CS184b: Computer Architecture
[Single Threaded Architecture: abstractions, quantification, and optimizations]

Day3: January 11, 2000
Instruction Set Architecture

Today

- Datapath review
- H&P view
- Questions about
- Themes
- Compilers
Terminology

• **Primitive Instruction** (*pinst*)
  – Collection of bits which tell a single bit-processing element what to do
  – Includes:
    • select compute operation
    • input sources in space
      – (interconnect)
    • input sources in time
      – (retiming)

Instructions

• Distinguishing feature of programmable architectures?
  – *Instructions* -- bits which tell the device how to behave
Single ALU Datapath

Datapath

[Datapath from PH (Fig. 5.1)]
Instructions

- Primitive operations for constructing (describing) a computation
- Need to do?
  - Interconnect (space and time)
  - Compute (intersect bits)
  - Control (select operations to run)

Detail Datapath

[Datapath from PH (Fig. 5.13)]
uCoded / Decoded

• uCoded
  – Bits directly control datapath
  – Horizontal vs. Vertical
  – Not abstract from implementation

• Decoded
  – more compressed
  – only support most common operations
  – abstract from implementation
  – time/area to decode to datapath control signals

H&P View

• ISA design done?
• Not many opportunities to completely redefine
• Many things mostly settled
  – at least until big technology perturbations arrive
• Implementation (uArch) is where most of the action is
• Andre: maybe we’ve found a nice local minima...
H&P Issues

- Registers/stack/accumulator
  - # operands, memory ops in instruction
- Addressing Modes
- Operations
- Control flow
- Primitive Data types
- Encoding

Register/stack/accumulator

- Driven largely by cost model
  - ports into memory
  - latency of register versus memory
  - instruction encoding (bits to specify)
Register/stack/accumulator

- **Today**: Load-Store, General Register arch.
- Registers more freedom of addressing than stack
- Load into register, then operate
  - not much longer than memory address
  - usually use more than once (net reduction)

Addressing Modes

- Minimal:
  - immediate
  - register
  - register indirect
- Others:
  - displacement
  - indirect (double derference)
  - auto increment/decrement (p[x++]=y)
  - scaled
Addressing Modes

- More / More capable
  - less instructions
  - potentially longer instructions
    - bits and cycle time
  - many operations (complicate atomicity of instructions)
    - Add (R2)+,(R3)+,(R4)+

Operations

- ALU/Arithmetic
  - add, sub, or, and, xor
  - compare
- Interconnect
  - move registers
  - load, store
- Control
  - jump
  - conditional branch
  - procedure call/return
Operations: ALU

- Small set of SIMD operations
- Covers very small fraction of the space of all $w \times w \rightarrow w$

Operations: Branching

- Models:
  - ops set condition codes, branch on condition codes
  - compare result to register, branch on register zero or one
  - comparison part of branch
Operations: Procedure call/return

• ? Save registers?
• Update PC
  – call target
  – return address
• Change stack and frame pointers
  – store old
  – install new

Operations: Procedure call/return

• **Question:** How much should instruction do?
• Lesson: High variance in work needs to be done
  – which registers need to save
  – best way to transfer arguments to procedures
  – better to expose primitives to the compiler and let it specialize the set of operations to the particular call
Data Types

- Powers of two from bytes to double worlds?
  - 8, 16, 32, 64
  - (very implementation driven decision)
- Floating Point types
- Are pointers integers?
- Alignment requirements

Encoding

- Variable vs. Fixed
- How complex is the decoding?
  - Fields in the same place…or have to be routed/muxed?
  - Sequential requirements in decode?
Detail Datapath

[Datapath from PH (Fig. 5.13)]

Encoding: RISC/Modern

[DLX Instruction Format from HP (Fig. 2.21)]
Operation Complexity

• Contradiction?
  – Providing primitives
  – including floating point ops

Local Minima?

• Self-Fulfilling?
  – How would we quantitatively validate need for a new operation?
  – [cue: bridge story]
  – This is what we use as primitives
  – Funny, we don’t find a need for other primitives…
Themes

- Common case fast
- Provide primitives (building blocks)
- Let compiler specialize to particular operation
- Make decode/operation simple so implementation is fast

Compilers

- 1960→1990 shift
  - increasing capability and sophistication of compilers
  - e.g.
    - inter-procedural optimization
    - register assignment (register usage)
    - strength reduction
    - dataflow analysis and instruction reordering
    - (some progress) alias analysis
Compilers

• Gap between programmer and Architecture
• Increasingly bridged by compiler
• Less need to make assembly language human programmable
• More opportunity for compiler to specialize, partial evaluate
  – (do stuff at compile time to reduce runtime)
• RISC: “Relegate Interesting Stuff to Compiler”

Implementation Significance

• André Agree: Implementation issues are significant in the design of ISA

• Many of these issues are more interesting when we discuss in light of implementation issues
ISA Driven by

- Implementation costs
- Compiler technology
- Application structure

- Can’t do good architecture in isolation from any of these issues.

Upcoming

- Next Time:
  - discuss RISC CISC
- Following that
  - pipelining ISA
  - (day ahead of original schedule since got today back)
Big Ideas

• Common Case
• Primitives
• Highly specialized instructions brittle
• Design pulls
  – simplify processor implementation
  – simplify coding
• Orthogonality (limit special cases)
• Compiler: fill in gap between user and hardware architecture