ISA: The Hardware – Software Interface

- Instruction Set Architecture (ISA) is where software meets hardware
  - In embedded systems, this boundary is often flexible
  - Understanding of ISA design is therefore important

- Instruction Set Definition
  - Operands: int32, uint32, int16, uint16, int8, uint8, float32, float64
  - Operations: four major types
    - Operator functions (add, shift, xor, mul, etc)
    - Memory access (load-word, store-byte, etc)
    - Control transfer (branch, jump, call, return, etc)
    - Privileged, and miscellaneous instructions (not generated by compiler)

- Good understanding of compiler translation is essential
ISA Design Considerations

- Simple target for compilers
- Support for OS and HLL features
- Support for important data types (floating-point, vectors)
- Code size
- Impact on execution efficiency (especially with pipelining)
- Backwards compatibility with legacy processors
- Provision for extensions
CISC vs RISC

- **CISC**
  - Assembly programming $\rightarrow$ HLL features as instruction
  - Small # registers, memory not that slow $\rightarrow$ memory operands
  - Code size must be small $\rightarrow$ variable length
  - Backward compatibility $\rightarrow$ complexity increases

- **RISC**
  - Compilers $\rightarrow$ Simple instructions
  - Large # registers, memory much slower than processor $\rightarrow$ load store architecture
  - Simple and fast decoding $\rightarrow$ fixed length, fixed format
Operators and their Instructions

- **Integer Arithmetic**
  - +   add
  - -   sub
  - *   mul
  - /   div
  - %   rem

- **Relational**

<table>
<thead>
<tr>
<th>C operator</th>
<th>Comparison</th>
<th>Reverse</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>seq</td>
<td>0</td>
<td>bnez</td>
</tr>
<tr>
<td>!=</td>
<td>seq</td>
<td>0</td>
<td>beqz</td>
</tr>
<tr>
<td>&lt;</td>
<td>slt, sltu</td>
<td>0</td>
<td>bnez</td>
</tr>
<tr>
<td>&lt;=</td>
<td>slt, sltu</td>
<td>0</td>
<td>beqz</td>
</tr>
<tr>
<td>&gt;</td>
<td>slt, sltu</td>
<td>1</td>
<td>bnez</td>
</tr>
<tr>
<td>&gt;=</td>
<td>slt, sltu</td>
<td>1</td>
<td>beqz</td>
</tr>
</tbody>
</table>
Operators continued...

- **Bit-wise logic**
  - `|` or
  - `&` and
  - `^` xor
  - `~` not

- **Boolean**
  - `||` (src1 != 0 or src2 != 0)
  - `&&` (src1 != 0 and src2 != 0)

- **Shifts**
  - `>>` (signed) shift-right-arithmetic
  - `>>` (unsigned) shift-right-logical
  - `<<` shift-left-logical
Operand Types

- Usually based on scalar types in C

<table>
<thead>
<tr>
<th>Type modifier</th>
<th>C type declarator</th>
<th>Machine type</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>int, long</td>
<td>uint32</td>
</tr>
<tr>
<td></td>
<td>short</td>
<td>uint16</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>uint8</td>
</tr>
<tr>
<td></td>
<td>long long</td>
<td>uint64</td>
</tr>
<tr>
<td>signed</td>
<td>int</td>
<td>int32</td>
</tr>
<tr>
<td></td>
<td>short</td>
<td>int16</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>int8</td>
</tr>
<tr>
<td></td>
<td>long long</td>
<td>int64</td>
</tr>
<tr>
<td>boolean</td>
<td></td>
<td>uint1</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>float32</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td>float64</td>
</tr>
<tr>
<td>&amp;&lt;type_specifier&gt;</td>
<td></td>
<td>uint32</td>
</tr>
</tbody>
</table>

- C defines integer promotion for expression evaluation
  - `int16 + int32` will be performed at 32-bit precision
    - First operand must be sign-extended to 32 bits
  - Similarly, `uint8 + int16` will be performed at 16-bit precision
    - First operand must be zero-extended to 16-bit precision
Instruction Operands - Registers

- Registers
  - How many registers operands should be specified?
    - 3: \( R1 = R2 + R3 \)
    - 2: \( R1 = R1 + R2 \)
    - 1: \(+R1\)

- 32-bit RISC architectures normally specify 3 registers for dyadic operations and 2 registers for monadic operations

- Compact 16-bit embedded architectures normally specify respectively 2 and 1 register in these cases
  - Introduces extra register copying
  - E.g.
    - `load r1, [address]`
    - `copy r2, r1`
    - `add r1, r3`
    - `sub r4, r2`  # this is simply a re-use of r1, but the value of r1 had to be copied into r2

- Accumulator architectures now dead, but accumulators still widely used in Digital Signal Processors (DSP).
  - E.g.
    - `load [address1]`
    - `add 23`
    - `store [address2]`
Instruction Operands - Literals

- Constant operands
  - E.g. add r1, r2, 45

- Jump or branch targets
  - Relative:
    - Normally used for if-then-else and loop constructs within a single function
    - Distances normally short – can be specified as 16-bit signed & scaled offset
    - Permits “position independent code” (PIC)
  - Absolute
    - Normally used for function call and return
    - But not all function addresses are compile-time constants, so jump to contents of register is also necessary

- Load/Store addresses
  - Relative
  - Absolute
How big do literals have to be?

- **Addresses**
  - Always 32 (or 64 bits)

- **Arithmetic operands**
  - Small numbers, representable in 5 – 10 bits are common

- **Literals are often used repeatedly at different locations**
  - Place as read-only data in the code and access relative to program counter register (e.g. MIPS16, ARM-thumb)

- **Branch offsets**
  - 10 bits catches most branch distances

- **32-bit RISC architectures provide 16-bit literals**

- **16-bit instructions must cope with 5 – 10 bits**
  - May extend literal using an instruction prefix
  - E.g. Thumb bx instruction
Decision Making and Branches

- Condition code based
  - `sub $1, $2`
  - Sets Z, N, C, V flags
  - Branch selects condition
    - `ble`: N or Z
  - (+) Sometimes condition set for free
  - (-) Extra state

- Condition register based
  - `slte $1, $2, $3`
  - `bnez $1 (or beqz $1)`
  - (+) Simple and reduces number of opcodes
  - (-) uses up register

- Compare and branch
  - `combt lte $1, $2`
  - (+) One instruction for a branch
  - (-) Too much work for an instruction
Memory Access Operations

- Memory operations are governed by:
  - Direction of movement (load or store)
  - Size of data objects (word, half-word, byte)
  - Extension semantics for load data (zero-ext, sign-ext)
Addressing Mode Frequency

- Bottom-line: few addressing modes account for most of the instructions in programs
Displacement Addressing and Data Classification

- Stack pointer and Frame pointer relative
  - 5 to 10 bits of offset is sufficient in most cases
- Register + offset
  - Generic form for accessing via pointers
  - Multi-dimensional arrays require address calculations
- PC relative addresses
  - Useful for locating commonly-used constants in a pool of constants located in the .text section
Encoding the Instruction Set

- How many bits per instruction?
  - Fixed-length 32-bit RISC encoding
  - Variable-length encoding (e.g. Intel x86)
  - Compact 16-bit RISC encodings
    - ARM Thumb
    - MIPS16

- Formats define instruction groups with a common set of operands
MIPS 32-bit Instruction Formats

- **R-type** (register to register)
  - three register operands
  - most arithmetic, logical and shift instructions

- **I-type** (register with immediate)
  - instructions which use two registers and a constant
  - arithmetic/logical with immediate operand
  - load and store
  - branch instructions with relative branch distance

- **J-type** (jump)
  - jump instructions with a 26 bit address
### MIPS R-type instruction format

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>reg rs</td>
<td>reg rt</td>
<td>reg rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

- **add** $1, $2, $3
  - special $2 $3 $1 add
- **sll** $4, $5, 16
  - special $5 $4 16 sll
### MIPS I-type instruction format

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td></td>
<td>reg rs</td>
<td>reg rt</td>
<td>immediate value/addr</td>
</tr>
</tbody>
</table>

Examples:
- `lw   $1, offset($2)`
- `beq  $4, $5, .L001`
- `addi $1, $2, -10`
- `lw   $2, $1    address offset`
- `beq  $4, $5, (PC - .L001) >> 2`
- `addi $2, $1       0xfff6`
MIPS J-type instruction format

6 bits

26 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>address</th>
</tr>
</thead>
</table>

call func

call | absolute func address >> 2
ISA Guidelines

- Regularity: operations, data types, addressing modes, and registers should be independent (orthogonal)

- Primitives, not solutions: do not attempt to match HLL constructs with special IS instructions

- Simplify tradeoffs: make it easy for compiler to make choices based on estimated performance