Classifying Instruction Set Architecture

Operands
- Number: (0,1,2,3)
- Type, Size: byte, int, float
- Location: memory or register (effective address)

Operations
- Type: add, sub, mul, ...
- How is it specified?

Internal Storage
- Stack
- Accumulator
- Register

Type and Size of Operands

- Designating Types of Operands
  - Encoded in Opcode
  - Tagged
    - Burroughs
    - Symbolics

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>1 Byte</td>
<td>ASCII, EBCDIC</td>
</tr>
<tr>
<td>Integer</td>
<td>4 Bytes</td>
<td>2’s complement</td>
</tr>
<tr>
<td>SP Floating Point</td>
<td>4 Bytes</td>
<td>IEEE Standard 754</td>
</tr>
<tr>
<td>DP Floating Point</td>
<td>8 Bytes</td>
<td>IEEE Standard 754</td>
</tr>
<tr>
<td>Character Strings, Packed Decimals, BCD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Designing a New System?

- 64-bit access path or two cycles for 64-bits?
- Support byte access?
- Implications of 64-bit address architecture?

Encoding an Instruction Set

- Instruction has to contain the Opcode and all the addresses
  - How to encode the addressing modes?
  - Complex encoding may reduce program size but may increase the complexity of implementation

- Architect has to balance
  - The desire to have as many registers and addressing modes as possible
  - The impact of the size of the register and addressing mode fields on the average instruction size
  - The desire to encode instructions into easy to handle lengths
Instruction Encoding

- Good code density but difficult to decode
- Single size for all
- Easy to decode but not as good code density
- Compromise between code density and ease of decode

Role of a Compiler

- Performance depends not only on the architecture but the compiler also
- Architectural choices affect
  - Quality of code
  - Complexity of building a good compiler

Compiler Issues

- Structure of Compilers
- Evaluation Criterion of Compilers
- Impact of Compiler Optimizations

Structure of a Compiler

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language Dependent</td>
<td>Transform language to common intermediate form</td>
</tr>
<tr>
<td>Machine Dependent</td>
<td>Procedure In-Lining Loop Transformation, etc.</td>
</tr>
<tr>
<td>Language dependent machine independent</td>
<td>High Level Optimization</td>
</tr>
<tr>
<td>Small Language Dependencies; slight Machine dependencies</td>
<td>Global Optimization</td>
</tr>
<tr>
<td>Highly Machine Dependent Language Dependent</td>
<td>Code Generation</td>
</tr>
<tr>
<td>Language Independent</td>
<td></td>
</tr>
</tbody>
</table>

Page 2
Evaluation Criteria of Compilers

- Correctness
- Speed of Compiled Code
- Fast Compilation
- Debugging Support
- Interoperability among Languages
- Size of compiled code

Complexity of writing a correct compiler is a major limiting factor on the amount of optimization that can be done

Transformations are carried out in order

Phase Ordering Problem - Have to assume information about later steps

Compiler Optimizations

- High Level Optimization
  - Done on source code
- Local Optimization
  - Optimize code within a straight-line code fragment
- Global Optimization
  - Extend the local optimization across branches and introduce a set of transformations aimed at optimizing loops
- Register Allocation
- Machine Dependent Optimization
  - Take advantage of specific architectural knowledge

Impact of Compiler Optimizations

- Level 0: unoptimized code
- Level 1: local optimization, code scheduling, and local register allocation
- Level 2: global optimization, loop transforms, and global register allocation
- Level 3: procedure integration

Impact of Compiler Technology on Architect's Decision

- How are variables allocated and addressed?
- How many registers are needed to allocate variables?

- Need to understand areas where data are allocated
- Register allocation effective for stack, global data;
- Virtually impossible for heap
- Pointer, aliases
Variable Addressing

- **Stack**
  - Used for local variables
  - Grown and shrunk on procedure call and return
  - Objects primarily scalars, addressed relative to SP
  - Used for activation record and not for evaluating expressions
  - Values rarely pushed or popped on stack
- **Global Data Area**
  - Used for statically declared objects - global variables, const
  - Usually arrays and other data structures
- **Heap**
  - Used for dynamic objects that do not use stack discipline
  - Objects accessed through pointers
  - Usually scalars

Desired properties of an architecture (How the architect can help the compiler writer)

*Make the frequent cases fast and rare case correct*

- Regularity/Orthogonality
- Provide primitives, not solutions
- Simplify tradeoffs among alternatives
- Provide instructions that bind the quantities known at compile time as constants

DLX Architecture

- Simple load/store architecture
- Design for pipelining efficiency, including a fixed instruction set encoding
- Efficiency as a compiler target

DLX Specifics

- **Registers**
  - 32 32-bit general-purpose registers (R0, R1, ..., R31)
  - R0 is always 0
  - 32 single-precision (32-bit) registers (F0, F1, ..., F31)
  - (can be used as 16 double-precision (64-bit) registers)
- **Data types**
  - 8-bit bytes, 16-bit half word, 32-bit words for integer data
  - 32-bit single precision and 64-bit double precision for floating point data
- **Addressing modes**
  - Immediate
  - Displacement: base register + 16-bit signed offset
  - Register deferred: place 0 in 16-bit offset
  - Absolute: use register R0 (which is always 0)
DLX Instruction Format

DLX Operations (load and store instructions)

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2, R3</td>
<td>Add</td>
<td>R3 = R1 + R2</td>
</tr>
<tr>
<td>ADDI R1, R2, #3</td>
<td>Add immediate</td>
<td>R2 = R1 + #3</td>
</tr>
<tr>
<td>LHI R1, R2</td>
<td>Load high immediate</td>
<td>R3 = R2 &lt;&lt; 24</td>
</tr>
<tr>
<td>SLT R1, R2, R3</td>
<td>Set less than</td>
<td>R3 = 1 if (R2 &lt; R3)</td>
</tr>
</tbody>
</table>

DLX Operations (arithmetic/logical instructions)

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2, R3</td>
<td>Add</td>
<td>R3 = R1 + R2</td>
</tr>
<tr>
<td>ADDI R1, R2, #3</td>
<td>Add immediate</td>
<td>R2 = R1 + #3</td>
</tr>
<tr>
<td>LHI R1, R2</td>
<td>Load high immediate</td>
<td>R3 = R2 &lt;&lt; 24</td>
</tr>
<tr>
<td>SLT R1, R2, R3</td>
<td>Set less than</td>
<td>R3 = 1 if (R2 &lt; R3)</td>
</tr>
</tbody>
</table>

DLX Operations (control-flow instructions)

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>J name</td>
<td>Jump</td>
<td>PC = PC + 4</td>
</tr>
<tr>
<td>JAL name</td>
<td>Jump and link</td>
<td>PC = PC + 4, R0 = #0</td>
</tr>
<tr>
<td>JAL R0</td>
<td>Jump and link</td>
<td>PC = PC + 4, R0 = R0</td>
</tr>
<tr>
<td>J R0</td>
<td>Jump register</td>
<td>PC = R0</td>
</tr>
<tr>
<td>SLLZ R0, #0</td>
<td>Branch equal zero</td>
<td>if (R0 = #0) then PC = PC + 4</td>
</tr>
<tr>
<td>SLLH R0, #0</td>
<td>Branch equal zero</td>
<td>if (R0 = #0) then PC = PC + 4</td>
</tr>
<tr>
<td>SLLW R0, #0</td>
<td>Branch equal zero</td>
<td>if (R0 = #0) then PC = PC + 4</td>
</tr>
</tbody>
</table>