ISA: The Hardware – Software Interface

- Instruction Set Architecture (ISA) is where software meets hardware
  - Understanding of ISA design is therefore important

- Instruction Set Components
  - Operands: int32, uint32, int16, uint16, int8, uint8, float32, float64
  - Addressing modes: how do we access data (in regs, memory, etc)
  - Operations: four major types
    - Operator functions (add, shift, xor, mul, etc)
    - Data movement (load-word, store-byte, etc)
    - Control transfer (branch, jump, call, return, etc)
    - Privileged, and miscellaneous instructions (not part of the application)

- Good understanding of compiler translation is essential
ISA Design Considerations

- Simple target for compilers
- Support for OS and programming language 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 instructions
  - Small # registers (but memory “fast”) \(\rightarrow\) in-memory operands
  - Code size must be small (transistors scarce) \(\rightarrow\) variable length instructions
  - Backward compatibility \(\rightarrow\) complexity grows over time

- **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
Instruction Classes

- Instructions that operate on data
  - Arithmetic & logic operations
  - Execution template: fetch operands, perform op, store result

- Instructions that move data
  - Move data between registers, memory, and I/O devices

- Instructions that change control flow
  - Re-direct control flow away from the next instruction
  - May be conditional or unconditional (including exceptions!)
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>

C operator Comparison Reverse Branch
Operators continued...

- **Bit-wise logic**
  
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>and</td>
</tr>
<tr>
<td><code>^</code></td>
<td>xor</td>
</tr>
<tr>
<td><code>~</code></td>
<td>not</td>
</tr>
</tbody>
</table>

- **Boolean**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>(src1 != 0 and src2 != 0)</td>
</tr>
</tbody>
</table>

- **Shifts**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt;&gt;</code></td>
<td>(signed) shift-right-arithmetic</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>(unsigned) shift-right-logical</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>shift-left-logical</td>
</tr>
</tbody>
</table>
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>unsigned</td>
<td>short</td>
<td>uint16</td>
</tr>
<tr>
<td>unsigned</td>
<td>char</td>
<td>uint8</td>
</tr>
<tr>
<td>unsigned</td>
<td>long long</td>
<td>uint64</td>
</tr>
<tr>
<td>signed</td>
<td>int</td>
<td>int32</td>
</tr>
<tr>
<td>signed</td>
<td>short</td>
<td>int16</td>
</tr>
<tr>
<td>signed</td>
<td>char</td>
<td>int8</td>
</tr>
<tr>
<td>signed</td>
<td>long long</td>
<td>int64</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
    - Result (signed or unsigned) determined by the result var’s type in the source code

This assumes a 32-bit machine!
Instruction Operands - Registers

- How many register-based 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 often specify respectively 2 and 1 register in these cases:
  - Reduces cost through fewer ports in the register file, less wire routing, etc.
  - “Destructive” ops → Requires extra register copying to preserve original values.
  - 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 concept still widely used in Digital Signal Processors (DSP).
  - E.g.
    
    ```
    load [address1]  
    add 23  
    store [address2]
    ```

  Register (accumulator) is implicit
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 for flexibility

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

- **Addresses**
  - Fixed & machine-specific: typically 32 or 64 bits

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

- **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. ARM Thumb `bx` instruction ➔ can branch conditionally to a target specified in a register, thus offering a 32-bit displacement
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)

```
Memory access
  load
    word lw
      signed lh
      unsigned lhu
    half-word
    byte lb
     signed lb
     unsigned lbu
  store
    word sw
    half-word sh
    byte sb
```
Memory Addressing Modes: Displacement

Displacement addressing is the most common memory addressing mode

- Register + offset
  - Generic form for accessing via pointers
  - Multi-dimensional arrays require address calculations

- Stack pointer and Frame pointer relative
  - 5 to 10 bits of offset is sufficient in most cases

- PC relative addresses
  - Used to modify control flow (e.g., upon a branch)
  - Also to access literals (see earlier slide)
Other Memory Addressing Modes

- Direct or absolute: useful for accessing constants and static data
- Auto-increment/decrement: useful for iterating over arrays or for stack push/pop operations
- Scaled: speeds up random array accesses
  
  *e.g.*, \( R7 = R5 + \text{Mem}[R1 + R2 \times d] \)
  
  where \( d \) is determined by the size of the data item being accessed (byte, hw, word, long)

- Memory indirect: in-memory pointer dereference
  
  *e.g.*, \( R3 = \text{Mem}[\text{Mem}[R1]] \)
Memory Addressing Mode Frequency

- Few addressing modes account for most memory accesses

H&P 5/e Fig. A.7
Instructions for Altering Control Flow

- Conditional (branches)
- (unconditional) Jumps
- Function calls and returns
- Exceptions & interrupts
  - Traps (instructions) vs events
  - Trigger a mode change

H&P 5/e Fig. A.11
Conditional Instruction Formats

- Condition code based (e.g., x86)
  - sub $1, $2
  - Sets Z, N, C, V flags
  - Branch selects condition
    - ble : N or Z
  - (+) Condition set for free (“side-effect” of instruction execution)
  - (-) Volatile state (next instruction may overwrite flags)

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

- Compare and branch
  - combt lte $1, $2
  - (+) One instruction per branch
  - (-) “Complex” instruction
Instruction Frequency by Type

Data from H&P 5/e Fig. A.13
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

- Orthogonal ISA: addressing modes are independent of the instruction type (i.e., all insts can use all addressing modes)
  - Great conceptually and for compilation
  - E.g., VAX-11: 256 opcodes * 13 addressing modes (mode encoded with each operand)
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>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

- **add** $1, $2, $3
- **sll** $4, $5, 16

Special formats:

<table>
<thead>
<tr>
<th>special</th>
<th>$2</th>
<th>$3</th>
<th>$1</th>
<th>add</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>special</th>
<th>$5</th>
<th>$4</th>
<th>16</th>
<th>sll</th>
</tr>
</thead>
</table>
MIPS I-type instruction format

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

- `lw $1, offset($2)`
- `beq $4, $5, .L001`
- `addi $1, $2, -10`

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>$2</td>
<td>$1</td>
<td>address offset</td>
</tr>
<tr>
<td>beq</td>
<td>$4</td>
<td>$5</td>
<td>(PC - .L001) &gt;&gt; 2</td>
</tr>
<tr>
<td>addi</td>
<td>$2</td>
<td>$1</td>
<td>0xffff6</td>
</tr>
</tbody>
</table>
### MIPS J-type instruction format

<table>
<thead>
<tr>
<th>6 bits</th>
<th>26 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>address</td>
</tr>
</tbody>
</table>

- **call func**
- **jal** absolute func address $\gg 2$
ISA Guidelines

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

- Primitives, not solutions: do not attempt to match HLL constructs through the instruction set

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