Lecture 4: ISA Examples

Kunle Olukotun
Gates 302
kunle@ogun.stanford.edu

http://www-leland.stanford.edu/class/ee282h/

ISA case studies

- We’ll look at examples of two instruction set architecture styles:
  » DEC VAX
    - conceptually elegant
    - end-of-an-era CISC design popular from the late 1970’s
    - much studied: many architectural measurements exist
  » DLX
    - based on the last decade of instruction-set measurement
    - 3-operand load/store many-register architecture
  » Intel 80x86
    - Will not cover in class but read appendix D
    - design influenced by generational history
    - halfway between accumulator and general-register machine
    - sells 100x any other ISA
Encoding an ISA

- Balance competing forces:
  - Decode complexity
  - Cycle time
  - Maximizing what can be done early in execution
  - Compiler/programmer convenience: many registers and addressing modes
  - Minimizing instruction count or instruction size

- Addressing modes are the hardest part
  - For small number of modes, encode implicitly in the opcode
  - For a large number of modes, use separate address specifier

The DEC VAX architecture

- Superset of many other ISAs, including IBM 360/370 and DEC PDP-11.
- VAX = “Virtual Address Extension: solve the problem of 16-bit addressing on the PDP-11, but be compatible:
  - Same data types
  - Same I/O bus (UNIBUS) plus others (MASSBUS, SBI)
  - PDP-11 direct emulation mode
  - Similar assembler syntax

- Large orthogonal instruction set, with support for:
  - High-level language constructs (procedure call w/ arguments)
  - Virtual memory management
  - Rapid context switching (Process Control Block: Regs+state)

- First implementation: the VAX-11/780
  - Internally microprogrammed, 96-bit word + writeable section
The VAX architecture

- General register machine: 16 32-bit registers
- Special register assignments:
  - R15: program counter
  - R14: current stack pointer
  - R13: stack frame pointer
  - R12: argument list pointer
  - R0-R5: state information for long interruptible instructions
- 32-bit addressing

VAX Instruction Encoding

<table>
<thead>
<tr>
<th>opcode</th>
<th>op spec 1</th>
<th>op spec 2</th>
<th>op spec 3</th>
<th>op spec 4</th>
<th>op spec 5</th>
<th>op spec 6</th>
</tr>
</thead>
</table>

- 1-2 byte opcodes followed by 0-6 operand specifiers, each of which may be up to 5 bytes.
- About 300 opcodes, most 8-bit (integer, floating point, character string, bitfields, decimal, CRC, POLY)
- Opcode implies datatype and size, and number of operands (A=B+C vs A=A+B)
- Operand specifiers indicate addressing modes
- Orthogonality: any opcode with any operands of any modes
- How do you encode?
VAX operand addressing

- 6-bit Literal: 0-63 positive integers
  3-bit exp+3-bit frac floating point (integers 1-16, 16ths, more)
- register, register deferred, optional autoincrement/decrement
- 1, 2, 4-byte displacement from register, optionally deferred
- Indexed (scaled): R*operandsize + next operand specifier
  (for array indexing)
- Clever encoding:
  » immediate = PC autoincrement
  » absolute = PC autoincrement deferred

Characteristics of RISC architectures?

- RISC ("Reduced Instruction Set Computer") architectures generally:
  » have fixed-size 32-bit instructions
  » are register-based with 32 or more registers
  » have only load/store memory operand access instructions
    (no ALU memory ops)...
  » No support complex data types
  » try to avoid special-case instructions
    – "orthogonality": any operation with any register, etc.
  » are willing to make architectural compromises to permit efficient
    pipelining
DLX

- DLX: a fictitious prototypical RISC machine
  - Simple load/store instruction set
  - Easily decoded and pipelined
  - Good compiler target (few opcodes, many registers, one (1) addressing mode)

- Overview
  - 32 32-bit general-purpose registers: R0 is always 0
  - 32 32-bit (single-precision) floating point registers, or 16 64-bit (double-precision) registers
  - Memory byte addressable, 32-bit addresses
  - Aligned data accesses
  - All instructions are 32 bits long, in 4 classes:
    - load and stores
    - ALU ops
    - branches and jumps
    - floating point

DLX Instructions
1 of 3

I-type instructions

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Load and stores (byte/halfword/word, signed/unsigned)
  - rd <=> Memory [rs1 + IMM]
- ALU operations with immediate operands
  - rd <= rs1 op IMM
- Conditional branches: no condition code!
  - if (rs1 == 0) PC += IMM
  - Longer distances require an additional jump
- Jump [and link] register
  - [r31 = PC+4] PC <= rs1
- Load immediate, load upper immediate
  - Used together to load a 32-bit value
DLX Instructions
2 of 3

R-type instructions

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opcode</td>
<td>rs1</td>
<td>rs2</td>
<td>rd</td>
<td>func</td>
</tr>
</tbody>
</table>

- Register-to-register instructions: 3-operand ALU ops
  - rd <= rs1 func rs2
    - add/sub/mul/divide, signed and unsigned
    - and/or/xor
    - shifts
    - compares: if true destination is 1, else 0
    - floating point add/sub/mul/divide, single/double precision
    - floating/integer conversion instructions

DLX Instructions
3 of 3

J-type instructions

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opcode</td>
<td>Offset</td>
</tr>
</tbody>
</table>

- Jump
  - PC += Offset
- Jump and link
  - r31 <= PC+4; PC += Offset
What DLX needs to do to emulate VAX addressing modes

- Assume:
  - VAX displacements fit in the DLX displacement field
  - The scale size is a power of 2

<table>
<thead>
<tr>
<th>VAX</th>
<th>DLX</th>
</tr>
</thead>
<tbody>
<tr>
<td>register-deferred add r1, (r2)</td>
<td>lw r3, 0(r2) add r1, r1, r3</td>
</tr>
<tr>
<td>displacement add r1, 200(r2)</td>
<td>lw r3, 200(r2) add r1, r1, r3</td>
</tr>
<tr>
<td>autoincrement add r1, (r2)+</td>
<td>lw r3, 0(r2) addui r2, r2, opsize add r1, r1, r3</td>
</tr>
<tr>
<td>displacement deferred add r1, @300(r2)</td>
<td>lw r3, 300(r2)</td>
</tr>
<tr>
<td>scaled add r1, 100(r2)[r3]</td>
<td>sll r4, r3, log2 opsize addu r4, r4, r2 lw r5, 100(r4) add r1, r1, r5</td>
</tr>
</tbody>
</table>

What addressing modes are most common?

Only about 30% of operands are memory references
Computing the DLX penalty for not having VAX addressing modes

- For VAX there are about 1.8 operands per instruction
- For GCC, memory operand modes are used as follows:
  - reg indirect 5.4% +1 instruction for DLX
  - displacement 20% +1 instruction for DLX
  - scaled 2.7% +3 instructions for DLX
  - displ. indirect 0.6% +2 instructions for DLX
  - autoincrement 1.2% +2 instructions for DLX
  - register 51% +0 instructions for DLX
  - immediate 19% +0 instructions for DLX

- Additional DLX instructions = 1.8 x (.054 x 1 + .20 x 1 + .027 x 3 + .006 x 2 + .012 x 2) = 1.8 x .371 = 0.66

Or, DLX/VAX of 1.66
DLX SPECfp92 Instruction Mix

<table>
<thead>
<tr>
<th>Instruction</th>
<th>other</th>
<th>our</th>
<th>hybrid</th>
<th>mult/fp1</th>
<th>mult/fp2</th>
<th>other</th>
<th>FP average</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1.6%</td>
<td>0.2%</td>
<td>0.1%</td>
<td>1.1%</td>
<td>3.0%</td>
<td>1.4%</td>
<td>1%</td>
</tr>
<tr>
<td>store</td>
<td>1.3%</td>
<td>0.1%</td>
<td>0.1%</td>
<td>0.1%</td>
<td>1.3%</td>
<td>1.3%</td>
<td>1%</td>
</tr>
<tr>
<td>add</td>
<td>13.6%</td>
<td>13.6%</td>
<td>0.4%</td>
<td>9.4%</td>
<td>4.3%</td>
<td>9.7%</td>
<td>11%</td>
</tr>
<tr>
<td>sub</td>
<td>0.3%</td>
<td>0.3%</td>
<td>0%</td>
<td>0.3%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>mul</td>
<td></td>
<td></td>
<td></td>
<td>0%</td>
<td></td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>div</td>
<td></td>
<td></td>
<td></td>
<td>0%</td>
<td></td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>comp</td>
<td>0.2%</td>
<td>3.1%</td>
<td>1.2%</td>
<td>0.9%</td>
<td>1.3%</td>
<td>2%</td>
<td>3%</td>
</tr>
<tr>
<td>load imm</td>
<td>2.7%</td>
<td>0.2%</td>
<td>0.2%</td>
<td>2.2%</td>
<td>0.8%</td>
<td>2%</td>
<td>1%</td>
</tr>
<tr>
<td>cond bruch</td>
<td>8.0%</td>
<td>8.0%</td>
<td>11.7%</td>
<td>8.3%</td>
<td>7.6%</td>
<td>8%</td>
<td>8%</td>
</tr>
<tr>
<td>uncond bruch</td>
<td>0.9%</td>
<td>0.4%</td>
<td>0.4%</td>
<td>0.4%</td>
<td>0.1%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>call</td>
<td>0.2%</td>
<td>1.9%</td>
<td></td>
<td>0.3%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>ret, ret rv</td>
<td>0.0%</td>
<td>1.9%</td>
<td></td>
<td>0.1%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>div</td>
<td>2.0%</td>
<td>0.2%</td>
<td>2.0%</td>
<td>2.0%</td>
<td>2.3%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>and</td>
<td>0.1%</td>
<td>0.1%</td>
<td></td>
<td>0.1%</td>
<td>0.1%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>or</td>
<td>0.3%</td>
<td>0.3%</td>
<td>0.1%</td>
<td>0.1%</td>
<td>0.1%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>other (not</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>load FP</td>
<td>21.3%</td>
<td>19.8%</td>
<td>24.1%</td>
<td>25.9%</td>
<td>21.8%</td>
<td>23%</td>
<td>23%</td>
</tr>
<tr>
<td>store FP</td>
<td>3.7%</td>
<td>11.4%</td>
<td>9.9%</td>
<td>10.9%</td>
<td>9.5%</td>
<td>9%</td>
<td>9%</td>
</tr>
<tr>
<td>add FP</td>
<td>8.8%</td>
<td>7.3%</td>
<td>3.6%</td>
<td>8.3%</td>
<td>2.4%</td>
<td>8%</td>
<td>8%</td>
</tr>
<tr>
<td>sub FP</td>
<td>3.8%</td>
<td>3.2%</td>
<td>3.0%</td>
<td>16.4%</td>
<td>5.9%</td>
<td>6%</td>
<td>6%</td>
</tr>
<tr>
<td>mul FP</td>
<td>12.8%</td>
<td>9.0%</td>
<td>9.8%</td>
<td>13.3%</td>
<td>20.6%</td>
<td>13%</td>
<td>13%</td>
</tr>
<tr>
<td>div FP</td>
<td>2.3%</td>
<td>1.6%</td>
<td>0.9%</td>
<td>0.7%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>compare FP</td>
<td>6.2%</td>
<td>6.6%</td>
<td>10.4%</td>
<td>9.5%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>mov reg FP</td>
<td>2.1%</td>
<td>1.6%</td>
<td>5.3%</td>
<td>0.9%</td>
<td>1.9%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>other FP</td>
<td>2.4%</td>
<td>8.4%</td>
<td>0.3%</td>
<td>0.3%</td>
<td>1.2%</td>
<td>3%</td>
<td>3%</td>
</tr>
</tbody>
</table>

FIGURE 3.27 DLX Instruction mix for five programs from SPECfp92. Note that integer register-register move instructions are included in the load instruction. Units entries here are the value 0.0%.

---

**ISA Design Example**

- **You propose a new version of DLX called DLX-lite**
- **DLX-lite only has a register-indirect memory addressing mode.**
- **You claim that DLX-lite reduces the CPI of loads and stores from 2.0 to 1.0**
- **The CPI of all other instructions is 1.0 on both processors**
- **Is DLX-lite faster on integer programs?**
Summary

- ISAs are shaped by implementations, compilers, and history
  - Limited hardware + primitive compilers --> stack architectures
  - HLL support + high memory cost --> orthogonal CISC
  - Limited hardware + much assembly programming
    --> small register set designs
  - Efficient pipelining + optimizing compilers + good caches
    --> RISC architectures

- What's coming?
  - More support for multiple instruction issue
  - Increased tolerance for long memory latency
  - Continued "complexification" of RISC, within reason
  - Even tighter compiler/ISA integration; compiler-generated hints for branches, memory accesses, etc.
  - Support for multiple threads of control

Other RISC variations:
PowerPC condition register

- Compromise between a global condition code later tested by branch instructions (hard to pipeline), and explicitly set compare results in registers (extra instructions):
  - A condition register (CR) that has 8 independent 4-bit condition codes:
    - one set implicitly by integer operations that request it
      "add" doesn't, "add." does
    - one set implicitly by floating-point operations that request it
    - six available to be set explicitly by compare instructions
  - Branch instructions can test any of the 8 condition codes

- Allows the compiler to reschedule and separate them sufficiently for high performance without destroying branch conditions before they are used.

- For combined conditions to reduce branches:
  and/or/nand/nor/xor between any CR bits, results to CR bit.
Other RISC variations:
DEC Alpha AXP (21064)

- Conditional register-move instruction
  » if (Ra <cond> 0) Rc = Rb
- Scaled addition/subtraction for array indexing
  » Rc = Ra + k x Rb, where k=4 or 8
- Byte manipulation instructions: compare, extract, insert, mask, zero
- PAL: Privileged Architecture Library
  » disabled instruction sequences with access to internal machine state; almost main-memory microcode
  » implementation-dependent
  » used for interrupt dispatching, virtual memory control, etc.