IEEE 754 Floating Point Arithmetic

Floating-Point Numbers

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$	Low precision and/or range
Rational	$\pm p/q$	Difficult arithmetic
Floating-point	$\pm s \times b^e$	Most common scheme
Logarithmic	$\pm \log_b x$	Limiting case of floating-point

Fixed-point numbers

$x = (0000\ 0000.0000\ 1001)_{two}$ Small number
$y = (1001\ 0000.0000\ 0000)_{two}$ Large number

Square of neither number representable

Floating-point numbers

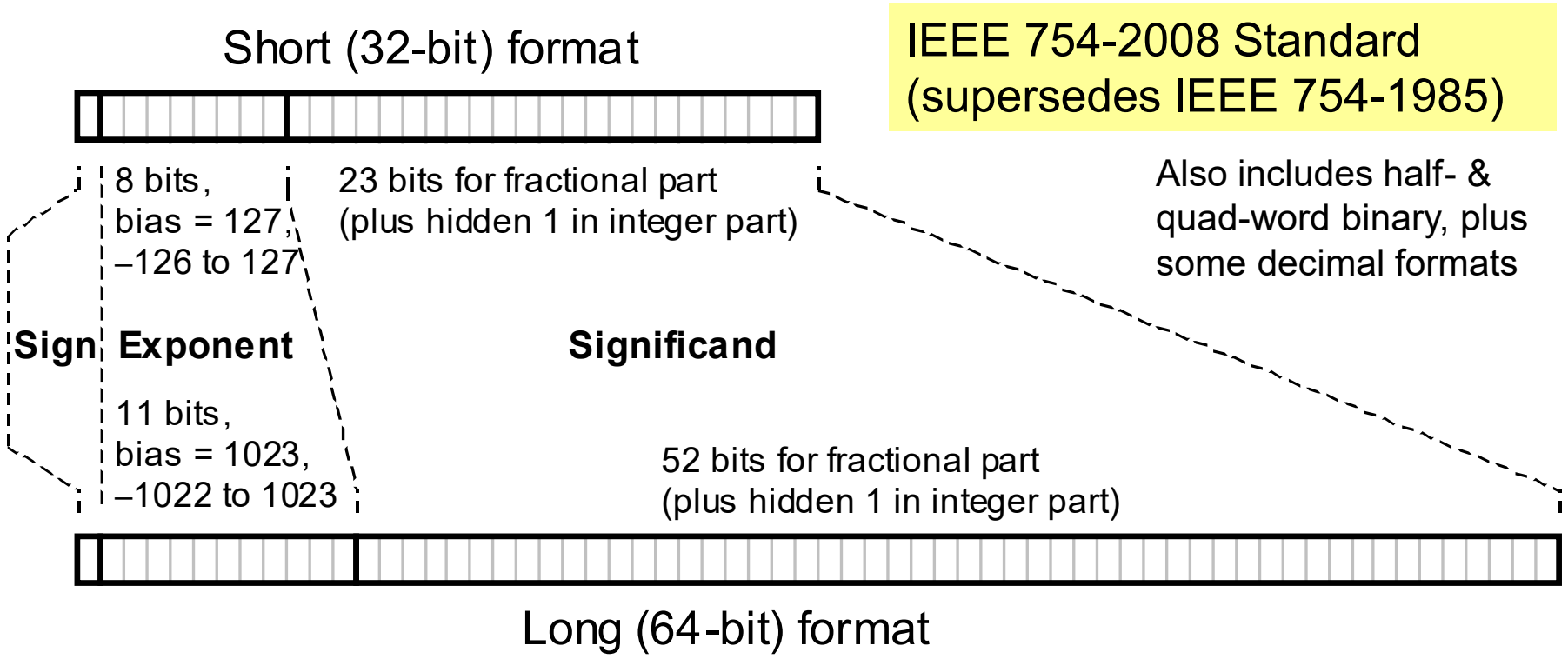
$x = \pm s \times b^e$ or $\pm \text{significand} \times \text{base}^{\text{exponent}}$

$x = 1.001 \times 2^{-5}$
$y = 1.001 \times 2^{+7}$

A floating-point number comes with two signs:

Number sign, usually appears as a separate bit
Exponent sign, usually embedded in the biased exponent

The IEEE Floating-Point Standard



IEEE 754-2008 Standard
(supersedes IEEE 754-1985)

Also includes half- &
quad-word binary, plus
some decimal formats

IEEE 754 standard floating-point number representation formats.

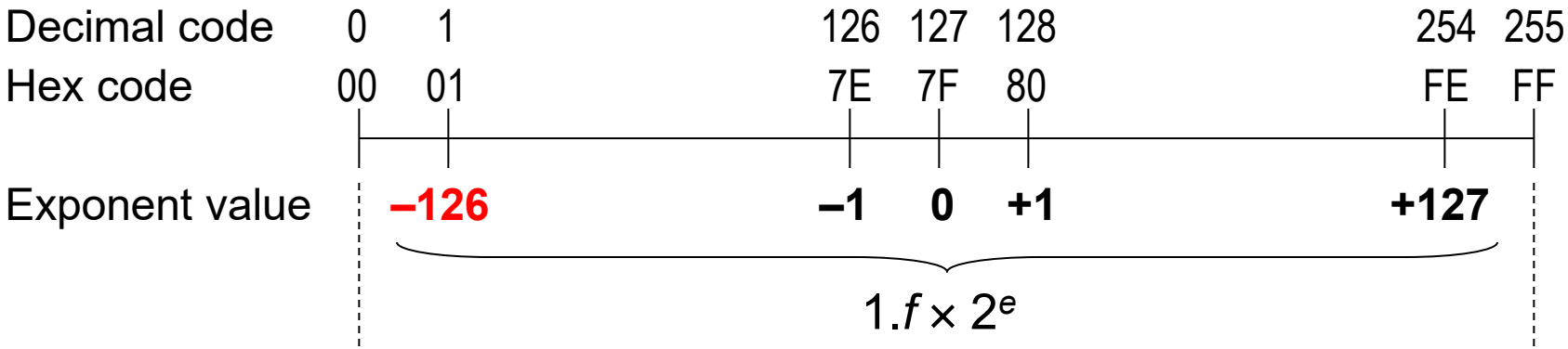
Overview of IEEE 754-2008 Standard Formats

Some features of the IEEE 754-2008 standard floating-point number representation formats

Feature	Single Precision(float32)	Double Precision (float64)
Word width (bits)	32	64
Significand bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (± 0)	$e + bias = 0, f = 0$	$e + bias = 0, f = 0$
Denormal	$e + bias = 0, f \neq 0$ represents $\pm 0.f \times 2^{-126}$	$e + bias = 0, f \neq 0$ represents $\pm 0.f \times 2^{-1022}$
Infinity ($\pm\infty$)	$e + bias = 255, f = 0$	$e + bias = 2047, f = 0$
Not-a-number (NaN)	$e + bias = 255, f \neq 0$	$e + bias = 2047, f \neq 0$
Ordinary number	$e + bias \in [1, 254]$ $e \in [-126, 127]$ represents $1.f \times 2^e$	$e + bias \in [1, 2046]$ $e \in [-1022, 1023]$ represents $1.f \times 2^e$
min	$2^{-126} \cong 1.2 \times 10^{-38}$	$2^{-1022} \cong 2.2 \times 10^{-308}$
max	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

Exponent Encoding

Exponent encoding in 8 bits for the single/short (32-bit) IEEE 754 format



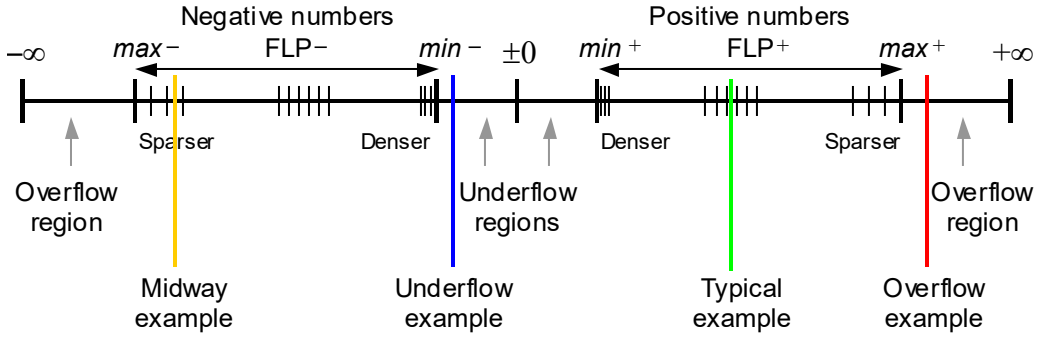
$f = 0$: Reserved for Representation of ±0

$f \neq 0$: Representation of subnormals, $0.f \times 2^{-126}$

$f = 0$: Representation of ±∞

$f \neq 0$: Representation of NaNs

Exponent encoding in 11 bits for the double/long (64-bit) format is similar



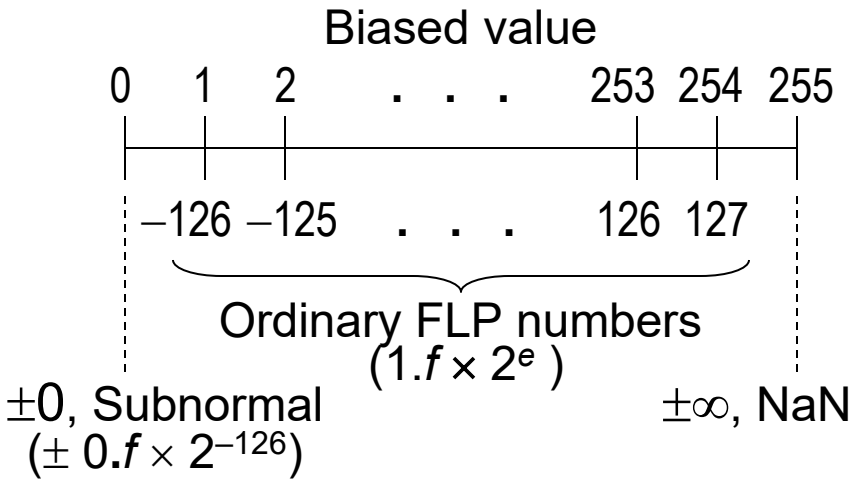
Special Operands and Subnormals

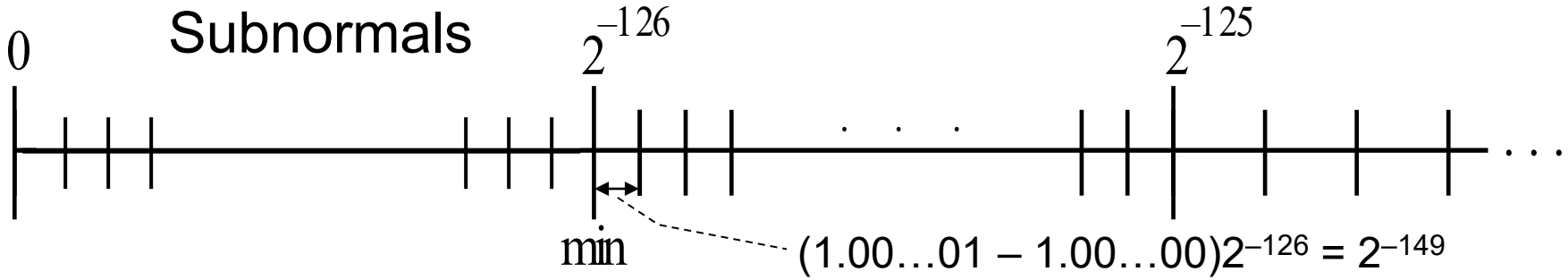
Operations on special operands:

Ordinary number ÷ (+∞) = ±0

(+∞) × Ordinary number = ±∞

NaN + Ordinary number = NaN





Subnormals in the IEEE single-precision format.

Exceptions in Floating-Point Arithmetic

Divide by zero

Overflow

Underflow

Invalid operation: examples include

Addition	$(+\infty) + (-\infty)$
Multiplication	$0 \times \infty$
Division	$0/0$ or ∞/∞
Square-rooting	operand < 0

Produce NaN as their results

It is important to realize that floating point arithmetic on a computer (or calculator) is always an approximation.

Addition

- To add two floating point numbers, the exponents must be equal. If they are not already equal, then they must be made equal by shifting the significand of the number with the smaller exponent.
- E.g., 10.375 + 6.34375 = 16.71875

$$1.0100110 \times 2^3$$
$$+\ 1.1001011 \times 2^2$$

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 0.1100110 \times 2^3 \\ \hline & 10.0001100 \times 2^3 \end{array}$$

16.75

Subtraction

As an example, consider $16.75 - 15.9375 = 0.8125$:

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.1111111 \times 2^3 \\ \hline \end{array}$$

Shifting 1.1111111×2^3 gives (rounding up) 1.0000000×2^4

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.0000000 \times 2^4 \\ \hline & 0.0000110 \times 2^4 \end{array}$$

$0.0000110 \times 2^4 = 0.11_2 = 0.75$ which is not exactly correct.

Multiplication and division

- For multiplication, the significands are multiplied and the exponents are added. Consider $10.375 \times 2.5 = 25.9375$:

$$\begin{array}{rl} & 1.0100110 \times 2^3 \\ \times & 1.0100000 \times 2^1 \\ \hline & 10100110 \\ + & 10100110 \\ \hline & 1.10011111000000 \times 2^4 \end{array}$$

$$1.1010000 \times 2^4 = 11010.000_2 = 26$$

- Division is more complicated, but has similar problems with round off errors.

Floating-Point Adders/Subtractors

Floating-Point Addition Algorithm

Assume $e1 \geq e2$; *alignment shift* (*preshift*) is needed if $e1 > e2$

$$(\pm s1 \times b^{e1}) + (\pm s2 \times b^{e2}) = (\pm s1 \times b^{e1}) + (\pm s2 / b^{e1-e2}) \times b^{e1}$$
$$= (\pm s1 \pm s2 / b^{e1-e2}) \times b^{e1} = \pm s \times b^{e}$$

Example:

Numbers to be added:

$x = 2^5 \times 1.00101101$
$y = 2^1 \times 1.11101101$ ← Operand with smaller exponent to be preshifted

Operands after alignment shift:

$x = 2^5 \times 1.00101101$
$y = 2^5 \times 0.000111101101$

Result of addition:

$s = 2^5 \times 1.01001011\underbrace{1101}$ ← Extra bits to be rounded off
$s = 2^5 \times 1.01001100$ ← Rounded sum

Normalisation

- **Same signs:**
- Possible 1-position normalizing right shift
- **Different signs:**
- Left shift, possibly
- by many positions
- **Overflow/underflow** during addition or normalization

FLP Addition Hardware

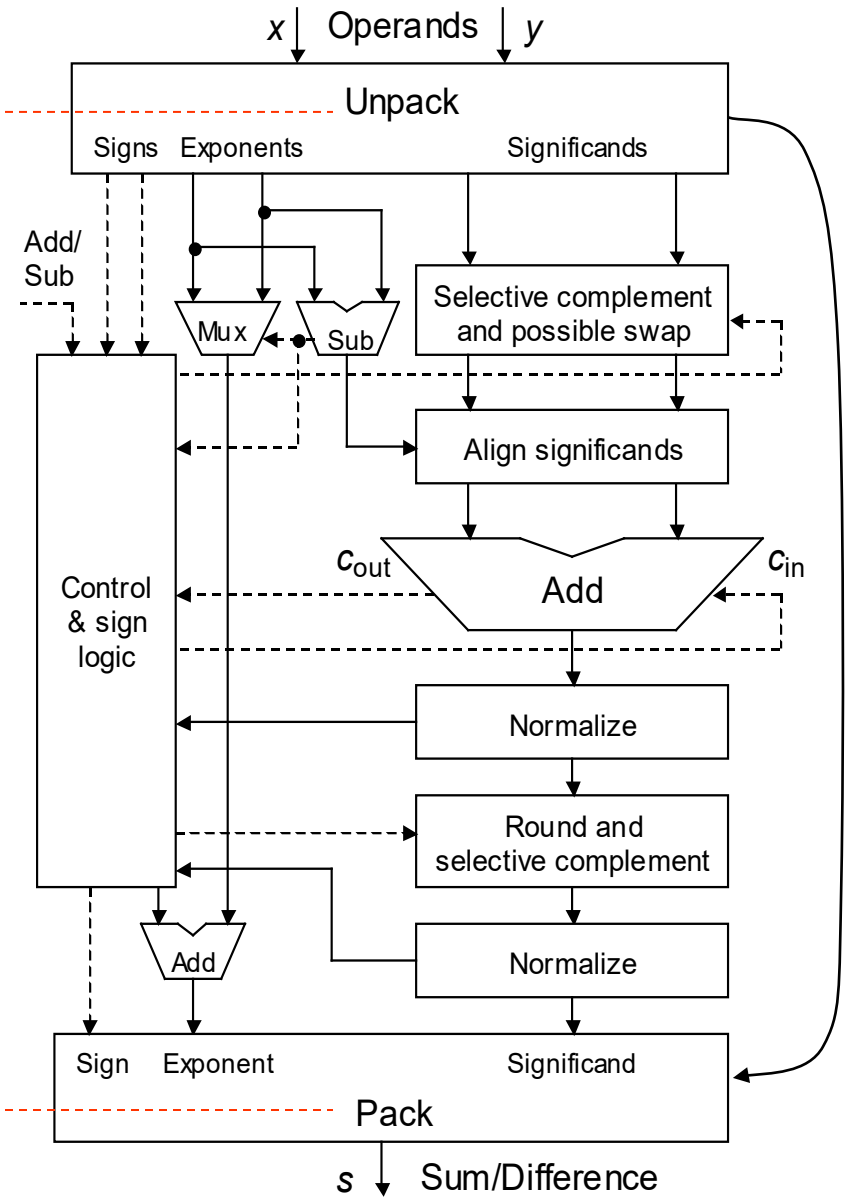
Isolate the sign, exponent, significand
Reinstate the hidden 1
Convert operands to internal format
Identify special operands, exceptions

Block diagram of a floating-point adder/subtractor.

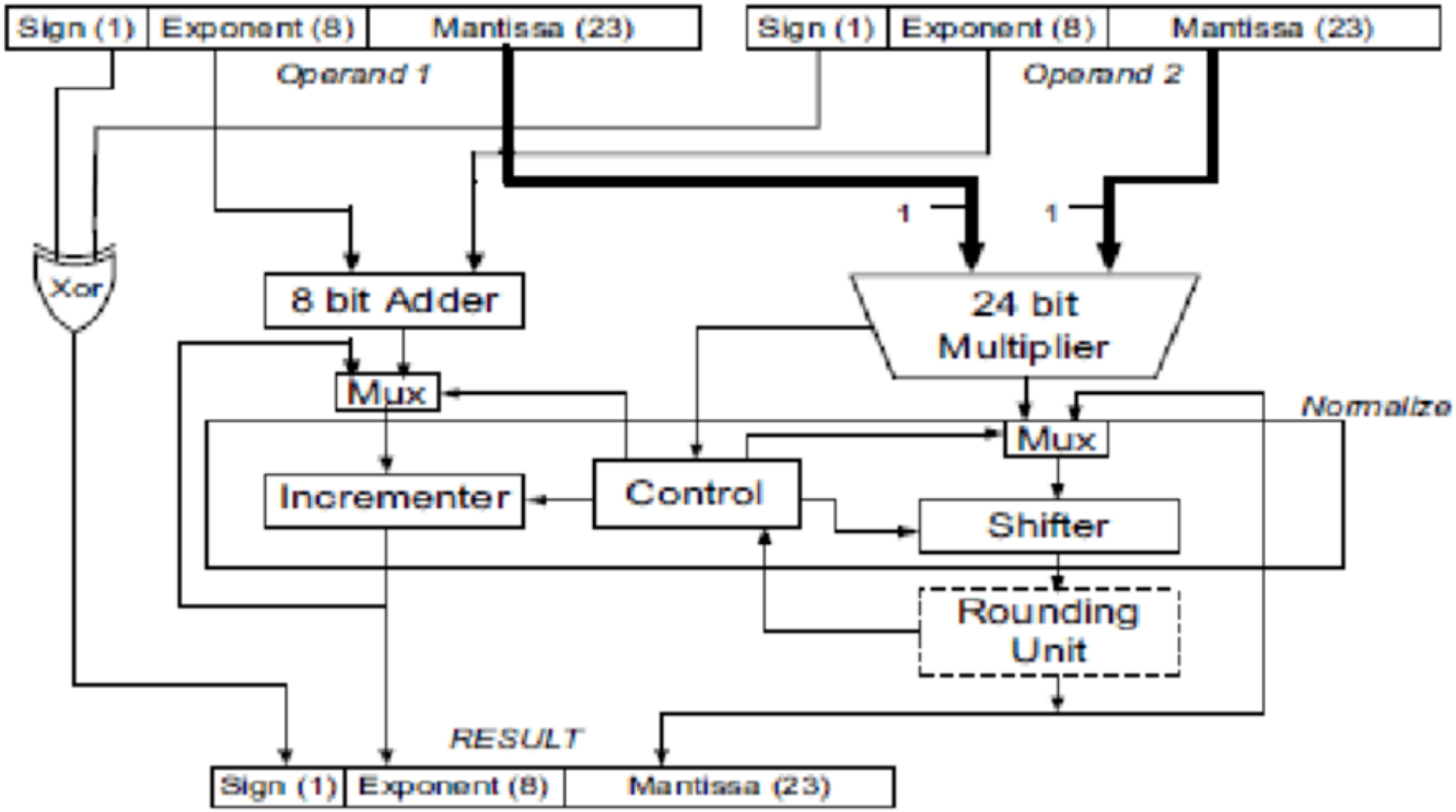
Other key parts of the adder:
Significand aligner (preshifter)
Result normalizer (postshifter), including leading 0s detector/predictor
Rounding unit
Sign logic

Converting internal to external representation, if required, must be done at the rounding stage

Combine sign, exponent, significand
Hide (remove) the leading 1
Identify special outcomes, exceptions



Architecture of 32-bit Single Precision Floating-Point Multiplier



RISC-V ISA

What is RISC-V

- RISC-V (pronounced "risk-five") is an Instruction Set Architecture (ISA) standard
  - An ISA is part of the abstract model of a computer that defines how the CPU is controlled by the software.
    - Interface between the hardware and the software
    - Specifies
      - What processor can do
      - How it gets done
  - *ISA does NOT specify implementation architecture/design/technology*
  - There was RISC-I, II, III, IV before
- Most ISAs: X86, ARM, PowerPC, MIPS, SPARC
  - Commercially protected by patents
  - Prevents open research efforts to reproduce systems
- RISC-V is open (royalty free)
  - Permits any person or group to construct compatible computers
  - Use associated software
- Originated in 2010 by researchers at UC Berkeley
  - Waterman, Lee, Asanović, David Patterson and students
- Current version
  - Unprivileged (user mode) ISA 20191213
  - Privileged (kernel mode) ISA 20211203

https://riscv.org/
https://en.wikipedia.org/wiki/RISC-V

Specifications and Software From riscv.org and github.com/riscv

- Specification from RISC-V website
  - https://riscv.org/specifications/

- RISC-V software includes
  - GNU Compiler Collection (GCC) toolchain (with GDB, the debugger)
    - https://github.com/riscv/riscv-tools
  - LLVM toolchain
  - A simulator ("Spike")
    - https://github.com/riscv/riscv-isa-sim
  - Standard simulator QEMU
    - https://github.com/riscv/riscv-qemu
- Operating systems support exists for Linux
  - https://github.com/riscv/riscv-linux
- A JavaScript ISA simulator to run a RISC-V Linux system on a web browser
  - https://github.com/riscv/riscv-angel
- Venus simulator
  - https://venus.cs61c.org/

Goals in Defining RISC-V ISA

- Completely open ISA freely available to academia and industry – royalty free
- Suitable for direct practical hardware implementation
- Decoupled from
  - particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of- order)
  - implementation technology (e.g., full-custom, ASIC, FPGA)
- RISC-V ISA includes
  - **Small base integer ISA**
    - usable by itself as a base for customized accelerators or educational purposes
  - **Optional standard extensions**
    - to support general-purpose software development
  - **Optional customisation extensions**
    - to support specific application needs
- Support for the revised 2008 IEEE-754 floating-point standard

RISC-V Implementations

- For RISC-V implementation, the UCB created **Chisel**, an open-source hardware construction language that is a specialized dialect of Scala.
  - **Chisel: Constructing Hardware In a Scala Embedded Language**
  - https://chisel.eecs.berkeley.edu/
  - https://github.com/ucb-bar/riscv-mini
- In-order Rocket core and chip generator
  - https://github.com/freechipsproject/rocket-chip
- UCB Sodor cores for education (single cycle, and 1-5 stages pipeline)
  - https://github.com/ucb-bar/riscv-sodor

RISC-V Implementations

- List of dev boards available at https://riscv.org/risc-v-developer-boards/
  - SiFive HiFive Unleashed
    - First Linux RISC-V Board
    - https://www.sifive.com/
    - https://github.com/sifive/freedom
- IIT-Madras is developing six RISC-V open-source CPU designs (SHAKTI) for six distinct usages
  - https://shaktiproject.bitbucket.io/index.html
- CDAC Trivandrum is developing RISC V Vega processor

RISC V Simulator for Assignment

- Use Venus RISC V simulator
  - VSCode plugin – preferable
    - Install VSCode – available for Windows, MacOS, Ubuntu
    - Install Venus RISC V plugin from VSCode Marketplace
      - https://marketplace.visualstudio.com/items?itemName=hm.riscv-venus
  - Web interface
    - https://venus.cs61c.org/
    - https://cs61c.org/su24/resources/venus-reference/

RISC-V ISA Principles

- Keep ISA very simple and extendable
  - Simplify hardware
  - Facilitate customisation
- Separated into multiple specifications
  - User-Level ISA spec (compute instructions)
  - Compressed ISA spec (16-bit instructions) Variable length instruction encoding
  - Privileged ISA spec (supervisor-mode instructions)
  - Others …
- ISA support is given by RV + word-width + extensions supported
  - e.g. RV32I means 32-bit RISC-V with support for the I(nteger) instruction set

```
Extensions
I: Integer Arithmetic & Logical Operations
M: Multiplication & Division
A: Atomics - LR/SC & fetch-and-op
F: Floating point (32-bit)
D: FP Double (64-bit)
Q: FP Quad (128-bit)
Zicsr: Control and status register support
Zifencei: Load/store fence
          (typically for multi-core)
C: Compressed instructions(16-bit)
J: Interpreted or JIT compiled languages
support
```

RISC-V ISA

- Both 32-bit and 64-bit address space variants
  - RV32 and RV64
- Easy to subset/extend for education/research
  - RV32IM, RV32IMA, RV32IMAFD, RV32G
- SPEC on the website
  - www.riscv.org

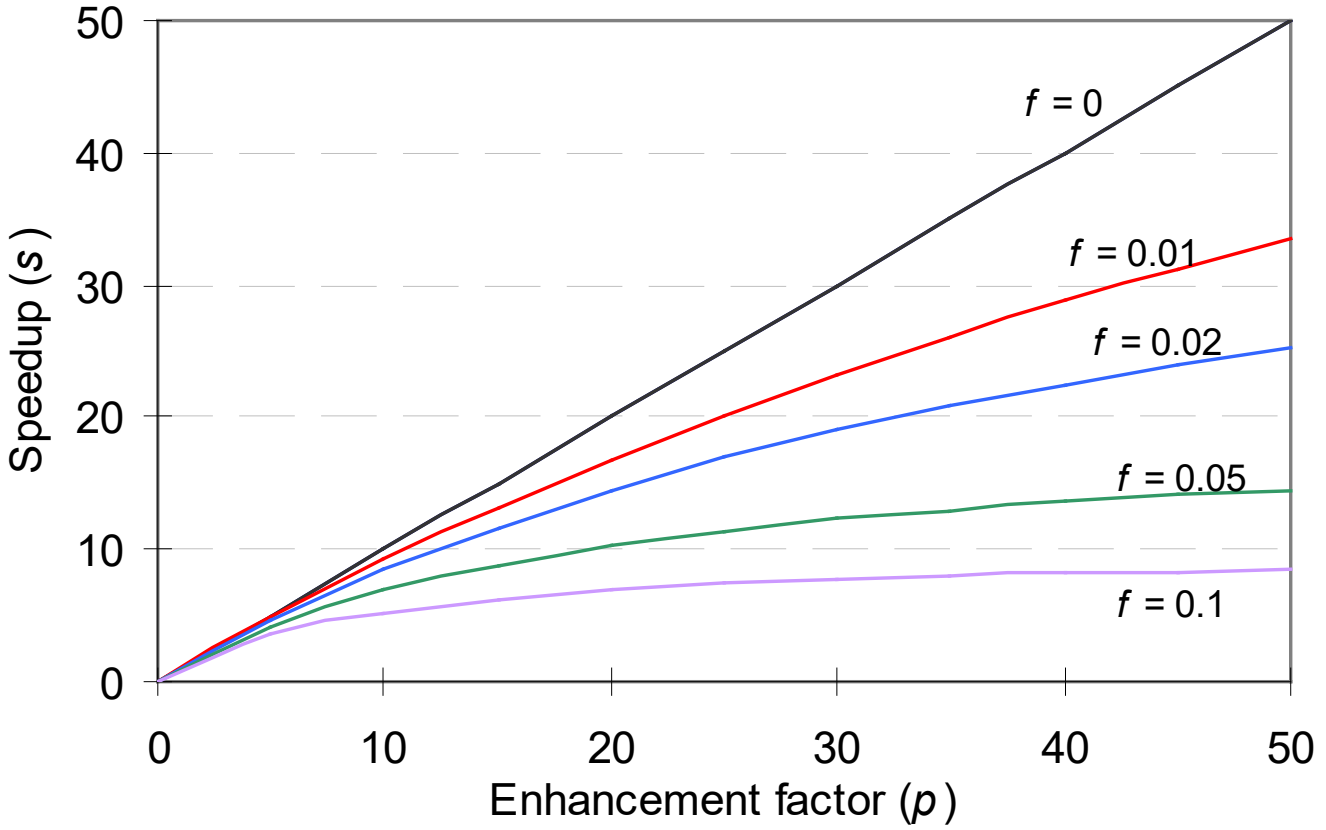
Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing;
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
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:
RV128I	A future base instruction set providing a 128-bit address space

RISC-V Dynamic Instruction Mix for SPECint2006

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

RISC-V dynamic instruction mix for the SPECint2006 benchmark

Performance Enhancement: Amdahl's Law



f = fraction unaffected

p = speedup of the rest

$$s = \frac{1}{f + (1-f)/p}$$

$$\leq min(p, 1/f)$$

Amdahl's law: speedup achieved if a fraction f of a task is unaffected and the remaining 1 – f part runs p times as fast.

Amdahl's Law Used in Design

A processor spends 30% of its time on flp addition, 25% on flp mult, and 10% on flp division. Evaluate the following enhancements, each costing the same to implement:

a. Redesign of the flp adder to make it twice as fast.
b. Redesign of the flp multiplier to make it three times as fast.
c. Redesign the flp divider to make it 10 times as fast.

Solution

a. Adder redesign speedup = 1 / [0.7 + 0.3 / 2] = 1.18
b. Multiplier redesign speedup = 1 / [0.75 + 0.25 / 3] = 1.20
c. Divider redesign speedup = 1 / [0.9 + 0.1 / 10] = 1.10

What if both the adder and the multiplier are redesigned?

RISC ISA Principles : Summary

Goal : Minimalist instruction set – Simplifies decoding hardware

- **Use general-purpose registers with a load-store architecture**
- **Support following addressing modes**
  - displacement (with an address offset size of 12 to 16 bits)
  - immediate (size 8 to 16 bits)
  - register indirect
- **Support following data sizes and types**
  - 8, 16, 32, 64-bit integers
  - 32 / 64-bit IEEE 754 floating-point numbers
- **Support following simple instructions, majority of typical application instructions**
  - load
  - store
  - add, subtract
  - move register- register
  - shift
  - compare equal
  - compare not equal
  - compare less
  - branch (with a PC- relative address at least 8 bits long),
  - jump
  - call
  - return
- **Use simple instruction encoding**
  - Fixed length instruction encoding for performance,
  - Variable length instruction encoding for reduced code size
- **Provide at least 16 general-purpose registers**
  - Often use separate floating-point registers.
  - Increase the total number of registers without disturbing
    - instruction coding format
    - speed of the general-purpose register file
- **Ensure all addressing modes apply to all data transfer instructions**

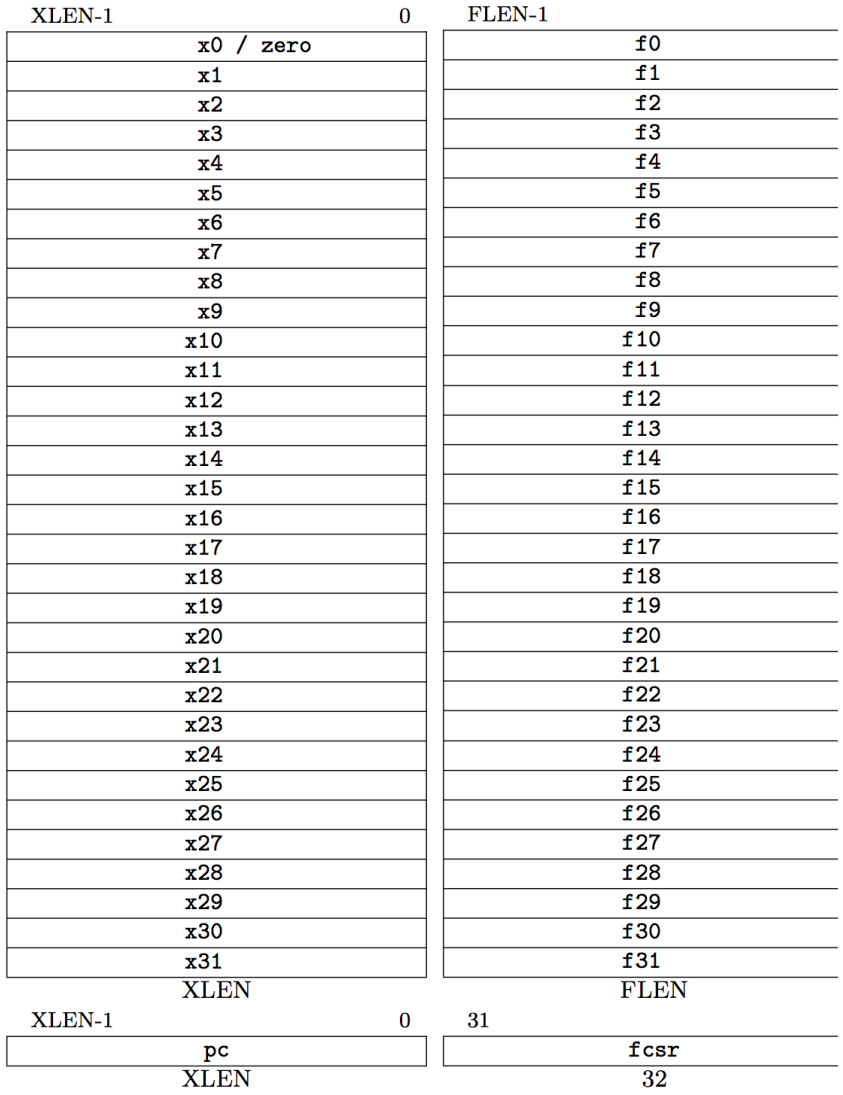
User Level (unprivileged) ISA

- Defines the normal instructions needed for computation
  - A mandatory Base integer ISA
    - I: Integer instructions:
      - ALU
      - Branches/jumps
      - Loads/stores
  - Standard Extensions
    - M: Integer Multiplication and Division
    - A: Atomic Instructions
    - F: Single-Precision Floating-Point
    - D: Double-Precision Floating-Point
    - *C: Compressed Instructions (16 bit)*

    - G = IMAFD: Integer base + four standard extensions
  - Optional extensions

RV32/64 Processor State

- Program counter (**pc**)
- 32 32/64-bit integer registers (**x0-x31**)
  - x0 always contains a 0
  - x1 to hold the return address on a call.
- 32 floating-point (FP) registers (**f0-f31**)
  - Each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)
- FP status register (**fsr**), used for FP rounding mode & exception reporting



RV64G In One Table

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

RISC-V Encoding Summary

Name	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-format	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-format	immed[11:0]		rs1	funct3	rd	opcode	Loads / immediate arithmetic
S-format	immed[11:5]	rs2	rs1	funct3	immed[4:0)	opcode	Stores
SB-format	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-format	immed[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-format	immed[31:12]				rd	opcode	Upper immediate format

ALU Instructions

Example instruction	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] ← 0^{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

Bit 31 – 12 of opcode

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

Load/Store Instructions

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

Sign extension

Zero padding

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. Both loads and stores are available for all the data types shown. Since RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

Control Flow Instructions

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.