Instruction-set Design Issues: what is the ML instruction format(s)

ML instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Dest. Operand</th>
<th>Source Operand 1</th>
<th>...</th>
</tr>
</thead>
</table>

1) Which instructions to include:
   - How many?
   - Complexity - simple “ADD R1, R2, R3”
     complex e.g., VAX
     “MATCHC substrLength, substr, strLength, str”
     looks for a substring within a string

2) Which built-in data types: integer, floating point, character, etc.

3) Instruction format:
   - Length (fixed, variable)
   - number of address (2, 3, etc)
   - field sizes

4) Number of registers

5) Addressing modes supported - how are the memory addresses of variables/data determining
### Number of Operands

<table>
<thead>
<tr>
<th>3 Address</th>
<th>2 Address</th>
<th>1 Address (Accumulator machine)</th>
<th>0 Address (Stack machine)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE (X ← Y)</td>
<td>MOVE (X ← Y)</td>
<td>LOAD M</td>
<td>PUSH M</td>
</tr>
<tr>
<td>ADD (X ← Y + Z)</td>
<td>ADD (X ← X + Y)</td>
<td>ADD M</td>
<td>ADD</td>
</tr>
<tr>
<td>SUB (X ← Y - Z)</td>
<td>ADD (X ← X - Y)</td>
<td>SUB M</td>
<td>SUB</td>
</tr>
<tr>
<td>MUL (X ← Y * Z)</td>
<td>MUL (X ← X * Y)</td>
<td>MUL M</td>
<td>MUL</td>
</tr>
<tr>
<td>DIV (X ← Y / Z)</td>
<td>DIV (X ← X / Y)</td>
<td>DIV M</td>
<td>DIV</td>
</tr>
</tbody>
</table>

**D = A + B * C**

<table>
<thead>
<tr>
<th>3 Address</th>
<th>2 Address</th>
<th>1 Address (Accumulator machine)</th>
<th>0 Address (Stack machine)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL D, B, C</td>
<td>MOVE D, B</td>
<td>LOAD B</td>
<td>PUSH B</td>
</tr>
<tr>
<td>ADD D, D, A</td>
<td>MUL D, C</td>
<td>MUL C</td>
<td>PUSH C</td>
</tr>
<tr>
<td></td>
<td>ADD D, A</td>
<td>ADD A</td>
<td>MUL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STORE D</td>
<td>PUSH A</td>
</tr>
</tbody>
</table>

**Load/Store Architecture** - operands for arithmetic operations must be from/to registers

LOAD R1, B
LOAD R2, C
MUL R3, R1, R2
LOAD R4, A
ADD R3, R4, R3
STORE R3, D
Flow of Control

How do we "jump around" in the code to execute high-level language statements such as if-then-else, while-loops, for-loops, etc.

Two Paths Possible

![Diagram showing flow of control with conditional and unconditional branches.]

Conditional branch - used to jump to "else" if x >= y

Unconditional branch - used to always jump "end if"

Labels are used to name spots in the code (memory) ("if:", "else:", and "end_if:" in below example)

Test-and-Jump version of the if-then-else (Used in MIPS)
if:
    bge x, y, else
    ... 
    j end_if
else:
    ... 
end_if:
**Set-Then-Jump** version of the if-then-else (Used in Pentium)

if:
```
    cmp x, y
    jge else
    ...
    j end_if
```
else:
```
    ...
```
end_if:

The "cmp" instruction performs \( x - y \) with the result used to set the condition codes:
- **SF** - (Sign Flag) set if result is < 0
- **ZF** - (Zero Flag) set if result = 0
- **CF** - (Carry Flag) set if unsigned overflow
- **OF** - (Overflow Flag) set if signed overflow

For example, the "jge" instruction checks to see if \( ZF = 1 \) or \( SF = 1 \), i.e., if the result of \( x - y \) is zero or negative.
Machine-Language Representation of Branch/Jump Instructions
(How are labels (e.g., “end_if”) in the code located?)

a) *direct/absolute addressing* - the memory address of where the label resides is put into the machine language instruction (EA, effective address = direct)
e.g., assume label "end_if" is at address $8000_{16}$

<table>
<thead>
<tr>
<th>AL instruction</th>
<th>ML instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>j end_if</td>
<td>Opcode 8000</td>
</tr>
</tbody>
</table>

```
end_if:
```

How *relocatable* is the code in memory if direct addressing is used? How many bits are needed to represent a direct address?

b) *Relative/PC-relative* - base-register addressing where the PC is the implicitly referenced register

<table>
<thead>
<tr>
<th>AL instruction</th>
<th>ML instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>while:</td>
<td>Opcode 8 9 40</td>
</tr>
<tr>
<td>bge R8, R9, end_while PC = 4000</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>&quot;end_while&quot; label 40 addresses from &quot;bge&quot;</td>
</tr>
</tbody>
</table>

```
b while
end_while: PC = 4040
```

Unconditional pc-relative branches are possible too.
Machine-Language Representation of Variables/Operands  
(How are labels (e.g., “sum”, “score”, etc.) in the code located???)

a) *Register* - operand is contained in a register

<table>
<thead>
<tr>
<th>AL instruction</th>
<th>ML instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r9, r4, r2</td>
<td>Opcode 9 4 2</td>
</tr>
</tbody>
</table>

b) *Direct/absolute addressing* - the memory address of where the label resides is put into the machine language instruction (EA, effective address = direct)

E.g., assume label "sum" is at address $8000_{16}$ and “score” is at address $8004_{16}$

<table>
<thead>
<tr>
<th>AL instruction</th>
<th>ML instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>add sum, sum, score</td>
<td>Opcode 8000 8000 8004</td>
</tr>
<tr>
<td></td>
<td>32 bits 32 bits 32 bits</td>
</tr>
</tbody>
</table>

c) *Immediate* - part of the ML instruction contains the value

<table>
<thead>
<tr>
<th>AL instruction</th>
<th>ML instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi r9, #2</td>
<td>Opcode 9 2</td>
</tr>
</tbody>
</table>
d) *Indirect* - operand is pointed at by an address in a memory

**AL instruction**

```
add r9, (A), r2
```

**ML instruction**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>9</th>
<th>8000</th>
<th>2</th>
</tr>
</thead>
</table>

**ML instruction**

```
add r9, (A), r2
```

**EA = (A)**

---

*Register Indirect* - operand is pointed at by an address in a register

**AL instruction**

```
add r9, (r4), r2
```

**ML instruction**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>9</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
</table>

**ML instruction**

```
add r9, (r4), r2
```

**EA = (r4)**

---

Lecture 2 - 7
Base-register addressing / Displacement - operand is pointed at by an address in a register plus offset

AL instruction
Load r9, 40(r2)

ML instruction
Opcode | 9 | 40 | 2
---|---|---|---

Register File

EA = (r2) + 40

Often the reference register is the stack pointer register to manipulate the run-time stack, or a global pointer to a block of global variables.
f) **Indexing** - ML instruction contains a memory address and a register containing an index

**AL instruction**

```
addindex r9, A(r2)
```

**ML instruction**

```
Opcode  9  8000  2
```

```
ML instruction

OP code  9  8000  2

Alt code 8000

Reg. File

| 10 |

EA = A + (r2)

Useful for array access.
Reduced Instruction Set Computers (RISC)

Two approaches to instruction set design:
1) CISC (Complex Instruction Set Computer) e.g., VAX or IBM 370
   1960’s: Make assembly language (AL) as much like high-level language
   (HLL) as possible to reduce the “semantic gap” between AL and HLL

Alleged Reasons:
- reduce compiler complexity and aid assembly language programming -
  compilers not too good at the time (e.g., they did not allocate registers
  very efficiently)
- reduce the code size - (memory limited at this time)
- improve code efficiency - complex sequence of instructions implemented
  in microcode (e.g., VAX “MATCHC substrLength, substr, strLength,
  str” that looks for a substring within a string)

Characteristics of CISC:
- high-level like AL instructions
- variable format and number of cycles
- many addressing modes (VAX 22 addressing modes)

Problems with CISC:
- complex hardware needed to implement more and complex instructions
  which slows the execution of simpler instructions
- compiler can rarely figure out when to use complex instructions (verified
  by studies of programs)
- variability in instruction format and instruction execution time made
  CISC hard to pipeline

2) RISC (mid 1980’s) Addresses these problems to improve speed.

(Table 9.1 - characteristics of some CISC and RISC processors)
Table 9.1 - characteristics of some CISC and RISC processors

<table>
<thead>
<tr>
<th></th>
<th>RISC</th>
<th>CISC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multiple register sets, often consisting of more than 256 registers</td>
<td>Single register set, typically 6 to 16 registers total</td>
</tr>
<tr>
<td></td>
<td>Three register operands allowed per instruction (e.g., add R1, R2, R3)</td>
<td>One or two register operands allowed per instruction (e.g., add R1, R2)</td>
</tr>
<tr>
<td></td>
<td>Parameter passing through efficient on-chip register windows</td>
<td>Parameter passing through inefficient off-chip memory</td>
</tr>
<tr>
<td></td>
<td>Single-cycle instructions (except for load and store)</td>
<td>Multiple-cycle instructions</td>
</tr>
<tr>
<td></td>
<td>Hardwired control</td>
<td>Microprogrammed control</td>
</tr>
<tr>
<td></td>
<td>Highly pipelined</td>
<td>Less pipelined</td>
</tr>
<tr>
<td></td>
<td>Simple instructions that are few in number</td>
<td>Many complex instructions</td>
</tr>
<tr>
<td></td>
<td>Fixed length instructions</td>
<td>Variable length instructions</td>
</tr>
<tr>
<td></td>
<td>Complexity in compiler</td>
<td>Complexity in microcode</td>
</tr>
<tr>
<td></td>
<td>Only load and store instructions can access memory</td>
<td>Many instructions can access memory</td>
</tr>
<tr>
<td></td>
<td>Few addressing modes</td>
<td>Many addressing modes</td>
</tr>
</tbody>
</table>
General Characteristics of RISC:

- emphasis on optimizing instruction pipeline
  a) one instruction completion per cycle
  b) register-to-register operations
  c) simple addressing modes
  d) simple, fixed-length instruction formats
- limited and simple instruction set and addressing modes
- large number of registers or use of compiler technology to optimize register usage
- hardwired control unit

RISC Instruction-Set Architecture (ISA) can be effectively pipelined