Overview

• Introduction computer design
• Central Processing Unit (CPU)
• Random Access Memory (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>Dec</th>
<th>x_2x_1x_0</th>
<th>( x_2 \times 4 + x_1 \times 2 + x_0 \times 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>( 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 )</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>( 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 )</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>( 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 )</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>( 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 )</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>( 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 )</td>
</tr>
</tbody>
</table>
Computer Architecture

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

- The RAM holds programs/data that direct action by 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 given command {read, write, or no-op}, input word, and an address. It produces an output word.
  • read – copies word at address $addr$ to $out\_wrd$
  • write – replaces word at address $addr$ with $in\_wrd$.
  • no-op – makes no changes.
Central Processing Unit (CPU)

- **Fetch-execute cycle**: Repeat forever:
  1. read an instruction from the RAM and
  2. 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 \((\text{prog}_\text{ctr})\).
- It holds address \((\text{PC})\) of next CPU instruction.
- Instructions normally executed from increasing RAM locations (e.g. PC, PC+1, PC+2).
- A \textbf{jump} to new instruction changes PC to a new address, say PC*.
- Jump conditioned on a \textbf{bit}, JUMPC, and unconditional, JUMP.
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 $5_{10} = 101_2$.

  
  \[
  \begin{align*}
  011 &= 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
  101 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\
  1000 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 8_{10}
  \end{align*}
  \]

• Subtraction (SUB) implemented by adding a negative number.
Multiplication Example

• In decimal notation: $11_{10} \times 23_{10}$

\[
(1 \times 10 + 1 \times 1) \times (2 \times 10 + 3 \times 1) = \\
(1 \times 10 + 1 \times 1) \times (2 \times 10) + (1 \times 10 + 1 \times 1) \times 3 = \\
(1 \times 10 + 1 \times 1) \times (2 \times 10) + (3 \times 10 + 3 \times 1) = \\
(1 \times 100 + 1 \times 10) \times 2 + (3 \times 10 + 3 \times 1) = \\
(2 \times 100 + 2 \times 10) + (3 \times 10 + 3 \times 1) = \\
(2 \times 100 + 5 \times 10 + 3 \times 1) = 253_{10}
\]
COMPARE and MOVE Instructions

• **SHIFT_LEFT A** shifts contents of register rega 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.

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

• **MOVE A B** moves the contents of register (or location) A to register (or location) B.
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 = 1
    • JUMPC implements a conditional branch.
An Example Multiplication Algorithm

- To multiply $x$ and $y$, set $u = 0$, $z = 0$. $z$ will be the product of $x$ and $y$. Method: **add $y$ to $z$ for $x$ times**
  - E.g. $x = 3$, $y = 2$, $u = 0$, $z = 0$. $z = y + y + y = 6$

### ALGORITHM

<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>(initially $z = 0$)</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>(Is $x = u = 0$?)</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**

Changed values are **red**

<table>
<thead>
<tr>
<th>Prog Step</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>u</th>
<th>CMP</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Initial conditions</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</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*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

• CPU only executes a READ when an external device places new data into input register. The device notifies the CPU by interrupting it.
  – An interrupt causes the CPU to request that the operating system determine source of interrupt.
Assembly Language

• CPU instructions are sequences of bits
  – Some instruction bits represent an opcode
  – Others represent DATA, e.g. addresses and values.
  – E.g. (OPCODE, DATA) = (01011, 01110010010)
    • The comma is used for clarity. It is not part of instruction.
  – 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 and 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 may spawn sub-processes to carry out subtasks, forming a tree.
  – Subtask returns values to parent, then halts.

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

• Memory holds words at consecutive locations
• For execution a program allocates temporary space to run and communicate.
  – **Text** holds the program.
  – **Data** holds values of initialized variables.
  – **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. $S = \text{square}(A,B)$.

- 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” or removes AR from the stack when $S$ finished.

- If an attacker changes AR, the attacker’s program starts, not $S$
### Calling a Sub-Program

#### Hypotenuse

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

![Diagram of a right triangle]
Memory Hierarchies

• Memory organized into registers, cache memory, RAM, disk, CDs, tape, etc.
• As we move up hierarchy, memories get slower and larger. Registers are fast.
• 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.

Program Sees:  
Actual Memory:  

![Diagram showing virtual memory concept]
Operating System Kernel

• The OS **kernel** performs basic OS functions
• Simplifies use of devices
  – “Drivers” – software to access devices, e.g. disks.
  – Drivers often supplied by a device vendor.
  – Drivers are integrated into OS for efficiency, thereby **introducing risk**.
  – **Risk occurs** because vendor might not use the same security standards as OS provider.
An Early Boot Program

- First few instructions are entered manually via toggle switches on the front panel!
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 is 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.
  – 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

• A **dictionary attack** builds a **lookup table** that contains the hash of each word in a dictionary.
  – Table entries = (hash(word), word) for word 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 $hash(#6apassword)$
  – Authentication table stores [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 a long time to compute?
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
  – Program allows more than N items to be put in memory
  – System data, e.g. AR, may be overwritten or changed
  – This could give control to attacker.
  – Buffer overflow vulnerability is a very serious error!
Buffer Overflow

Normal Execution

After Buffer Overflow

Stack

return address

buffer

return address

buffer

Data

Program Code

Stack

malicious code

return address

overwritten

Data

Program Code
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. Place stack and libraries at different location in each computer. Called **Address Space Layout Randomization (ASLR)**
   - These are very good defenses!
Virtual Machines (VMs)

• **A VM is program that simulates a real machine.**
  – Applications are run on copies of a 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.