Appendix A: Instruction Set Principles
Chapter Overview

- Classification
- Memory Addressing
- Operand Types
- Basic Operations
- Encoding Instruction Sets
- The Role of Compilers
- Putting it all together: The RISC-V Architecture
This illustration is a common presentation of the application/operating system/hardware components. It can be deceiving in that it seems to show that the application runs through the O/S to the hardware layer. In order to gain a clear view of system operation, it is first necessary to break this view and remember that all of the application and operating system services run through the system CPU/Core one at a time. We will restructure this view in the next few slides.

Recall that the hardware control unit is configured to control access to certain resources (instructions, registers, memory, devices, etc) based on some configuration data regarding the program/task that it is currently executing. Generally this means that the system executes in either user mode (where some accesses are not allowed) or privileged mode (where access to all system resources is allowed). This configuration data is generally set when the task is scheduled into the CPU/Core and generally resides in one or more CPU (not general purpose) registers.

Before we examine further the system level view, it may be instructive to first review the operation of the control unit and execution of programs/tasks within the CPU/Core.
Consider the operation of the control unit as a sequential algorithm (see pseudo-code). This example highlights two locations where hardware interrupts continued interpretation of instruction execution. The first occurs when an attempt is made to access a protected resource when operating in user mode. The second trap is a timeout trap that occurs when the system is running in user mode and the timer reaches zero. This timeout trap is used to facilitate time sharing of the CPU/Core by the running programs. There are numerous other traps in the system but these will suffice to illustrate our needs.

```
while true do
    instrRegister = MM[PC];
    PC++;
    decode instrRegister;
    if (executing in user mode and a privileged resource is being accessed) then
        trap the program/terminate execution;
    else
        allow the instruction to be executed;
    end
    if (executing in user mode and timeout has occurred) then
        trap the program/context swap;
    end
end
```

One other comment that needs to be made here is with respect to virtual memory. With virtual memory, the system remaps the addresses from the virtual (program) space to the physical space where the O/S places the program page. Thus, the hardware will also have to support traps (virtual memory faults) when the mapping of these spaces has not already been performed (by the O/S). Furthermore, if you think about it, you will realize that when the user mode program/task accesses memory, it is accessing a privileged resource that the system must protect. Since a running program has a large number of memory accesses, we cannot have the O/S services process all of these requests. Thus, we migrate the address translation process into the hardware so that user programs can run most efficiently and yet the shared hardware resources (memory) are protected from errant (or malicious) access by non-privileged tasks. Similar techniques are used in virtualization for performance.

All tasks, including operating system tasks run sequentially, one at a time, through the CPU/Core. When a user mode task requests access to a privileged service, the hardware prevents it from doing so directly. When the trap occurs, the operating system interrupt service routines will look for the reason for the trap and service the request on behalf of the user task that initiated the request (provided the O/S approves/allows the request to be performed).
Contrasting the conventional view (shown in a previous slide), the accompanying figure restructures the relation of the operating system tasks and applications to the executing hardware platform (CPU/Core). All of the O/S tasks and all of the application programs are actually executed through the CPU/Core one at a time in a time-shared manner. In general all of the application programs are executed in user mode and most (but not necessarily all) operating system tasks are run in privileged mode.

Now from the perspective of virtualization, one possible organization is to simply add another runtime layer/mode to the hardware platform (hypervisor mode), add a box for the virtual machine state (interrupts, etc) and add another box containing the hypervisor (or Virtual Machine Monitor, VMM) to our picture:

BTW: multiple cores do not really change the picture much. You simply have another hardware resource that can simultaneously process another instruction stream. The O/S just has to schedule the Core (of course there are shared memory issues, to be addressed, but that’s another topic altogether).
Classification

- General-purpose register (GPR) architectures
  - Register-memory architectures
  - Load-store architectures
- Stack architectures
- Memory-memory architectures
- Accumulator architecture
Memory Addressing

- Little Endian (word address = rightmost byte address)
- Big Endian (word address = leftmost byte address)
- The significance of alignment
The book begins to really open up and clearly show examples of quantitative measures to support design decisions. Examine the graphs of quantitative measures and corresponding prose closely. Establish this habit for the remainder of your studies in this textbook.
Frequency of Addressing Modes

- **TeX**
  - Memory indirect: 1%
  - Scaled: 0%
  - Register indirect: 3%
  - Immediate: 24%
  - Displacement: 32%

- **spice**
  - Memory indirect: 6%
  - Scaled: 16%
  - Register indirect: 11%
  - Immediate: 39%
  - Displacement: 40%

- **gcc**
  - Memory indirect: 1%
  - Scaled: 6%
  - Register indirect: 11%
  - Immediate: 43%
  - Displacement: 55%
Distance of Operand Displacements

Percentage of displacement

Number of bits of displacement

Floating-point average

Integer average
Frequency of Immediates

- **Loads**
  - Floating-point average: 22%
  - Integer average: 23%

- **ALU operations**
  - Floating-point average: 19%
  - Integer average: 25%

- **All instructions**
  - Floating-point average: 16%
  - Integer average: 21%
Size of Immediates

- Percentage of immediates:
  - 0% to 45% intervals

- Number of bits needed for immediate:
  - Floating-point average
  - Integer average

- Graph shows the distribution of immediates by percentage and the number of bits needed for each.
## Top 10 Instructions for 80x86

<table>
<thead>
<tr>
<th>Rank</th>
<th>80x86 instr</th>
<th>Frequency (% total executed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move (reg-to-reg)</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
</tr>
</tbody>
</table>
Control Flow Instructions

Frequency of branch instructions

- Call/return: 8% (Floating-point average) - 19% (Integer average)
- Jump: 10% (Floating-point average) - 6% (Integer average)
- Conditional branch: 82% (Floating-point average) - 75% (Integer average)
Bits of Branch Displacement

- Integer average
- Floating-point average

Percentage of distance vs. Bits of branch displacement

0% 1% 2% 3% 4% 5% 6% 7% 8% 9% 10% 11% 12% 13% 14% 15% 16% 17% 18% 19% 20%
Relational Tests in Conditional Branches

- **Not equal**: 5% (Floating-point average), 2% (Integer average)
- **Equal**: 16% (Floating-point average), 18% (Integer average)
- **Greater than or equal**: 0%, 11%
- **Greater than**: 0%, 0%
- **Less than or equal**: 44%, 33%
- **Less than**: 34%, 35%

Frequency of comparison types in branches
Additional Comments

- PC relative addressing
- Indirect jumps
- Caller saving/Callee saving
- Instruction set encoding: 16-bit instrs vs compression
  - impact on caches
The Role of Compilers

- Optimization (multi-pass [phases])
- Effective optimization has dramatic impact: 25%-90%
- Register allocation
- How to help the compiler writer:
  - Provide regularity
  - Provide primitives not solutions
  - Simplify trade-offs among alternatives
  - Provide abilities to bind quantities known at compile time
Impact of Compiler Optimization

The chart illustrates the impact of compiler optimization on various programs. It shows the percentage of unoptimized instructions executed at different levels of optimization.

- **lucas, level 3**: 11% unoptimized instructions executed.
- **lucas, level 2**: 12% unoptimized instructions executed.
- **lucas, level 1**: 21% unoptimized instructions executed.
- **lucas, level 0**: 100% unoptimized instructions executed.
- **mcf, level 3**: 76% unoptimized instructions executed.
- **mcf, level 2**: 76% unoptimized instructions executed.
- **mcf, level 1**: 84% unoptimized instructions executed.
- **mcf, level 0**: 100% unoptimized instructions executed.

The bars represent different types of operations:
- Branches/calls
- Floating-point ALU ops
- Loads-stores
- Integer ALU ops

The x-axis represents the percentage of unoptimized instructions executed, ranging from 0% to 100%.
The RISC-V instruction set is used throughout the textbook in examples and to discuss program structures that can impact design decisions. You will not be examined on your knowledge of the RISC-V instruction set, but familiarity with it will assist you in later chapters of this textbook.