy is designed to simulate one large, fast memory.**
Virtual Memory

- **Virtual memory** simulates large real memory.
- Programmer sees a contiguous memory. It is actually split into pieces by the OS.
Operating System Kernel

• The OS **kernel** performs basic OS functions
• Simplifies use of devices
  – “Drivers”, code to access devices, supplied by vendor
  – Drivers are integrated into OS for efficiency, thereby increasing risk
An Early Boot Program

• On DEC PDP-1 (1960) memory initially empty. First few instructions must be entered via toggle switches.
Operating System Kernel

• Kernel contains BIOS (basic input/output system), a tiny program that loads (boots) kernel at startup.
  – BIOS is frequent target of attacks.
  – If BIOS corrupted, very difficult to discover and fix.
Logging

• An OS will log certain types of event, such as
  – User authentications
  – Attempts to violate security policies
  – Unexpected exits from programs
  – Outward network connections made by processes

• OS logging is valuable in maintaining security.
• Logs record status of an OS (forensic analysis)
Password Authentication

• Users are typically identified by passwords.

• Computer stores hash or compression of each password, not the word itself.
  – A hash function is easy to compute, difficult to invert.

• Password file contains pairs (ID, hash(psswrd)), e.g. such as (jes, x^sr$1); (avd, ysae)
Password Authentication

• In a dictionary attack build a lookup table that contains the hash of each word in a dictionary.
  – Table entries = (hash(word), word) for words in dictionary

• Given a hash from OS password file, find a user’s password by finding the hash in this table!

• How can we thwart a dictionary attack?
Preventing a Dictionary Attack

• To make such attacks harder, a word (salt) is concatenated with a password before hashing.
  – E.g. if salt=#6a, compute $hash(#6a\text{password})$
  – Store [ID, salt, $hash(salt\text{password})$]
  – salt is not secret but is different on each computer

• If attacker knows $hash$ in advance but not salt, must do a full search once salt is discovered.

• Can the attack be made even harder?
  – What if hash function takes long time to compute?
Application Security Problems

• **Privilege escalation**
  – Occurs when program at security level A is able to run at level B, B > A, giving it unauthorized access.

• **Buffer Overflow**
  – Programmer allocates memory for N data items
  – But program allows more than N items to be stored in memory
  – Consequence: *data is unintentionally overwritten*
  – This is a programming error!
Buffer Overflow

• An important and common vulnerability

Normal Execution

After Buffer Overflow
Three Protections Against Buffer Overflow

1. Insert known **canary** string. If overwritten, overflow detected.

2. Don’t let instructions execute from within stack. Called **Data Execution Protection (DEP)**

3. Change location of the stack and libraries in each computer. Called **Address Space Layout Randomization (ASLR)**
   
   • These are very good defenses!
Virtual Machines (VMs)

• A VM is a program that simulates an OS.
  – Applications are run using copies of simulated OS
  – If simulated OS is corrupted, real OS is not affected.

• A VM is called **sandbox** – isolates an application

• *Cloud computing* consists of thousands of computers each running multiple VMs.
  – Leads to great efficiencies. Few idle cycles.
Review

- Short history of computers
- Architecture of CPU and RAM
- Simple assembly-level program
- Assembler and compiler
- Brief intro to operating systems
- Operating system services
  - Logging, password authentication and attacks
- Application security
  - Buffer overflow and how to prevent it.