Instruction Set Architecture (ISA)

- Specification of the set of native commands implemented by a particular CPU design

- Defines
  - set of operations,
  - instruction format,
  - hardware supported data types,
  - named storage,
  - addressing modes,
  - sequencing
Instruction Set Architecture (II)

- ISA different from the micro-architecture - set of processor design techniques used to implement an ISA
- Example: Intel and AMD processors support a nearly identical ISA, but have very different micro-architectures
- CISC: Complex Instruction Set Computing
  - Execution of a single instruction can involve multiple low level operations and take an arbitrary number of cycles
  - Minimizes number of (assembler) instructions
- RISC: Reduced Instruction Set Computing
  - Supports only few, very simple operations

CISC vs. RISC

Example: multiplying two numbers

- CISC:
  mult 2:3, 5:2
- RISC
  load A, 2:3
  load B, 5:2
  prod A, B
  store 2:3, A

Image source: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/riscisc/
Internal storage

- Stack architecture: operands are implicitly on the top of the stack
- Accumulator architecture: one operand is implicitly the accumulator
- General purpose register architectures: operands have to be made available by explicit load operations
  - Dominant form in today’s systems

Internal storage (II)

- Example: $C = A + B$

<table>
<thead>
<tr>
<th>Stack:</th>
<th>Accumulator:</th>
<th>Load-Store:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push $A$</td>
<td>Load $A$</td>
<td>Load $R1,A$</td>
</tr>
<tr>
<td>Push $B$</td>
<td>Add $B$</td>
<td>Load $R2,B$</td>
</tr>
<tr>
<td>Add</td>
<td>Store $C$</td>
<td>Add $R3,R1,R2$</td>
</tr>
<tr>
<td>Pop $C$</td>
<td></td>
<td>Store $R3,C$</td>
</tr>
</tbody>
</table>
Internal storage (III)

- Advantage of general purpose register architectures vs. stack architectures:
  - Registers are fast
  - Easier and faster to evaluate complex expressions, e.g. 
    \((A*B)-(B*C)-(A*D)\)
  - Registers can hold temporary variables
  - Reduces memory traffic
  - A register can be named with fewer bits than main memory

Addressing modes

- How does an ISA specify the address an object will access?

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4, R3</td>
<td>(\text{Regs}[R4] \leftarrow \text{Regs}[R4]+\text{Regs}[R3])</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4, #3</td>
<td>(\text{Regs}[R4] \leftarrow \text{Regs}[R4]+3)</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4, (R1)</td>
<td>(\text{Regs}[R4] \leftarrow \text{Regs}[R4]+\text{Mem}[\text{Regs}[R1]])</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4, 100(R1)</td>
<td>(\text{Regs}[R4] \leftarrow \text{Regs}[R4]+\text{Mem}[100+\text{Regs}[R1]])</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R4, @R3</td>
<td>(\text{Regs}[R4] \leftarrow \text{Regs}[R4]+\text{Mem[Mem[Regs[R3]]]})</td>
</tr>
</tbody>
</table>
Addressing modes (II)

- Addressing modes must match
  - Ability of compilers to use them
  - Hardware characteristics
- Which modes are most commonly used?
  - Displacement
  - Immediate
  - Register indirect
- Size of address for displacement mode?
  - Typically 12-16 bits
- Size of the immediate field?
  - 8-16 bits

Internal storage (IV)

- Two major GPR architectures:
  - 2 or 3 operands for ALU instructions
    - 3 operands: 2 source, 1 result
    - 2 operands: 1 operand is both source and result
  - How many operands can be memory addresses?

<table>
<thead>
<tr>
<th>No. of memory addresses</th>
<th>Max. no. of operands</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>Register-register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(load-store arch.)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Register-memory</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Memory-memory</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Memory-memory</td>
</tr>
</tbody>
</table>
Memory alignment (I)

- Memory is typically aligned on a multiple of word boundaries
- Best case:
  - accessing misaligned address leads to performance problems since it requires accessing multiple words
- Worst case:
  - hardware does not allow misaligned access

Memory alignment (II)

<table>
<thead>
<tr>
<th>Width of object</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2 bytes (half word)</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 bytes (half word)</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 bytes (word)</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 bytes (word)</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 bytes (word)</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Type and size of operands (I)

- How is the type of an operand designated?
  - Encoded in the opcode
  - Annotated by tags

- Common operand types:
  - Character - 8 bits
  - Half word - 16 bits, 2 bytes
  - Word - 32 bits, 4 bytes
  - Single precision floating point - 32 bits, 4 bytes
  - Double precision floating point - 64 bits, 8 bytes

Type and size of operands (II)

- Encoding of characters:
  - ASCII
  - UNICODE

- Encoding of integers:
  - Two’s complement binary numbers

- Encoding of floating point numbers:
  - IEEE standard 754
  - No uniform representation of the data type long
double
Operations in the Instruction Set

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logical</td>
<td>Integer arithmetic: add, subtract, and, or, multiple, divide</td>
</tr>
<tr>
<td>Data transfer</td>
<td>Load, store, move</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jump, procedure call, return, traps</td>
</tr>
<tr>
<td>System</td>
<td>OS call, virtual memory management</td>
</tr>
<tr>
<td>Floating point</td>
<td>Floating point arithmetic: add, multiply, divide, compare</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal add, multiply</td>
</tr>
<tr>
<td>String</td>
<td>String move, string compare, string search</td>
</tr>
<tr>
<td>Graphics</td>
<td>Pixel and vertex operations, compression</td>
</tr>
</tbody>
</table>

Flow Control instructions

- Four types of different control flow changes
  - Conditional branches
  - Jumps
  - Procedure calls
  - Procedure returns

- How to specify the destination address of a flow control instruction?
  - PC-relative: Displacement to the program counter (PC)
  - Register indirect: name a register containing the target address
Flow Control instructions (II)

- Register indirect jumps also required for
  - Case/switch statements
  - Virtual functions in C++
  - Function pointers in C
  - Dynamic shared libraries (dll in Windows, .so in UNIX)
- Procedure invocation: global variables could be accessed by multiple routines
  → location of the variable needs to be known
- Options for saving registers:
  - Caller saving
  - Callee saving
  → due to the possibility of separate compilation, many compilers store any global variable that may be accessed during a call

Encoding an Instruction Set

- How are instructions encoded into a binary representation
  - Affects size of compiled program
  - Affects implementation of the processor
- Decision depends on range of addressing modes supported
  - Variable encoding
    - Individual instructions can vary widely in number of operands, size and amount of work to be performed
  - Fixed encoding
    - Easy to decode

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>Address specifier 2</th>
<th>Address field 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Address field 1</td>
<td>Address field 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Architecture: MIPS64 (I)

- Load-store architecture
  - 32 64bit GPR registers (R0,R1,...R31)
  - R0 contains always 0
  - 32 64bit floating point registers (F0,F1,...F31)
- When using 32bit floating point numbers
  - the other 32 bits of the FP registers are not used or
  - Instructions available for operating 2 32bit FP operations on a single 64bit register
- Data types:
  - 8 bit bytes, 16bit half-words, 32bit words, 64 bit double words
  - 32bit and 64bit floating point data types

Example architecture: MIPS64 (II)

- Addressing modes:
  - Immediate
  - Displacement
- Displacement field and immediate field are both 16bit wide
- Register indirect addressing accomplished by using 0 in the displacement field
- Absolute addressing accomplished by using R0 as the base register
Example architecture: MIPS64(III)

- All instructions are 32bit wide (fixed encoding): 6bit opcode
- Addressing modes are encoded in the opcode

\[
\text{LD R1, 30(R2)} \quad \text{load double word}
\]

\[
\begin{align*}
\text{Regs[R1]} & \leftarrow 64 \\
\text{Mem[30+Regs[R2]]} & \leftarrow n
\end{align*}
\]

with \( n \) load \( n \) bits

\[
\text{LW R1, 60(R2)} \quad \text{load word}
\]

\[
\begin{align*}
\text{Regs[R1]} & \leftarrow 64 \\
\text{Mem[60+Regs[R2]]} & \leftarrow 32 \quad \# \quad \text{Mem[60+Regs[R2]]}
\end{align*}
\]

with \( \text{Regs[R2]} \) indicating a bit-field selection,
- e.g. \( \text{Regs[R2]}_0 \) is the sign bit of \( R2 \)
- e.g. \( \text{Regs[R2]}_{53..63} \) last byte of \( R2 \)

with \( X^n \) replicate a bit field
- e.g. \( \text{Regs[R2]}_{0..23} \leftarrow 0^{24} \) set high order three bytes to 0

with \( \# \) concatenate two fields

Example architecture: MIPS64(IV)

Thus

\[
\begin{align*}
\text{Reg[R1]} & \leftarrow 64 \\
\text{Mem[60+Regs[R2]]} & \leftarrow 32 \quad \# \quad \text{Mem[60+Regs[R2]]}
\end{align*}
\]

Replicate the sign bit of memory address [60+Regs[R2]] on the first 32 bits of Regs[1]