Outline

• Instruction Sets in General
• MIPS Assembly Programming
• Other Instruction Sets
  • Goals of ISA Design
  • RISC vs. CISC
  • Intel x86 (IA-32)

What Makes a Good ISA?

• Programmability
  • Easy to express programs efficiently?
• Implementability
  • Easy to design high-performance implementations (i.e., microarchitectures)?
• Compatibility
  • Easy to maintain programmability (implementability) as languages and programs (technology) evolve?

Programmability

• Easy to express programs efficiently?
  • For whom?
  • Human
    • Want high-level coarse-grain instructions
      • As similar to HLL as possible
      • This is the way ISAs were pre-1985
        • Compilers were terrible, most code was hand-assembled
  • Compiler
    • Want low-level fine-grain instructions
      • Compiler can’t tell if two high-level idioms match exactly or not
      • This is the way most post-1985 ISAs are
        • Optimizing compilers generate much better code humans do

Implementability

• Every ISA can be implemented
  • But not every ISA can be implemented well
    • Bad ISA $\rightarrow$ bad microarchitecture (slow, power-hungry, etc.)
• Classic high-performance implementation techniques
  • Pipelining, parallel execution, out-of-order execution
    • We’ll discuss these later in the semester
• Certain ISA features make these difficult
  • Variable length instructions
  • Implicit state
  • Variable formats
  • Coarse granularity
Compatibility

- Nobody buys new hardware… if it means you have to buy new software, too
  - Intel was the first company to realize this
  - ISA must stay stable, no matter what (microarch. can change)
    - x86 is one of the ugliest ISAs EVER, but survives
  - Intel then forgot this lesson: IA-64 (Itanium) is new ISA
- **Backward compatibility**: very important
  - New processors must support old programs (can’t drop features)
- **Forward (upward) compatibility**: less important
  - Old processors must support new programs
  - New processors only re-define opcodes that trapped in old ones
  - Old processors emulate new instructions in low-level software
  - Can also be used to ensure backwards compatibility

---

RISC vs. CISC

- **RISC**: reduced-instruction set computer
  - Coined by P+H in early 80’s
- **CISC**: complex-instruction set computer
  - Not coined by anyone, term didn’t exist before “RISC”
  - Religious war (one of several) started in mid 1980’s
    - RISC (MIPS, Alpha) “won” the technology battles
    - CISC (IA32 = x86) “won” the commercial war
      - Compatibility a stronger force than anyone (but Intel) thought
      - Intel beat RISC at its own game … more on this soon

---

The Setup

- Pre 1980
  - Bad compilers
  - Complex, high-level ISAs
  - Slow multi-chip micro-programmed implementations
    - Vicious feedback loop
- Around 1982
  - Advances in VLSI made single-chip microprocessor possible…
    - Speed by integration, on-chip wires much faster than off-chip
      - …but only for very small, very simple ISAs
  - Compilers had to get involved in a big way
- **RISC manifesto**: create ISAs that…
  - Simplify single-chip implementation
  - Facilitate optimizing compilation

---

The RISC Tenets

- **Single-cycle execution**
  - CISC: many multi-cycle ops
- **Hardwired control**
  - CISC: microcode
- **Load/store architecture**
  - CISC: register memory and memory memory instructions
- **Few memory addressing modes**
  - CISC: many modes
- **Fixed instruction format**
  - CISC: many formats and lengths
- **Reliance on compiler optimizations**
  - CISC: hand assemble to get good performance
PowerPC ISA

- RISCy, very similar to MIPS
- Indexed Addressing (register+register)
  \[ \text{lw } \$t1, \$a0, \$s3 \Rightarrow \$t1 = \text{mem}[\$a0 + \$s3] \]
- Update Addressing
  \[ \text{lw } \$t1, 4(\$a0) \Rightarrow \$t1 = \text{mem}[\$a0 + 4]; \$a0 \equiv 4; \]
- Load/Store Multiple
- Counter Register
  - \text{bc loop, cfr} \equiv 0
  - decrement cfr, if cfr \equiv 0 go to loop

Intel 80x86 ISA (aka x86 or IA-32 now)

- Long history
- Binary compatibility
- 1978: 8086, 16-bit, registers have dedicated uses
- 1980: 8087, added floating point (stack)
- 1982: 80286, 24-bit
- 1985: 80386, 32-bit, new insts \(\rightarrow\) GPR almost
- 1989-95: 80486, Pentium, Pentium II
- 1997: Added MMX instructions (for graphics)
- 1999: Pentium III
- 2002: Pentium 4
- 2004: “Nocona” 64-bit extension (to keep up with AMD)

Intel x86: The Penultimate CISC

- VAX was ultimate CISC, but x86 (IA-32) is close
  - Variable length instructions: 1-16 bytes
  - Few registers: 8 and each one has a special purpose
  - Multiple register sizes: 8,16,32 bit (for backward compatibility)
    - AMD Hammer, Intel IA-64 also have 64-bit
  - Accumulators for integer, stack for FP
  - Multiple addressing modes: indirect, scaled, displacement
  - Register-register, memory-register, and memory-register insns
  - Condition codes
  - Two-level virtual addressing (segments)
  - Instructions for memory stack management (push, pop)
  - Instructions for manipulating strings (entire loop in one instruction)

80x86 Registers and Addressing Modes

- Eight 32-bit GPRs
  - EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- Six 16-bit Registers for code, stack, & data
- 2-address ISA
  - One operand is both source and destination
- NOT a Load/Store ISA
  - One operand can be in memory
80x86 Addressing Modes

- Register Indirect
  - `mem[reg]`
  - not ESP or EBP
- Base + displacement (8 or 32 bit)
  - `mem[reg + const]`
  - not ESP or EBP
- Base + scaled Index
  - `mem[reg + (2\text{scale} \times \text{index})]`
  - scale = 0, 1, 2, 3
  - base any GPR, Index not ESP
- Base + scaled Index + displacement
  - `mem[reg + (2\text{scale} \times \text{index}) + \text{displacement}]`
  - scale = 0, 1, 2, 3
  - base any GPR, Index not ESP

Condition Codes

- Both PowerPC and x86 ISA have condition codes
- Special HW register, that has values set as side effect of instruction execution
- Example conditions
  - Zero
  - Negative
- Example use
  ```
  subi $t0, $t0, 1
  bz loop
  ```

80x86 Instruction Encoding

- Variable Size 1-byte to 17-bytes
- Jump (JE) 2-bytes
- Push 1-byte
- Add Immediate 5-bytes
- W bit says 32-bits or 8-bits
- D bit indicates direction
  - `memory \rightarrow \text{reg or reg} \rightarrow \text{memory}`
  - `movw EBX, [EDI + 45]`
  - `movw [EDI + 45], EBX`

Decoding x86 Instructions

- Is a nightmare!
- Instruction length is variable from 1 to 17 bytes!
- Prefixes, postfixes
- Crazy "formats" → register specifiers move around
- But key instructions not terrible
- Yet, everything must work correctly
How Intel Won Anyway

- X86 won because it was the first 16-bit chip by 2 years
  - IBM put it into its PCs because there was no competing choice
  - Rest is historical inertia and “financial feedback”
    - X86 is most difficult ISA to implement and do it fast but…
    - Because Intel sells the most processors…
    - It has the most money…
    - Which it uses to hire more and better engineers…
    - Which it uses to maintain competitive performance …
    - And given equal performance compatibility wins…
  - So Intel sells the most processors…

- Moore’s law has helped Intel in a big way
  - Most engineering problems can be solved with more transistors

Current Approach: Pentium Pro

- Instruction decode logic translates into μops
- Fixed-size instructions moving down execution path
- Execution units see only μops
- Faster instruction processing with backward compatibility
- Execution unit as fast as RISC machines like MIPS
  - Complex decoding
  - We work with MIPS to keep decoding simple/clean
  - Learn x86 on the job!

Aside: Complex Instructions

- More powerful instructions → not necessarily faster execution
  - E.g., string copy or polynomial evaluation

- Option 1: use “repeat” prefix on memory-memory move inst
  - Custom string copy

- Option 2: use a loop of loads and stores through registers
  - General purpose move through simple instructions

- Option 2 is faster on same machine, e.g., Pentium III

Concluding Remarks

1. Keep it simple and regular
   - Uniform length instructions
   - Fields always in same places
2. Keep it simple and fast
   - Small number of registers
3. Make sure design can be pipelined (will learn soon)
4. Make the common case fast

- Compromises inevitable → there is no perfect ISA
Outline

• Instruction Sets in General
• MIPS Assembly Programming
• Other Instruction Sets