

Evolution of ISAs

Instruction set architectures have changed over computer generations with changes in the

- cost of the hardware
- density of the hardware
- design philosophy
- potential performance gains

One way to characterize ISAs:

- number of addresses/instruction
- regularity of instruction formats
- number of addressing modes

Number & Type of Operands/Instruction

One address & an implied accumulator register

- hardware was expensive & huge
- **accumulator architecture**: EDSAC (1949)
 - `load AddressB # accum = Memory[AddressB]`
 - `add AddressC # accum = accum + Memory[AddressC]`
 - `store AddressA # Memory[AddressA] = accum`

One address & a few special purpose registers

- **extended accumulator** (special-purpose register) architecture: Intel 8086
- registers for:
 - data
 - addresses
 - segment pointers
 - special, e.g., PC

Number & Type of Operands/Instruction

General-purpose registers

- **register-memory architectures**
 - one operand is in memory: IBM 360 (1964)
`add reg10, AddressA`
- **memory-memory architectures**
 - all operands can be in memory: VAX 780 (1977)
`add AddressA, AddressB, AddressC`
- **load-store architectures**
 - CDC 6600 (1963), Cray 1
 - current RISCs (1982 and on)

One address & no registers

- **stack architectures**: Burroughs 5000, Intel 8087
- use the top of the stack for other, implied operands

```
push AddressC # increment stack pointer
                # TOS = Memory[AddressB]

push AddressB # do it again

add             # add top 2 locations; result on TOS]

pop AddressA  # Memory[AddressA] = TOS
                # decrement stack pointer
```

Regularity of Instruction Formats

Started with 1 format

- for ease of programming (programming on the binary level!)

Then 3 or 4 formats, not necessarily the same length

- assembly language & compilers made programming easier

More formats

- small, low density, expensive memory
- CPU-to-memory bottleneck
- ISA's:
 - reflected high-level language operations
 - had complicated addressing modes (*next*)
 - had variable length instructions
 - IBM 360: instructions can be 2,4 or 6 bytes
 - Intel x86: 1 to 17 bytes
 - DEC VAX: 1 to 54 bytes
- Examples:
 - IBM 360: 5
 - Intel x86: lots
 - DEC VAX: also lots
 - **orthogonal design**: all opcodes can be used with any addressing mode & any information unit

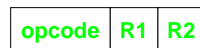
Regularity of Instruction Formats

Back to fixed length instructions, few formats

- memory is large & cheap
- simple encoding facilitates faster hardware interpretation of instructions

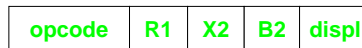
IBM 360 Formats

RR format



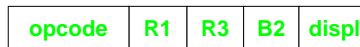
$R1 \leftarrow R1 \text{ op } R2$

RX format



$R1 \leftarrow R1 \text{ op mem}[X2+B2+\text{displ}]$

RS format



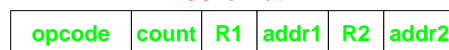
$\text{mem}[B2+\text{displ}] \leftarrow R1 \text{ op } R3$

SI format



$\text{mem}[B2+\text{displ}] \leftarrow \text{mem}[B2+\text{displ}] \text{ op value}$

SS format



$\text{mem}[\text{addr2}+R2+i] \leftarrow \text{mem}[\text{addr1}+R1+i], 0 \leq i < \text{count}$

Addressing Modes

Start with immediate, direct, indirect (also called deferred)

indirect: register contains the address of the operand

Then index registers

special registers for an array index

Then index + base

allow the sum of 2 registers to be an address

Even more....

VAX Addressing Modes

Immediate

different addressing modes for constants of different sizes

different addressing modes for data and address constants

Register: **reg**

the register contains the operand

Register deferred: **(reg)**

the register contains the address of the operand

Autoincrement: **(reg)+**

the register contains the address of the operand & is incremented by the size of the operand *after* it's accessed

Autodecrement: **-(reg)**

the address in the register is decremented *before* the access

Autoincrement deferred: **@(reg)+**

address in the register is a *pointer* to the address of the operand & is incremented by the size of the operand

PC-relative:

both regular & deferred

VAX Addressing Modes

Displacement: $\text{displ}(\text{reg})$

separate addressing modes for each displacement length (one for each information unit) which is stored only in the number of bits needed & then sign extended
both regular & deferred

Indexed:

used in conjunction with other addressing modes

the contents of the **index register** is multiplied by the size of the operand in bytes & added to the contents of the other register

- **register deferred indexed:** $(\text{reg})[\text{IndexReg}]$
- **autoincrement indexed:** $(\text{reg}) + [\text{IndexReg}]$
- **autodecrement indexed:** $-(\text{reg})[\text{IndexReg}]$
- **autoincrement deferred indexed:** $@(\text{reg}) + [\text{IndexReg}]$
- **displacement indexed:** $\text{displ}(\text{reg})[\text{IndexReg}]$
- **displacement deferred indexed:**
 $\{\text{@displ}(\text{reg})\}[\text{IndexReg}]$

add the displacement & the contents of reg to form a pointer to the base address (the address where the base address is stored); the base address is fetched & added to the adjusted index in IndexReg to form the operand address; the operand is then fetched

Intel x86

85% of the microprocessors in the world (not counting embedded processors)

Only 8 GPRs (other registers are special purpose)

Register-memory architecture

- 2 operand instructions; 1 is both source & destination
- Addressing relies on segments (code, stack & static data)

String instructions in addition to computation, data transfer, control

Condition codes instead of condition registers

No regularity in the ISA

- ISA for 8 bits & an extended ones for 16, 32 & 64 bits
- Lots of addressing modes (but fewer than the VAX)
 - some can't use certain registers
- Variable length instructions, variable length opcodes (*later*)
 - addresses in bytes, not instructions
- Encoding is complicated (see Figure 3.35)

RISC Vs. CISC

RISC

general philosophy:

simple instructions execute faster than complex instructions

- less to do
- fewer choices; therefore it takes less time to decide what is being executed now (smaller circuitry)
- *simplicity leads to regularity* in the hardware design easier to get the hardware right & to debug it

use simple instructions as building blocks for more complex operations

short cycle time & single-cycle instructions; therefore more instructions executed per time unit

- few instructions, simple instructions
- few, simple addressing modes
- fixed-length instructions (32 bits)
- few instruction formats: (almost) fixed fields within an instruction
- load/store architecture
- hardwired control (*later*)
- conducive to **pipelining** because each instruction takes about the same amount of time to execute (*later*)
- expose the implementation to the compiler/programmer

RISC Vs. CISC

CISC

general philosophy:

get faster execution by having a better match between the high-level operations & the hardware operations

direct execution of one instruction in hardware is faster than many instructions in software

tight encoding & fewer instructions

⇒ fewer bytes brought in from memory

developed when:

- memory was expensive
- caching was not so widespread

- large number of instructions

examples:

- `subgtz` for loop indexing
- search for a substring
- evaluate a polynomial

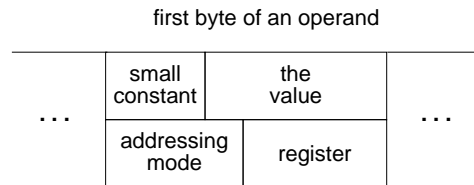
- many addressing modes
- variable-length instructions

opcode	info unit	# operands	...
--------	-----------	------------	-----

RISC Vs. CISC

CISC, cont'd.

- lots of instruction formats; use the same fields for different purposes



- memory-to-memory architecture
- **microprogrammed** control (*later*)
- difficult to pipeline because instructions take vastly different amount of time to execute (*later*)
- implementation hidden from the compiler/programmer; separation of architecture & implementation

RISC Vs. CISC

Why the change to RISC?

- performance studies showed that:
 - few instructions were used most of the time (VAX study: 15/90%. 26/95%)
 - very complex instructions were never generated
- increase in technology density
 - instruction caches for loops
- advances in compiler technology that enabled code to be scheduled to hide operation latencies in a pipeline (*later*)

Bottom line:

- fast instruction execution, pipelining, compiler support for pipelining, onchip caches, more general-purpose registers
more important than
- encoded instructions & functionality in hardware
- result: lots of fast instructions executed more quickly than slower, fewer instructions

RISC VS. CISC

A comparison is difficult today

- Pentium Pro hardware translates instructions into **uops** (very RISC-like micro operations) & pipelines uop execution
- improvements in implementation that are somewhat architecture-independent
 - superscalar execution (wide issue width)
 - speculative execution
 - out-of-order execution & register renaming
 - branch prediction with 96% accuracy
 - huge on-chip caches
 - multithreading
- majority of microprocessor market ⇒ \$\$\$ ⇒ more engineers on microprocessor design teams & better fabrication lines

MIPS is Not the Only RISC

RISC architectures began (again) in the 1980's in reaction to more complex CISCs

- Cray & CDC 6600 were early load-store architectures
- research at IBM on the IBM 801 (the first RISC processor) became the RT/PC ⇒ RS6000 ⇒ PowerPC
- research at Stanford on the MIPS led to MIPS Rxxx
- research at Berkeley on the RISC I & II led to Sun SPARC processors
- HP Precision
- DEC/Compaq Alphas (the fastest cycle time today)