INSTRUCTION SET ARCHITECTURE

CHAPTER # 02

INTRODUCTION

- First, we present a taxonomy of instruction set alternatives and give some qualitative assessment of the advantages and disadvantages of various approaches.
- Second, we present and analyze some instruction set measurements that are largely independent of a specific instruction set.
- Third, we address the issue of languages and compilers and their bearing on instruction set architecture.
- Finally, the "Putting It All Together" section shows how these ideas are reflected in the RISC-V instruction set, which is typical of RISC architectures.

INTRODUCTION

- Desktop computing emphasizes the performance of programs with integer and floating-point data types, with little regard for program size. For example, code size has never been reported in the five generations of SPEC benchmarks.
- Servers today are used primarily for database, file server, and Web applications, plus some time-sharing applications for many users. Hence, floating-point performance is much less important for performance than integers and character strings, yet virtually every server processor still includes floating point instructions.

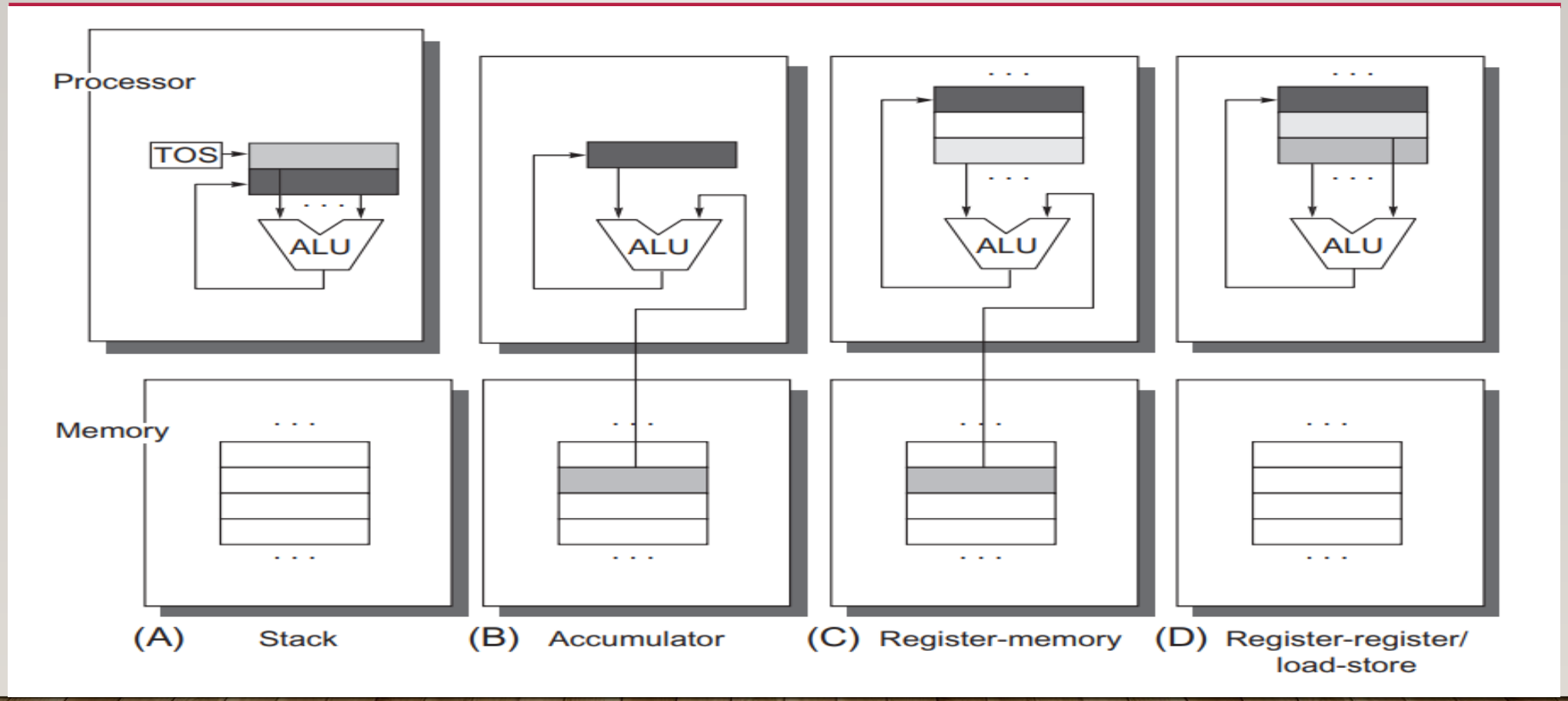
INTRODUCTION

- Personal mobile devices and embedded applications value cost and energy, so code size is important because less memory is both cheaper and lower energy, and some classes of instructions (such as floating point) may be optional to reduce chip costs, and a compressed version of the instructions set designed to save memory space may be used.
- Recent 80x86 microprocessors, including all the Intel Core microprocessors built in the past decade, use hardware to translate from 80x86 instructions to RISC-like instructions and then execute the translated operations inside the chip. They maintain the illusion of 80x86 architecture to the programmer while allowing the computer designer to implement a RISC-style processor for performance. There remain, however, serious disadvantages for a complex instruction set like the 80x86.

CLASSIFYING INSTRUCTION SET ARCHITECTURES

- The type of internal storage in a processor is the most basic differentiation, so in this section we will focus on the alternatives for this portion of the architecture.
- The major choices are a stack, an accumulator, or a set of registers.
- Operands may be named explicitly or implicitly:
- The operands in a stack architecture are implicitly on the top of the stack.
- In an accumulator architecture one operand is implicitly the accumulator.
- The general-purpose register architectures have only explicit operands—either registers or memory locations.

CLASSIFYING INSTRUCTION SET ARCHITECTURES



CLASSIFYING INSTRUCTION SET ARCHITECTURES

- As the figures show, there are really two classes of register computers.
- One class can access memory as part of any instruction, called register-memory architecture, and the other can access memory only with load and store instructions, called load-store architecture.
- The major reasons for the emergence of general-purpose register (GPR) computers are twofold. First, registers—like other forms of storage internal to the processor—are faster than memory.
- Second, registers are more efficient for a compiler to use than other forms of internal storage.

CLASSIFYING INSTRUCTION SET ARCHITECTURES

- How many registers are sufficient?
- The answer, of course, depends on the effectiveness of the compiler. Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables. Modern compiler technology and its ability to effectively use larger numbers of registers has led to an increase in register counts in more recent architectures.

CLASSIFYING INSTRUCTION SET ARCHITECTURES

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

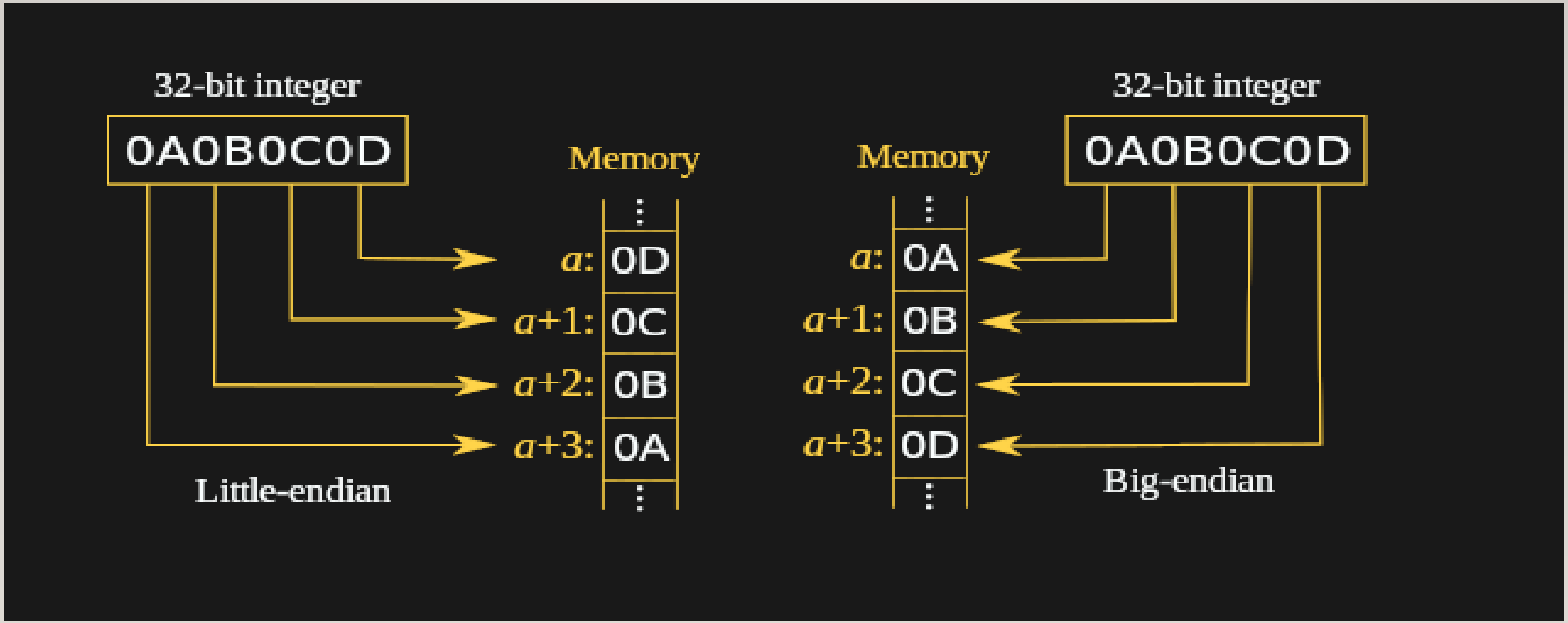
CLASSIFYING INSTRUCTION SET ARCHITECTURES

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C)	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density	Operands are not equivalent because a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

MEMORY ADDRESSING

- How is a memory address interpreted? That is, what object is accessed as a function of the address and the length? All the instruction sets discussed in this book are byte addressed and provide access for bytes (8 bits), half words (16 bits), and words (32 bits). Most of the computers also provide access for double words (64 bits).
- There are two different conventions for ordering the bytes within a larger object.
- Little Endian and Big endian .

MEMORY ADDRESSING



MEMORY ADDRESSING

- A second memory issue is that in many computers, accesses to objects larger than a byte must be aligned. An access to an object of size s bytes at byte address A is aligned if A mod s= 0. Figure shows the addresses at which an access is aligned or misaligned.

Memory address	Alignment (8 bit)	Alignment (16 bit)	Alignment (32 bit)	Alignment (64 bit)
0x0000_0000	Aligned	Aligned	Aligned	Aligned
0x0000_0001	Aligned	Non Aligned	Non Aligned	Non Aligned
0x0000_0002	Aligned	Aligned	Non Aligned	Non Aligned
0x0000_0003	Aligned	Non Aligned	Non Aligned	Non Aligned
0x0000_0004	Aligned	Aligned	Aligned	Non Aligned
0x0000_0005	Aligned	Non Aligned	Non Aligned	Non Aligned
0x0000_0006	Aligned	Aligned	Non Aligned	Non Aligned
0x0000_0007	Aligned	Non Aligned	Non Aligned	Non Aligned
0x0000_0008	Aligned	Aligned	Aligned	Aligned

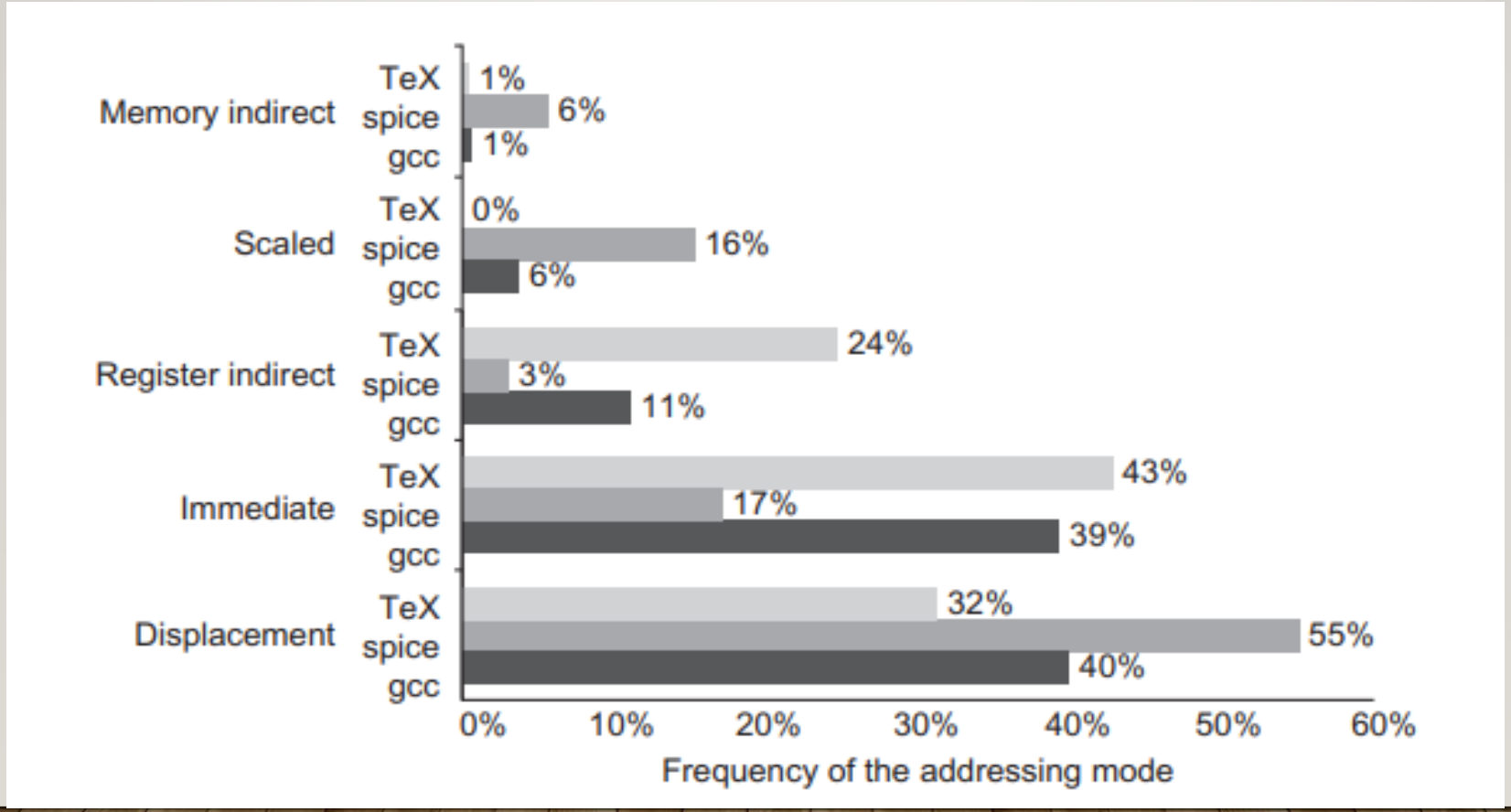
ADDRESSING MODES

- Given an address, we now know what bytes to access in memory. In this subsection we will look at addressing modes—how architectures specify the address of an object they will access.
- Addressing modes specify constants and registers in addition to locations in memory.
- When a memory location is used, the actual memory address specified by the addressing mode is called the effective address.

ADDRESSING MODES

Addressing mode	Example instruction	Meaning
Register	Add R4,R3	Regs[R4] ← Regs[R4] + Regs[R3]
Immediate	Add R4,3	Regs[R4] ← Regs[R4] + 3
Displacement	Add R4,100(R1)	Regs[R4] ← Regs[R4] + Mem[100 + Regs[R1]]
Register indirect	Add R4,(R1)	Regs[R4] ← Regs[R4] + Mem[Regs[R1]]
Indexed	Add R3,(R1 + R2)	Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]
Direct or absolute	Add R1,(1001)	Regs[R1] ← Regs[R1] + Mem[1001]
Memory indirect	Add R1,@(R3)	Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]
Autoincrement	Add R1,(R2)+	Regs[R1] ← Regs[R1] + Mem[Regs[R2]] Regs[R2] ← Regs[R2] + *d*
Autodecrement	Add R1,-(R2)	Regs[R2] ← Regs[R2] - *d* Regs[R1] ← Regs[R1] + Mem[Regs[R2]]
Scaled	Add R1,100(R2)[R3]	Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * *d*]

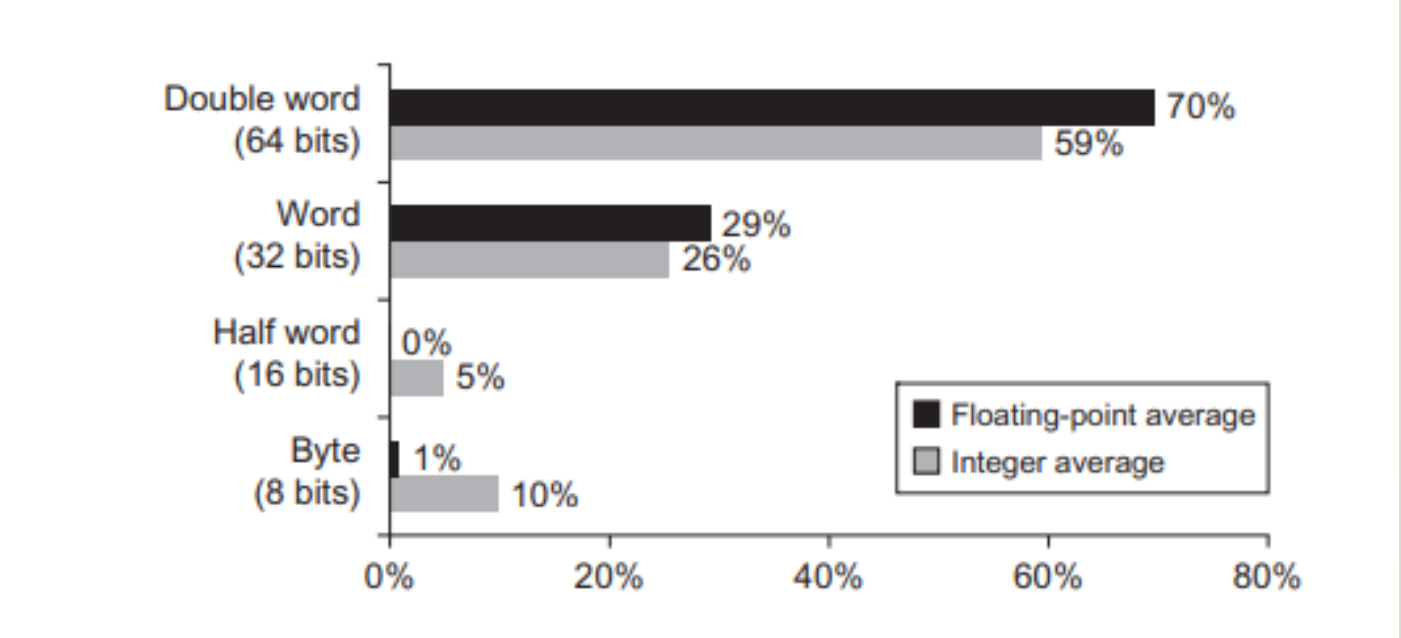
ADDRESSING MODES



TYPE AND SIZE OF OPERANDS

- How is the type of an operand designated? Usually, encoding in the opcode designates the type of an operand—this is the method used most often.
- Alternatively, the data can be annotated with tags that are interpreted by the hardware.
- Let's start with desktop and server architectures. Usually the type of an operand—integer, single-precision floating point, character, and so on—effectively gives its sizes.
- Common operand types include character (8 bits), half word (16 bits), word (32 bits), single-precision floating point (also 1 word), and double precision floating point (2 words). Integers are almost universally represented as two's complement binary numbers.

TYPE AND SIZE OF OPERANDS



OPERATIONS IN THE INSTRUCTION SET

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

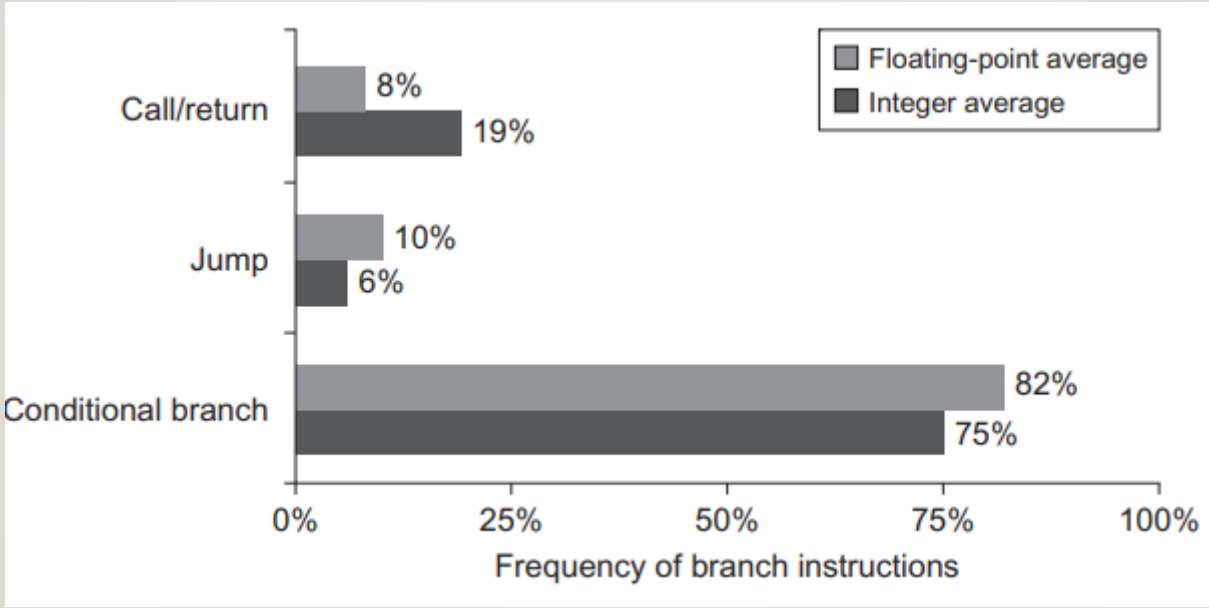
OPERATIONS IN THE INSTRUCTION SET

Rank	80x86 instruction	Integer average % total executed)
1	Load	22%
2	Conditional branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move register-register	4%
9	Call	1%
10	Return	1%
Total		**96%**

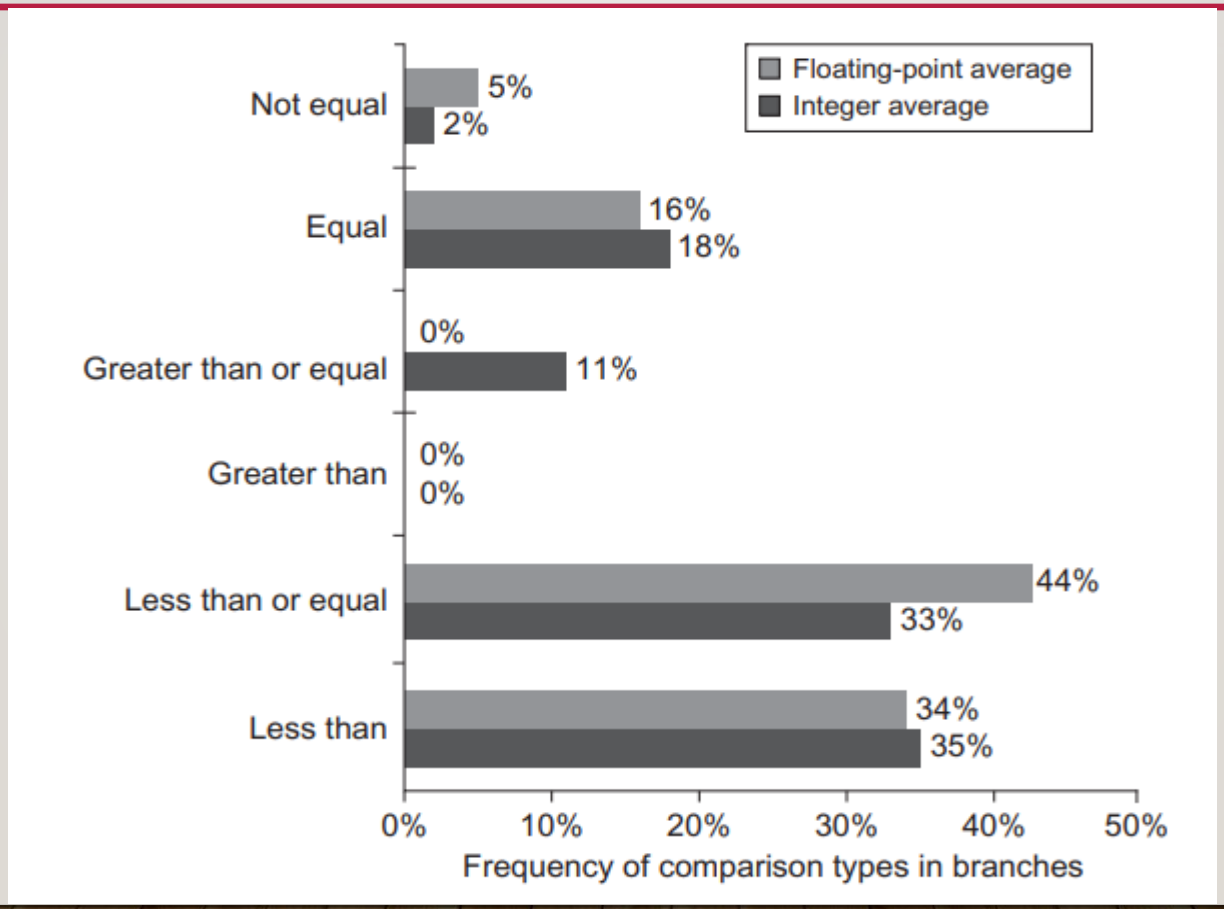
INSTRUCTIONS FOR CONTROL FLOW

- We can distinguish four different types of control flow change:
- ■ Conditional branches
- ■ Jumps
- ■ Procedure calls
- ■ Procedure returns

INSTRUCTIONS FOR CONTROL FLOW

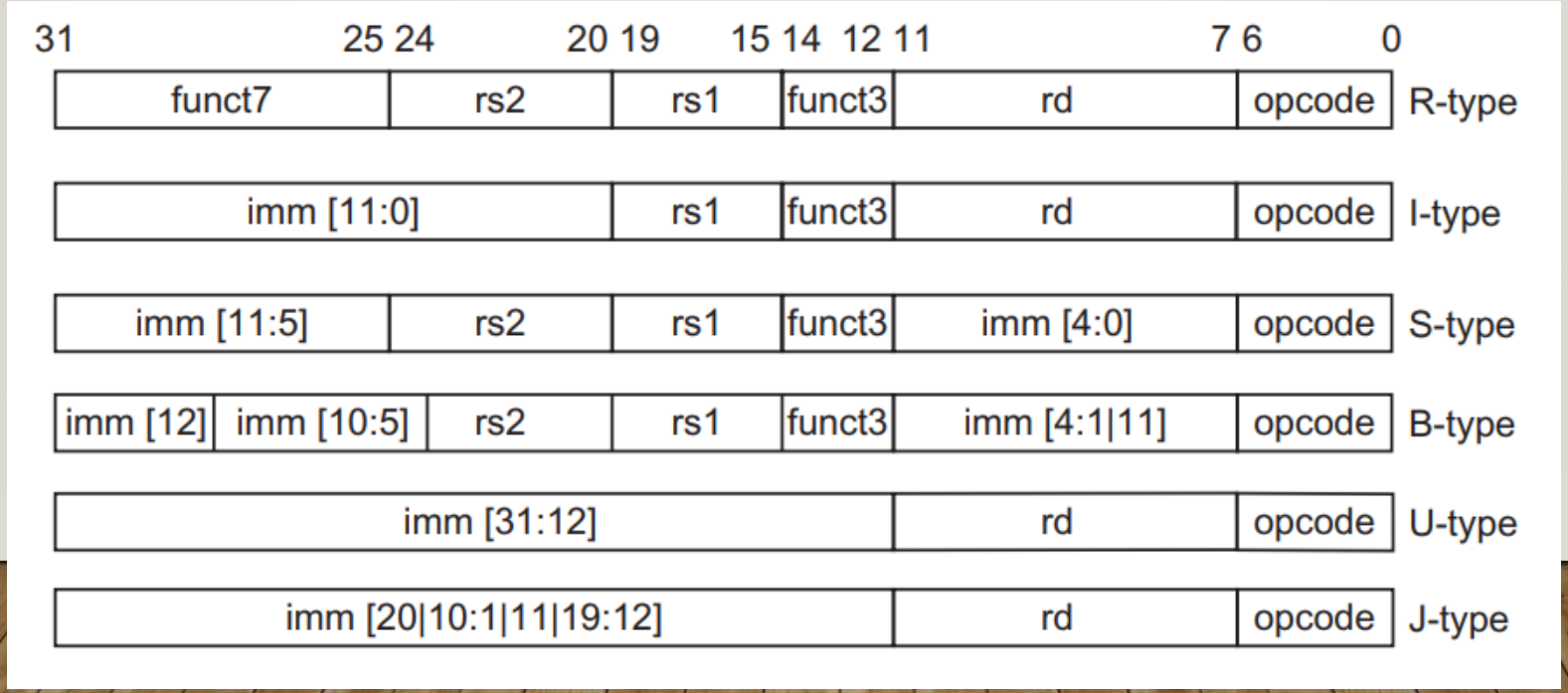


INSTRUCTIONS FOR CONTROL FLOW



ENCODING AN INSTRUCTION SET

- There are two basic choices on encoding: fixed length and variable length.
- All ARMv8 and RISC-V instructions are 32 bits long, which simplifies instruction decoding.

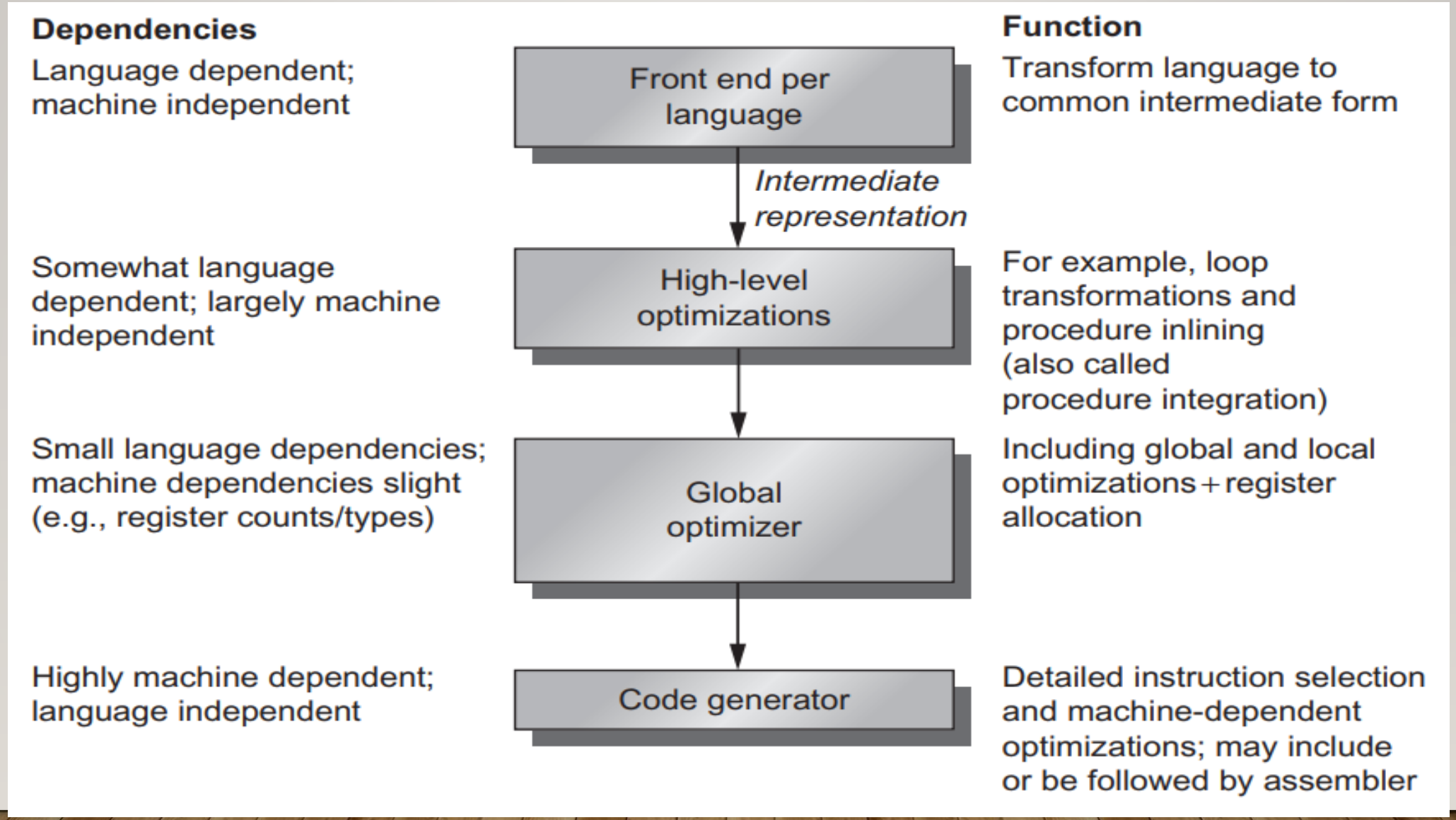


THE ROLE OF COMPILERS

- In this section, we discuss the critical goals in the instruction set primarily from the compiler viewpoint. It starts with a review of the anatomy of current compilers.
- Next we discuss how compiler technology affects the decisions of the architect, and how the architect can make it hard or easy for the compiler to produce good code.
- We conclude with a review of compilers and multimedia operations, which unfortunately is a bad example of cooperation between compiler writers and architects

THE ROLE OF COMPILERS

- A compiler writer's first goal is correctness—all valid programs must be compiled correctly.
- The second goal is usually speed of the compiled code.
- Typically, a whole set of other goals follows these two, including fast compilation, debugging support, and interoperability among languages.
- The complexity of writing a correct compiler is a major limitation on the amount of optimization that can be done.


Dependencies
Language dependent; machine independent
Somewhat language dependent; largely machine independent
Small language dependencies; machine dependencies slight (e.g., register counts/types)
Highly machine dependent; language independent
Front end per language
Intermediate representation
High-level optimizations
Global optimizer
Code generator
Function
Transform language to common intermediate form
For example, loop transformations and procedure inlining (also called procedure integration)
Including global and local optimizations + register allocation
Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

RISC-V ARCHITECTURE

- RISC-V ("RISC Five") is a modern RISC instruction set developed at the University of California, Berkeley, which was made free and openly adoptable in response to requests from industry.
- In addition to a full software stack (compilers, operating systems, and simulators), there are several RISC-V implementations freely available for use in custom chips or in field-programmable gate arrays.
- Developed 30 years after the first RISC instruction sets, RISC-V inherits its ancestors' good ideas—a large set of registers, easy-to-pipeline instructions, and a lean set of operations—while avoiding their omissions or mistakes.

REGISTERS ALLOCATION

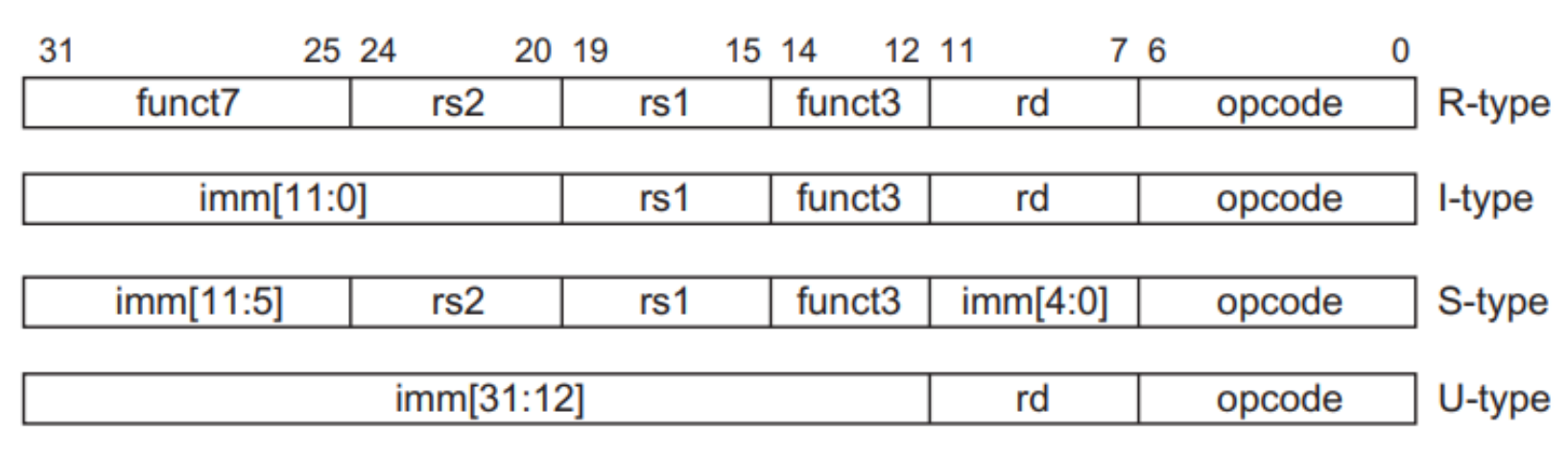
Register	Name	Use
x0	zero	The constant value 0
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function arguments/return values
x12-x17	a2-a7	Function arguments
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporaries
f0-f7	ft0-ft7	FP temporaries
f8-f9	fs0-fs1	FP saved registers
f10-f11	fa0-fa1	FP function arguments/return values
f12-f17	fa2-fa7	FP function arguments
f18-f27	fs2-fs11	FP saved registers
f28-f31	ft8-ft11	FP temporaries

ADDRESSING MODES FOR RISC-V DATA TRANSFERS

- The only data addressing modes are immediate and displacement, both with 12-bit fields.
- Register indirect is accomplished simply by placing 0 in the 12-bit displacement field, and limited absolute addressing with a 12-bit field is accomplished by using register 0 as the base register.
- As it is a load-store architecture, all references between memory and either GPRs or FPRs are through loads or stores.

RISC-V INSTRUCTION FORMAT

- Because RISC-V has just two addressing modes, these can be encoded into the opcode.
- Following the advice on making the processor easy to pipeline and decode, all instructions are 32 bits with a 7-bit primary opcode.



RISC-V INSTRUCTION FORMAT

Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

RISC-V OPERATIONS

- There are four broad classes of instructions:
- loads and stores,
- ALU operations,
- branches and jumps, and
- floating-point operations.