RISC vs CISC
In the Beginning...

- 1964 -- The first ISA appears on the IBM System 360
- In the “good” old days
  - Initially, the focus was on usability by humans.
  - Lots of “user-friendly” instructions (remember the x86 addressing modes).
  - Memory was expensive, so code-density mattered.
  - Many processors were *microcoded* -- each instruction actually triggered the execution of a builtin function in the CPU. Simple hardware to execute complex instructions (but CPIs are very, very high)
- ...so...
  - Many, many different instructions, lots of bells and whistles
  - Variable-length instruction encoding to save space.
- ... their success had some downsides...
  - ISAs evolved organically.
  - They got messier, and more complex.
Things Changed

• In the modern era
  • Compilers write code, not humans.
  • Memory is cheap. Code density is unimportant.
  • Low CPI should be possible, but only for simple instructions
  • We learned a lot about how to design ISAs, how to let them evolve gracefully, etc.

• So, architects started with with a clean slate...
Reduced Instruction Set Computing (RISC)

- Simple, regular ISAs, mean simple CPUs, and simple CPUs can go fast.
  - Fast clocks.
  - Low CPI.
  - Simple ISAs will also mean more instruction (increasing IC), but the benefits should outweigh this.

- Compiler-friendly, not user-friendly.
  - Simple, regular ISAs, will be easy for compilers to use
  - A few, simple, flexible, fast operations that compiler can combine easily.
  - Separate memory access and data manipulation
    - Instructions access memory or manipulate register values. Not both.
    - “Load-store architectures” (like MIPS)
## Instruction Formats

### R-Type
- **Opcode**: 6 bits
- **rs**: 5 bits
- **rt**: 5 bits
- **rd**: 5 bits
- **shamt**: 5 bits
- **funct**: 6 bits

**Arithmetic**: `Register[rd] = Register[rs] + Register[rt]`

**Register Indirect Jumps**: `PC = PC + Register[rs]`

### I-Type
- **Opcode**: 6 bits
- **rs**: 5 bits
- **rt**: 5 bits
- **Immediate**: 16 bits

**Arithmetic**: `Register[rd] = Register[rs] + Imm`

**Branches**: If `Register[rs] == Register[rt]`, goto `PC + Immediate`

**Memory**: `Memory[Register[rs] + Immediate] = Register[rt]`

**Register**: `Register[rt] = Memory[Register[rs] + Immediate]`

### J-Type
- **Opcode**: 6 bits
- **Address**: 26 bits

**Direct Jumps**: `PC = Address`

**Syscalls, break, etc.**
RISC Characteristics of MIPS

- All instructions have
  - \(\leq 1\) arithmetic op
  - \(\leq 1\) memory access
  - \(\leq 2\) register reads
  - \(\leq 1\) register write
  - \(\leq 1\) branch
  - It needs a small, fixed amount of hardware.

- Instructions operate on memory or registers \textit{not both}
  - “Load/Store Architecture”

- Decoding is easy
  - Uniform opcode location
  - Uniform register location
  - Always 4 bytes -> the location of the next PC is to know.

- Uniform execution algorithm
  - Fetch
  - Decode
  - Execute
  - Memory
  - Write Back

- Compiling is easy
  - No complex instructions to reason about
  - No special registers

- The HW is simple
  - A skilled undergrad can build one in 10 weeks.
  - 33 instructions can run complex programs.
CISC: x86

• x86 is the prime example of CISC (there were many others long ago)
  • Many, many instruction formats. Variable length.
  • Many complex rules about which register can be used when, and which addressing modes are valid where.
  • Very complex instructions
  • Combined memory/arithmetic.
  • Special-purpose registers.
  • Many, many instructions.

• Implementing x86 correctly is almost intractable
Mostly RISC: ARM

• ARM is somewhere in between
  • Four instruction formats. Fixed length.
  • General purpose registers (except the condition codes)
  • Moderately complex instructions, but they are still “regular” -- all instructions look more or less the same.

• ARM targeted embedded systems
  • Code density is important
  • Performance (and clock speed) is less critical
  • Both of these argue for more complex instructions.
  • But they can still be regular, easy to decode, and crafted to minimize hardware complexity

• Implementing an ARM processor is also tractable for 141L, but it would be harder than MIPS
RISCing the CISC

• Everyone believes that RISC ISAs are better for building fast processors.
• So, how do Intel and AMD build fast x86 processors?
  • Despite using a CISC ISA, these processors are actually RISC processors inside.
  • Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor.
  • The preceding was a dramatization. MIPS instructions were used for clarity and because I had some laying around.
  • No x86 instruction were harmed in the production of this slide.

```
  movb $0x05, %al
  movl -4(%ebp), %eax
  movl %eax, -4(%ebp)
  movl %R0, -4(%R1,%R2,4)
  movl %R0, %R1

  ori $t0, $t0, 5
  lw $t0, -4($t1)
  sw $t0, -4($t1)
  slr $at, $t2, 2
  add $at, $at, $t1
  sw $t0, k($at)
  ori $t0, $t0, $zero
```

VLIWing the CISC

- We can also get rid of x86 in software.
- Transmeta did this.
  - They built a processor that was completely hidden behind a “soft” implementation of the x86 instruction set.
  - Their system would translate x86 instruction into an internal VLIW instruction set and execute that instead.
  - Originally, their aim was high performance.
  - That turned out to be hard, so they focused low power instead.
- Transmeta eventually lost to Intel
  - Once Intel decided it cared about power (in part because Transmeta made the case for low-power x86 processors), it started producing very efficient CPUs.
The End