ECE 152
Introduction to Computer Architecture

Instruction Set Architecture (ISA)
Copyright 2005 Daniel J. Sorin
Duke University

Slides are derived from work by
Amir Roth (Penn) and Alvy Lebeck (Duke)
Spring 2005

Readings
• Patterson and Hennessy
  • Chapter 2
    • Read this chapter as if you’d have to teach it
    • Appendix A (reference for MIPS instructions and SPIM)
      • Read as much of this chapter as you feel you need

What Is an ISA?
• ISA
  • The “contract” between software and hardware
  • If software does X, hardware promises to do Y
    • Functional definition of operations, modes, and storage locations supported by hardware
    • Precise description of how software can invoke and access them
  • Strictly speaking, ISA is the architecture, i.e., the interface between the hardware and the software
  • Less strictly speaking, when people talk about architecture, they’re also talking about how the architecture is implemented
How Would You Design an ISA?

- What kind of interface should the hardware present to the software?
- Types of instructions?
- Instruction representation?
- How do we get from instruction 1 to 2 (or to 7 instead)?
- Software’s view of storage? Where do variables live?
- Does the hardware help to support function calls? If so, how?

Microarchitecture

- ISA specifies what hardware does, not how it does it
  - No guarantees regarding
    - How operations are implemented
    - Which operations are fast and which are slow
    - Which operations take more power and which take less
  - These issues are determined by the microarchitecture
    - Microarchitecture = how hardware implements architecture
    - Can be any number of microarchitectures that implement the same architecture (Pentium and Pentium 3 are almost the same architecture, but are very different microarchitectures)
- Class project is to build MIPS 152/16 processor
  - I specify the architecture
  - You design the microarchitecture, with the goal of making it as fast as possible (while still correct in all cases!)

Aspects of ISAs

- We will discuss the following aspects of ISAs
  1. The Von Neumann (pronounced NOY-muhn) model
     - Implicit structure of all modern ISAs
  2. Format
     - Length and encoding
  3. Operations
  4. Operand model
     - Where are operands stored and how do address them?
  5. Datatypes and operations
  6. Control

- Running example: MIPS
  - MIPS ISA designed to match actual pattern of use in programs

(1) The Sequential (Von Neumann) Model

- Implicit model of all modern ISAs
  - Often called Von Neumann, but in ENIAC before
- Basic feature: the program counter (PC)
  - Defines total order of dynamic instructions
    - Next PC is PC++ unless insn says otherwise
  - Order and named storage define computation
    - Value flows from insn X to Y via storage A iff...
    - X names A as output, Y names A as input...
    - And Y after X in total order
- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instruction X finishes before insn X+1 starts
(2) Instruction Format

- **Length**
  1. Fixed length
     - 32 or 64 bits (depends on architecture – MIPS 152/16 is 16 bit)
     + Simple implementation: compute next PC using only PC
     - Code density: 32 or 64 bits for a NOP?
  2. Variable length
     - Complex implementation
     + Code density
  3. Compromise: two lengths
     • Example: MIPS16

- **Encoding**
  • A few simple encodings simplify decoder implementation
  • You will appreciate simple encodings when building MIPS 152/16

---

MIPS Format

- **Length**
  - 32-bits
  - MIPS16: 16-bit variants of common instructions for density

- **Encoding**
  - 3 formats, simple encoding
  - Q: how many operation types can be encoded in 6-bit opcode?

<table>
<thead>
<tr>
<th>Type</th>
<th>Op(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Rd(5)</th>
<th>Sh(5)</th>
<th>Func(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I-type</td>
<td>Op(6)</td>
<td>Rs(5)</td>
<td>Rt(5)</td>
<td></td>
<td></td>
<td>Immed(16)</td>
</tr>
<tr>
<td>J-type</td>
<td>Op(6)</td>
<td></td>
<td></td>
<td>Target(26)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

(3) Operations

- Operation type encoded in instruction **opcode**
- Many types of operations
  - Integer arithmetic: add, sub, mul, div, mod/rem (signed/unsigned)
  - FP arithmetic: add, sub, mul, div, sqrt
  - Integer logical: and, or, xor, not, sll, srl, sra
  - Packed integer: padd, pmul, pand, por… (saturating/wraparound)
- What other operations might be useful?
- More operation types == better ISA??
- DEC VAX computer had LOTS of operation types
  - E.g., instruction for polynomial evaluation (no joke!)
  - But many of them were rarely/never used
  - We’ll talk more about this issue later …

---

(4) Operations Act on Operands

- If you’re going to add, you need at least 3 operands
  - Two source operands, one destination operand
- Question #1: Where can operands come from?
- Question #2: And how are they specified?
  - Running example: \( A = B + C \)
    - Several options for answering both questions
- Criteria for evaluating operand models
  - Metric I: **static code size**
  - Metric II: **data memory traffic**
  - Metric III: **instruction latency**
Operand Model I: Memory Only

- **Memory only**

  \[ \text{add } A,B,C \quad \text{mem}[A] = \text{mem}[B] + \text{mem}[C] \]

Operand Model II: Stack

- **Stack**: top of stack (TOS) is implicit in instructions

  \[
  \begin{align*}
  &\text{push } B \quad \text{stack}[\text{TOS}++] = \text{mem}[B] \\
  &\text{push } C \quad \text{stack}[\text{TOS}++] = \text{mem}[C] \\
  &\text{add} \quad \text{stack}[\text{TOS}++] = \text{stack}[\text{TOS}++] + \text{stack}[\text{TOS}++] \\
  &\text{pop } A \quad \text{mem}[A] = \text{stack}[\text{TOS}++] 
  \end{align*}
  \]

Operand Model III: Accumulator

- **Accumulator**: implicit single-element stack

  \[
  \begin{align*}
  &\text{load } B \quad \text{ACC} = \text{mem}[B] \\
  &\text{add } C \quad \text{ACC} = \text{ACC} + \text{mem}[C] \\
  &\text{store } A \quad \text{mem}[A] = \text{ACC}
  \end{align*}
  \]

Operand Model: Registers

- **General-purpose registers**: multiple explicit accumulators

  \[
  \begin{align*}
  &\text{load } B,R_1 \quad R_1 = \text{mem}[B] \\
  &\text{add } C,R_1 \quad R_1 = R_1 + \text{mem}[C] \\
  &\text{store } R_1,A \quad \text{mem}[A] = R_1 \\
  &\text{Load-store}: \text{GPR and only loads/stores access memory} \\
  &\text{load } B,R_1 \quad R_1 = \text{mem}[B] \\
  &\text{load } C,R_2 \quad R_2 = \text{mem}[C] \\
  &\text{add } R_1,R_2,R_1 \quad R_1 = R_1 + R_2 \\
  &\text{store } R_1,A \quad \text{mem}[A] = R_1
  \end{align*}
  \]
**Operand Model Pros and Cons**

- **Metric I: static code size**
  - Number of instructions needed to represent program, size of each
  - Want many implicit operands, high level instructions
  - Good → bad: memory, stack, accumulator, load-store

- **Metric II: data memory traffic**
  - Number of bytes moved to and from memory
  - Want as many long-lived operands in on-chip storage
  - Good → bad: load-store, accumulator, stack, memory

- **Metric III: instruction latency**
  - Want low latency to execute instruction
  - Good → bad: load-store, accumulator, stack, memory

- **Upshot**: most current ISAs are load-store

**How Many Registers?**

- Registers faster than memory, have as many as possible?
  - **No!**
  - One reason registers are faster is that there are fewer of them
  - Small is fast (hardware truism)
  - Another is that they are directly addressed (no address calc)
  - More of them, means larger specifiers
  - Means fewer registers per instruction or indirect addressing
  - Not everything can be put in registers
    - Structures, arrays, anything pointed-to
    - Although compilers are getting better at putting more things in
    - More registers means more saving/restoring them

- **Upshot**: trend to more registers: 8(IA-32) → 32(MIPS) → 128(IA-64)

**MIPS Operand Model**

- MIPS is load-store
  - 32 32-bit integer registers
    - Actually 31: r0 is hardwired to value 0 → why?
    - Other conventions as well
      - Certain registers conventionally used for special purposes
  - 32 32-bit FP registers
    - Can also be treated as 16 64-bit FP registers
  - HI,LO: destination registers for multiply/divide

- Integer register conventions
  - Allows separate function-level compilation and fast function calls

**Memory Addressing**

- ISAs assume “virtual” address size
  - Most 32-bit, but increasingly 64-bit
  - Program can name $2^{32}$ bytes (4GB) or $2^{64}$ bytes (16PB)
  - ISA point? no room for even one address in a 32-bit instruction

- **Addressing mode**: way of specifying address
  - Displacement: $\text{ld } R1, (R2)$
    - $R1 = \text{mem}[R2]$
  - Indirect: $\text{ld } R1, R8 (R2)$
    - $R1 = \text{mem}[R2 + R8]$
  - Index-base: $\text{ld } R1, (R2, R3)$
    - $R1 = \text{mem}[R2 + R3]$
  - Memory-indirect: $\text{ld } R1, R8 (R2)$
    - $R1 = \text{mem}[\text{mem}[R2]]$
  - Auto-increment: $\text{ld } R1, (R2) +$
    - $R1 = \text{mem}[R2 + *]$
  - Scaled: $\text{ld } R1, (R2, R3, 32, 8)$
    - $R1 = \text{mem}[R2 + R3*32 + 8]$

- What high-level program idioms are these used for?
MIPS Addressing Modes

- MIPS implements only displacement
  - Why? Experiment on VAX (ISA with every mode) found distribution
  - Disp: 61%, reg-ind: 19%, scaled: 11%, mem-ind: 5%, other: 4%
  - 80% use displacement or register indirect (=displacement 0)

- I-type instructions: 16-bit displacement
  - Is 16-bits enough?
  - Yes! VAX experiment showed 1% accesses use displacement >16

<table>
<thead>
<tr>
<th>l-type</th>
<th>Op(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Immed(16)</th>
</tr>
</thead>
</table>

Addressing Issue: Endian-ness

- **Byte Order**
  - **Big Endian**: byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, SPARC, HP PA-RISC
  - **Little Endian**: byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC/Compaq Alpha

Another Addressing Issue: Alignment

- **Alignment**: require that objects fall on address that is multiple of their size.
  - 32-bit integer
    - Aligned if address % 4 = 0
    - Aligned: lw @XXXX00, lh @XXXX00
    - Not: lw @XXXX10, lh @XXXX10
  - 64-bit integer?
    - Aligned if ?
    - Question: what to do with unaligned accesses (uncommon case)?
      - Support in hardware? Makes all accesses slow
      - Trap to software routine? Possibility
  - MIPS? ISA support: unaligned access using two instructions:
    - lw @XXXXX10 = lw1 @XXXXX10; lwr @XXXXX10