



**ISAs**  
**revisited**



# Resources/readings

Intel® 64 and IA-32 Architectures Software Developer Manuals

Arm A64 Instruction Set Architecture

RISC and CISC comparison paper (1991)

RISC vs CISC power struggles paper (2013)

Agner Fog's instruction tables (for uop analysis)

# Terminology review

ISA (Instruction set architecture): interface between high-level programming language and hardware

Instructions

Registers

Memory models

I/O model

Microarchitecture: hardware implementation of ISA

Today: how do different ISAs compare? Does choice of ISA limit HW performance?

# Back in the day: accumulator architectures

Single-register architecture (register called “accumulator”)

All arithmetic operations have the accumulator as source and destination

e.g. ADD 200 means: add value at mem. address 200 to accumulator and store result in accumulator

Born of necessity (registers were expensive!)

Fun reading: [PDP-8 instruction set](#)

# Computations in memory, or not?

Load/store or register-register architectures: all arithmetic operations done in registers (need to load from memory into register first)

RISC-V, MIPS, Arm

Register-memory architectures: arithmetic operations can be done using combination of registers and memory addrs

x86

# Comparing 64-bit ADD instructions

## RISC-V

ADD, ADDW (32-bit add), ADDI, ADDIW

## Armv8

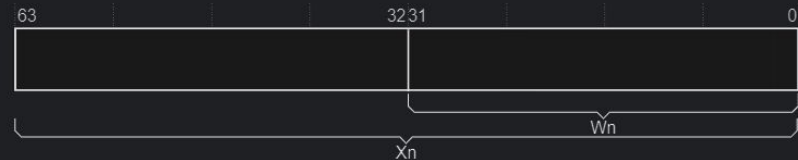
Same mnemonic (ADD), different machine instructions

image  
source

ADD W0, W1, W2 (32-bit) ADD X0, X1, X2 (64-bit) ADD X0, X1, W2, SXTW  
(sign-extended) ADD X0, X1, #42

Each AArch64 64-bit general-purpose register (X0-X30) also has a 32-bit (W0-W30) form.

Figure 4.2. 64-bit register with W and X access.



# x86-64 ADD

## ADD—Add

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
04 ib	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
05 iw	ADD AX, imm16	I	Valid	Valid	Add imm16 to AX.
05 id	ADD EAX, imm32	I	Valid	Valid	Add imm32 to EAX.
REX.W + 05 id	ADD RAX, imm32	I	Valid	N.E.	Add imm32 sign-extended to 64-bits to RAX.
80 /0 ib	ADD r/m8, imm8	MI	Valid	Valid	Add imm8 to r/m8.
REX + 80 /0 ib	ADD r/m8*, imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m8.
81 /0 iw	ADD r/m16, imm16	MI	Valid	Valid	Add imm16 to r/m16.
81 /0 id	ADD r/m32, imm32	MI	Valid	Valid	Add imm32 to r/m32.
REX.W + 81 /0 id	ADD r/m64, imm32	MI	Valid	N.E.	Add imm32 sign-extended to 64-bits to r/m64.
83 /0 ib	ADD r/m16, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m16.
83 /0 id	ADD r/m32, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m32.
REX.W + 83 /0 ib	ADD r/m64, imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m64.
00 /r	ADD r/m8, r8	MR	Valid	Valid	Add r8 to r/m8.
REX + 00 /r	ADD r/m8*, r8*	MR	Valid	N.E.	Add r8 to r/m8.
01 /r	ADD r/m16, r16	MR	Valid	Valid	Add r16 to r/m16.
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.
02 /r	ADD r8, r/m8	RM	Valid	Valid	Add r/m8 to r8.
REX + 02 /r	ADD r8*, r/m8*	RM	Valid	N.E.	Add r/m8 to r8.
03 /r	ADD r16, r/m16	RM	Valid	Valid	Add r/m16 to r16.
03 /r	ADD r32, r/m32	RM	Valid	Valid	Add r/m32 to r32.
REX.W + 03 /r	ADD r64, r/m64	RM	Valid	N.E.	Add r/m64 to r64.

### NOTES:

\*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

# RISC vs CISC

RISC-V, MIPS, Arm are RISC (Reduced Instruction Set Computer) architectures

x86 is CISC (Complex Instruction Set Computer) architecture

- Allows register-memory instrs

- Allows variable-length instruction encodings

- Allows instructions that take longer than 1 cycle

*Except...* this distinction is becoming less useful



Choose whether each of the following statements corresponds more to RISC or CISC

- Code size is larger
- Decoder is more complicated
  - Pipelining is harder
- Single instruction takes more work (control, energy)
- Fewer general-purpose registers are available



What would be necessary to implement x86  
add instruction variations in the  
microarchitecture?

# Micro-ops

Translation of complex machine instruction (macro-op) to multiple steps

Microarchitecture dependent (Intel doesn't provide documentation on this)

For example, `addq 8(%rdi) %rax` might be translated into:

add 8 to rdi

load that address from memory

add that value to rax

store the result in rax

We will come back to this for out-of-order

**CISC becomes RISC-ier:** control unit has an easier job with each uop; in turn makes pipelining easier

In practice, compilers such as gcc *also* prioritize the RISC-er instructions (Blem et al paper)



Why does x86 have so many instructions?



Why might a variable-length instruction  
encoding be useful?

# RISC-V C extension

**RISC becomes CISC-ier!**

Sometimes used for embedded applications

RVC uses a simple compression scheme that offers shorter 16-bit versions of common 32-bit RISC-V instructions when:

- the immediate or address offset is small, or
- one of the registers is the zero register (`x0`), the ABI link register (`x1`), or the ABI stack pointer (`x2`), or
- the destination register and the first source register are identical, or
- the registers used are the 8 most popular ones.

The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary. With the addition of the C extension, JAL and JALR instructions will no longer raise an instruction misaligned exception.



Why were new ISAs after about 1982 mostly  
RISC?



# Arm vs. Intel: market incentives

Arm claims 99% of premium smartphones use their chips

Prioritizes low energy over performance

Licenses chips + gets royalties instead of manufacturing chips

Apple's switch from x86 to Arm driven by need for flexibility

Already were making Arm-based chips for iPhones and iPads

Don't need to rely on manufacturing issues (deadlines, quality control) of other company

Intel used to power through based on market dominance

- IBM's adoption of Intel in the 80s fueled rise

- Intel in turn was able to spend more \$\$ on R&D