DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Computer Architecture

Lecture 2: Instruction Set Principles (Appendix A)

Chih-Wei Liu 劉志尉

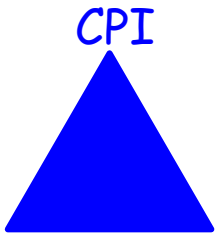
National Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

Instruction Set Architecture (ISA)

- ISA: the portion of the computer visible to the programmer or compiler writer
- Outline:
  - 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.

Computer Performance



$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	X
Organization		X	X
Technology			X

ISA Design Issue

- Where are operands stored?
- How many explicit operands are there?
- How is the operand location specified?
- What type & size of operands are supported?
- What operations are supported?

Before answering these questions, let's consider more about

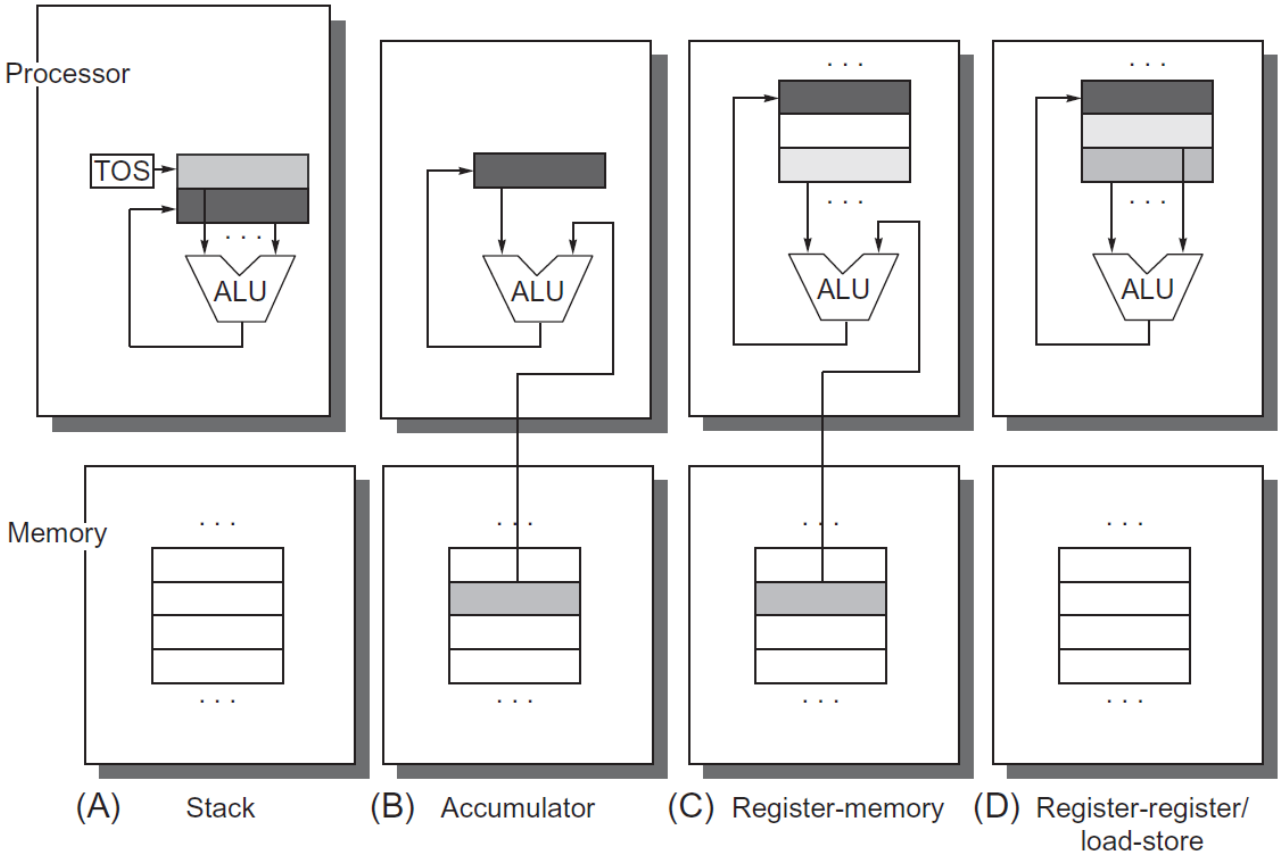
- Data operands (source and destination)
- Memory addressing modes
- Operations

Classifying ISAs

- Type of internal storage in a processor is the most basic differentiation:
  - A stack
  - An accumulator
  - A set of registers
- Operands may be named explicitly or implicitly
  - In a stack architecture, all Operands, including two source operands and one result, are implicit on the top of the stack (TOS).
  - In an accumulator architecture, one operand and the result are implicitly the accumulator and the other is form memory.
  - The general-purpose register architectures have only explicit operands—either registers or memory locations.
    - Register-memory architecture
    - Memory-memory architecture
    - Register-register architecture (load-store architecture)

Four Classes of ISAs (1/2)

- The arrows indicate whether the operand is an input or the result of the arithmetic-logical unit (ALU) operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result.



Four Classes of ISAs (2/2)

- The code sequence for **`C = A + B`** for four classes of instruction sets.

Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R3,R1,B`	`Load R2,B`
`Add`	`Store C`	`Store R3,C`	`Add R3,R1,R2`
`Pop C`			`Store R3,C`

Assume that `A`, `B`, and `C` all belong in memory and that the values of `A` and `B` cannot be destroyed.

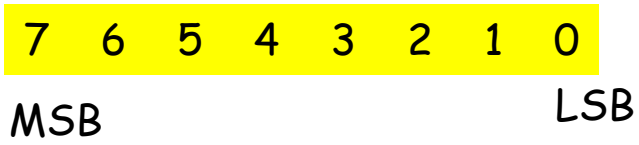
Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has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

Memory Addressing for Data Operand

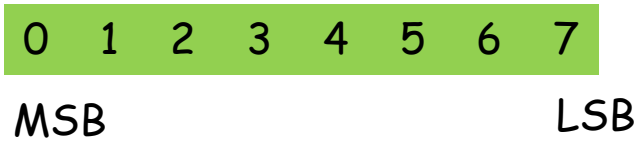
- Most processors are byte-addressable and provide access for
  - Byte or character (8-bit)
  - Half-word or short integer (16-bit)
  - Integer word or single-precision floating point (32-bit)
  - Double-word or long integer or double-precision floating point (64-bit)
- How memory addresses are interpreted and how they are specified?
  - Little Endian or Big Endian
    - for ordering the bytes within a larger object within memory
  - Alignment or misaligned memory access
    - for accessing to an abject larger than a byte from memory
  - Addressing modes
    - for specifying constants, registers, and locations in memory

Byte-Order ("Endianness")

- **Little Endian**
  - The byte order put the byte whose address is "xx…x000" at the least-significant position in the double word
  - E.g. Intel, DEC, …
  - The bytes are numbered as

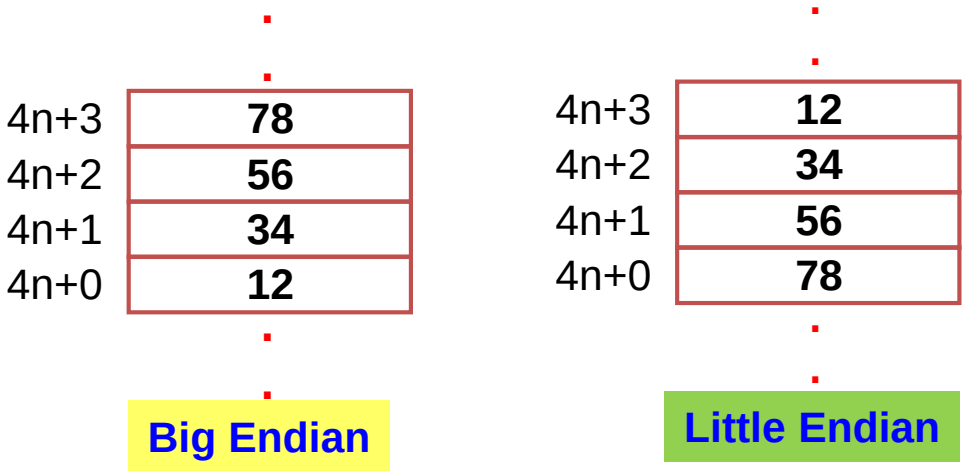


- **Big Endian**
  - The byte order put the byte whose address is "xx…x000" at the most-significant position in the double word
  - E.g. MIPS, IBM, Motorolla, Sun, HP, …
  - The bytes are numbered as



Little or Big Endian ?

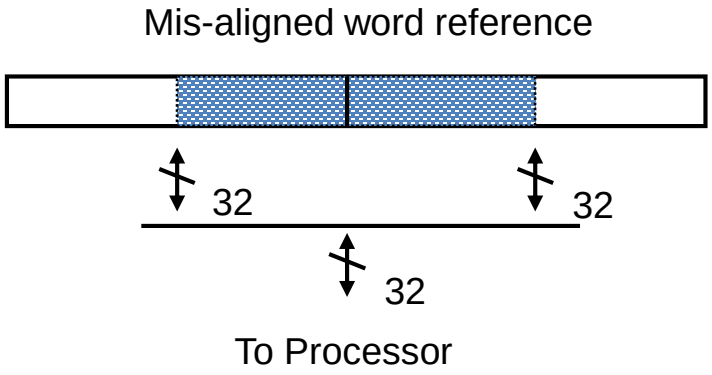
- No absolute advantage for one over the other, but
  - Byte order is a problem when exchanging data among computers
- Example
  - In C, **`int num = 0x12345678; // a 32-bit word`**,
  - how is `num` stored in memory?



  - Little Endian ordering fails to match the normal ordering of words when strings are compared. Strings appear “SDRAWKCAB” (backwards) in the registers.

Data Alignment

- An access to object of size S bytes at byte address A is called aligned if $A \bmod S = 0$.
  - The instruction is typically aligned on a word.
- Access to an unaligned operand may require more memory accesses !!

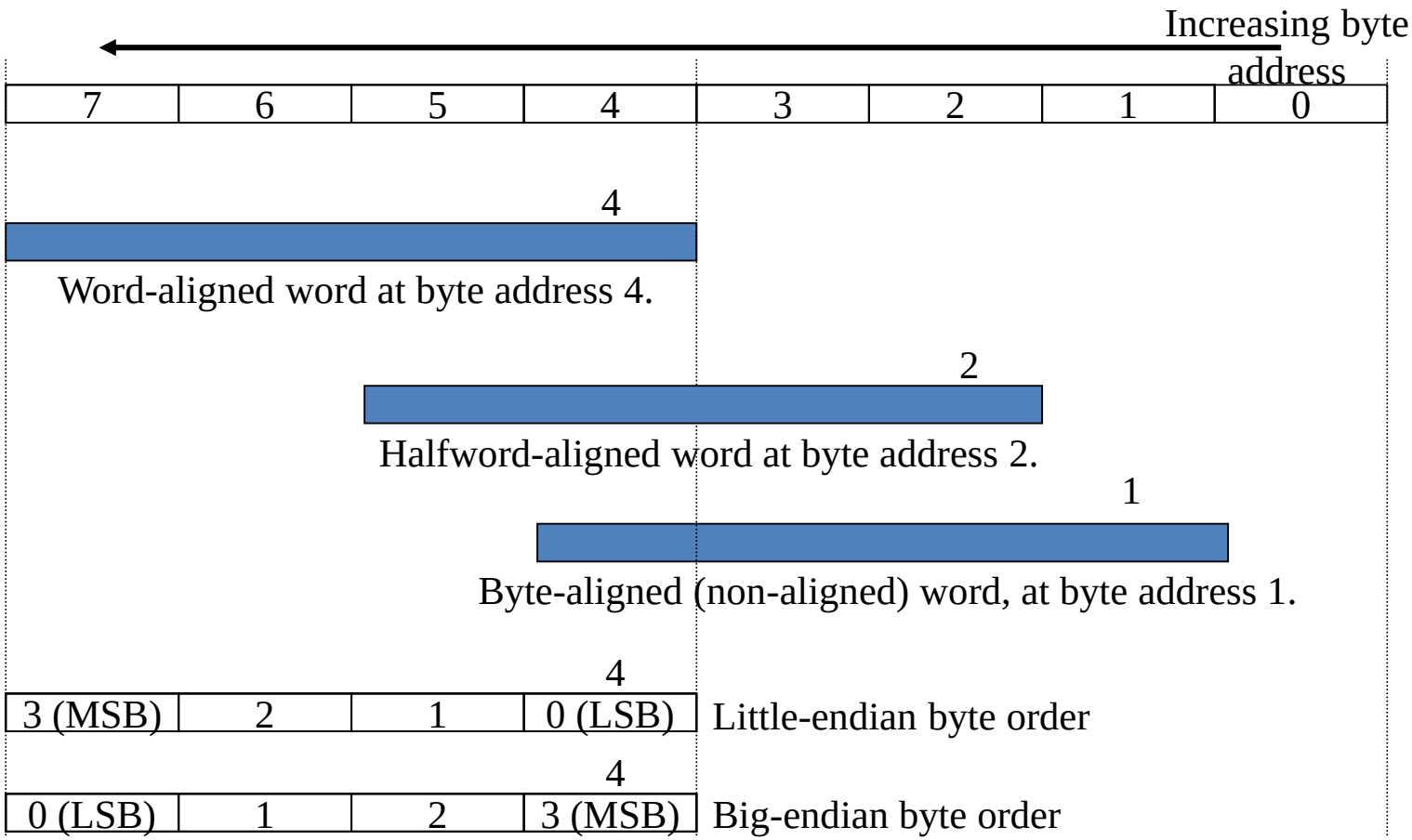


Remarks

- Unrestricted Alignment
  - Software is simple
  - Hardware must detect misalignment and make more memory accesses
  - Expensive logic to perform detection
  - Can slow down all references
  - Sometimes required for backwards compatibility
- Restricted Alignment
  - Software must guarantee alignment
  - Hardware detects misalignment access and traps
  - No extra time is spent when data is aligned

- Since we want to ***make the common case fast***, having restricted alignment is often a better choice, unless compatibility is an issue.

Summary: Endians & Alignment



Addressing Mode ?

- It answers the question:
  - Where can operands/results be located?

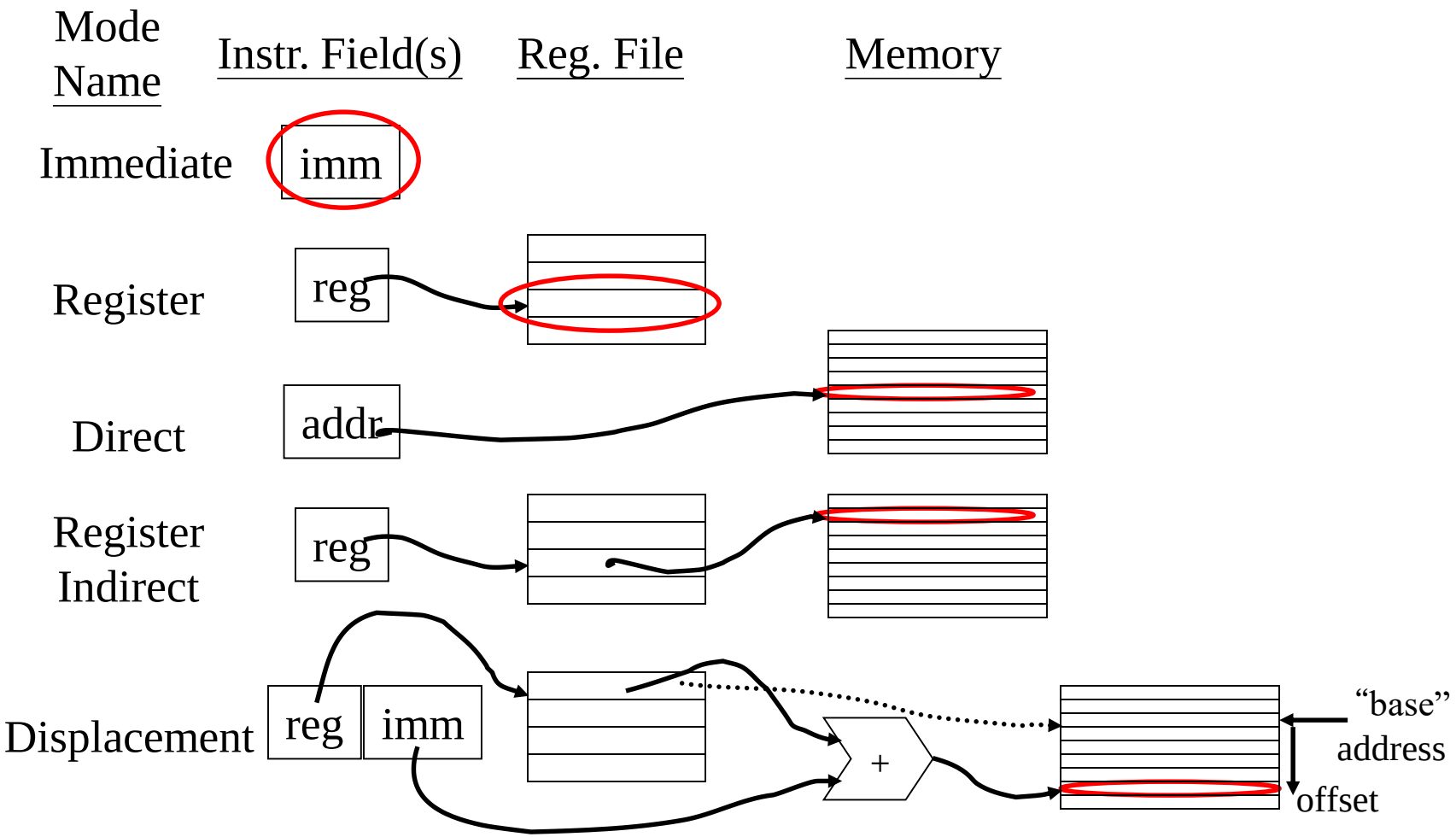
- Recall that we have two types of storage in computer : registers and memory
  - A single operand can come from either a register or a memory location
  - Addressing modes offer various ways of specifying the specific location

Addressing Mode Example

`R:Register; M:Memory`

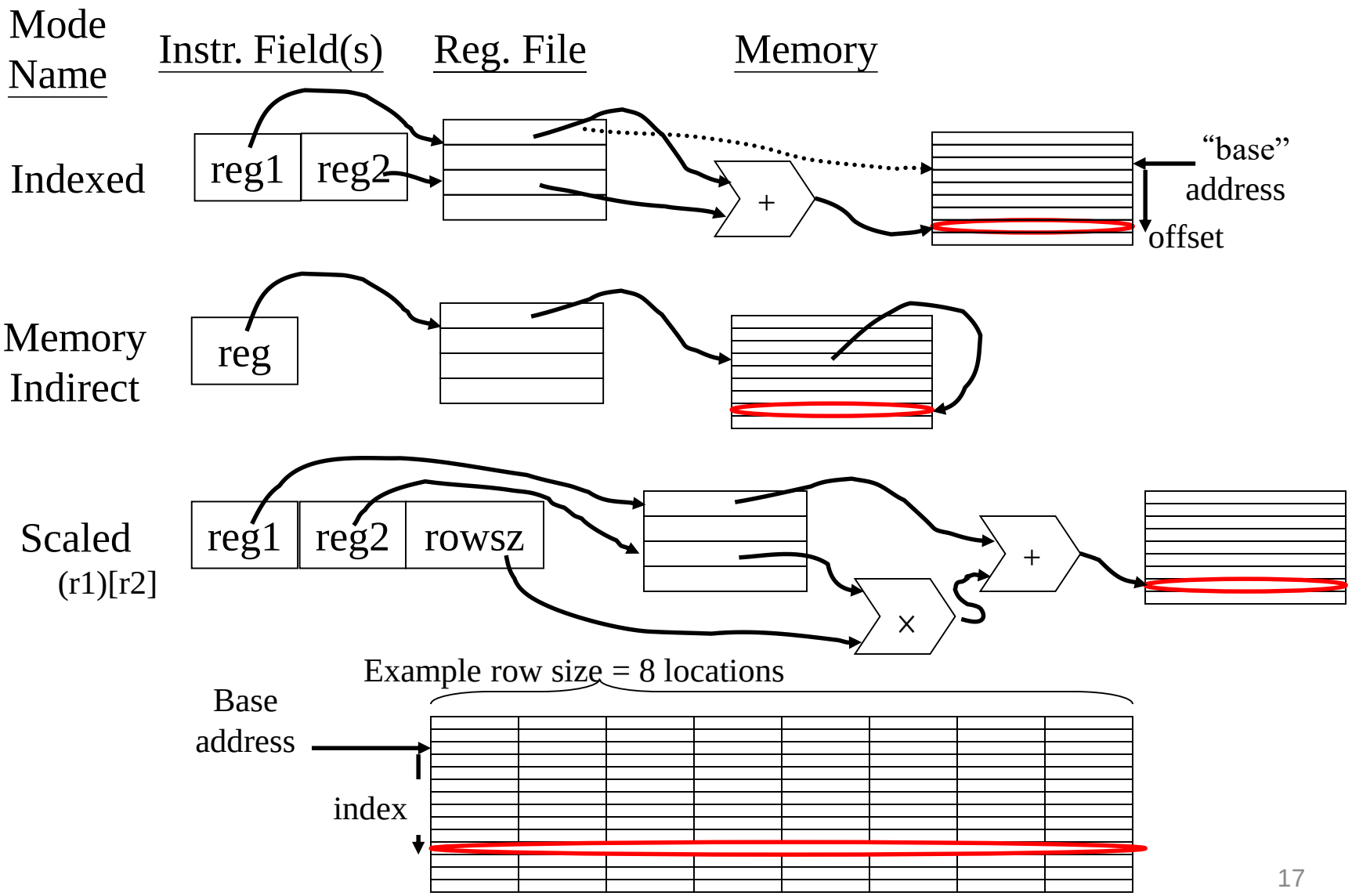
Addressing Mode	Example	Action
1. Register direct	`Add R1, R2, R3`	`R1 <- R2 + R3`
2. Immediate	`Add R1, R2, #3`	`R1 <- R2 + 3`
3. Register indirect	`Add R1, R2,(R3)`	`R1 <- R2 + M[R3]`
4. Displacement	`LD R1, 100(R2)`	`R1 <- M[100 + R2]`
5. Indexed	`LD R1, (R2 + R3)`	`R1 <- M[R2 + R3]`
6. Direct	`LD R1, (1000)`	`R1 <- M[1000]`
7. Memory Indirect	`Add R1, R2, @(R3)`	`R1 <- R2 + M[M[R3]]`
8. Auto-increment	`LD R1, (R2)+`	`R1 <- M[R2]`
		`R2 <- R2 + d`
9. Auto-decrement	`LD R1, (R2)-`	`R1 <- M[R2]`
		`R2 <- R2 – d`
10. Scaled	`LD R1, 100(R2)[R3]`	`R1 <- M[100+R2+R3*d]`

Addressing Modes Visualization (1/2)



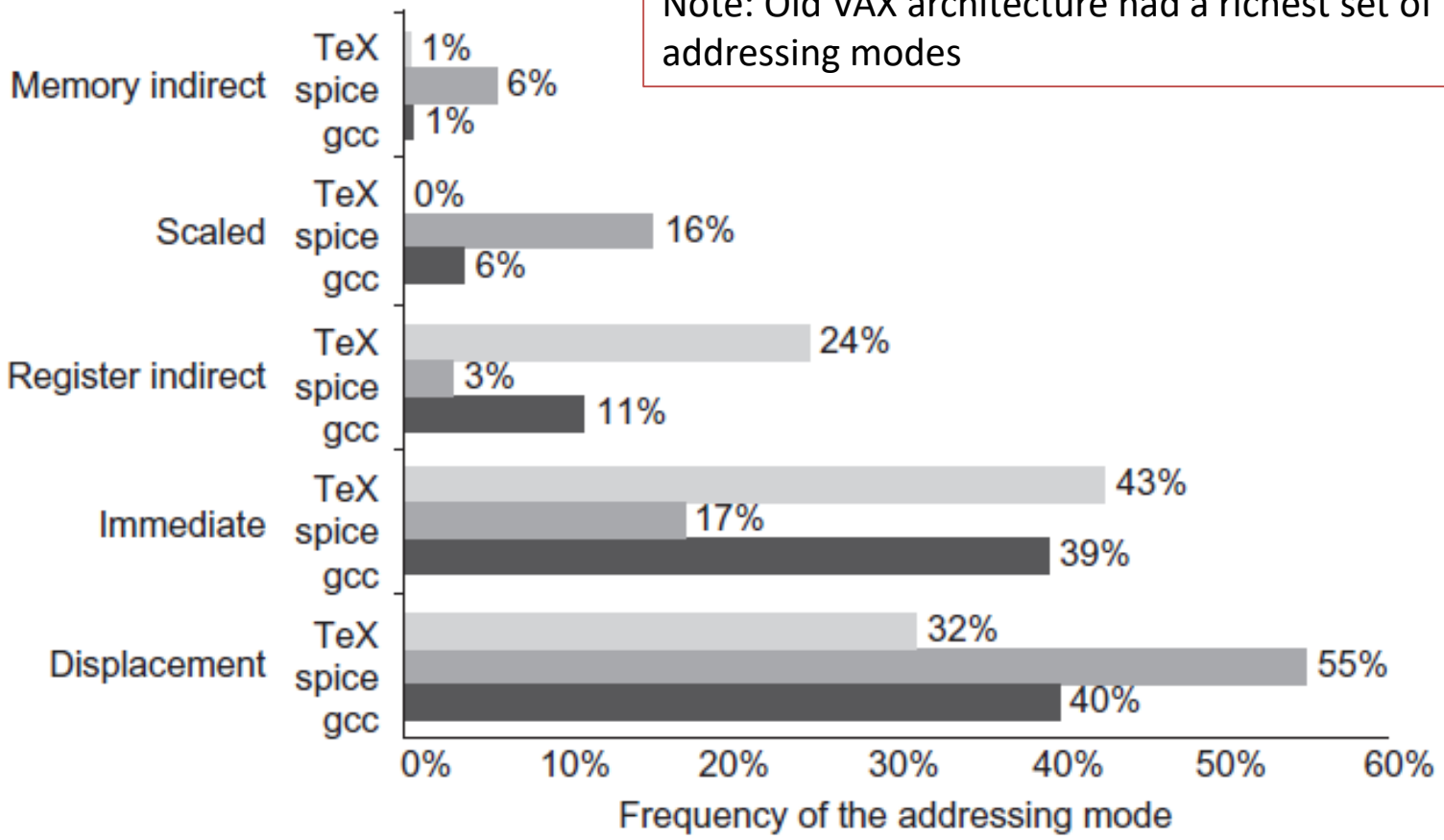
all your base are belong to us

Addressing Modes Visualization (2/2)



Summary of Use of Addressing Modes in Three Programs on VAX Machine

Note: Old VAX architecture had a richest set of addressing modes



How Many Addressing Mode ?

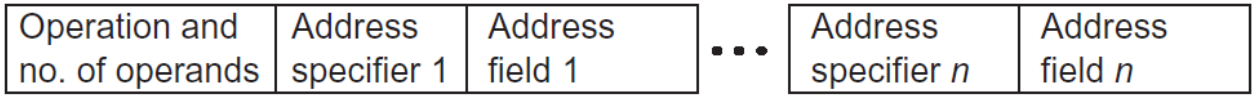
- A Tradeoff: complexity vs. instruction count
  - Should we add more modes?
    - Depends on the application class
    - Special addressing modes for DSP/GPU processors
      - Modulo or circular addressing, bit reverse addressing, stride or gather/scatter addressing, …
    - Some DSPs rely on hand-coded libraries for using novel addressing modes
- Need to support at least three types of addressing mode
  - Displacement, immediate, and register indirect
- For 32-bit fixed-width instruction encoding
  - The size of the address for displacement mode is 12—16 bits
  - The size of immediate field is 8—16 bits

Operations in ISA

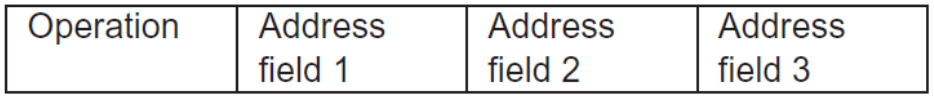
Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual 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

Encoding an Instruction Set

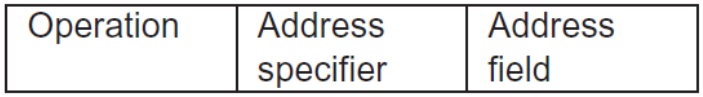
- Instructions are generally specified by some “fields.”
- Variable length encoding vs. Fixed-width encoding vs. Hybrid encoding

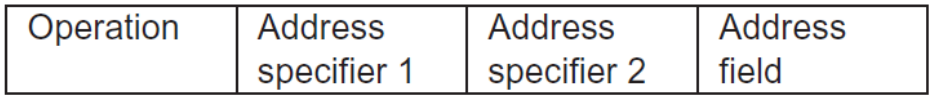


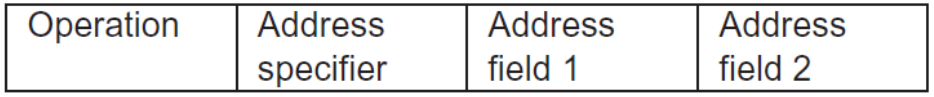
(A) Variable (e.g., Intel 80x86, VAX)



(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)



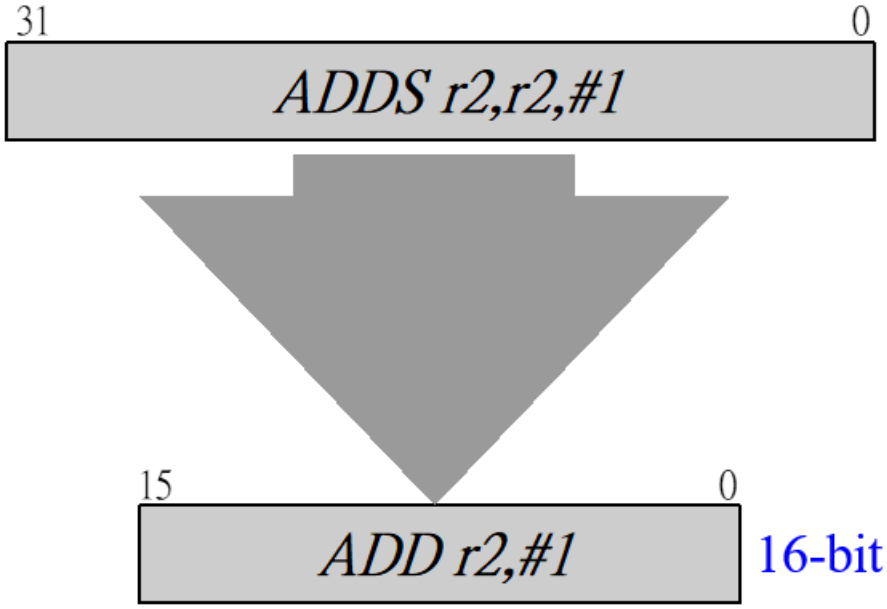




(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

Reduced Code Size in RISCs

- Hybrid instruction encoding
  - 32-bit normal-mode instruction
  - 16-bit narrow-mode instruction
    - Support fewer operations, smaller address and immediate fields, fewer registers, and the two-address format
    - Thumb mode in ARM, RV32IC (C standing for compressed)



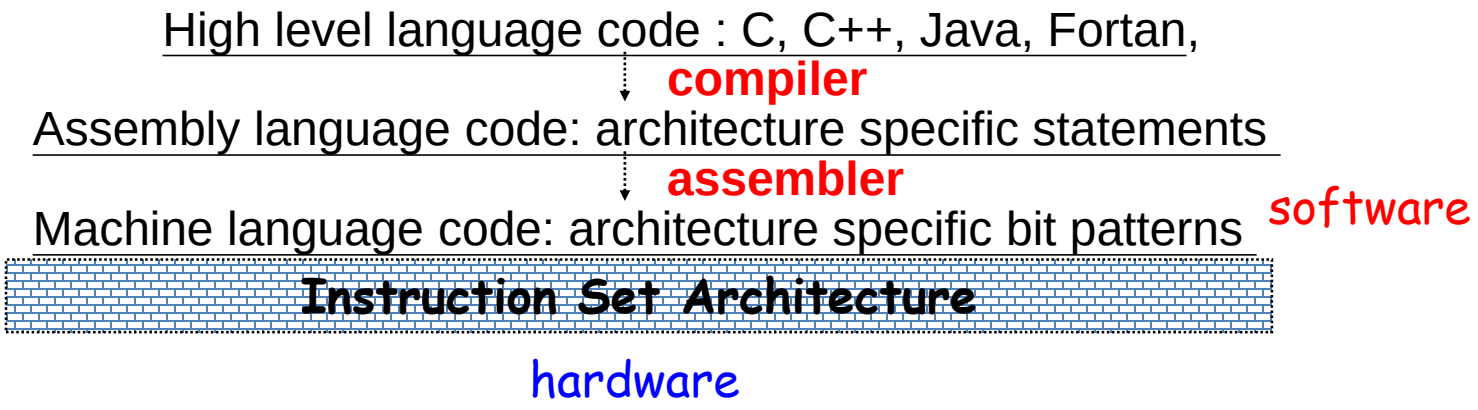
32-bit ARM instruction

- Source and destination registers identical
- Only low registers are used
- Constants are of limited size
- Inline barrel shifter is not used, Conditional execution is not used, …

16-bit Thumb instruction

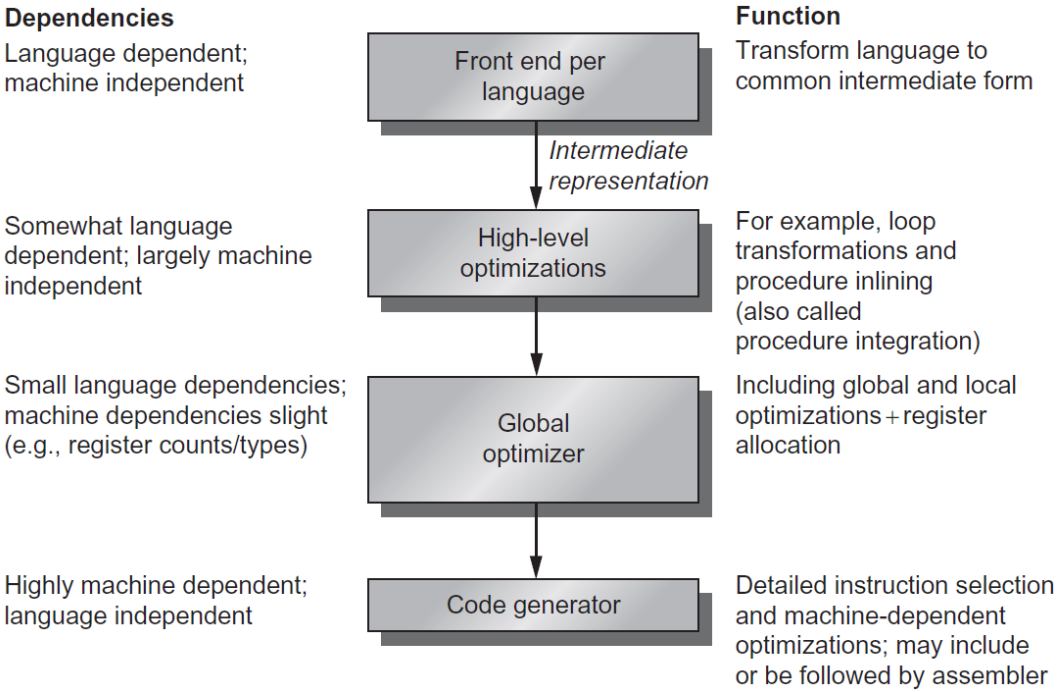
ISA Summary

- A specification of a standardized programmer-visible interface to hardware, comprises of:
  - A set of instructions
    - instruction types
    - with associated argument fields, assembly syntax, and machine encoding.
  - A set of named storage locations
    - registers
    - memory
  - A set of addressing modes (ways to name locations)
  - Often an I/O interface
    - memory-mapped



The Role of Compilers

- The structure of recent compiler



- High-level optimizations are often done on the source with output fed to later optimization passes.
- Local optimizations optimize code only within a straight-line code fragment (called a basic block by compiler people).
- Global optimizations extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.
- Register allocation associates registers with operands.
- Machine-dependent optimizations attempt to take advantage of specific architectural knowledge.

RISC-V Architecture

Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

- A freely licensed open standard, similar to many of the RISC architectures.
- RISC-V is the load-store architecture.
- RISC-V has three base instructions sets: RV32I, RV32E, RV64I, and a reserved spot for a future fourth: RV128I.
- All the extensions extend one of the base instruction sets, for example RV64IMAFD (also known as RV64G, for short), it refers to the base 64-bit instruction set with extensions M, A, F, and D.

Registers for RV64IMAFD

- RV64G has 32 64-bit general-purpose registers (GPRs) (or called integer registers), named x0, x1, … , x31.
- x0 is always 0.
- The extension part of RV64G contains a set of floating point registers (FPRs), named f0, f1, … , f31. Both single- and double-precision floating-point operations (32-bit and 64-bit) are provided.
- FPRs of RV64G are either 32 32-bit registers (which can hold single-precision FP values) or 32 64-bit registers (when holding one single precision FP, the other half of the FPR is unused).

Addressing Modes for RISC-V Data Transfers

- RV64G memory is byte addressable with a 64-bit address and uses Little Endian byte numbering.
- The only data addressing modes for RISC-V 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.
- Embracing zero gives us four effective modes, although only two are supported in the architecture.
- Memory accesses need not be aligned; however, it may be that unaligned accesses run extremely slow.

RV64G Operations

- RV64G supports four broad classes of instructions: loads and stores, ALU operations, branches and jumps, and floating-point operations.

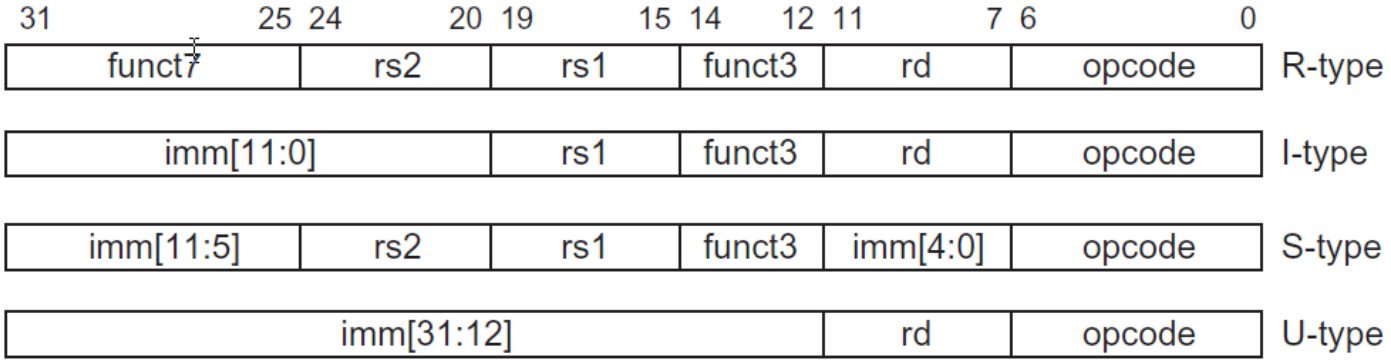
Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement + contents of a GPR*
lb, lbu, sb	Load byte, load byte unsigned, store byte (to/from integer registers)
lh, lhu, sh	Load half word, load half word unsigned, store half word (to/from integer registers)
lw, lwu, sw	Load word, store word (to/from integer registers)
ld, sd	Load doubleword, store doubleword
Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
add, addi, addw, addiw, sub, subi, subw, subiw	Add and subtract, with both word and immediate versions
slt, sltu, slti, sltiu	set-less-than with signed and unsigned, and immediate
and, or, xor, andi, ori, xori	and, or, xor, both register-register and register-immediate
lui	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
auipc	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
sll, srl, sra, slli, srli, srai, sllw,slliw, srli, srliw, srai, sraiw	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
mul, mulw, mulh, mulhsu, mulhu, div,divw, divu, rem, remu, remw, remuw	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions
Control	*Conditional branches and jumps; PC-relative or through register*
beq, bne, blt, bge, bltu, bgeu	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
jal,jalr	Jump and link address relative to a register or the PC
Floating point	*All FP operation appear in double precision (.d) and single (.s)*
flw, fld, fsw, fsd	Load, store, word (single precision), doubleword (double precision)
fadd, fsub, fmult, fiv, fsqrt, fmadd, fmsub, fnmadd, fnmsub, fmin, fmax, fsgn, fsgnj, fsjnx	Add, subtract, multiply, divide, square root, multiply-add, multiply-subtract, negate multiply-add, negate multiply-subtract, maximum, minimum, and instructions to replace the sign bit. For single precision, the opcode is followed by: .s, for double precision: .d. Thus fadd.s, fadd.d
feq, flt, fle	Compare two floating point registers; result is 0 or 1 stored into a GPR
fmv.x.*, fmv.*.x	Move between the FP register abd GPR, "*" is s or d
fcvt.*.l, fcvt.l.*, fcvt.*.lu, fcvt.lu.*, fcvt.*.w, fcvt.w.*, fcvt.*.wu, fcvt.wu.*	Converts between a FP register and integer register, where "*" is S or D for single or double precision. Signed and unsigned versions and word, doubleword versions

Example instruction	Instruction name
ld x1,80(x2)	Load doubleword
lw x1,60(x2)	Load word
lwu x1,60(x2)	Load word unsigned
lb x1,40(x3)	Load byte
lbu x1,40(x3)	Load byte unsigned
lh x1,40(x3)	Load half word
flw f0,50(x3)	Load FP single
fld f0,50(x2)	Load FP double
sd x2,400(x3)	Store double
sw x3,500(x4)	Store word
fsw f0,40(x3)	Store FP single
fsd f0,40(x3)	Store FP double
sh x3,502(x2)	Store half
sb x2,41(x3)	Store byte

Example instrucmtion	Instruction name
add x1,x2,x3	Add
addi x1,x2,3	Add immediate unsigned
lui x1,42	Load upper immediate
sll x1,x2,5	Shift left logical
slt x1,x2,x3	Set less than

Example instruction	Instruction name
jal x1,offset	Jump and link
jalr x1,x2,offset	Jump and link register
beq x3,x4,offset	Branch equal zero
bgt x3,x4,name	Branch not equal zero

RISC-V Instruction Format



- All instructions are 32 bits with a 7-bit primary opcode.
- 4 four major instruction types, providing 12-bit fields for displacement addressing, immediate constants, or PC-relative branch addresses.

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