What is an ISA?

A hardware-software interface

The instruction-set architecture (ISA) defines:

- The state of the program: Processor registers, memory.
- What instructions do: Semantics of instructions, how they update state.
- How instructions are represented: Syntax (bit encodings)

... selected so that implications of the above on hardware design/compiler design are optimal.

- Example: If the register specifier is found in different places in different instructions, what is the problem?

Why is the ISA important?

It fixes the hardware/software interface for several generations of processors.

IBM realized early the value of a fixed ISA, but “stuck” with bad decisions for long time.

ISA decisions affect—

- Memory cost of the machine
  - Short vs. long bit encodings
  - High vs. low semantic meaning per instruction
- Hardware design
  - Simple, uniform-complexity ops → efficient pipeline.

Don’t build hardware for instructions that never get used.
Compiler and programming language issues:

- How much can compiler exploit ISA to optimize performance?
- How well does ISA support high-level language constructs?

**RISC vs. CISC**

In the early days of computing, instructions were simple, and there weren’t very many different kinds.

However, architects noticed that the same sequence of operations was being performed time and again, and realized that it would be quicker to do it in hardware than in software.

Besides, it would simplify the compiler.

As the years went by, more and more functionality was moved into hardware.

In fact, the instructions became so complicated that it wasn’t cost-effective to build hardware to decode and execute them. Instead, each instruction were treated as a “micro-procedure” that was implemented in *microcode*.

By the early 1980s, most computers were microcoded and had a large instruction set (several hundred opcodes) and many different addressing modes. Observers began to argue that this was bad.

- These instructions require too much microcode.
  - The chip area devoted to control store could better be used for other purposes, e.g., cache memory.
  - A microcode bug could only be fixed by a hardware (not software) modification.
- It took too long to fetch and decode these instructions.
  - Many multiway branches are required; each takes at least a cycle—in every instruction.
  - Several fetches were required, and operands cannot be decoded until they are fetched.
• Having too many instructions complicates life for the compiler writer. The compiler writer may not have time to figure out how to use all instructions effectively.

For example, compare a register-file architecture which requires all operands to be in registers with one that also allows operands to come from memory.

This led to a move to simplify instruction sets. (See the comment on the front page of Appendix C of H&P.)

And instruction sets did become simpler for a time, but then complexity began to creep back in.

However, software is more advanced today, and that makes compromises possible.

• The x86 instruction set is very complicated, but the engines that execute it are really RISCs.

E.g., the AMD K7 takes x86 instructions and divides them into two categories (http://www.arstechnica.com/cpu/3q99/k7_theory/k7-two-1.html).
  ◦ Simple instructions, that can be decoded by hardware,
  ◦ Complicated instructions, that must be decoded by microcode.

Fortunately, the vast majority of instructions are simple instructions, so the performance hit of microcode is acceptable.

• Transmeta’s Crusoe processor (see H&P, pp. 367–370) translates x86 instructions to its own VLIW instruction set.

Initially, instructions are interpreted by software.

If a basic block is executed enough times, it can be translated into an equivalent Crusoe code sequence.

This is similar to “just-in-time” compilation, used for similar reasons in Java systems.
ISA Design Decisions & Outline

- Style of operand specification: stack, accumulator, registers, etc.
- Operand access limitations
- Addressing modes for operands
- Semantics:
  - Mix of operations supported
  - Control transfers
- Encoding tradeoffs
- Compiler influence
- Example: MIPS

Styles of instruction sets

The operands of most instructions are addresses. (Occasionally, though, an operand may be something else, like a condition code.)

An instruction may have zero, one, two, or three addresses.

**Three-address instructions**: Two source addresses and one destination address. To add R1 to R2 giving R3, we could write

\[
\text{Add } R3, R1, R2
\]

The addresses may either name registers or main-memory locations.

**Two-address instructions**: The destination address also serves as one of the source addresses. To add R1 to R2 (overwriting R2), we could write

\[
\text{Add } R2, R1
\]

To add R1 to R2 giving R3, we would need to write

\[
\text{Move } R3, R1 \\
\text{Add } R3, R2
\]
One-address instructions: Instructions implicitly reference an accumulator as the destination and one of the source operands. A load instruction copies a value from main memory into the accumulator; a store instruction copies the contents of the accumulator back to memory. To add A to B giving C, we can write

\[
\begin{align*}
\text{Load} & \quad \text{A} \\
\text{Add} & \quad \text{B} \\
\text{Store} & \quad \text{C}
\end{align*}
\]

Zero-address instructions: Most instructions implicitly reference the top of a stack. Unary operations use the top item on the stack; binary operations use the top two.

\[
\begin{align*}
\text{Push} & \quad \text{A} \\
\text{Push} & \quad \text{B} \\
\text{Add} & \quad \\
\text{Pop} & \quad \text{C}
\end{align*}
\]

In summary, here are sequences for adding A to B giving C in all four instruction styles.

<table>
<thead>
<tr>
<th>Stack (0-address)</th>
<th>Accumulator (1-address)</th>
<th>Register-memory (2-address)</th>
<th>Load-store (3-address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1, A</td>
<td>Load R1, A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1, B</td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store R1, C</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td>Store C, R3</td>
<td></td>
</tr>
</tbody>
</table>

Why stacks, accumulators

Stacks:

- Very compact format
  - All calculation operations take zero operands
  - Example use: Java bytecode (low network b/w)
• Theoretically shortest code for implementing arithmetic expressions
  All HP calculator fanatics know this

Accumulator:

• Also a very compact format
• Less dependence on memory than stack-based

For both:

• Compact implies memory efficient
• Good if memory is expensive

**Why registers?**

Faster than memory

• Latency: raw access time (once address is known)
  ◦ Cache access: 2 cycles (typical)
  ◦ Register access: 1 cycle
  ◦ Register file typically smaller than data cache.
  ◦ Register file doesn’t need tag-check logic.

• Bandwidth: more practical to multiport a register file
  ◦ Instruction-level parallelism (ILP) requires a large number of operand ports.

• Compiler considerations. Registers can hold variables. This speeds up the program and improves code density.

• ILP requirements
  ◦ Allows for concurrent execution. Suppose we have the expression
    \[(A * B) = (B * C) - (A * D)\]
    A stack instruction set makes it hard to perform the operations concurrently, since operands are hidden on the stack.
And it may have to load an operand multiple times.

- High-performance scheduling (ILP) requires detecting data dependent/independent operations early in pipeline.

- Register “addresses” are known at instruction-decode time.

Memory addresses are known quite late due to address computation.