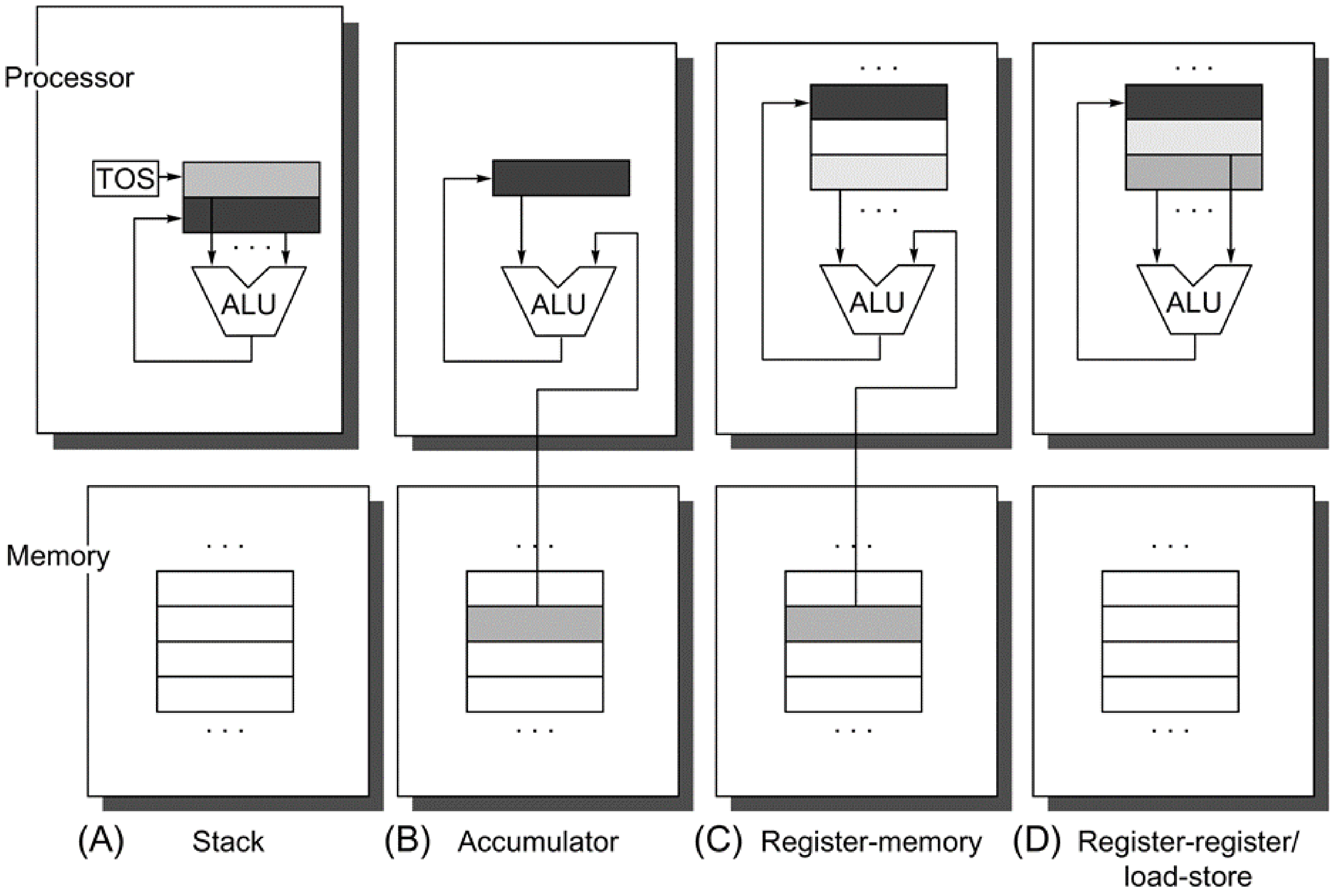
CS60003: High Performance Computer Architecture

Instruction Set Principles and RISC-V

IIT KHARAGPUR

Instructor:
Prof. Rajat Subhra Chakraborty
Professor
Dept. of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India 721302
E-mail: rschakraborty@cse.iitkgp.ac.in

Different Instruction Set Architectures



- **Most modern processors are load-store (including RISC-V)**
  - **Every new architecture developed after 1980!**
- **x86 processors support register-memory instructions**
- **Load-store architectures usually have relatively large number of General-purpose Registers (GPRs)**

H&P CA:A QA (6th. Ed.)

Different Instruction Set Architectures

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

The code sequence for `C = A + B` for four classes of instruction sets. Note that the **Add** instruction has implicit operands for stack and accumulator architectures and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure A.1 shows the **`Add`** operation for each class of architecture.

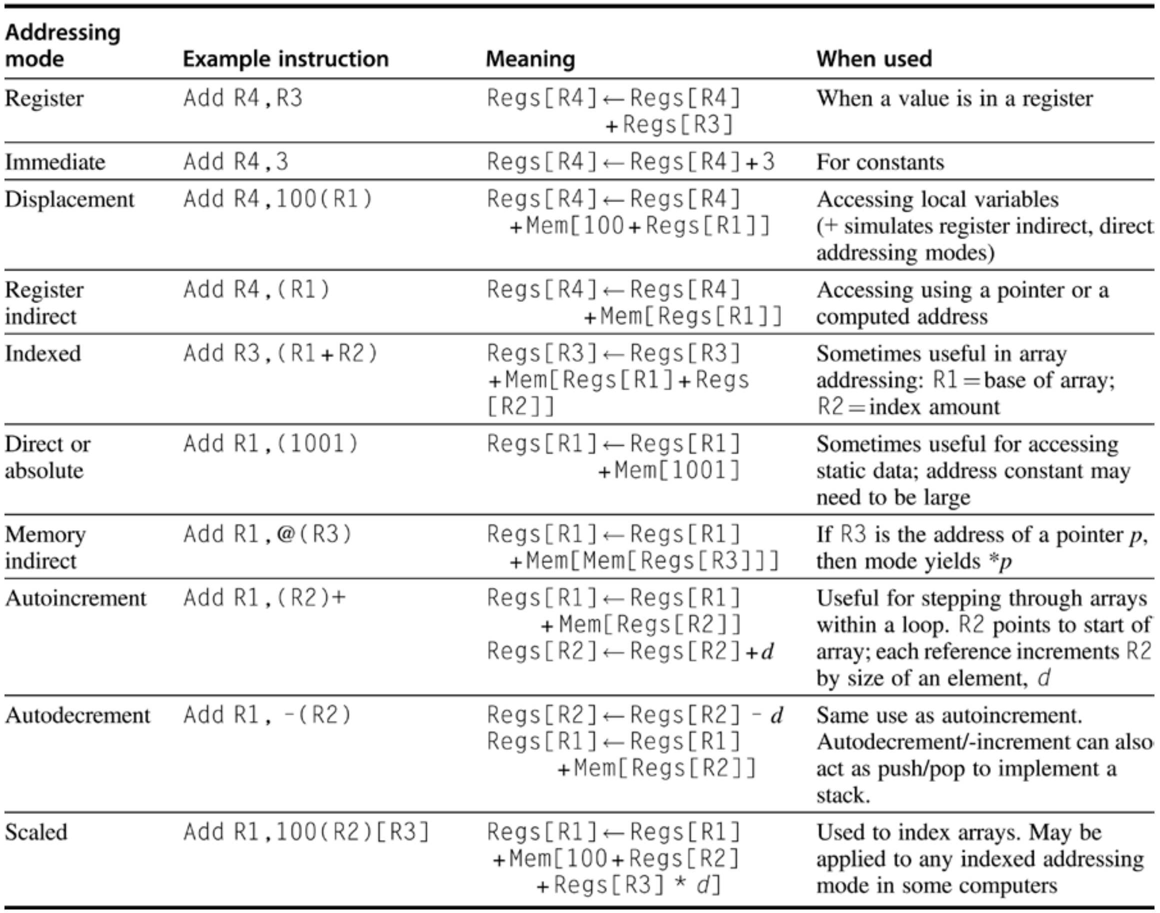
RISC-V Registers

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

RISC-V registers, names, usage, and calling conventions. In addition to the 32 general-purpose registers (x0–x31), RISC-V has 32 floating-point registers (f0–f31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number. The registers that are preserved across a procedure call are labeled “Callee” saved.

H&P CA:A QA (6th. Ed.)

Common Addressing Modes

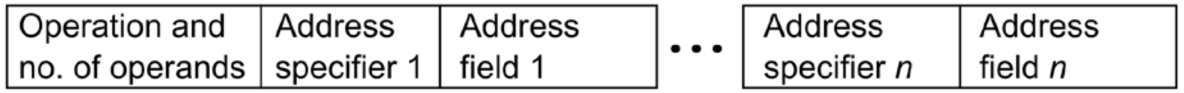


Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	Regs[R4]←Regs[R4] +Regs[R3]	When a value is in a register
Immediate	Add R4,3	Regs[R4]←Regs[R4]+3	For constants
Displacement	Add R4,100(R1)	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	Add R4,(R1)	Regs[R4]←Regs[R4] +Mem[Regs[R1]]	Accessing using a pointer or a computed address
Indexed	Add R3,(R1+R2)	Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs [R2]]	Sometimes useful in array addressing: R1=base of array; R2=index amount
Direct or absolute	Add R1,(1001)	Regs[R1]←Regs[R1] +Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	Add R1,@(R3)	Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer *p*, then mode yields **p*
Autoincrement	Add R1,(R2)+	Regs[R1]←Regs[R1] +Mem[Regs[R2]] Regs[R2]←Regs[R2]+*d*	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*
Autodecrement	Add R1, -(R2)	Regs[R2]←Regs[R2] - *d* Regs[R1]←Regs[R1] +Mem[Regs[R2]]	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	Regs[R1]←Regs[R1] +Mem[100+Regs[R2] +Regs[R3] * *d*]	Used to index arrays. May be applied to any indexed addressing mode in some computers

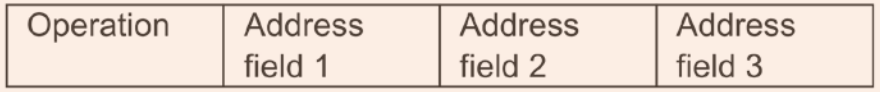
- **RISC architectures in general support relatively few addressing modes**
- **RISC-V supports:**
  - **Register, Immediate, Displacement**
  - **Can simulate two others:**
    - **Register indirect: Displacement with zero offset**
    - **Direct: Displacement with zero register (x0)**
  - **RISC-V also supports PC-relative addressing, for branch and jump instructions**

H&P CA:A QA (6th. Ed.)

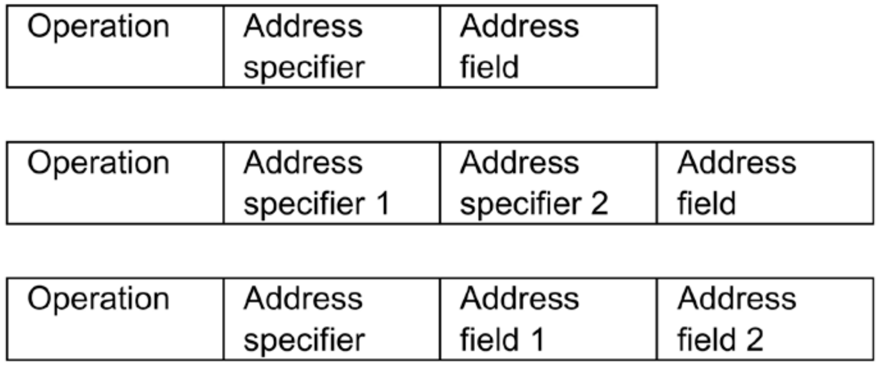
Major Types of Instruction Encoding



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



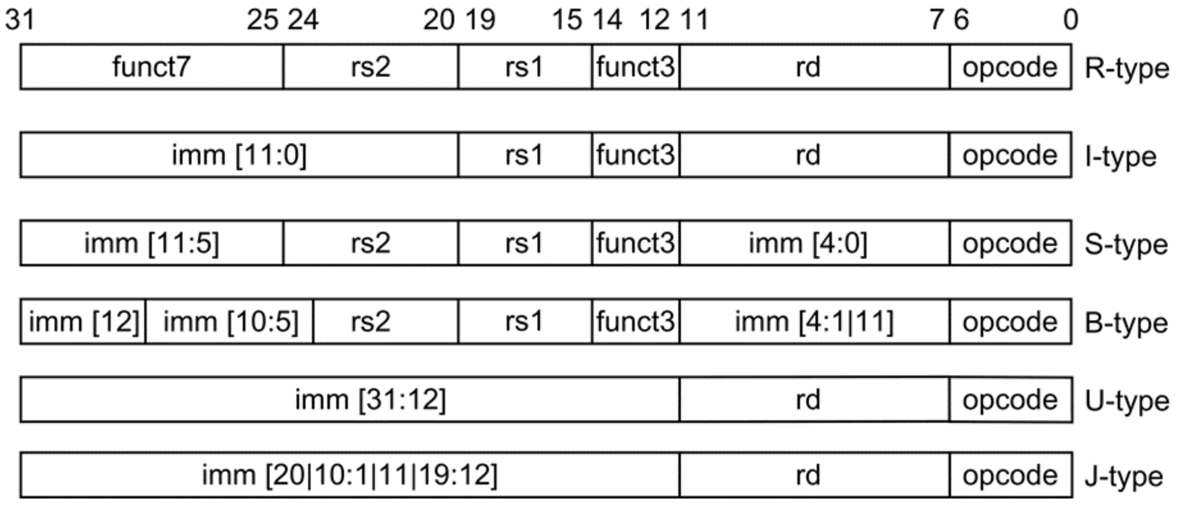
(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)



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

Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, because unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode. It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.

RISC-V Instruction Encoding (layout)

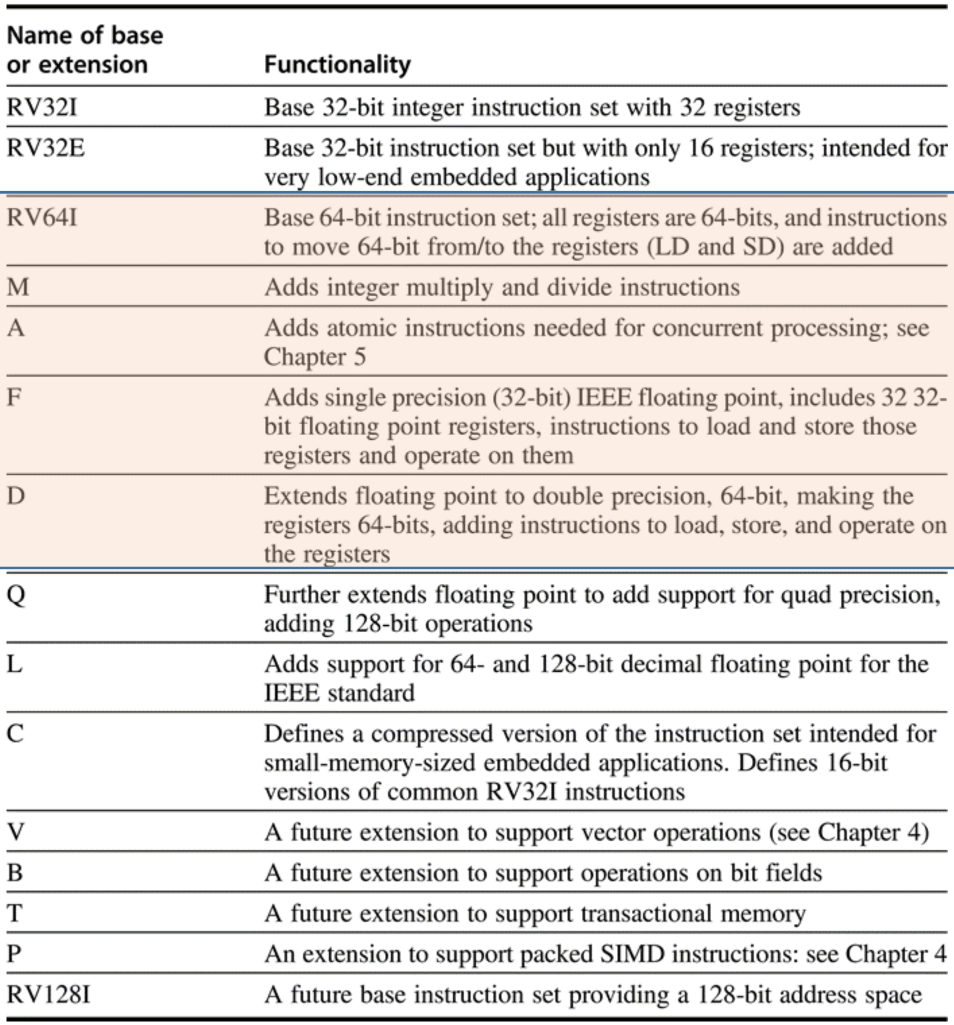


Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store 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

The RISC-V instruction layout. There are two variations on these formats, called the SB and UJ formats; they deal with a slightly different treatment for immediate fields.

H&P CA:A QA (6th. Ed.)

RISC-V Instruction Set Extensions



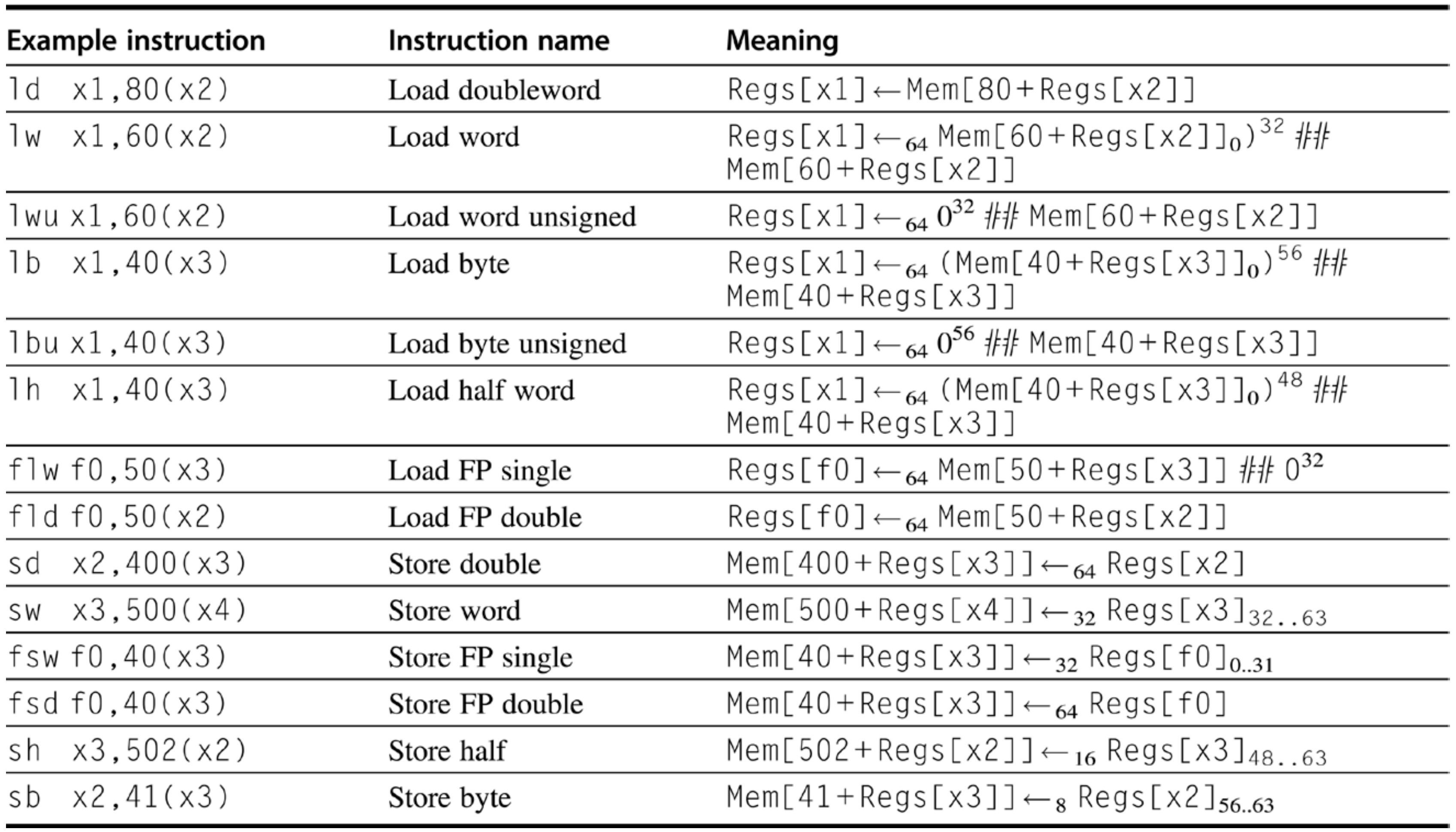
Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit 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

RV64IMAFD ≡ RV64G

RISC-V has three base instructions sets (and a reserved spot for a future fourth); all the extensions extend one of the base instruction sets. An instruction set is thus named by the base name followed by the extensions. For example**, RISC-V64IMAFD** refers to the base 64-bit instruction set with extensions M, A, F, and D. For consistency of naming and software, this combination is given the abbreviated name: **RV64G**, and we use RV64G through most of this text.

H&P CA:A QA (6th. Ed.)

RISC-V Load and Store Instructions



Example instruction	Instruction name	Meaning
ld x1,80(x2)	Load doubleword	Regs[x1] $\leftarrow$ Mem[80+Regs[x2]]
lw x1,60(x2)	Load word	Regs[x1] $\leftarrow_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
lwu x1,60(x2)	Load word unsigned	Regs[x1] $\leftarrow_{64}$ 0^{32} ## Mem[60+Regs[x2]]
lb x1,40(x3)	Load byte	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
lbu x1,40(x3)	Load byte unsigned	Regs[x1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[x3]]
lh x1,40(x3)	Load half word	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
flw f0,50(x3)	Load FP single	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x3]] ## 0^{32}
fld f0,50(x2)	Load FP double	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x2]]
sd x2,400(x3)	Store double	Mem[400+Regs[x3]] $\leftarrow_{64}$ Regs[x2]
sw x3,500(x4)	Store word	Mem[500+Regs[x4]] $\leftarrow_{32}$ Regs[x3]$_{32..63}$
fsw f0,40(x3)	Store FP single	Mem[40+Regs[x3]] $\leftarrow_{32}$ Regs[f0]$_{0..31}$
fsd f0,40(x3)	Store FP double	Mem[40+Regs[x3]] $\leftarrow_{64}$ Regs[f0]
sh x3,502(x2)	Store half	Mem[502+Regs[x2]] $\leftarrow_{16}$ Regs[x3]$_{48..63}$
sb x2,41(x3)	Store byte	Mem[41+Regs[x3]] $\leftarrow_{8}$ Regs[x2]$_{56..63}$

Notation is confusing: uses sign-bit to have index-0, most significant byte to have index 56..63 etc.!

The load and store instructions in RISC-V. Loads shorter than 64 bits are available in both sign-extended and zero-extended forms. All memory references use a single addressing mode. Of course, both loads and stores are available for all the data types shown. Because RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

H&P CA:A QA (6th. Ed.)

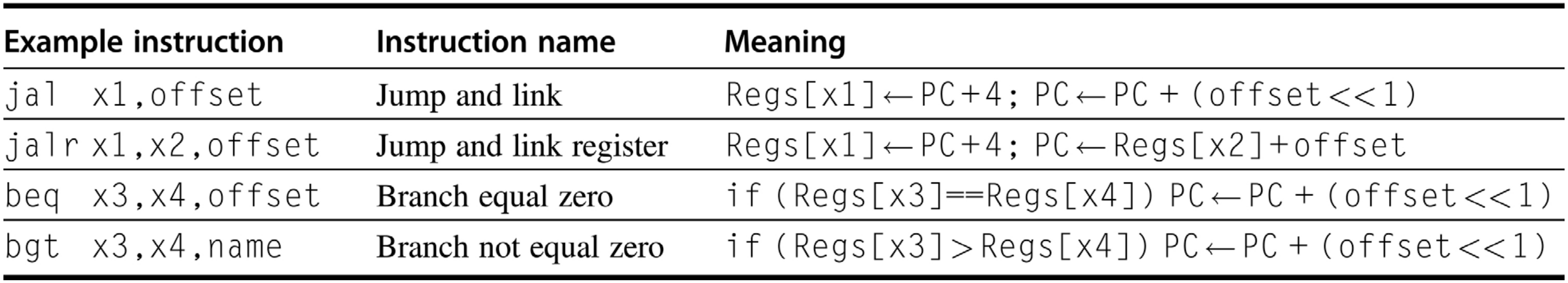
RISC-V Basic (integer) ALU Instructions Examples

Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	Regs[x1]←Regs[x2]+Regs[x3]
`addi x1,x2,3`	Add immediate unsigned	Regs[x1]←Regs[x2]+3
`lui x1,42`	Load upper immediate	Regs[x1]← s^{32} ##42##0^{12}
`sll x1,x2,5`	Shift left logical	Regs[x1]←Regs[x2]<<5
`slt x1,x2,x3`	Set less than	if (Regs[x2]<Regs[x3]) Regs[x1]←1 else Regs[x1]←0

Note: the 64-bit instructions often sign extends the results to occupy 64 bits
Logic operations: **and, or, xor, andi, ori, xori**

The basic ALU instructions in RISC-V are available both with register-register operands and with one immediate operand. LUI uses the U-format that employs the rs1 field as part of the immediate, yielding a 20-bit immediate.

RISC-V Branch and Jump Instructions Examples



Example instruction	Instruction name	Meaning
jal x1,offset	Jump and link	Regs[x1]←PC+4; PC←PC + (offset<<1)
jalr x1,x2,offset	Jump and link register	Regs[x1]←PC+4; PC←Regs[x2]+offset
beq x3,x4,offset	Branch equal zero	if (Regs[x3]==Regs[x4]) PC←PC + (offset<<1)
bgt x3,x4,name	Branch not equal zero	if (Regs[x3]>Regs[x4]) PC←PC + (offset<<1)

Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

RISC-V Floating-point Instructions Examples

Instruction type/opcode	Instruction meaning
Floating point	*FP operations on DP and SP formats*
`fadd.d`, `fadd.s`	Add DP, SP numbers
`fsub.d`, `fsub.s`	Subtract DP, SP numbers
`fmul.d`, `fmul.s`	Multiply DP, SP floating point
`fmadd.d`, `fmadd.s`, `fnmadd.d`, `fnmadd.s`	Multiply-add DP, SP numbers; negative multiply-add DP, SP numbers
`fmsub.d`, `fmsub.s`, `fnmsub.d`, `fnmsub.s`	Multiply-sub DP, SP numbers; negative multiply-sub DP, SP numbers
`fdiv.d`, `fdiv.s`	Divide DP, SP floating point
`fsqrt.d`, `fsqrt.s`	Square root DP, SP floating point
`fmax.d`, `fmax.s`, `fmin.d`, `fmin.s`	Maximum and minimum DP, SP floating point
`fcvt._._`, `fcvt._._u`, `fcvt._u._`	Convert instructions: `FCVT.x.y` converts from type `x` to type `y`, where `x` and `y` are `L` (64-bit integer), `W` (32-bit integer), `D` (DP), or `S` (SP). Integers can be unsigned (`U`)
`feq._`, `flt._`,`fle._`	Floating-point compare between floating-point registers and record the Boolean result in integer register; “__”=`S` for single-precision, `D` for double-precision
`fclass.d`, `fclass.s`	Writes to integer register a 10-bit mask that indicates the class of the floating-point number ($-\infty$, $+\infty$, -0, $+0$, NaN, …)
`fsgnj._`, `fsgnjn._`, `fsgnjx._`	Sign-injection instructions that changes only the sign bit: copy sign bit from other source, the oppositive of sign bit of other source, XOR of the 2 sign bits

H&P CA:A QA (6th. Ed.)

Thank you