Instruction Set Principles and Architectures

COE 501

Computer Architecture

Prof. Muhamed Mudawar

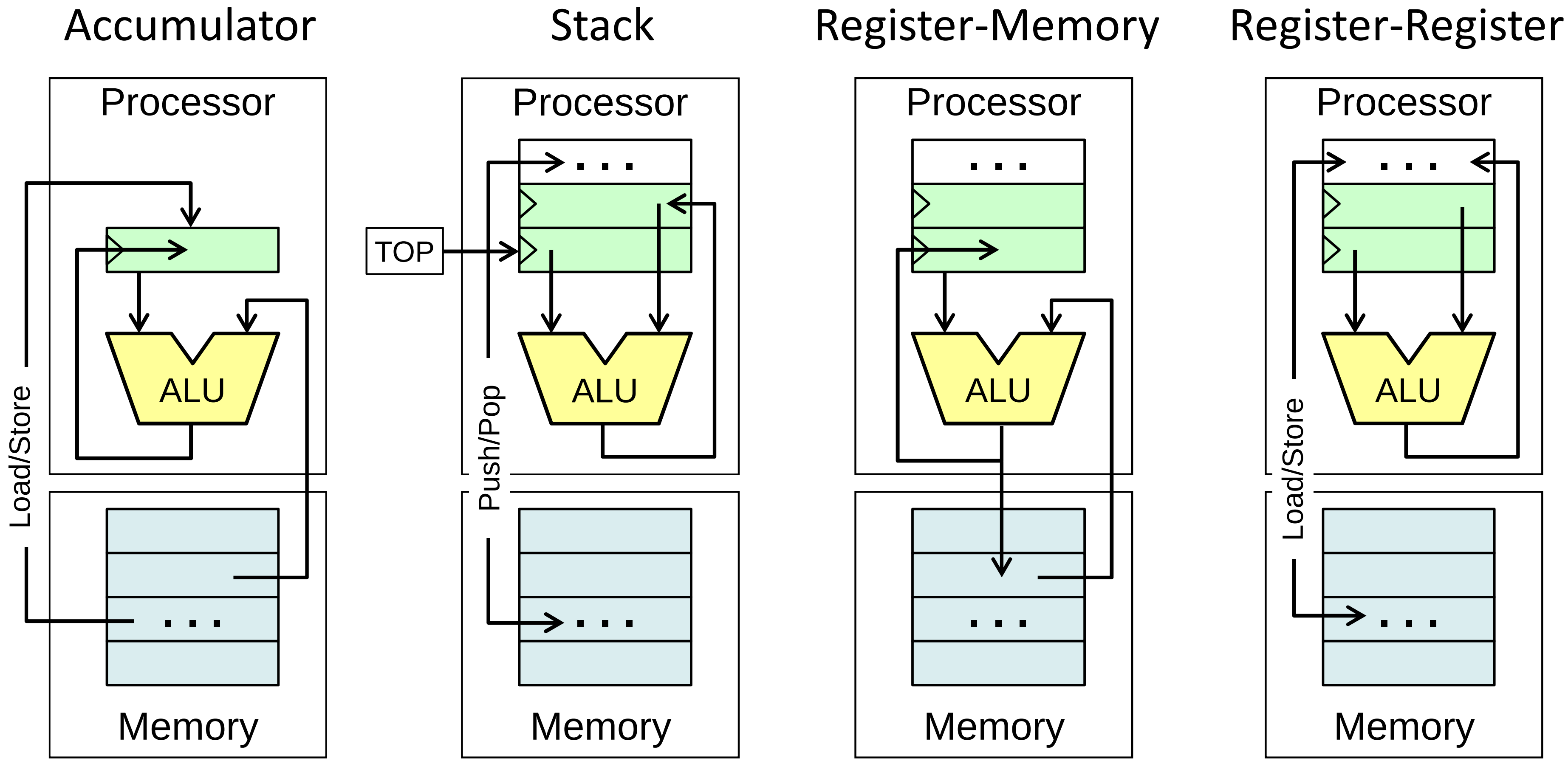
Computer Engineering Department

King Fahd University of Petroleum and Minerals

Instruction Set Architecture

- Critical interface between software and hardware
- Set of instructions, each is directly executed in hardware
- Programmer's visible instruction set
- Programmer's visible state (registers and memory)
- Lasts through generations (backward compatibility)
- Used in desktops, servers, and embedded applications
- Provides convenient functionality to higher level software
- Permits an efficient implementation at lower levels

Evolution of Instruction Sets



	Load [A]	Push [A]	Load R1, [A]	Load R1, [A]
C=A+B	Add [B]	Push [B]	Add R1, [B]	Load R2, [B]
	Store [C]	Add	Store R1, [C]	Add R3, R1, R2
		Pop [C]		Store R3, [C]

Classifying Instruction Sets

- Early Instruction Set Architectures
  - Accumulator-based or Stack-based
  - Replaced with General-Purpose Register (GPR) architectures
- Three classes or general-purpose register architectures

1. Register-Register (or Load-Store) Architecture (RISC)
   - Can access memory only via load and store instructions
2. Register-Memory Architecture (CISC)
   - Can access a memory location as part of any instruction
3. Memory-Memory Architecture (not used today)
   - Can access two or three memory locations per instruction
   - Large variation in instruction size and work per instruction (CPI)

Variety of Instruction Formats

- **Zero-address format**: Stack machines
  - ADD   Stack[SP-1] ← Stack[SP] + Stack[SP-1]
  - Usually top of stack is kept in high-speed registers
- **One-address format**: Accumulator machines
  - ADD [X]   AC ← AC + Memory[X]
- **Two-address format**: destination = first source operand
  - ADD R1, R2   Reg[R1] ← Reg[R1] + Reg[R2]
  - ADD R1, [X]   Reg[R1] ← Reg[R1] + Memory[X]
  - ADD [X], [Y]   Memory[X] ← Memory[X] + Memory[Y]
- **Three-address format**: most RISC architectures
  - ADD R1, R2, R3   Reg[R1] ← Reg[R2] + Reg[R3]

Memory Addressing

- ❖ Most architectures define memory as byte addressable
- ❖ A memory address can provide access to …
  - ✧ A byte (8 bits), 2 bytes, 4 bytes, 8 bytes, or more bytes
- ❖ The word size is defined differently by architectures
  - ✧ The word size = 2 bytes (Intel x86), 4 bytes (MIPS), or larger
- ❖ Two conventions for ordering bytes within a larger object

1. Little Endian byte ordering

x+3	x+2	x+1	x	
Byte 3	Byte 2	Byte 1	Byte 0	32-bit Register

   - ✧ Memory address X = address of least-significant byte (Intel x86)

2. Big Endian byte ordering

x	x+1	x+2	x+3	
Byte 0	Byte 1	Byte 2	Byte 3	32-bit Register

   - ✧ Memory address X = address of most-significant byte (SPARC)

Memory Alignment

- Address A must be multiple of data size: A mod size = 0
  - Why? because misalignment complicates hardware implementation

<table>
<tr><th colspan="16">Address mod 16 = Lower 4 bits of address in hexadecimal</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td></tr>
<tr><td colspan="2">Aligned-2</td><td colspan="2">Aligned-2</td><td colspan="2">Aligned-2</td><td colspan="2">Aligned-2</td><td colspan="2">Aligned-2</td><td colspan="2">Aligned-2</td><td colspan="2">Aligned-2</td><td colspan="2">Aligned-2</td></tr>
<tr><td colspan="4">Aligned (4 bytes)</td><td colspan="4">Aligned (4 bytes)</td><td colspan="4">Aligned (4 bytes)</td><td colspan="4">Aligned (4 bytes)</td></tr>
<tr><td colspan="8">Aligned (8 bytes)</td><td colspan="8">Aligned (8 bytes)</td></tr>
<tr><td></td><td colspan="8">Misaligned (8 bytes)</td><td colspan="4">Misaligned (4 bytes)</td><td colspan="2">Misalign-2</td><td></td></tr>
<tr><td colspan="2"></td><td colspan="8">Misaligned (8 bytes)</td><td colspan="4">Misaligned (4 bytes)</td><td colspan="2">Aligned-2</td></tr>
<tr><td colspan="3"></td><td colspan="8">Misaligned (8 bytes)</td><td colspan="4">Misaligned (4 bytes)</td><td></td></tr>
<tr><td></td><td colspan="2">Misalign-2</td><td></td><td colspan="8">Misaligned (8 bytes)</td><td colspan="4">Aligned (4 bytes)</td></tr>
<tr><td></td><td colspan="4">Misaligned (4 bytes)</td><td colspan="8">Misaligned (8 bytes)</td><td colspan="3"></td></tr>
<tr><td colspan="2"></td><td colspan="4">Misaligned (4 bytes)</td><td colspan="8">Misaligned (8 bytes)</td><td colspan="2"></td></tr>
<tr><td colspan="3"></td><td colspan="4">Misaligned (4 bytes)</td><td colspan="8">Misaligned (8 bytes)</td><td></td></tr>
</table>

Addressing Modes (Commonly Used)

- How instructions specify the addresses of their operands
- Operands can be in registers, constants, or in memory

Mode	Example	Meaning	When used
Register	ADD R1, R2, R3	R1 ← R2 + R3	Values in registers
Immediate	ADD R1, R2, 100	R1 ← R2 + 100	For constants
Register Indirect	LD R1, [R2]	R1 ← Mem[R2]	R2 contains address
Displacement	LD R1, [R2, 8]	R1 ← Mem[R2 + 8]	Address local variables
Absolute	LD R1, [1000]	R1 ← Mem[1000]	Address static data
Indexed	LD R1, [R2, R3]	R1 ← Mem[R2 + R3]	R2=base, R3=index
Scaled Index	LD R1, [R2, R3, s]	R1 ← Mem[R2 + R3 << s]	s = scale factor

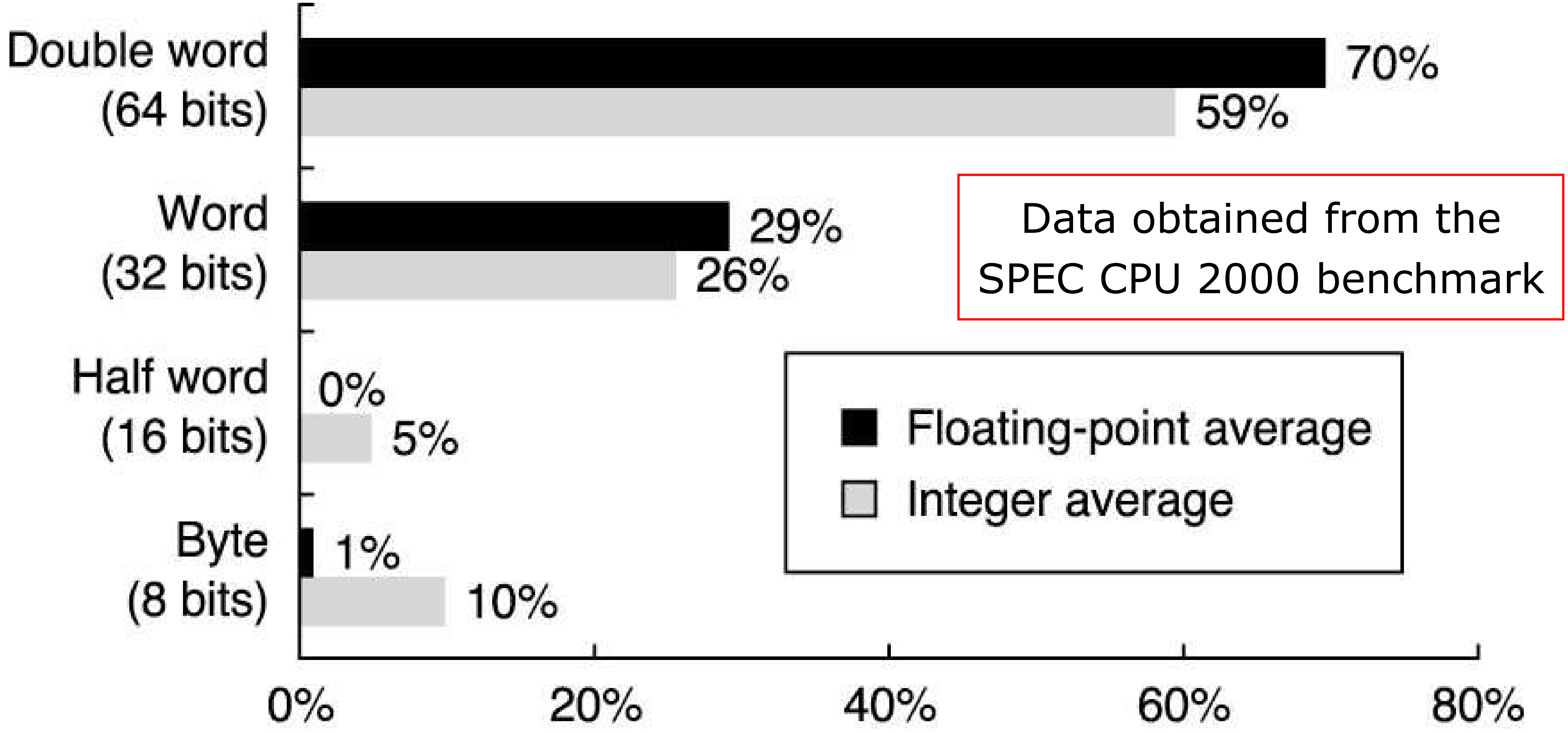
Pre-update	LD R1, [R2, 8] !	R2 ← R2 + 8
		R1 ← Mem[R2]
			Address is pre-updated
			Using pointer to traverse array

Post-update	LD R1, [R2], 8	R1 ← Mem[R2]
		R2 ← R2 + 8
			Address is post-updated
			Using pointer to traverse array

Types and Size of Operands

- Common operand types:
  - ASCII character = 1 byte (64-bit register can store 8 characters)
  - Unicode character or Short integer = 2 bytes = 16 bits (half word)
  - Integer = 4 bytes = 32 bits (word size on many RISC processors)
  - Single-precision float = 4 bytes = 32 bits (word size)
  - Long integer = 8 bytes = 64 bits (double word)
  - Double-precision float = 8 bytes = 64 bits (double word)
  - Extended-precision float = 10 bytes = 80 bits (Intel architecture)
  - Quad-precision float = 16 bytes = 128 bits (quad word)
- 32-bit versus 64-bit architectures
  - 64-bit architectures support 64-bit operands & memory addresses
  - Older architectures were 32-bit (can address 4 GB of memory)

Data Accesses by Size



Data obtained from the SPEC CPU 2000 benchmark

The double-word data type is used for double-precision floating-point and for 64-bit memory addresses

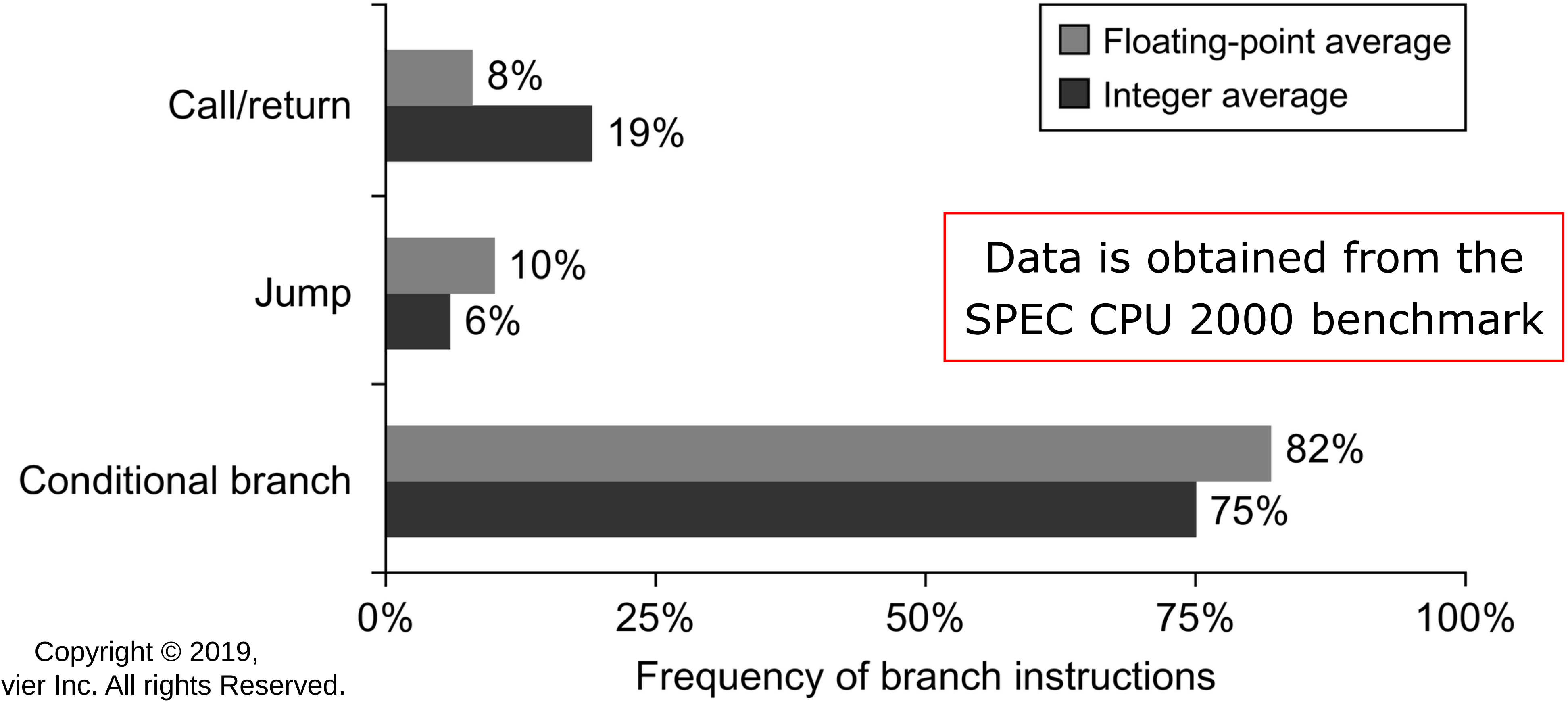
Operations in the Instruction Set

- Integer Arithmetic and Logical
  - Integer arithmetic: ADD, SUB, SHIFT, MUL, DIV, etc.
  - Logical operations: AND, OR, XOR, NOR, etc.
- Data Transfer and Data Conversions
  - Load, Store, Move data between registers
  - Convert data between different formats: integer, floating-point, …
- Control: branch, jump, procedure call, return, and traps
- System: Operating system calls and memory management
- Binary and Decimal Floating-point operations
- Graphics: pixel and vertex operations, compression, etc.
  - SIMD instructions operate in parallel on many data elements

Breakdown of Control Instructions

- ❖ Procedure Call and Return
- ❖ Unconditional Jump
- ❖ Conditional Branch

Conditional Branch clearly dominate



Data is obtained from the SPEC CPU 2000 benchmark

Addressing Modes for Control

How to specify the target address for control instructions?

- ❖ PC-relative addressing for branch instructions
  - ✧ PC-relative offset is added to the program counter (PC)
  - ✧ The target instruction is often near the branch instruction
  - ✧ Position independent code: can be loaded anywhere in memory
- ❖ As a register (or memory) containing the target address
  - ✧ For procedure return and indirect jumps
  - ✧ For case or switch statements
  - ✧ For methods in object-oriented languages
  - ✧ For high-order functions or function pointers
  - ✧ For dynamically shared libraries that are loaded/linked at runtime
- ❖ As a direct address in the instruction format

Conditional Branch Options

Option	Examples	How Tested	Advantages	Disadvantages
Condition Code (Z, N, C, V)	Intel x86, ARM, PowerPC	Tests special bits set by ALU and compare instructions	Set as a side effect of some ALU instructions	Extra state, constrains ordering of instructions
Condition Register	Alpha, MIPS	Comparison result put in a register	Simple	Extra compare instruction for general condition
Compare and Branch	MIPS, PA-RISC	Compare is part of the branch	One instruction rather than two for a branch	May be too much work for pipelined execution

- ❖ Different techniques are used for branches based on integer versus floating-point comparisons

Procedure Call Options

- At a minimum, the return address should be saved
  - In a special link register, in a GPR, or in memory on the stack
- Some architectures can save/restore many registers
  - The compiler should select which registers to save and restore
- Two basic conventions to preserve registers
  - By the caller before making a procedure CALL (Caller-Saved)
  - Inside the procedure before modifying registers (Callee-Saved)
- Software conventions to reduce register saving
  - Which registers should be preserved by the caller and which ones should be preserved inside the procedure

Encoding an Instruction Set

- Variable Encoding
  - Instruction length is a variable number of bytes
  - Allows all addressing modes to be used with all operations
  - Examples: Intel x86 and VAX
- Fixed Encoding
  - All instructions have a single fixed size, typically 32 bits
  - Combines the addressing mode with the opcode
  - Examples: MIPS, ARM, Power, SPARC, etc.
- Hybrid Encoding
  - Few instruction lengths ➔ reduces the variability in length
  - Compressed encoding of some frequently used instructions
  - Examples: micro MIPS and ARM Thumb

Instruction encoding impacts the code size and ease of decoding inside the processor

Things to Remember …

- Major reasons for GPR architectures
  - Registers are faster than memory and reduce memory traffic
  - General-Purpose Registers are easier for a compiler to use
  - Register-Register architectures are simpler than Register-Memory
- Programs with aligned memory references run faster
  - Misalignment requires multiple aligned memory references
- Addressing modes specify …
  - Registers, constants, and memory locations
  - Simple addressing modes are frequently used
  - 32 bits can address at most 4GB, 64 bits can address 16 Exabytes
- Most frequently used instructions are the simplest ones
- Instruction encoding impacts size and ease of decoding

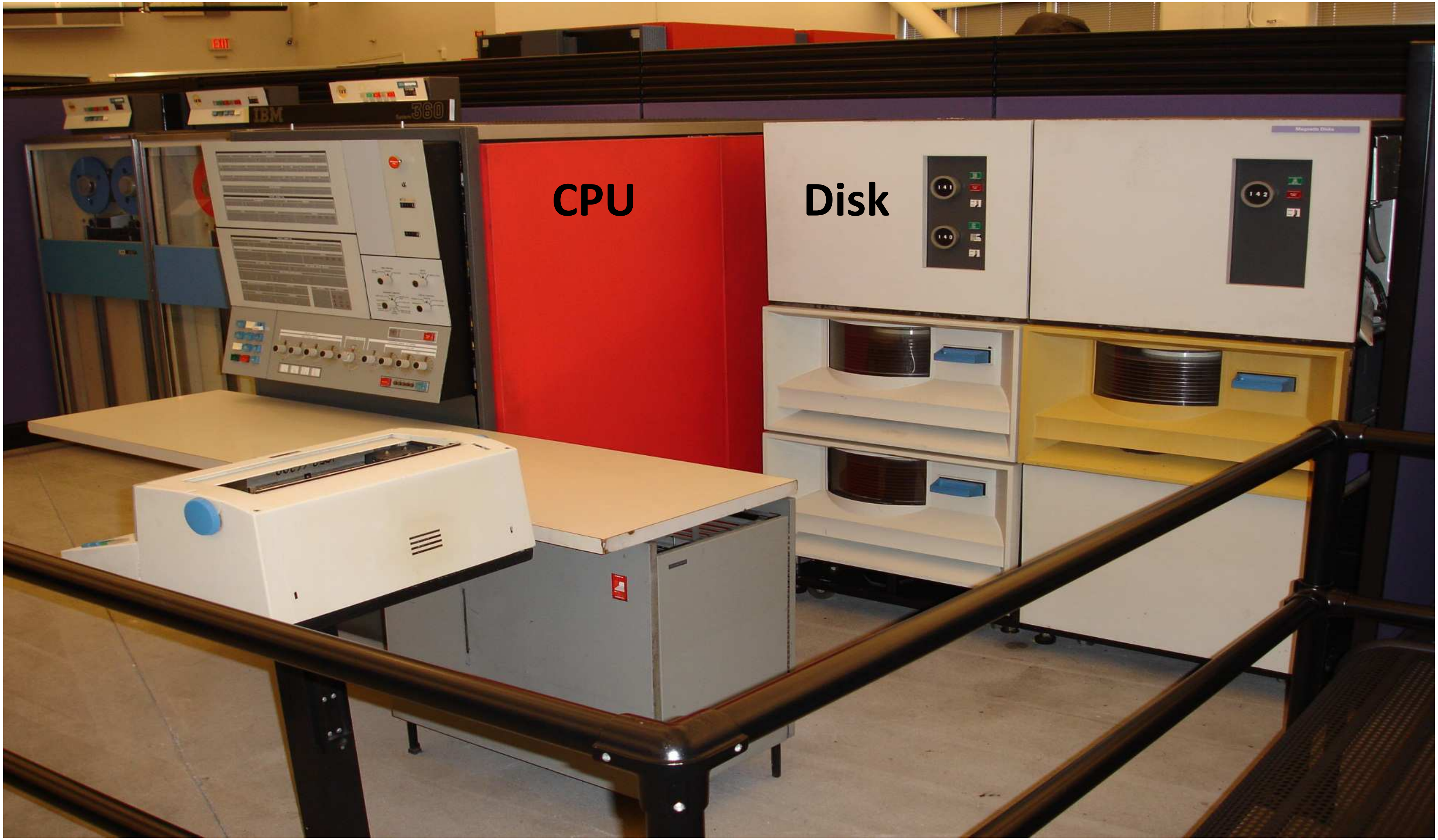
IBM 360 Architecture

- The term “Computer Architecture” was coined by IBM in 1964
- First true Instruction Set Architecture (ISA)
  - Portable software on different models, compiler, assembler, linker
  - Milestone: one of the most successful computers in history
- IBM 360 ISA hid the technological differences between models
  - Model 30 (64 KB, 0.03 MIPS), Model 67 (1 MB, virtual memory, 1 MIPS)
- Machine is capable of supervising itself
  - IBM Operating System/360
  - Dynamic Address Translation to support time-sharing
- General method for connecting I/O devices (simple to assemble)
- Built-in hardware fault checking to reduce down time

IBM 360 Architecture (cont'd)

- Processor State: 32-bit machine with 24-bit addresses
  - 16 General-Purpose 32-bit Registers
  - 4 Floating-Point 64-bit Registers
  - Instruction Address register, Condition codes
- Data Formats
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
  - This is why bytes are 8-bit long today
- 64-bit Floating-point precision
  - Model 91: Out-of-Order execution for scientific computing
- Instruction Types and Formats
  - Register-Register, Register-Memory, and Memory-Memory
  - 2-Byte RR format, 4-Byte RX format, and 6-Byte SS format

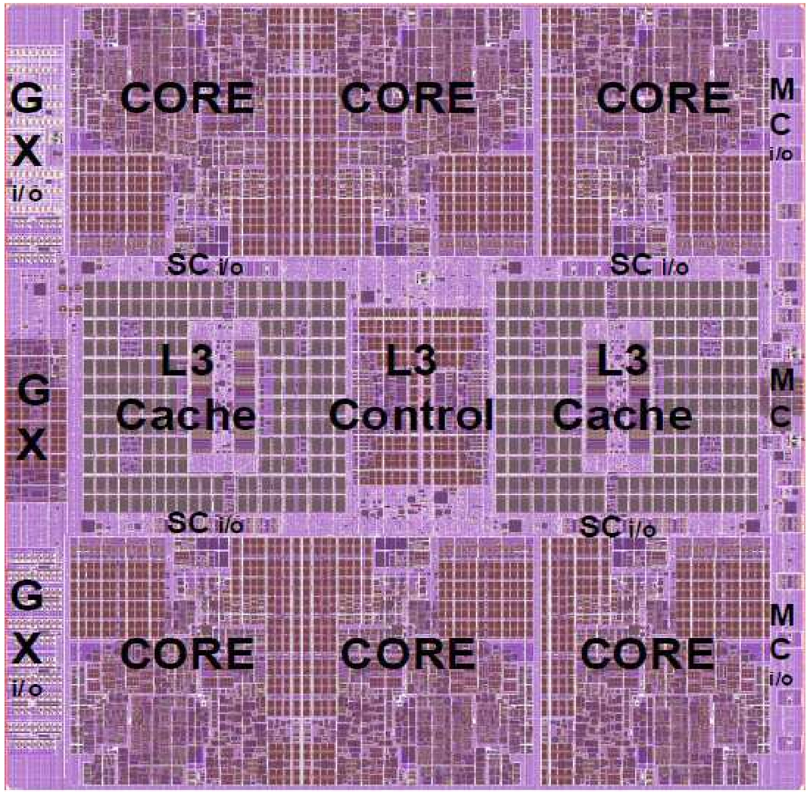
IBM 360 Model 30



IBM 360: 50 years later zSeries z12

The IBM zSeries

z12 Die [2012]



- ❖ Six-core design (large cores)
- ❖ 2.75 billion transistors (597 mm^2)
- ❖ 32 nm technology (13 layers)
- ❖ The z12 runs at 5.5 GHz to 6 GHz
- ❖ Power = 300 Watts (liquid cooling)
- ❖ I-Cache: 64KB L1 + 1MB L2 per core
- ❖ D-cache: 96KB L1 + 1MB L2 per core
- ❖ On-chip shared L3: 48MB eDRAM
- ❖ 64-bit virtual addressing
  - ✧ Original S/360 was 24-bit, S/370 was 32-bit
- ❖ Out-of-order superscalar pipeline
  - ✧ Six execution units per core
  - ✧ Optimized for single-thread performance

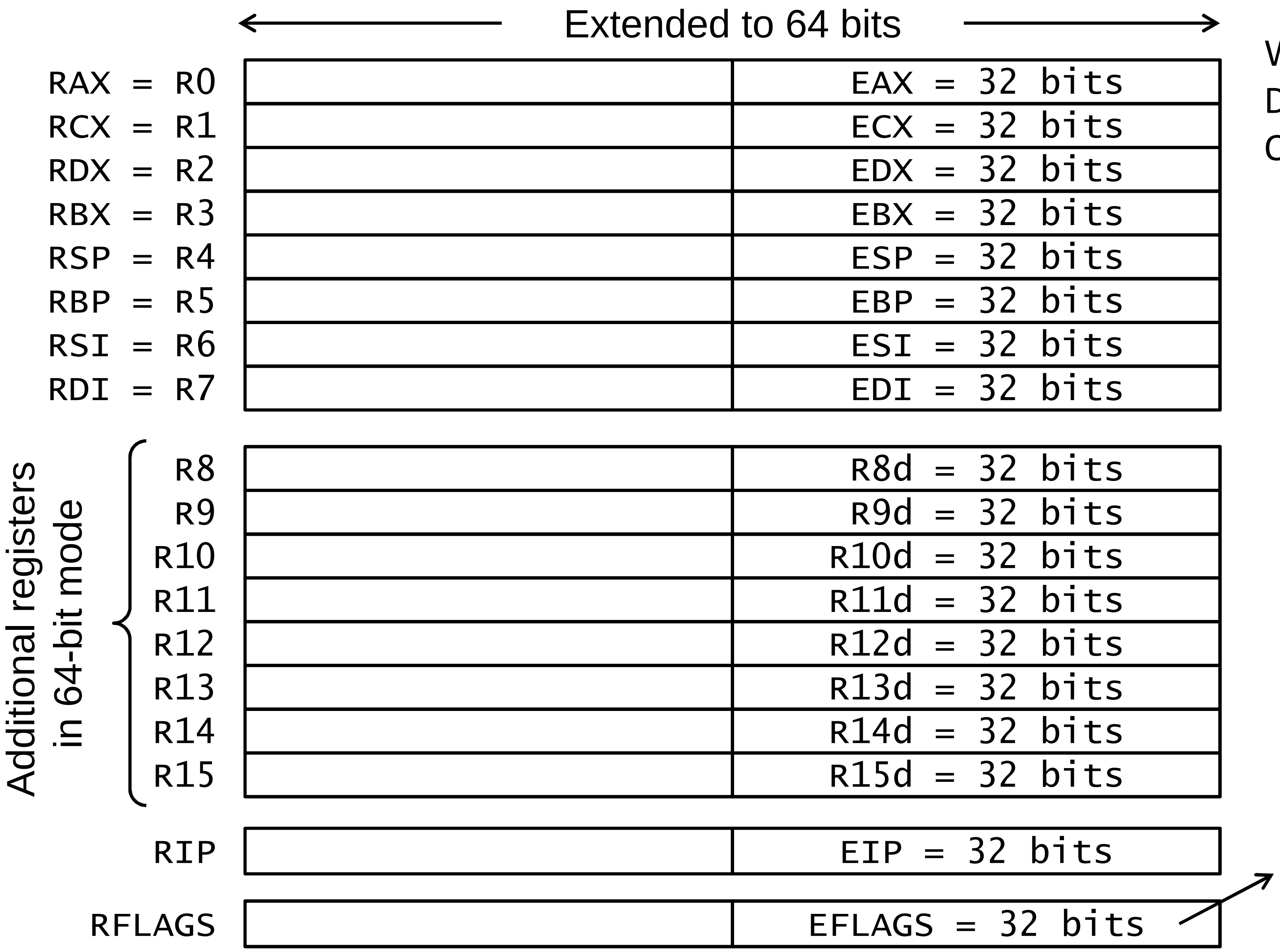
Intel x86 Architecture

- Difficult to understand and impossible to love!
- Developed by independent groups (over 30+ years)
  - 8086 (1978): 16-bit registers, 20-bit address, segmentation
  - 8087 (1980): FP coprocessor, FP instructions, FP register stack
  - 80286 (1982): 24-bit address space, protected mode
  - 80386 (1985): 32-bit architecture, Paging (4 KB pages), MMU
  - 80486 (1989): pipelined, on-chip caches and x87 FPU (80-bit)
  - Pentium (1993): two pipelines U&V, 64-bit databus, MMX
  - Pentium Pro (1995): µop translation, Out-of-order, L2 cache
  - Pentium III (1999): SSE instructions, 128-bit XMM registers
  - Pentium 4 (2001): deeply pipelined, SSE2, hyper-threading

Intel x86 Architecture (cont'd)

- Further developments …
  - AMD64 (2003): AMD extended Intel x86 architecture to 64 bits
  - Intel x86-64 (2004): Intel adopted AMD64, added SSE3
  - Intel Core (2006): 64-bit integer, low-power, multi-core, SSE4
  - Intel Core i3/i5/i7 (2008): L3 cache, QuickPath interconnect
  - Intel Atom (2008): In order execution, low-power, on-die GPU
  - AVX: Advanced Vector eXtension (2008): 256-bit YMM registers
  - AVX-512: expands AVX into 512-bit ZMM registers
  - Intel Xeon Phi (2012): Many Integrated Cores (MIC)
    - 62 Cores (Pentium), AVX-512, 4 threads/core, 1+ Teraflops
- Market Success ≠ Technical Elegance

Intel x86-64 Basic Registers



Extended to 64 bits

RAX = R0	EAX = 32 bits
RCX = R1	ECX = 32 bits
RDX = R2	EDX = 32 bits
RBX = R3	EBX = 32 bits
RSP = R4	ESP = 32 bits
RBP = R5	EBP = 32 bits
RSI = R6	ESI = 32 bits
RDI = R7	EDI = 32 bits

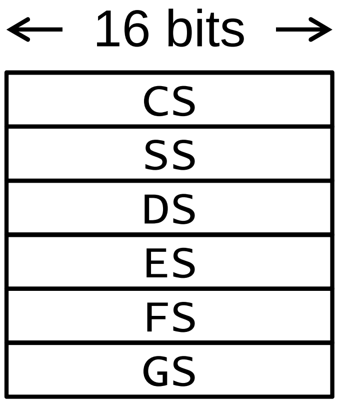
Additional registers in 64-bit mode

R8	R8d = 32 bits
R9	R9d = 32 bits
R10	R10d = 32 bits
R11	R11d = 32 bits
R12	R12d = 32 bits
R13	R13d = 32 bits
R14	R14d = 32 bits
R15	R15d = 32 bits

RIP	EIP = 32 bits
RFLAGS	EFLAGS = 32 bits

Word	=	16 bits
Double	=	32 bits
Quad	=	64 bits

Segment Registers



16 bits

CS
SS
DS
ES
FS
GS

CF = Carry Flag

OF = Overflow Flag

ZF = Zero Flag

SF = Sign Flag

MOV Instruction

- MOV has different meanings according to source and destination
- Three types of source operands:
  - Immediate: constant encoded in the instruction
  - Source Register: register number is encoded in the instruction
  - Memory: address is computed according to memory addressing mode
- Two types of destination operands: Register or Memory
  - However, Memory to Memory transfer is not allowed

Instruction	Meaning	Comment
`MOV Rd, Rs`	`Rd = Rs`	`Register copy`
`MOV Rd, Imm`	`Rd = Imm`	`Initialize Rd with Immediate`
`MOV Rd, [mem]`	`Rd = [mem]`	`Load register Rd from memory`
`MOV [mem], Rs`	`[mem] = Rs`	`Store register Rs in Memory`
`MOV [mem], Imm`	`[mem] = Imm`	`Store immediate in Memory`

Data Movement Instructions

Instruction	Meaning	Comment
`MOVZX Rd, src`	`Rd = zero_extend(src)`	Move with zero extend
`MOVSX Rd, src`	`Rd = sign_extend(src)`	Move with sign extend
`PUSH  src`	`RSP -= 8 ; [RSP] = src`	Push src value on stack
`POP   dest`	`dest = [RSP] ; RSP += 8`	Pop top of stack
`XCHG  dest, src`	`{dest,src} = {src,dest}`	Exchange src with dest
`LEA   Rd, [mem]`	`Rd = address_of(mem)`	Load effective address

- MOVZX and MOVSX: src can be a register or memory location
  - Value is copied into destination register Rd with zero or sign extension
- PUSH and POP use the Stack Pointer register RSP
- XCHG: exchange two registers or a register with memory
- Not all instructions are listed, only the commonly used ones

Arithmetic Instructions

Instruction	Meaning	Comment
ADD Rd, Rs	Rd = Rd + Rs	Register to Register
ADD Rd, Imm	Rd = Rd + Imm	Register Immediate
ADD Rd, [mem]	Rd = Rd + [mem]	Source Memory
ADD [mem], Rs	[mem] = [mem] + Rs	Destination Memory
ADD [mem], Imm	[mem] = [mem] + Imm	Destination Memory
SUB dest, src	dest = dest – src	Multiple opcodes
ADC dest, src	dest = dest + src + CF	Add with Carry Flag
SBB dest, src	dest = dest – src – CF	Subtract with Borrow
NEG dest	dest = -dest	Negate (2's complement)
INC dest	dest = dest + 1	Faster than: ADD dest, 1
DEC dest	dest = dest – 1	Faster than: SUB dest, 1

- Arithmetic Instruction update flags in RFLAGS

  CF = Carry Flag, OF = Overflow Flag, SF = Sign Flag, ZF = Zero Flag

Logic and Shift Instructions

Instruction	Meaning	Comment
`AND dest, src`	`dest = dest & src`	Bitwise AND
`OR  dest, src`	`dest = dest | src`	Bitwise OR
`XOR dest, src`	`dest = dest ^ src`	Bitwise XOR
`NOT dest`	`dest = ~dest`	Bitwise NOT
`SHL dest, src`	dest = dest $<<_0$ src	Shift Left (insert zeros)
`SHR dest, src`	dest = dest $_0>>$ src	Shift Right (insert zeros)
`SAR dest, src`	dest = dest $_s>>$ src	Shift Arithmetic Right
`ROR dest, src`		Rotate Right
`ROL dest, src`		Rotate Left

- ❖ Destination (dest) can be a register Rd or memory location
- ❖ Source (src) can be a register Rs, immediate, or memory location
- ❖ Destination and source cannot be both memory locations

Integer Multiply and Divide Instructions

Instruction	Meaning	Comment
MUL src	RDX:RAX = RAX * src	64 × 64 bits = 128 bits
IMUL src	RDX:RAX = RAX * src	Signed Multiplication
IMUL dest, src	dest = dest * src	Multiple opcodes

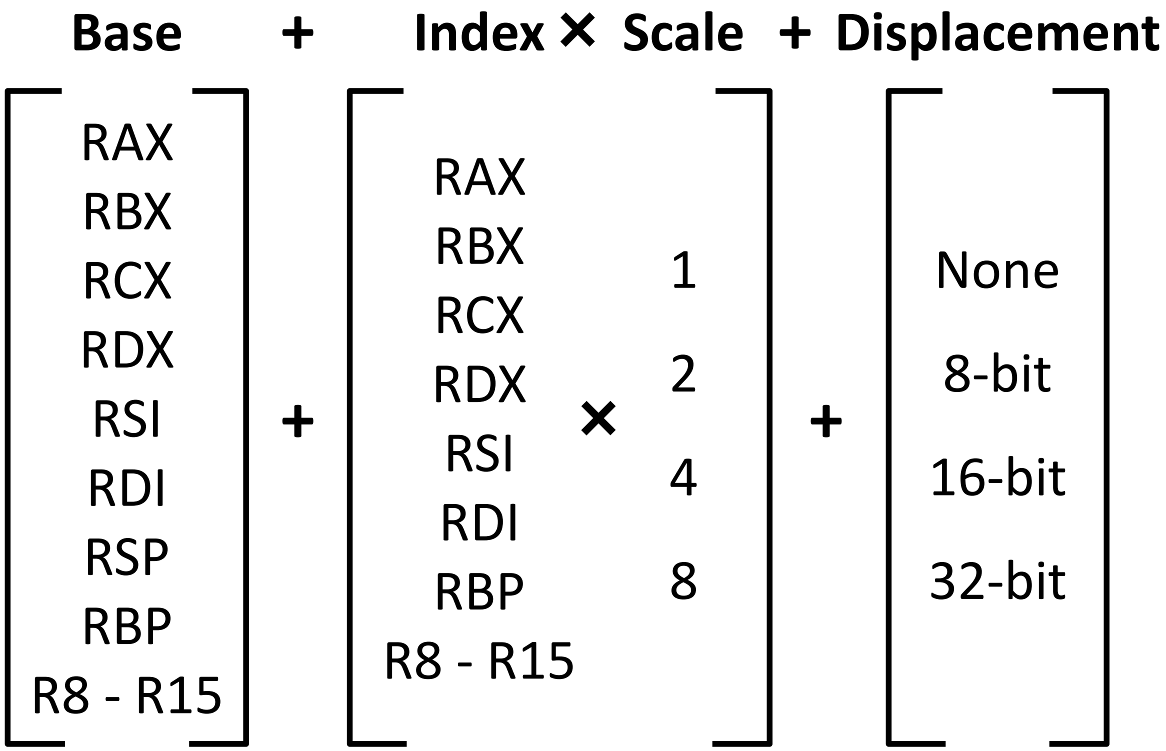
DIV src	RAX = RDX:RAX / src
	RDX = RDX:RAX % src
		Unsigned Division
		RDX = remainder

IDIV src	RAX = RDX:RAX / src
	RDX = RDX:RAX % src
		Signed Division
		RDX = remainder

- MUL does unsigned multiplication, IMUL does signed multiply
  - 128-bit result is written to RDX (upper 64 bits) and RAX (lower 64 bits)
- IMUL can have 2 operands: 64-bit result is written to destination register or memory. Upper 64-bit of product is discarded.

Intel x86 Memory Addressing Modes

- Base Register: any general purpose register (16 registers)
- Index Register: any general purpose register, except RSP
- Scale factor: 1, 2, 4, or 8 multiplied by the index value
- Displacement: optional 8-bit, 16-bit, or 32-bit constant value



Base + Index × Scale + Displacement

Base: RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8 - R15

+

Index: RAX, RBX, RCX, RDX, RSI, RDI, RBP, R8 - R15 × Scale: 1, 2, 4, 8

+

Displacement: None, 8-bit, 16-bit, 32-bit

Examples:

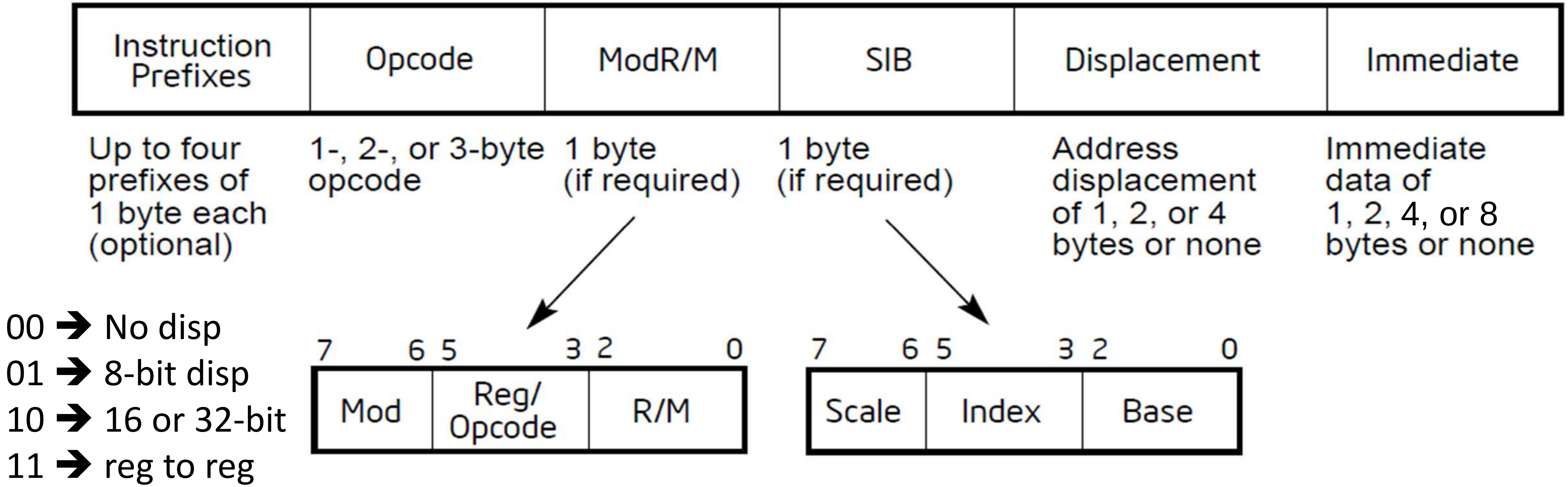
```
mov eax, [rbx]
mov [rbx + 16], rdx
add r10, [r11 + rsi]
and r12, [rdi*4 + 100]
sub [r8 + r9*8 – 100], rax
```

Flow Control Instructions

Instruction	Meaning	Comment
`JMP   target`	`RIP = target`	`Unconditional Jump`
`JMP   Rs/[mem]`	`RIP = Rs/[mem]`	`Indirect Jump`
`CMP   src1,src2`	`Compute (src1 – src2)`	`Only flags are modified`
`Jcond target`	`if (cond) RIP = target`	`Conditional Jump`
`CALL target`	`Push(RIP); RIP=target`	`Push Return Addr on stack`
`CALL Rs/[mem]`	`Push(RIP); RIP=Rs/[mem]`	`Indirect Call`
`RET`	`RIP = pop()`	`Pop & Jump to return addr`
`RET   Imm`	`RIP = pop(); RSP+=Imm`	`Return & pop Imm bytes`

- Conditional Jump Instructions:
  - JZ/JE (ZF=1), JNZ/JNE (ZF=0), JC (CF=1), JNC (CF=0), JO, JNO, JS, JNS
  - Signed: JL (SF ≠ OF), JGE (SF = OF), JLE (SF ≠ OF or ZF = 1), JG
  - Unsigned: JB (CF = 1), JAE (CF = 0), JBE (CF = 1 or ZF = 1), JA

Intel x86-64 Instruction Format



Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
Up to four prefixes of 1 byte each (optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes or none	Immediate data of 1, 2, 4, or 8 bytes or none

ModR/M:

7 6	5 3	2 0
Mod	Reg/ Opcode	R/M

SIB:

7 6	5 3	2 0
Scale	Index	Base

00 ➔ No disp
01 ➔ 8-bit disp
10 ➔ 16 or 32-bit
11 ➔ reg to reg

- Variable instruction length and complex encoding ☹
- REX (Reg Extension) prefix to address R8 to R15 in 64-bit mode
- Addressing modes (ModR/M and SIB bytes)
  - Base or scaled index with 8, 16, or 32-bit displacement
- Immediate operand (if needed), can be 8 bytes in 64-bit mode

Complex Encoding of x86 Instructions

a. JE EIP + displacement

4	4	8
JE	Condition	Displacement

Some Instructions can be very long (up to 17 bytes)

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

Top 10 Integer Instructions for Intel x86

1. Load: 22% (read from memory)
2. Conditional branch: 20%
3. Compare: 16%
4. Store: 12% (write to memory)
5. Add: 8%
6. And: 6%
7. Sub: 5%
8. Move register-register: 4%
9. Call: 1% (function call)
10. Return: 1% (function return)

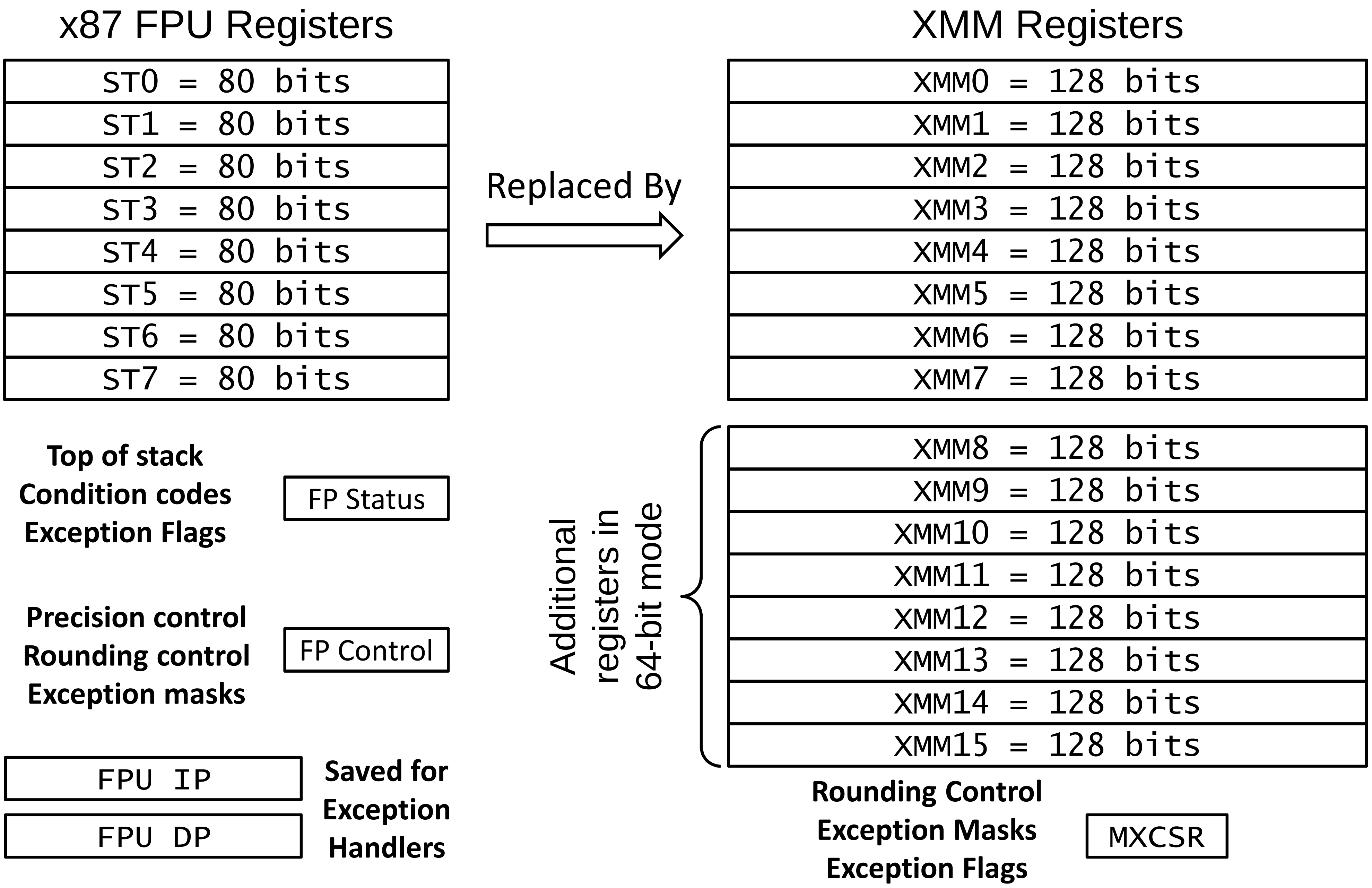
Total = 96% of instructions executed

Percentages are based on five SPEC INT 92 programs

The most widely executed instructions are the simplest operations of an instruction set

Top-10 instructions account for 96% of instructions executed

Intel x86-64 FPU & XMM Registers



x87 FPU Registers

ST0 = 80 bits

ST1 = 80 bits

ST2 = 80 bits

ST3 = 80 bits

ST4 = 80 bits

ST5 = 80 bits

ST6 = 80 bits

ST7 = 80 bits

Top of stack
Condition codes
Exception Flags

FP Status

Precision control
Rounding control
Exception masks

FP Control

FPU IP

FPU DP

Saved for
Exception
Handlers

Replaced By

XMM Registers

XMM0 = 128 bits

XMM1 = 128 bits

XMM2 = 128 bits

XMM3 = 128 bits

XMM4 = 128 bits

XMM5 = 128 bits

XMM6 = 128 bits

XMM7 = 128 bits

Additional registers in 64-bit mode

XMM8 = 128 bits

XMM9 = 128 bits

XMM10 = 128 bits

XMM11 = 128 bits

XMM12 = 128 bits

XMM13 = 128 bits

XMM14 = 128 bits

XMM15 = 128 bits

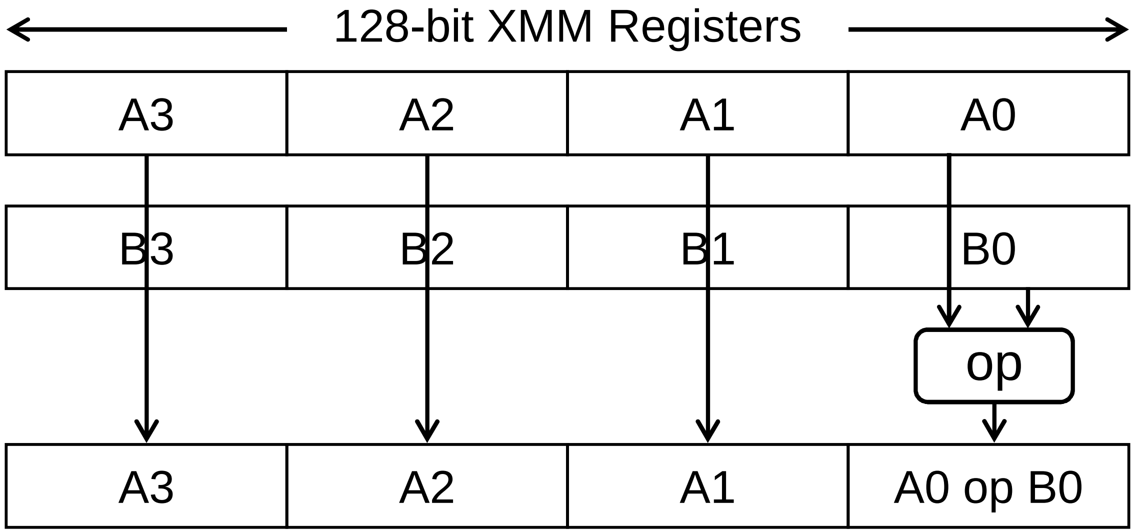
Rounding Control
Exception Masks
Exception Flags

MXCSR

SSE Instruction Set

- SSE = Streaming SIMD Extension
  - SIMD instructions operate in parallel on multiple data packed in a register
- SSE Instructions consist of the following:
  - Data movement instructions
  - Arithmetic Instructions
  - Logical Instructions
  - Comparison Instructions
  - Conversion Instructions
- The SSE instruction set introduced 70 new instructions
- SSE2 added 144 more instructions to SSE
- SSE3 added 13 more instructions
- SSE4 added 54 more instructions

SSE Scalar Instructions

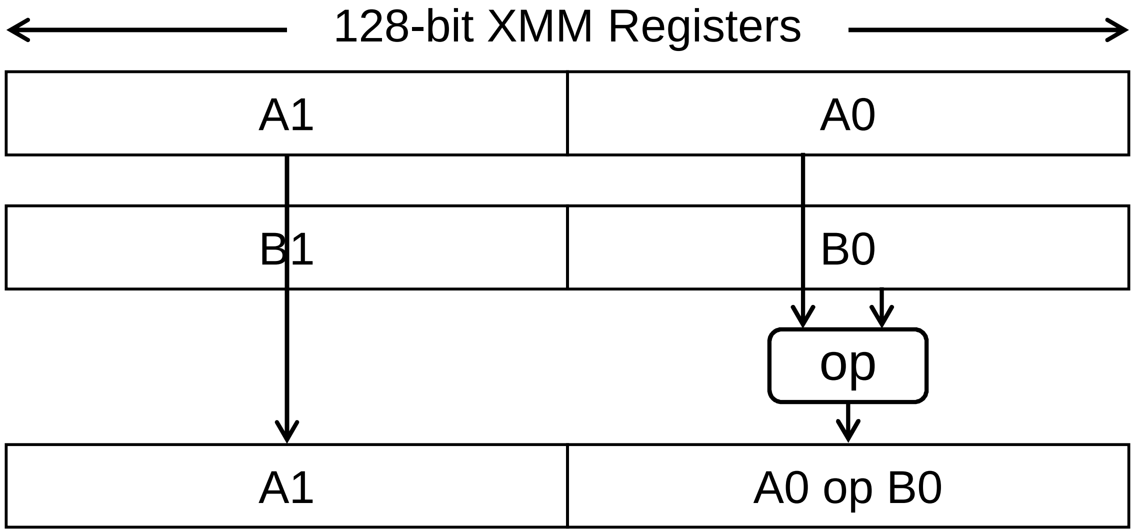


Scalar Single-Precision Floating-Point Instructions (SSE)

MOVSS, ADDSS, SUBSS, …

MULSS, DIVSS, SQRTSS, …

MAXSS, MINSS, CMPSS, …



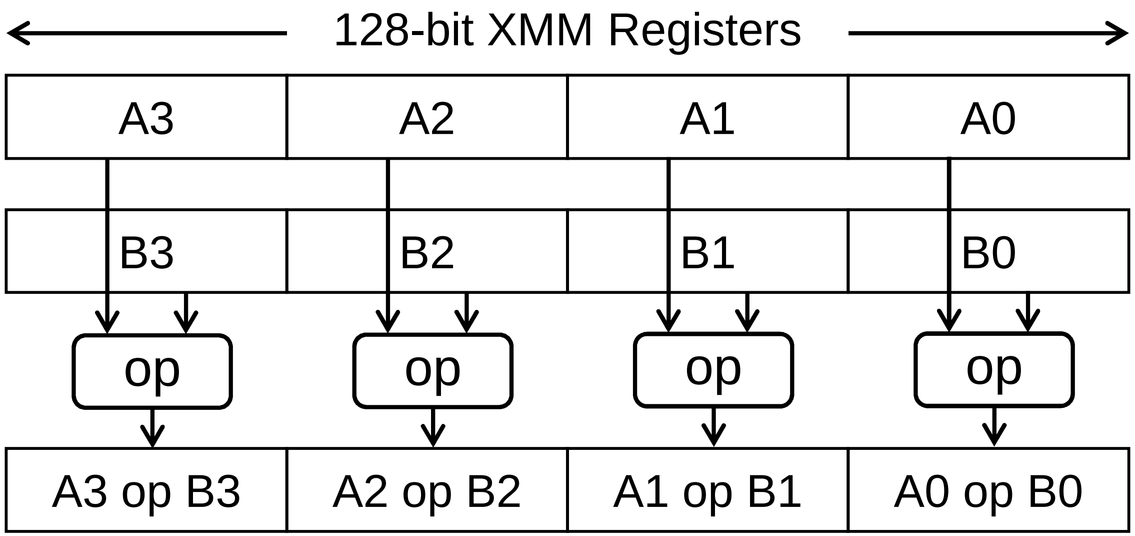
Scalar Double-Precision Floating-Point Instructions (SSE2)

MOVSD, ADDSD, SUBSD, …

MULSD, DIVSD, SQRTSD, …

MAXSD, MINSD, CMPSD, …

SSE Parallel (SIMD) Instructions

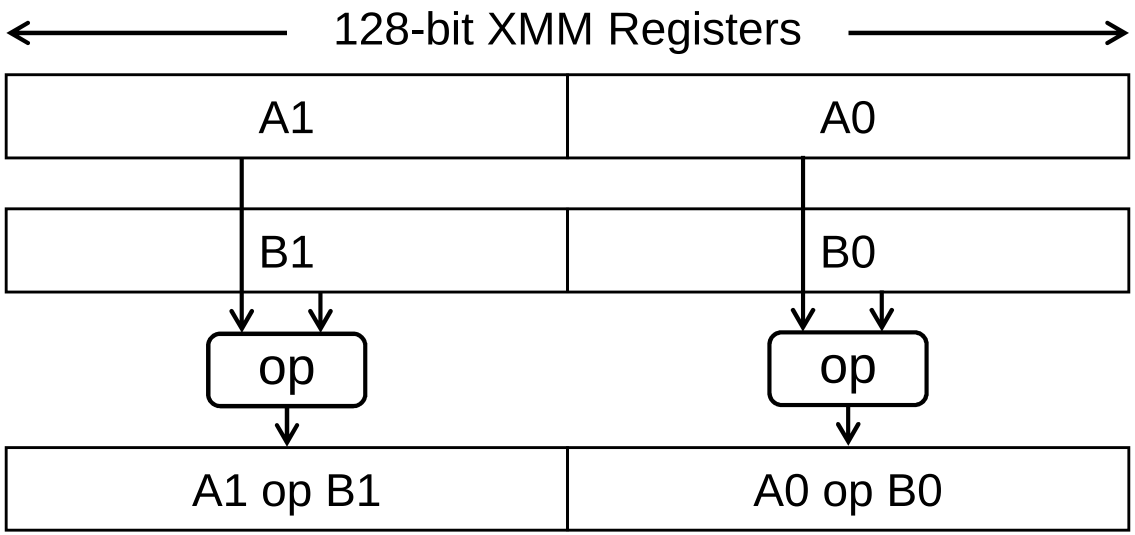


Packed Single-Precision Floating-Point Instructions (SSE)

MOVAPS, MOVUPS, …

ADDPS, SUBPS, MULPS, …

MAXPS, MINPS, CMPPS, …



Packed Double-Precision Floating-Point Instructions (SSE2)

MOVAPD, MOVUPD, …

ADDPD, SUBPD, MULPD, …

MAXPD, MAXPD, CMPPD, …

SSE/2 Data Movement Instructions

Instruction	Meaning
`MOVSS  dest, src`	`Move Scalar (S=32-bit float) from src to dest`
`MOVSD  dest, src`	`Move Scalar (D=64-bit float) from src to dest`
`MOVAPS dest, src`	`Move Aligned Packed floats (16 bytes)`
`MOVUPS dest, src`	`Move Unaligned Packed floats (16 bytes)`
`MOVAPD dest, src`	`Move Aligned Packed double-precision floats`
`MOVUPD dest, src`	`Move Unaligned Packed double-precision floats`
`MOVD   dest, src`	`Move Double-word (32 bits) between GPR and XMM`
`MOVQ   dest, src`	`Move Quad-word (64 bits) between GPR and XMM`

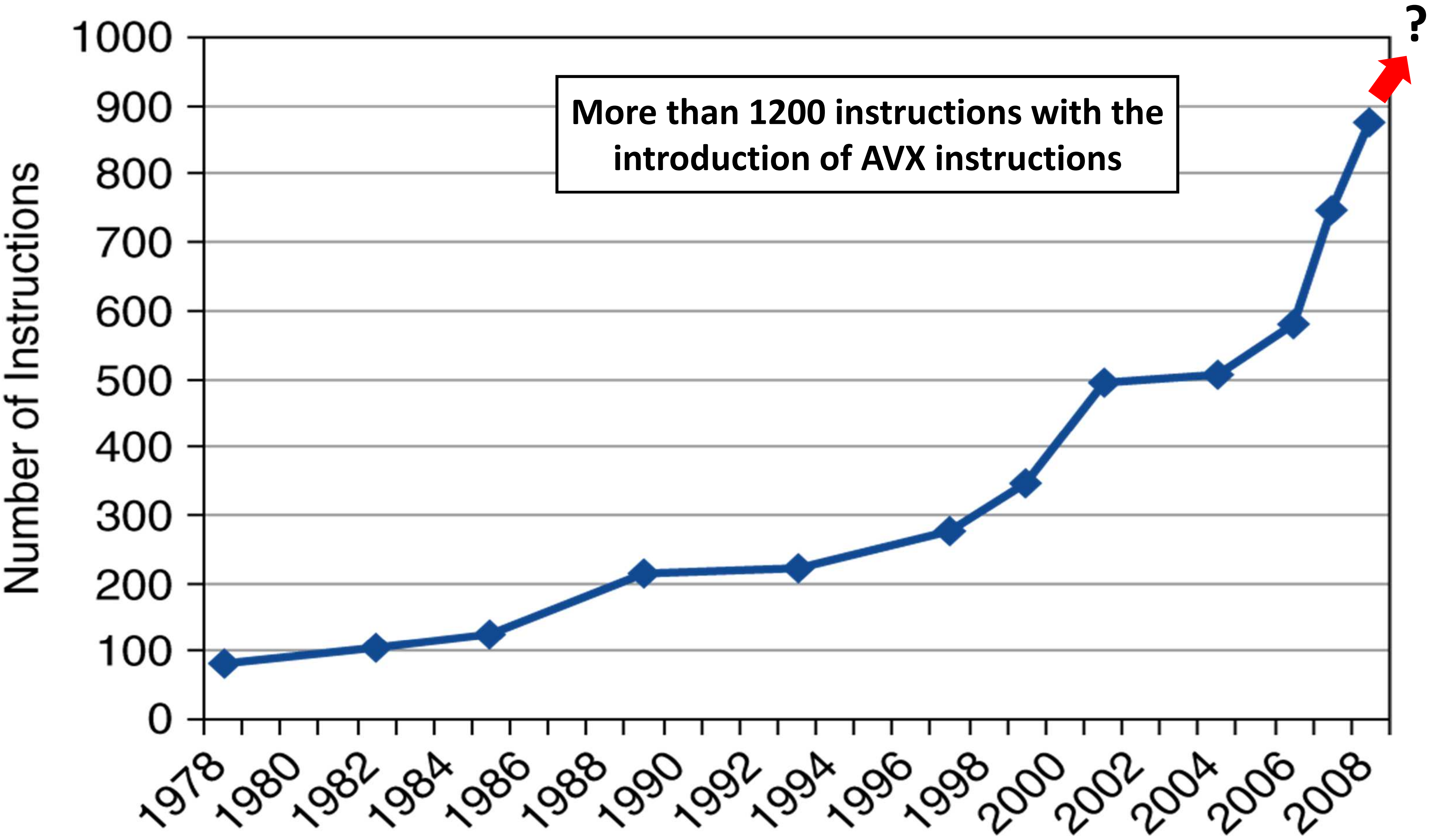
- dest: can be xmm register or [mem]
- src: can be xmm register or [mem]
- However, memory to memory operations are not allowed
- MOVD and MOVQ: either **`dest`** or **`src`** is an integer GPR

SSE/2 Floating-Point Instructions

Instruction	Meaning
`ADDSS dest, src`	`Add Scalar S=32-bit floats (low 32-bit)`
`ADDPS dest, src`	`Add Packed S=32-bit floats (4 elements)`
`ADDSD, ADDPD`	`Add Scalar/Packed D=64-bit floats (2 elements)`
`SUBSS, SUBPS`	`Subtract Scalar/Packed S=32-bit floats`
`SUBSD, SUBPD`	`Subtract Scalar/Packed D=64-bit floats`
`MULSS, MULPS`	`Multiply Scalar/Packed S=32-bit floats`
`MULSD, MULPD`	`Multiply Scalar/Packed D=64-bit floats`
`DIVSS, DIVPS`	`Divide   Scalar/Packed S=32-bit floats`
`DIVSD, DIVPD`	`Divide   Scalar/Packed D=64-bit floats`
`MAXSS, MAXPS`	`Maximum  Scalar/Packed S=32-bit floats`
`MAXSD, MAXPD`	`Maximum  Scalar/Packed D=64-bit floats`
`CMPSS, CMPPS`	`Compare  Scalar/Packed S=32-bit floats (8 cond)`
`CMPSD, CMPPD`	`Compare  Scalar/Packed D=64-bit floats (8 cond)`

- This is only a short list of some important SSE/SSE2 instructions

Intel x86 Instruction Set Expansion



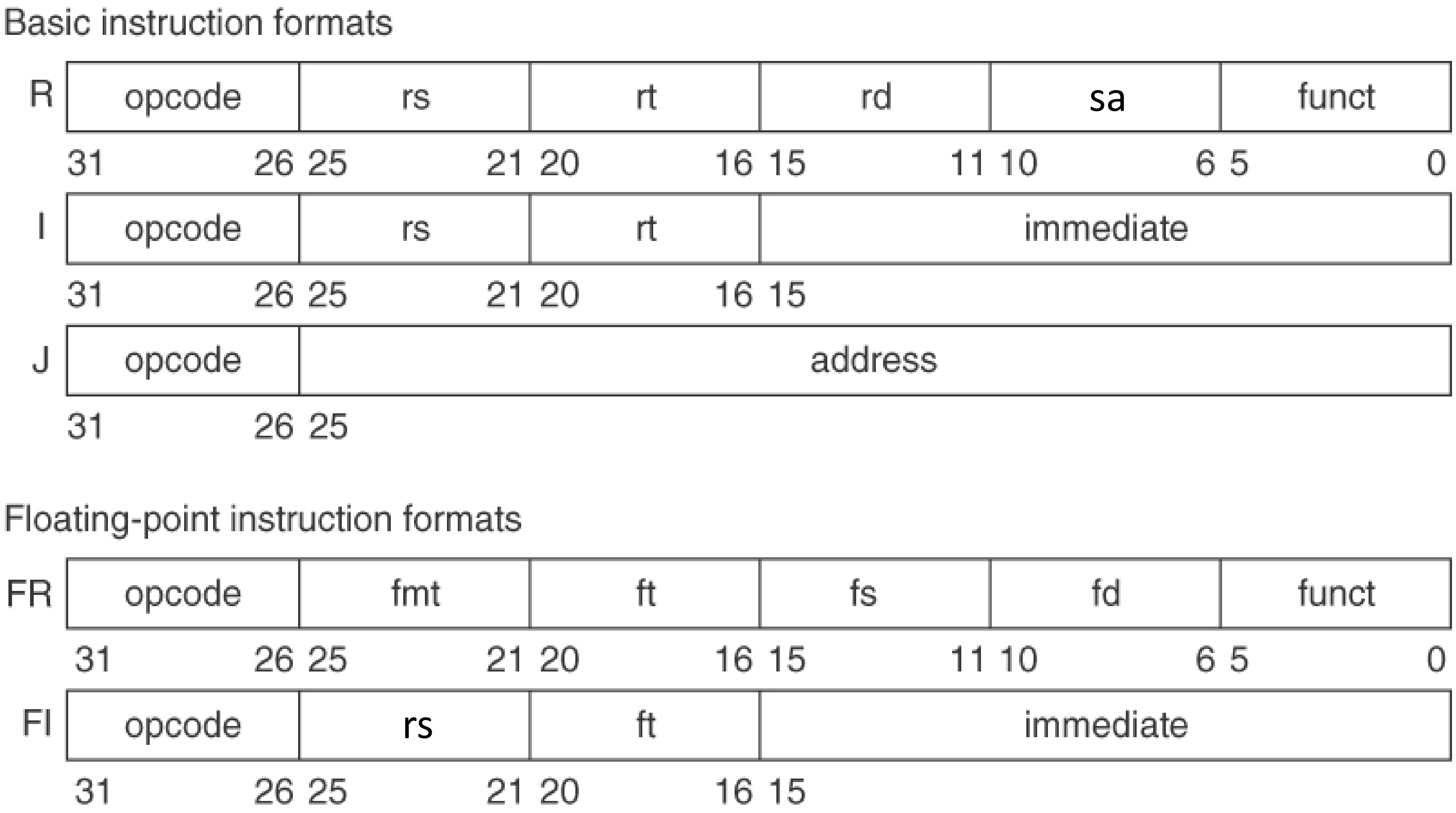
More than 1200 instructions with the introduction of AVX instructions

The MIPS Architecture

- Announced in 1985: MIPS I,II,III,IV,V, MIPS32, MIPS64
- MIPS64 has 32 × 64-bit general-purpose registers
  - Named R0 to R31 (also known as integer registers)
  - Register R0 is always zero and cannot be written
- There are also 32 × 64-bit floating-point registers
  - Named F0 to F31 for double-precision FP numbers
  - Single-precision FP numbers use the lower 32-bit of the register
- Integer and Floating-Point data types for MIPS64
  - 8-bit bytes, 16-bit half words, 32-bit words, and 64-bit long words
  - 32-bit single-precision and 64-bit double precision
- Latest MIPS64 release eliminated the HI and LO registers
  - Multiply and Divide instructions write their results into GPR registers

MIPS Instruction Formats

- ❖ All instructions are 32 bits with a 6-bit primary opcode
- ❖ These are the main instruction formats, not the only ones



Basic instruction formats

R	opcode	rs	rt	rd	sa	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15		
J	opcode	address				
	31 26	25				

Floating-point instruction formats

FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	rs	ft	immediate		
	31 26	25 21	20 16	15		

MIPS Load and Store Instructions

- ❖ Load/Store instructions use the I-Format with 16-bit displacement

Instruction	Name	Meaning
LD Rt, Imm(Rs)	Load double word	Reg[Rt] $\leftarrow_{64}$ Mem[Reg[Rs] + Imm]
LW Rt, Imm(Rs)	Load word	Reg[Rt] $\leftarrow_{32}$ Mem[Reg[Rs] + Imm] (sign-extend)
LH Rt, Imm(Rs)	Load half word	Reg[Rt] $\leftarrow_{16}$ Mem[Reg[Rs] + Imm] (sign-extend)
LB Rt, Imm(Rs)	Load byte	Reg[Rt] $\leftarrow_{8}$ Mem[Reg[Rs] + Imm] (sign-extend)
LWU Rt, Imm(Rs)	Load word unsigned	Reg[Rt] $\leftarrow_{32}$ Mem[Reg[Rs] + Imm] (zero-extend)
LHU Rt, Imm(Rs)	Load half unsigned	Reg[Rt] $\leftarrow_{16}$ Mem[Reg[Rs] + Imm] (zero-extend)
LBU Rt, Imm(Rs)	Load byte unsigned	Reg[Rt] $\leftarrow_{8}$ Mem[Reg[Rs] + Imm] (zero-extend)
SD Rt, Imm(Rs)	Store double word	Mem[Reg[Rs] + Imm] $\leftarrow_{64}$ Reg[Rt]
SW Rt, Imm(Rs)	Store word	Mem[Reg[Rs] + Imm] $\leftarrow_{32}$ Reg[Rt] (lower 32-bit)
SH Rt, Imm(Rs)	Load half word	Mem[Reg[Rs] + Imm] $\leftarrow_{16}$ Reg[Rt] (lower 16-bit)
SB Rt, Imm(Rs)	Load byte	Mem[Reg[Rs] + Imm] $\leftarrow_{8}$ Reg[Rt] (lower 8-bit)

MIPS Floating-Point Load and Store

Instruction	Name	Meaning
LDC1 Ft, Imm(Rs)	Load double to FP	Reg[Ft] $\leftarrow_{64}$ Mem[Reg[Rs] + Imm]
LWC1 Ft, Imm(Rs)	Load word to FP	Reg[Ft] $\leftarrow_{32}$ Mem[Reg[Rs] + Imm] (zero-extend)
SDC1 Ft, Imm(Rs)	Store FP double	Mem[Reg[Rs] + Imm] $\leftarrow_{64}$ Reg[Ft]
SWC1 Ft, Imm(Rs)	Store FP word	Mem[Reg[Rs] + Imm] $\leftarrow_{32}$ Reg[Ft] (lower 32-bit)

- ❖ Coprocessor 1 (C1) means the Floating-Point unit
- ❖ The FI-Format is used for floating-point load/store instructions
- ❖ Displacement Addressing: Address = Reg[Rs] + Imm16
- ❖ Data should be aligned in memory
- ❖ Loading less than 64 bits ➔ Data is extended to 64 bits
- ❖ Storing less than 64 bits ➔ Lower bit are written to memory

MIPS64 ALU Instructions

- ❖ ALU instructions can be Register-Register or Register-Immediate
  - ✧ DADD is used for 64-bit integer addition, ADD for 32-bit integer addition

Instruction	Meaning
DADD Rd, Rs, Rt	Reg[Rd] ← Reg[Rs] + Reg[Rt] (64-bit integer addition)
DSUB Rd, Rs, Rt	Reg[Rd] ← Reg[Rs] – Reg[Rt] (64-bit integer subtraction)
DADDU / DSUBU	Same as DADD / DSUB, but Ignore Overflow
DADDI Rt, Rs, Imm	Reg[Rt] ← Reg[Rs] + Imm (immediate can be negative)
DADDIU Rt, Rs, Imm	Same as DADDI, but Ignore Overflow
DSLL, DSRL, DSRA	Shift Left, Shift Right Logical, Shift Right Arithmetic
DSLLV, DSRLV, DSRAV	Same as DSLL, DSRL, DSRA, but with a variable amount
AND, OR, XOR, NOR	R-type bitwise logic instructions (64-bit operands)
ANDI, ORI, XORI	I-type bitwise logic (16-bit immediate is zero-extended)
SLT, SLTU, SLTI, SLTIU	Set Less Than, Unsigned, Immediate, (Result is 0 or 1)

MIPS64 Multiply and Divide Instructions

- ❖ Multiplication of 64-bit integers produces a 128-bit product
  - ✧ Low and High 64-bit of the product are computed using two instructions
- ❖ Division of 64-bit integers produces a quotient and remainder
  - ✧ Results are written to a register Rd ➔ LO and HI registers are eliminated

Instruction	Meaning
DMUL Rd, Rs, Rt	Rd = Low 64-bit of Signed 64-bit integer multiplication
DMUH Rd, Rs, Rt	Rd = High 64-bit of Signed 64-bit integer multiplication
DMULU Rd, Rs, Rt	Rd = Low 64-bit of Unsigned 64-bit integer multiplication
DMUHU Rd, Rs, Rt	Rd = High 64-bit of Unsigned 64-bit integer multiplication
DDIV Rd, Rs, Rt	Rd = Quotient of Signed 64-bit integer division
DMOD Rd, Rs, Rt	Rd = Modulo (Remainder) of Signed 64-bit integer division
DDIVU Rd, Rs, Rt	Rd = Quotient of Unsigned 64-bit integer division
DMODU Rd, Rs, Rt	Rd = Modulo (Remainder) of Unsigned 64-bit integer division

MIPS Floating-Point Instructions

Instruction	Meaning
ADD.S Fd, Fs, Ft	Reg[Fd] ← Reg[Fs] + Reg[Ft] (32-bit double-precision add)
ADD.D Fd, Fs, Ft	Reg[Fd] ← Reg[Fs] + Reg[Ft] (64-bit double-precision add)
SUB.S, SUB.D	FP Subtract (FR-format), Single and Double-precision
MUL.S, MUL.D	FP Multiply (FR-format), Single and Double-precision
DIV.S, DIV.D	FP Divide (FR-format): Single and Double-precision
MADDF.S, MADDF.D	FP Fused Multiply-Add: Reg[Fd] ← Reg[Fd] + Reg[Fs] × Reg[Ft]
SEL.S, SEL.d	Select: Reg[Fd] ← Reg[Fd].bit0 ? Reg[Ft] : Reg[Fs]
CVT.x.y Fd, Fs	Convert: Reg[Fd] ← convert_from_format_y_to_x (Reg[Fs])
CMP.cond.S (or .D)	Compare: Reg[Fd] ← compare_cond (Reg[Fs], Reg[Ft])

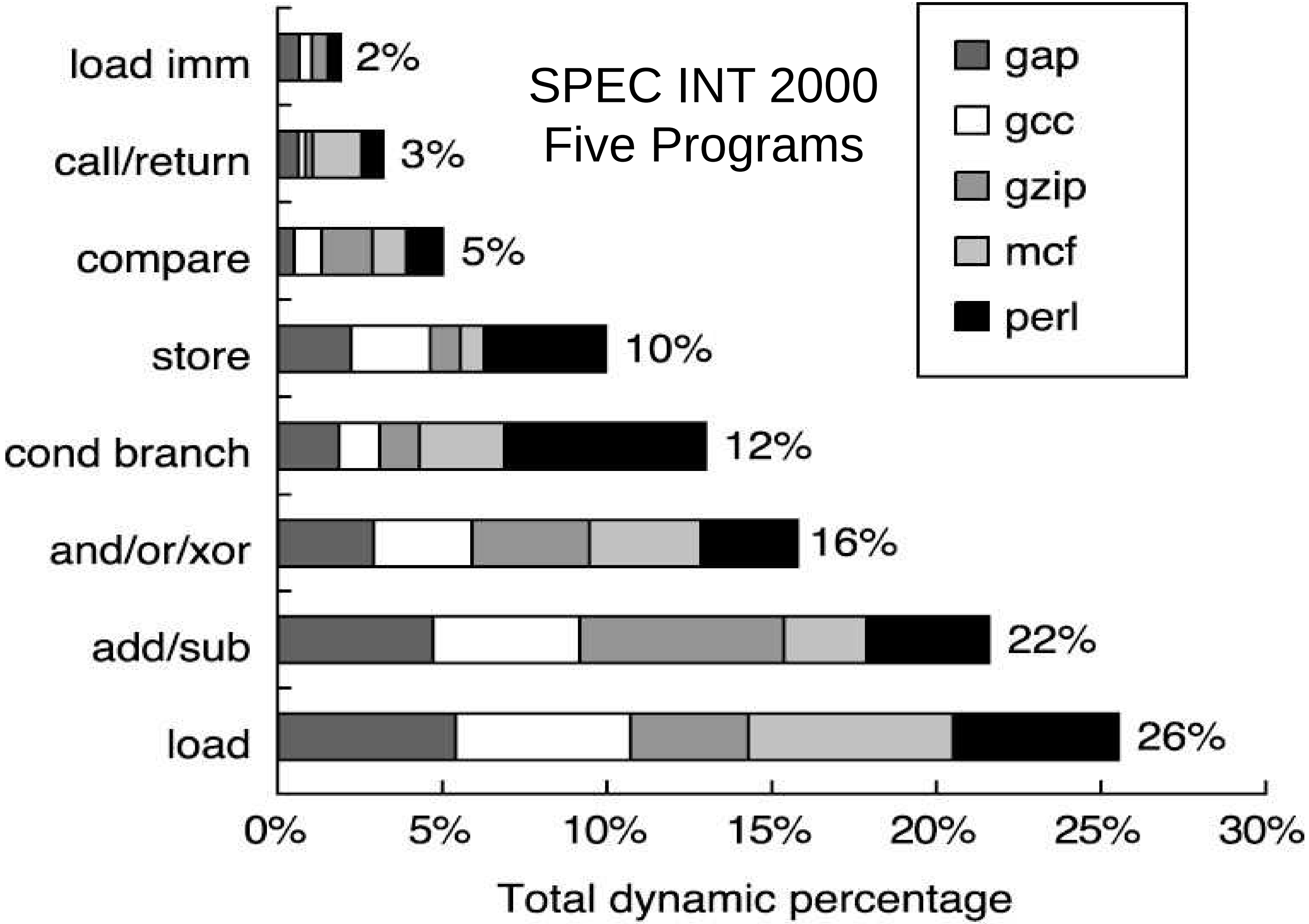
- ❖ FCSR: Floating-point Control and Status Register
  - ✧ Controls the FPU: Rounding mode, enables and reports FP exceptions
- ❖ CMP (compare) instruction: result is written to register Fd

MIPS Control Flow Instructions

Instruction		Meaning
J	target	Jump within current 256 MB region (J-Format: 26-bit target addr)
JAL	target	Jump And Link (J-Format): Reg[R31] ← RA, PC ← target addr
JALR	Rd, Rs	Jump And Link Register (R-Format): Reg[Rd] ← RA, PC ← Reg[Rs]
JR	Rs	Jump Register (R-Format), PC ← Reg[Rs]
BEQ	Rs, Rt, Offset	Branch on Equal (I-Format): if (Reg[Rs] == Reg[Rt])
BNE	Rs, Rt, Offset	Branch on Not Equal (I-Format): if (Reg[Rs] != Reg[Rt])
BLTZ	Rs, Offset	Branch on Less Than Zero (I-Format): if (Reg[Rs] < 0)
BGTZ, BLEZ, BGEZ		Branch (I-Format): if (Reg[Rs] > 0), if (Reg[Rs] <= 0), if (Reg[Rs] >= 0)
BC1EQZ, BC1NEZ		Branch (FI-Format): if (Reg[Ft].bit0 == 0), if (Reg[Ft].bit0 != 0)
SYSCALL, ERET		System Call exception, Exception Return to user code

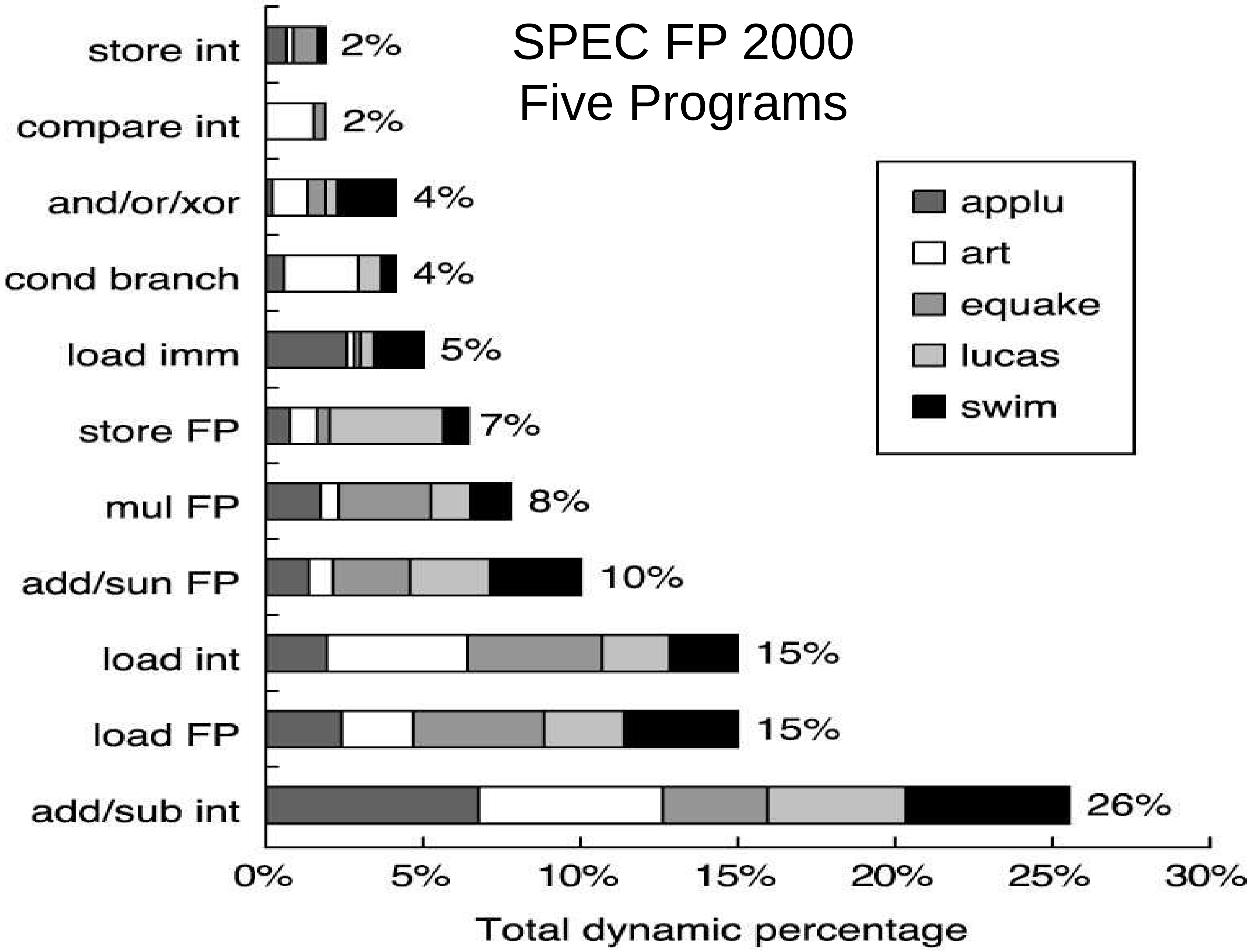
- Branch Target Address: PC-Relative
  - PC ← PC + 4 + Offset × 4

MIPS Instruction Set Usage



SPEC INT 2000
Five Programs

MIPS Instruction Set Usage (cont'd)



Fallacies and Pitfalls

- Fallacy: Complex and Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down instruction execution
  - Compilers are good at making fast code from simple instructions
- Fallacy: You can design a flawless architecture
  - All architecture design involves tradeoffs
- Fallacy: Use assembly code for high performance
  - Modern compilers are better at dealing with modern processors
- Pitfall: Innovating ISA without accounting for the compiler
- Pitfall: Designing “high-level” instructions for specific languages

What Makes a Good Instruction Set?

- Provides a simple software interface
- Allows simple, fast, efficient hardware implementations
  - But across 25+ year time frame
- Instruction set changes continually (ISA revisions & extensions)
  - Technology allows larger CPU over time
  - Technology constraints changes (power versus performance)
  - Compiler, programming style, applications change
- Software compatibility negatively impacts ISA innovation
- New instruction set can be justified only by a new large market and technological advances