ISA?

- ISA components:
  - Data types supported (e.g., bytes, half words, words—signed, unsigned)
  - Registers as a storage of data or information
    - General registers: e.g., R0~R7
    - Special registers: e.g., PSR (processor status register), return address from exception, ...
  - Processor modes
    - User mode, privileged mode, ...
  - Register view in different modes
  - Instruction definitions
    - Basic semantics: e.g., add, multiply
    - Exception behaviors: e.g., load – misaligned access, TLB miss, ...
    - Some instructions may behave differently in different modes
    - Some instructions are not available in user mode (e.g., privileged instructions)
    - ...
  - ABI (application binary interface)
    - ISA
    - Defines low-level binary interface between an application and the OS
      - Calling convention
      - System call mechanism
      - Binary format, ...

ISA design considerations

- Target application
  - General-purpose processor
  - Application-specific processor

- Roadmap

- Properties generally considered desirable
  - Completeness
  - Orthogonality
  - Regularity and simplicity
  - Compactness–code size

- Ease of programming
- Ease of implementation
- Ease of extension
Machine instructions

- Operation – “what” to do?
  - How many operations?
  - What kind of operations?
    - Early processors omitted “multiply”
    - Floating-point operations were also absent
- Operands – what do we operate on?
  - How many operands?
  - Where are they stored?
  - How to specify?
- Format
  - Fixed N byte or variable size
  - What are necessary subfields?
  - How are subfields laid out?

Operand location

- Stack
  - Implicit – use data on the top of the stack
- Accumulator
  - Implicit – there is one accumulator
- Register
- Memory

Number of operands

- Stack
  - 0 address
  - add
  - tos ← tos + next
- Accumulator
  - 1 address
  - add A
  - acc ← acc + mem[A]
- Register/memory
  - 2 addresses
  - add A B
  - A ← A + B
  - 3 addresses
  - add A B C
  - A ← B + C
- Load store architectures disallow operation and memory access in a single instruction
Sample code sequence

\[ C = A + B \]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</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 R3, R1, B</td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store R3, C</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td>Store R3, C</td>
<td></td>
</tr>
</tbody>
</table>

Specifying operands

- **"Addressing mode"**
  - Register direct: add R1, R2, R3
  - Immediate: add R1, R1, #1
  - Memory:
    - Direct (absolute): load R1, @(10000)
    - Register indirect: load R1, @(R2)
    - Memory indirect: load R1, @((R2))
    - Displacement: load R1, @(R2 + 100)
    - Indexed: load R1, @(R2 + R3)
    - Scaled: load R1, @(R2 + R3 \times d + 100)
    - Auto-increment/decrement: load R1, @((R2 + /-))

Usage of addressing modes

Endian-ness and data alignment

- **Big endian vs. little endian**
  - Defines byte ordering inside a larger data type stored in memory

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\text{(addr)} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\text{(addr)} & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\end{array}
\]

- **Alignment**
  - Data A is aligned if \((\text{addr}(A) \% \text{sizeof}(A)) == 0\) is true
  - What is the implication on hardware implementation (esp. memory system)?
Displacement size

Immediate size

Operations

- Arithmetic
  - add, sub, mul, div, ...
- Logical
  - and, or, xor, not, ...
- Shift
  - shift-left, shift-right, shift-right-arithmetic, ...
- Memory access
  - load, store, prefetch, ...
- Control
  - branches, jumps, procedure call/return, ...
- System-related
  - trap, break, ...

Frequent operations

<table>
<thead>
<tr>
<th>Rank</th>
<th>80x86 instruction</th>
<th>Integer average (%) 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 register-register</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>
<tr>
<td>Total</td>
<td></td>
<td>96%</td>
</tr>
</tbody>
</table>
Control flow instructions

- Target description
  - Instruction with an immediate value
    - PC-relative or absolute
  - In register
    - Typically absolute

- Condition description
  - Condition bit (zero, carry, overflow, …)
    - sub R1, R2, R3; bz LABEL
  - Condition register
    - cmp R1, R2, R3; bgtz R1, LABEL
  - Compare-and-branch
    - bgt R1, R2, LABEL

Branch distance

Comparison types
Instruction format

- At least 16 GPRs
- Provide regularity
  - Orthogonality
  - No restrictions on register usage
- Provide primitives, not solutions
  - e.g., HLL-like operations
- Simplify trade-offs in alternatives
- Provide instructions that bind constants

MIPS example

- Thirty-two 32-bit GPR
  - R0 wired to 0
  - Separate 32/16 SP/DP (single-/double-precision) FP registers
- Byte/half/word/dword, SP/DP FP data types
- Immediate and displacement addressing modes
- 32-bit fixed instruction encoding

ARM example

- Sixteen 32-bit GPR
  - R14 is link register
  - R15 is PC
- Different CPU modes have different register view
- Immediate and displacement addressing modes
- “Thumb” mode supports 16-bit instruction encoding
Summary

- Instruction set design requires understanding of
  - Application
  - Roadmap
  - Properties affecting hardware implementation
  - Properties affecting software design (e.g., application, compiler, …)

- Binary compatibility has been a key market driver
  - Legacy binaries

- Dynamic binary translation technology may weaken the dependency