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

Different approaches to branching

How does new PC get determined?

- RISCV jumps (not branches): direct (jal) and indirect (jalr)

How does comparison get computed?

- RISCV: comparison and branch in one instruction
- x86, arm: compare, then branch (`cmp eax 0`, `jne end`)
- **Requires status/flag/condition code register**

Related: predicated instructions (we'll come back to this for DLP)

image source

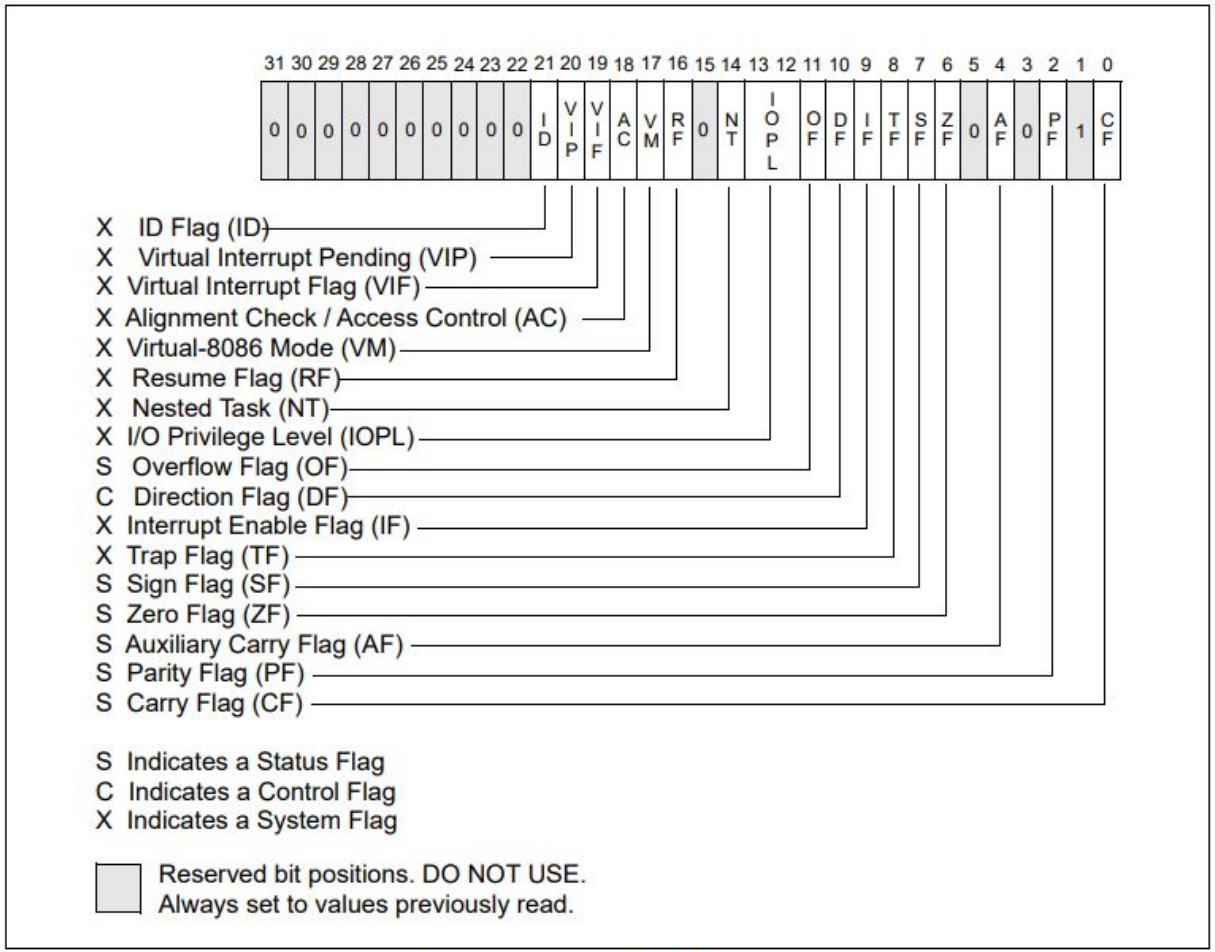


Figure 3-8. EFLAGS Register

(from Intel manual linked on first slide)

Different approaches to memory

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

RISCV

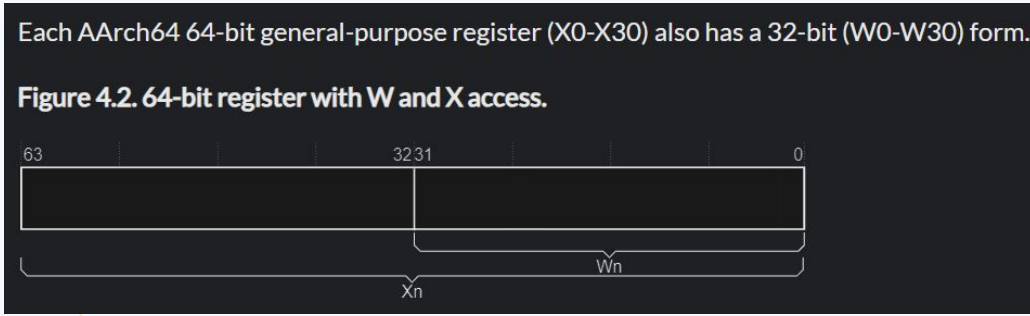
ADD, ADDW (32-bit add), ADDI, ADDIW

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



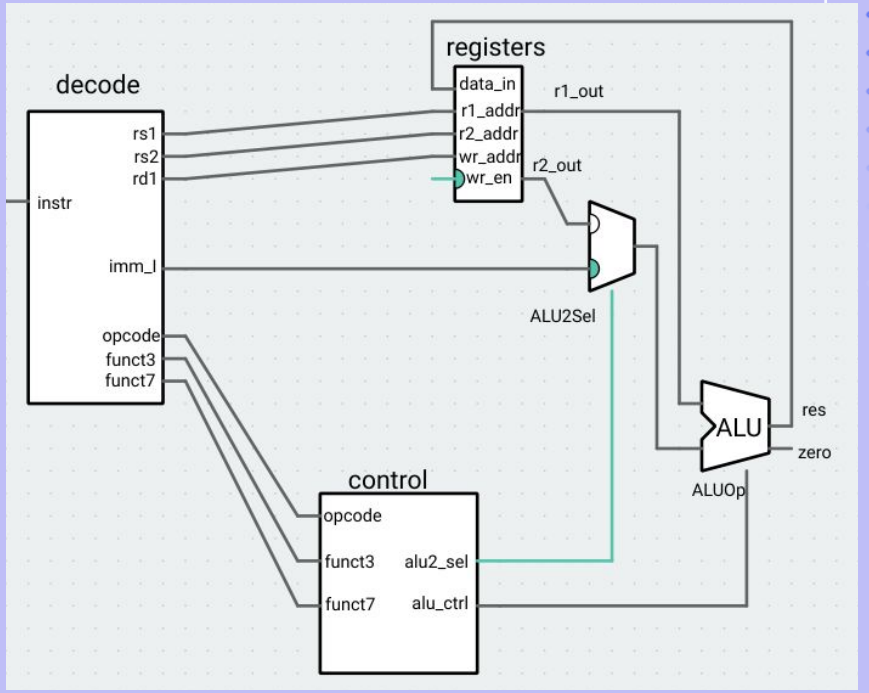
x86-64 ADD

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 ib	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.

? ? ?

What would we have to add to our single-stage processor to implement the x86 add variations?



Classification of ISAs: RISC/CISC

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

x86 is **CISC** (Complex Instruction Set Computer) architecture

- Typically a larger set of instructions
- Allows register-memory instrs
- Allows variable-length instruction encodings
- Allows instructions that take longer than 1 cycle

Except... this distinction is becoming less useful

- Blem et al. paper summed up: it's up to the microarchitecture

What’s typically the case?

- Larger code size
- Pipelining is harder
- Single instruction takes more work
- Fewer available GPRs
- More complicated decoder

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?

Extending ISAs

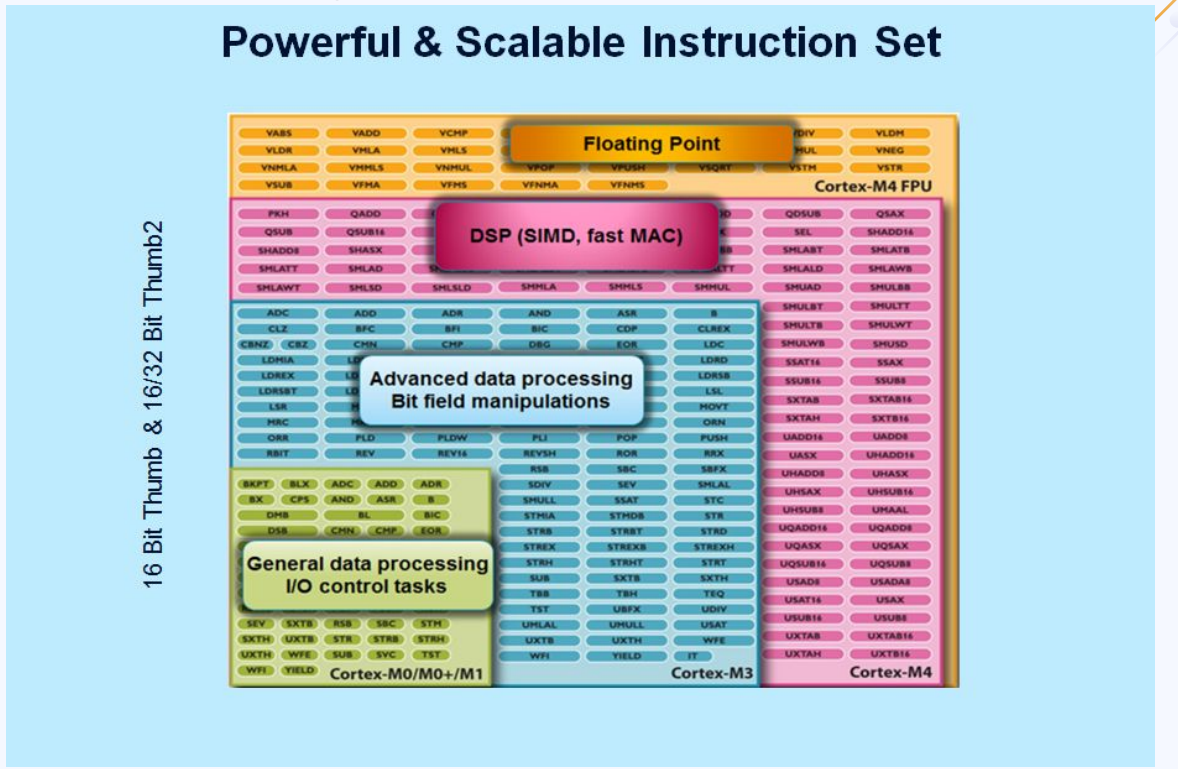
RISC becomes CISC-ier!

Chapter 21

Extending RISC-V

Base	Version
RV32I	2.0
RV32E	1.9
RV64I	2.0
RV128I	1.7
Extension	**Version**
M	2.0
A	2.0
F	2.0
D	2.0
Q	2.0
L	0.0
C	2.0
B	0.0
J	0.0
T	0.0
P	0.1
V	0.2
N	1.1

image source



Why might a variable-length instruction encoding be useful?

RISC becomes CISC-ier!

RISC-V C extension

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

Concluding remarks from Blem et al

While our study shows that RISC and CISC ISA traits are irrelevant to power and performance characteristics of modern cores, ISAs continue to evolve to better support exposing workload-specific semantic information to the execution substrate. *On x86, such changes include the transition to Intel64 (larger word sizes, optimized calling conventions and shared code support), wider vector extensions like AVX, integer crypto and security extensions (NX), hardware virtualization extensions and, more recently, architectural support for transactions in the form of HLE. Similarly, the ARM ISA has introduced shorter fixed length instructions for low power targets (Thumb), vector extensions (NEON), DSP and bytecode execution extensions (Jazelle DBX), Trustzone security, and hardware virtualization support. Thus, while ISA evolution has been continuous, it has focused on enabling specialization and has been largely agnostic of RISC or CISC. Other examples from recent research include extensions to allow the hardware to balance accuracy and reliability with energy efficiency [15, 13] and extensions to use specialized hardware for energy efficiency [18].*