Instruction Set Architecture (ISA)

- ISAs in General
  - Using MIPS as primary example
  - MIPS Assembly Programming
  - Other ISAs

![Diagram showing application, OS, compiler, firmware, CPU, I/O, memory, digital circuits, and gates & transistors.]

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/method calls? If so, how?
  - Should the hardware support other features that are specific to certain HLLs (e.g., garbage collection for Java)?

Microarchitecture

- ISA specifies what hardware does, not how it does it
  - No guarantees regarding these issues:
    - 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 4 are almost the same architecture, but are very different microarchitectures)
  - Class project is to build Duke152-S12 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 – Duke152/32 is 32 bit)
     + Simple implementation: compute next PC using only this PC
     - Code density: 32 or 64 bits for a NOP (no operation) insn?
  2. Variable length
     - Complex implementation
     + Code density
  3. Compromise: two lengths
     - Example: MIPS

- **Encoding**
  - A few simple encodings simplify decoder implementation
  - You’ll appreciate simple encodings when building Duke152/32

---

MIPS Format

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

- **Encoding**
  - 3 formats, simple encoding, 6-bit opcode (type of operation)
  - ICQ: how many operation types can be encoded in 6-bit opcode?

<table>
<thead>
<tr>
<th>R-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>I-type</td>
<td>Op(6)</td>
<td>Rs(5)</td>
<td>Rd(5)</td>
<td>Immed(16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-type</td>
<td>Op(6)</td>
<td>Target(26)</td>
<td></td>
<td></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
  - Bit-wise/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 (ICQ: Why not?)
    - 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
  - Note: operands don’t have to be unique (e.g., A = B + A)
  - 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 execution latency**
Operand Model I: Memory Only

- **Memory only**
  
  ```
  ```

Operand Model II: Stack

- **Stack**: top of stack (TOS) is implicit operand in all insns
  
  ```
  push B // stack[TOS++] = mem[B]
  push C // stack[TOS++] = mem[C]
  add // stack[TOS++] = stack[-TOS] + stack[-TOS]
  pop A // mem[A] = stack[-TOS]
  ```

Operand Model III: Accumulator

- **Accumulator**: implicit single-element stack
  
  ```
  load B // ACC = mem[B]
  add C // ACC = ACC + mem[C]
  store A // mem[A] = ACC
  ```

Operand Model IV: Registers

- **General-purpose registers**: multiple explicit accumulators
  
  ```
  load R1,B // R1 = mem[B]
  add R1,C // R1 = R1 + mem[C]
  store A,R1 // mem[A] = R1
  ```

- **Load-store**: GPR and only loads/stores access memory
  
  ```
  load R1,B // R1 = mem[B]
  load R2,C // R2 = mem[C]
  add R3,R2,R1 // R3 = R1 + R2
  store A,R3 // mem[A] = R3
  ```

You may remember that the ECE52 protocomputer has an accumulator ISA.
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: many 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
  - Smaller storage structures are faster (hardware truism)
  - Another is that they are directly addressed (no address calc)
  - More registers → larger specifiers → fewer regs per instruction
  - 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
    - At procedure calls and context switches
  - 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 → ICQ: why?
    - Also, certain registers conventionally used for special purposes
      - We’ll talk more about these conventions later
  - 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
  - Note: “function”, “method”, and “procedure” are equivalent terms in this course
  - We’ll discuss this more when we get to procedure calls

---

Memory Operand Addressing

- ISAs assume “virtual” address size
  - Either 32-bit or 64-bit
  - Program can name 2^32 bytes (4GB) or 2^64 bytes (16EB)
  - ISA impact? no room for even one address in a 32-bit instruction

- **Addressing mode**: way of specifying address
  - (Register) Indirect: 1d r1, (r2) R1=mem[R2]
  - Displacement: 1d r1, 8(r2) R1=mem[R2+8]
  - Index-base: 1d r1, (r2, r3) R1=mem[r2+r3]
  - Memory-indirect: 1d r1, @[r2] R1=mem[mem[r2]]
  - Auto-increment: 1d r1, (r2)+ R1=mem[r2+1]
  - Scaled: 1d r1, (r2, r3, 32, 8) R1=mem[r2+r3*32+8]

- ICQ: What HLL program idioms are these used for?
MIPS Addressing Modes

- MIPS implements only displacement addressing mode
  - 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

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 [% is symbol for "mod"]
    - Aligned: lw @XXXX0
    - Not: lw @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 @XXXX10 = lw1 @XXXX10; lw2 @XXXX10

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

Datatypes

- Datatypes
  - Software view: property of data
  - Hardware view: data is just bits, property of operations
- Hardware datatypes
  - Integer: 8 bits (byte), 16b (half), 32b (word), 64b (long)
  - IEEE754 FP: 32b (single-precision), 64b (double-precision)
  - Packed integer: treat 64b int as 8 8b int's or 4 16b int's
MIPS Datatypes (and Operations)

- Datatypes: all the basic ones (byte, half, word, FP)
  - All integer operations read/write 32-bits
    - No partial dependences on registers
  - Only byte/half variants are load/store
    \(lb, lbu, lh, lh\), \(sb, sh\)
  - Loads sign-extend (or not) byte/half into 32-bits

- Operations: all the basic ones
  - Signed/unsigned variants for integer arithmetic
  - Immediate variants for all instructions
    \(add, addu, addi, addiu\)
  - Regularity/orthogonality: all variants available for all operations
    - Makes compiler’s “life” easier