Instruction Set Architecture (ISA)

- ISAs in General
  - Using MIPS as primary example
- MIPS Assembly Programming
- Other ISAs
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 Duke152/16 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 – Duke152/16 is 16 bit)
     + Simple implementation: compute next PC using only this PC
     – Code density: 32 or 64 bits for a NOP (no operation)?
  2. Variable length
     – Complex implementation
     + Code density
  3. Compromise: two lengths
     • Example: MIPS$_{16}$

- **Encoding**
  • A few simple encodings simplify decoder implementation
  • You’ll appreciate simple encodings when building Duke152/16
MIPS Format

- **Length**
  - 32-bits
  - MIPS\textsubscript{16}: 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?

\begin{tabular}{|c|c|c|c|c|c|c|}
  \hline
  R-type & Op(6) & Rs(5) & Rt(5) & Rd(5) & Sh(5) & Func(6) \\
  \hline
  I-type & Op(6) & Rs(5) & Rt(5) & & Immed(16) \\
  \hline
  J-type & Op(6) & & & Target(26) \\
  \hline
\end{tabular}
(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 (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**
  
  ```
  ```

  international symbol for **Arithmetic Logic Unit (ALU)** – a piece of logic that performs arithmetic, bitwise logic, shifts, etc.
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]

Intel x86 ISA uses a stack architecture for its floating point computations
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

You may remember that the ECE52 protocomputer has an accumulator ISA
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
  ```
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