

CHAPTER 3

Arithmetic for Computers

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University

Outline

× means not covered in this file

Chapter Goals

- To understand …
  - the representation of real numbers,
  - arithmetic algorithms,
  - hardware that follows these algorithms—and
  - the implications of all this for instruction sets

- To show how this knowledge to make arithmetic-intensive programs go much faster

Integer Addition

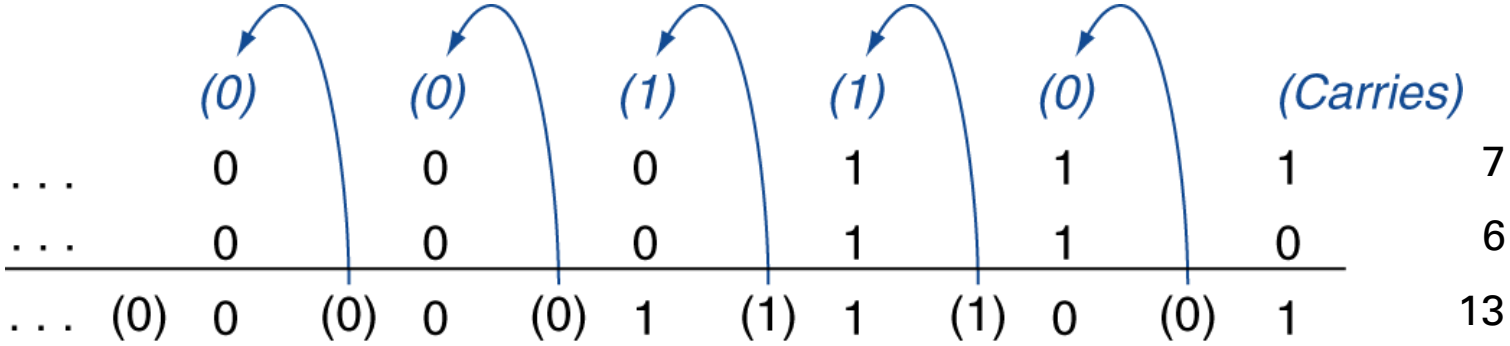


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

- Example of 7 + 6
- Overflow cannot occur
  - when adding operands with different signs, since the sum must be no larger than one of the operands (if the operands fit in 32 bits)
- Overflow occurs if the result is out of range
  - Adding +num and –num operands → no overflow
  - Adding two + num operands
    - Overflow if result *sign* is 1
  - Adding two –num operands
    - Overflow if result *sign* is 0

Integer Subtraction

- Concept: *negate* the second operand, and *add*
  - E.g., c-b = c +(-b)

- Example: 7 + (-6)
  - -6 uses two's compliment representation

+7: 0000 0000 … 0000 0111
–6: 1111 1111 … 1111 1010
+1: 0000 0000 … 0000 0001

- Overflow cannot occur
  - when the signs of the operands are *the same*
- Overflow occurs if the result is out of range
  - Subtracting two +num or two –num operands → no overflow
  - Subtracting + num from –num operand (e.g., 8 – (-2))
    - Overflow if result sign is 1 (<0)
  - Subtracting –num from + num operand
    - Overflow if result sign is 0 (>=0)

- The above shows how to detect overflow for two's complement numbers in a computer
- What about overflow with unsigned integers?
  - The compiler can easily check for unsigned overflow using a branch instruction
    - Addition has overflowed if the sum is less than either of the addends
    - Subtraction has overflowed if the difference is greater than the minuend

- What will happen when overflow occurs?
  → Saturating operations in multimedia apps.
  - E.g., clipping in audio, saturation in video
  - On overflow, result is largest representable value
  - Compare w/ 2's-complement modulo arithmetic

Basic Arithmetic Logic Unit (ALU)

- ALU is hardware that performs addition, subtraction, and usually logical operations, such as AND and OR
- Fig. A.5.1 shows the logical unit for AND/OR
- Fig. A.5.2 shows the 1-bit adder

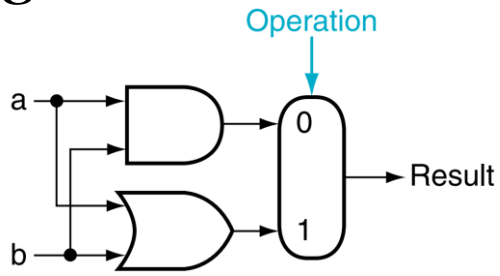


FIGURE A.5.1 The 1-bit logical unit for AND and OR.

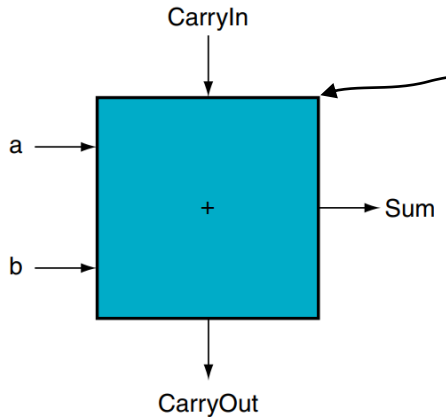


FIGURE A.5.2 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has three inputs and two outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

- Input: a, b, CarryIn
- Output: Sum, CarryOut
- The outputs against the inputs are listed in the truth table below
- The values in the table can be as of the equation:
  CarryOut = (b · CarryIn)+(a · CarryIn)+(a · b)+(a · b · CarryIn)

Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

FIGURE A.5.3 Input and output specification for a 1-bit adder

More on ALUs

- Fig. A.5.6 shows a 1-bit ALU derived by combining the adder with the earlier component in Fig. A.5.1
  - The Operation signal controls the action to be performed
- The full 32-bit ALU is created by connecting adjacent 1-bit ALUs as in Fig. A.5.7
- Please read through Section A.5 Constructing a basic arithmetic logic unit
  - E.g., see Fig. A.5.9 to know how the subtraction is done w/ the 1-bit ALU (Fig. A.5.10 for handling overflow)
  - See Fig. A.5.11 for a 32-bit ALU with the new 1-bit-ALU design

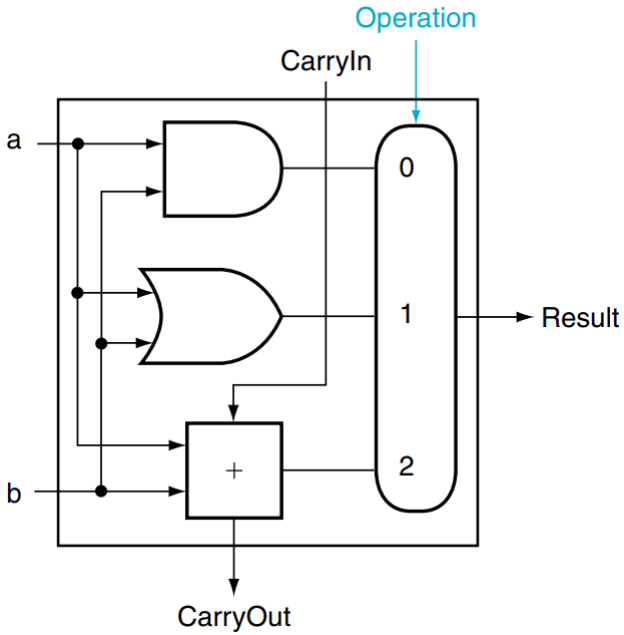


FIGURE A.5.6 A 1-bit ALU that performs AND, OR, and addition (see Figure A.5.5)

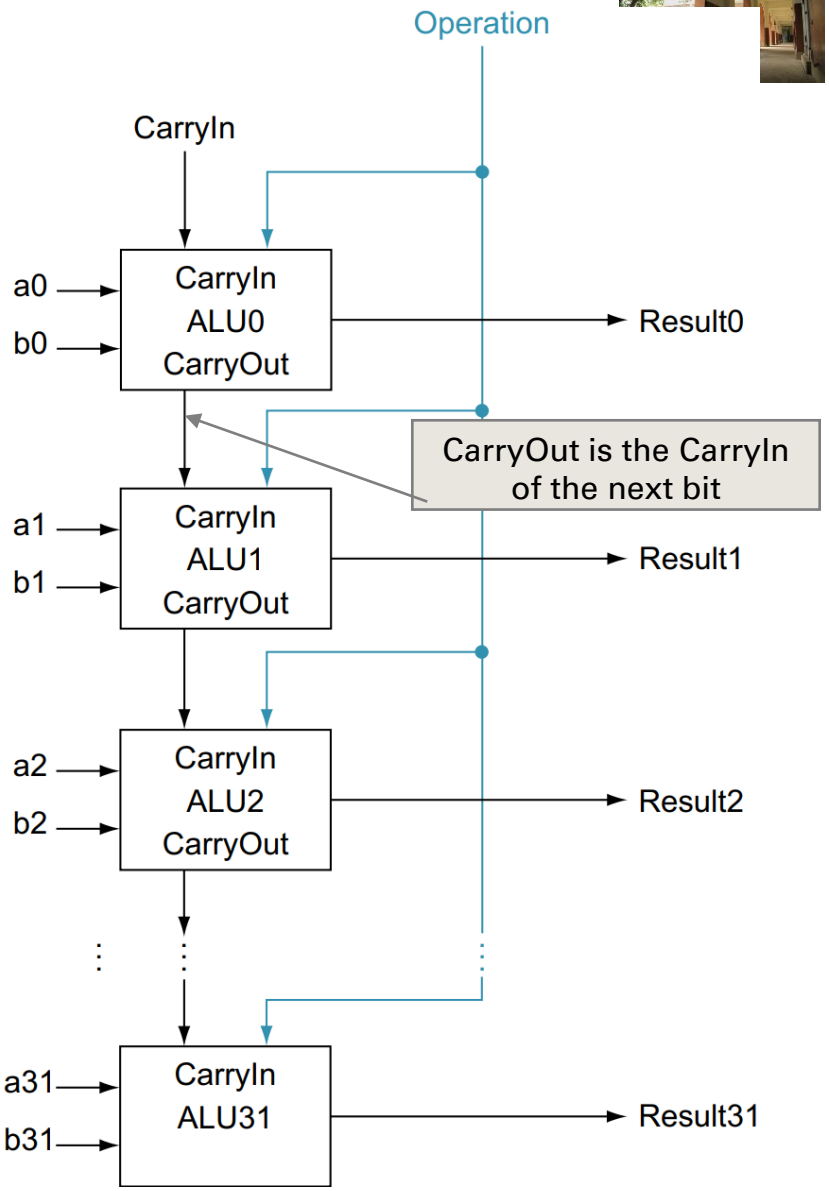
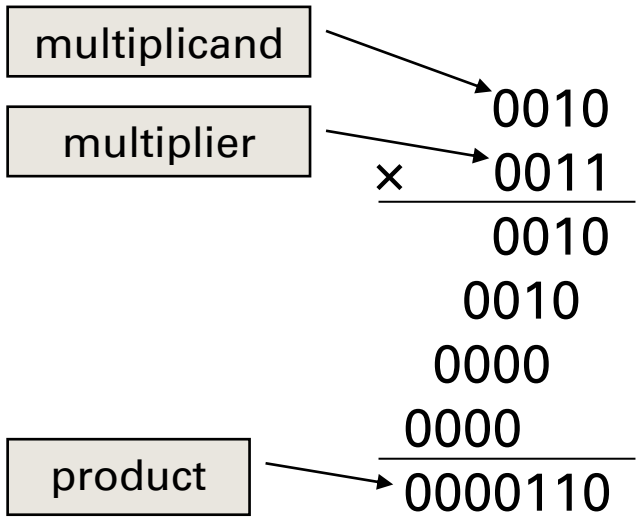


FIGURE A.5.7 A 32-bit ALU constructed from 32 1-bit ALUs. CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry

Multiplication

- Procedure for binary multiplication
  1. take the digits of the multiplier one at a time from right to left,
  2. multiply the multiplicand by the single digit of the multiplier, and
  3. shift the intermediate product one digit to the left of the earlier intermediate products

- The length of product
  - ➢is considerably larger than the number in either the multiplicand or the multiplier (→ overflow may occur)



Multiplication (32-bit Data)

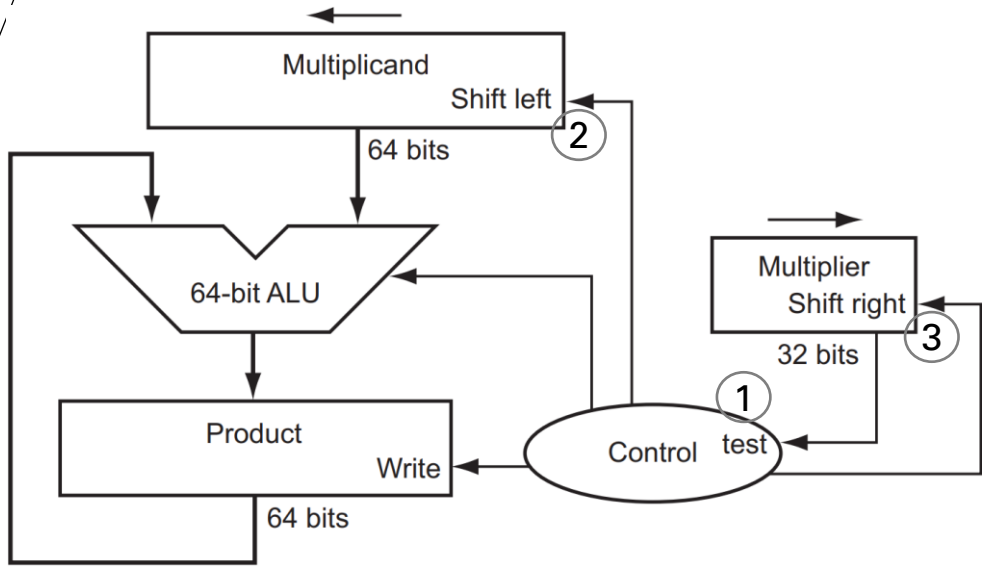


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix A describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the **product initialized to 0**. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

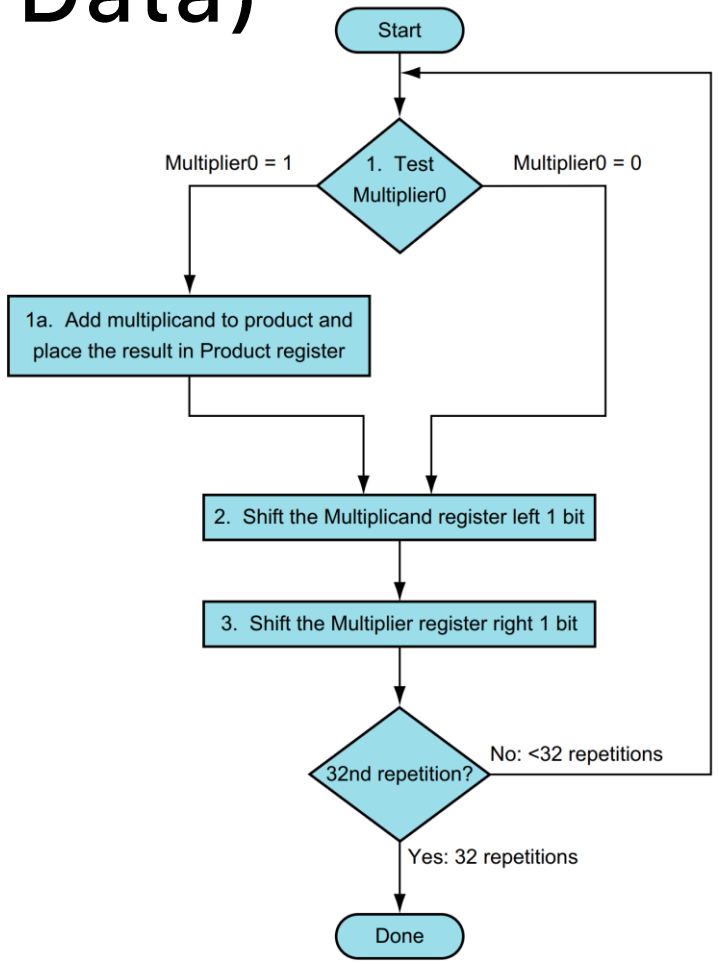


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times

- Three steps in Fig. 3.4 (the control flow of binary multiplication in the previous page)

1. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register
2. The left shift in step 2 has the effect of moving the intermediate operands to the left
3. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration

- These three steps are repeated 32 times to obtain the product

- The multiplier is in the 32-bit multiplier register
- It needs a 64-bit multiplicand register (needing to shift 32 times on a 32-bit number)
- The 64-bit product register is initialized to 0
- NOTE: You can refer to Fig. 3.6 for detailed information. Be aware that the following example is different from the contents listed in Fig. 3.6.

Multiplication (Example: Initial)

- Initial state of the example
- It has 4-bit multiplier, and 8-bit ALU, multiplicand and product

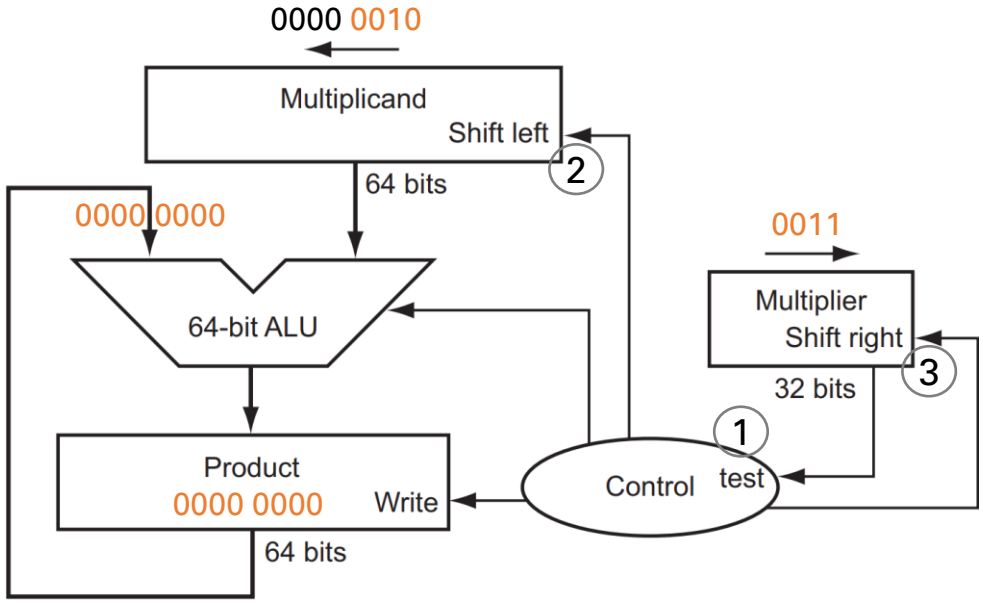


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix A describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

```
    0010
  × 0011
    0010
   0010
  0000
 000
0000110
```

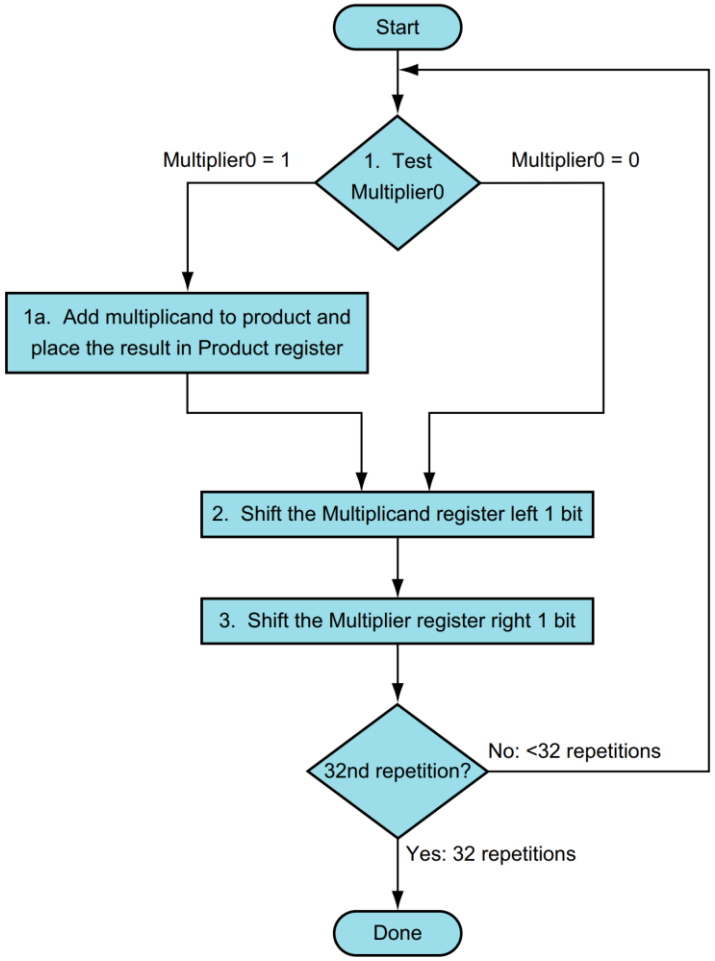


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times

Multiplication (Example: 1st iter)

- Step 1 of the example

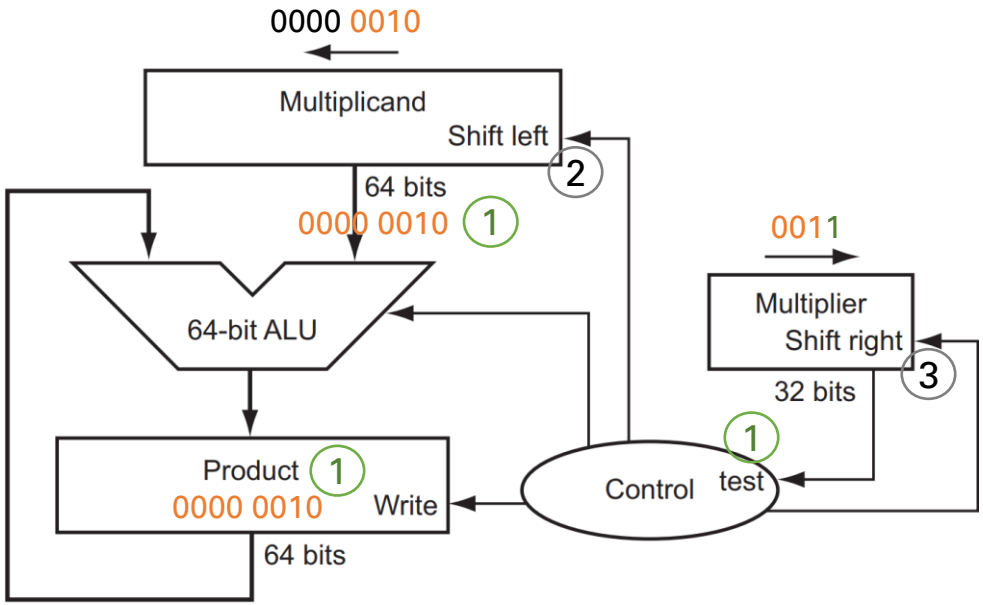


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix A describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

```
    0010
  × 0011
  ------
    0010
   0010
  0000
 000
 -------
 0000110
```

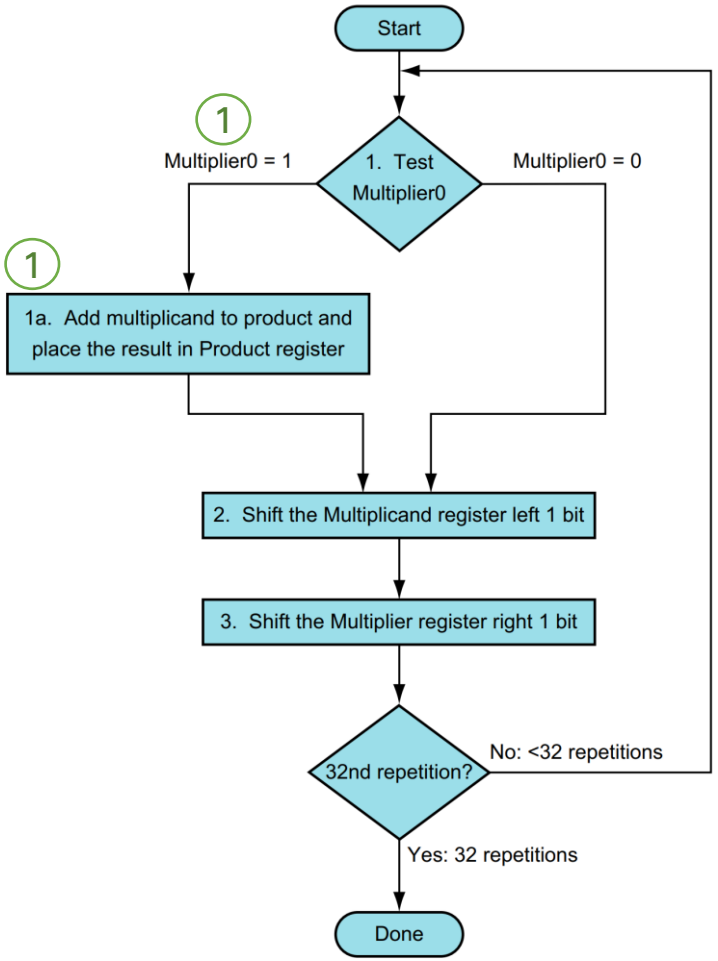
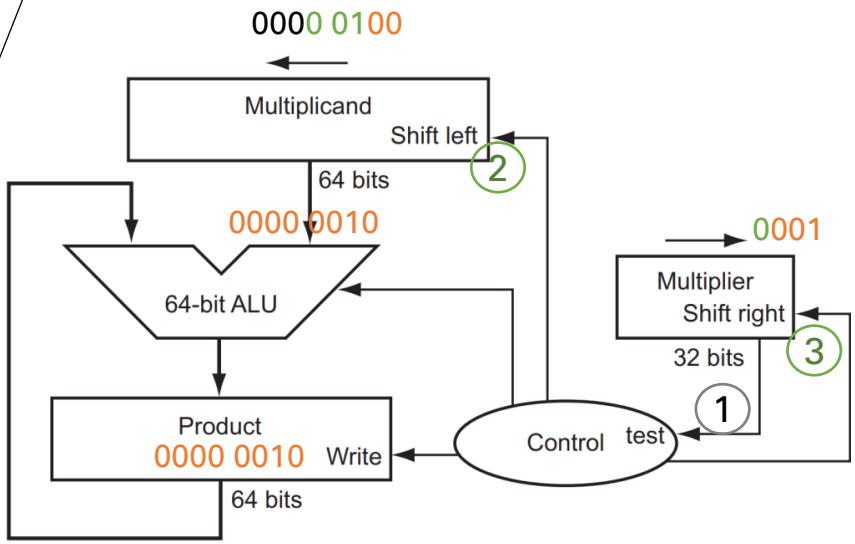


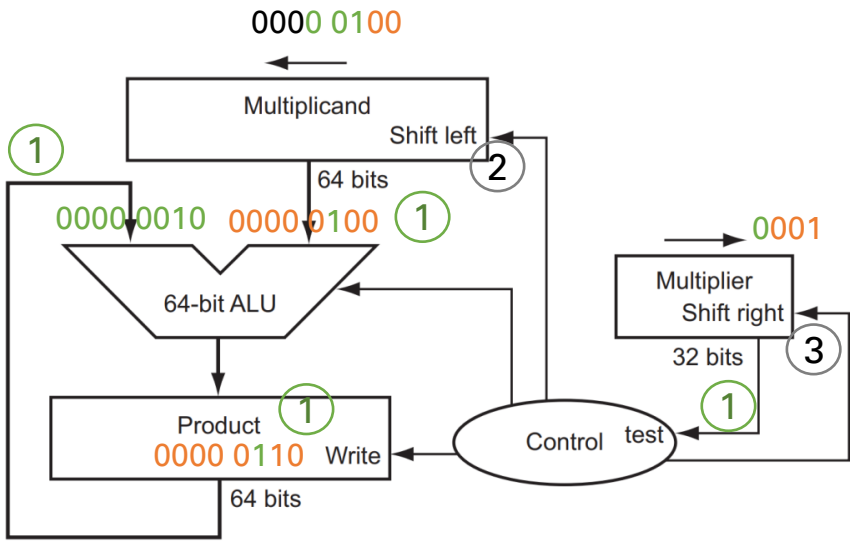
FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times

Multiplication (Example: 2nd and 3rd iter)

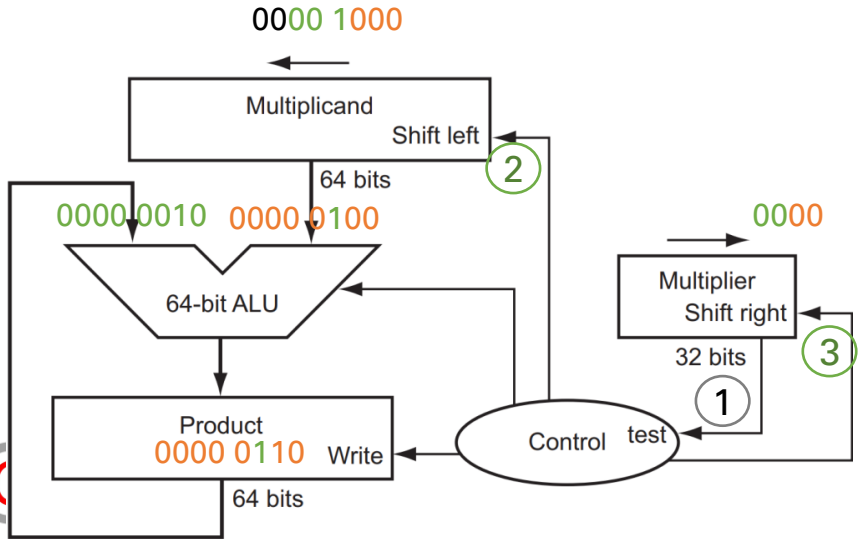
- Step 2 and 3



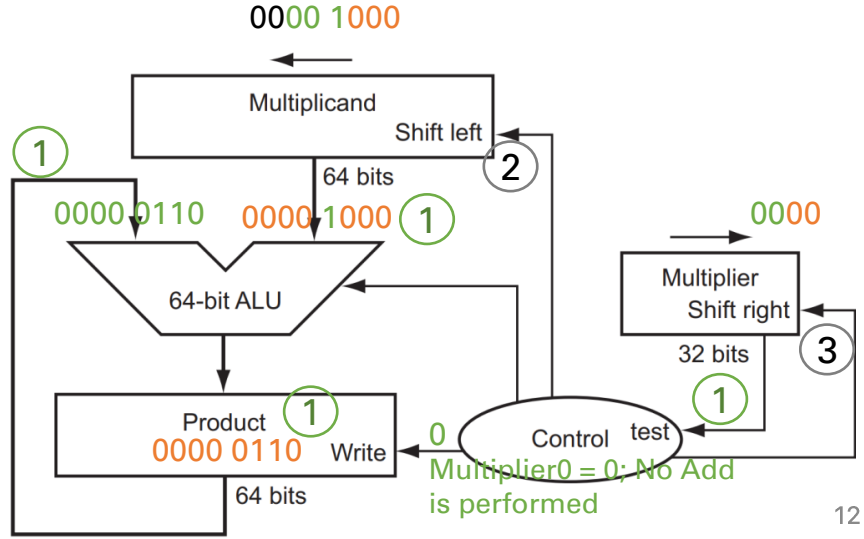
- Step 1 of the 2nd iteration



- Step 2 and 3 of the 2nd iteration

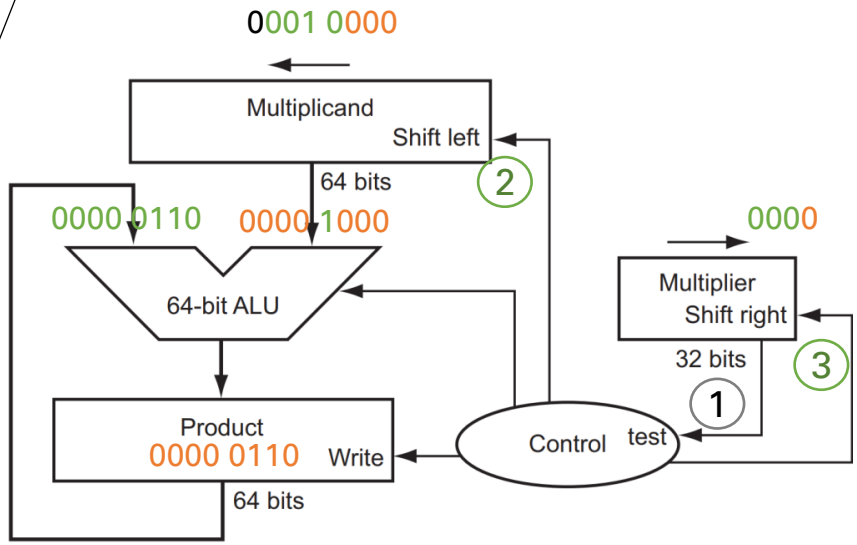


- Step 1 of the 3rd iteration

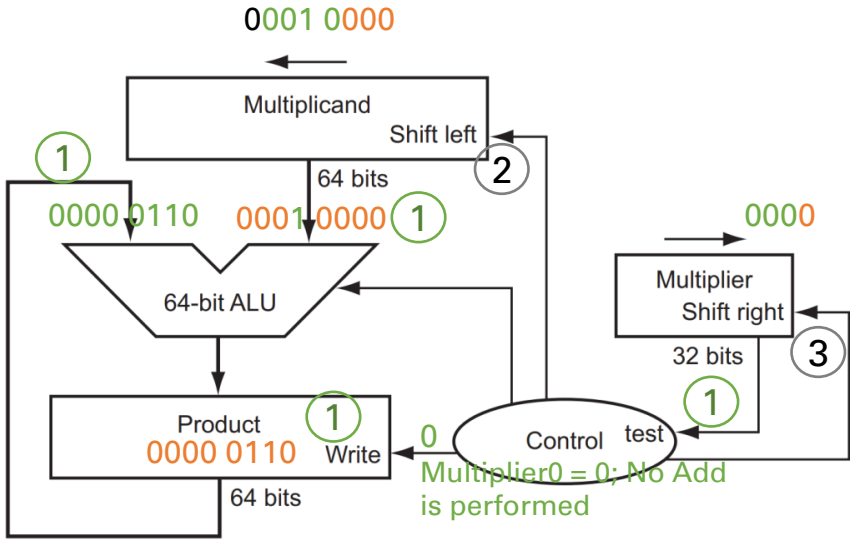


Multiplication (Example: 4th iter)

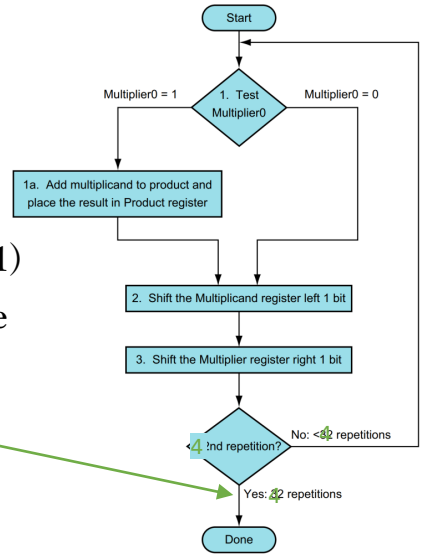
- Step 2 and 3 of the 3rd iteration



- Step 1 of the 4th iteration



- Step 2 and 3 of the 4th iteration will be performed after the above (step 1)
- Then, the computation is done since 4 repetitions have been performed



RISC-V Multiplication Inst.

- Four multiply instructions to produce 64-bit products
  - mul: multiply
    - Gives the integer 32-bit product (from the lower 32-bit of the 64-bit product)
  - mulh: multiply high
    - Gives the upper 32 bits of the 64-bit product (if the both operands are signed)
  - mulhu: multiply high unsigned
    - Gives the upper 32 bits of the 64-bit product (if the both operands are unsigned)
  - mulhsu: multiply high signed/unsigned
    - Gives the upper 32 bits of the 64-bit product (if one operand is signed and the other unsigned)
  - Use mulh result to check for 64-bit overflow
    - Check p. 198-199 of textbook for details

- Example: code sequence to get the product of unsigned multiplication

```
mulh    rdh, rs1, rs2
mul     rdl, rs1, rs2
```

  - The multiplicand and multiplier are kept in rs1 and rs2, respectively
  - The high/low 32 bits data are in rdh and rdl, respectively

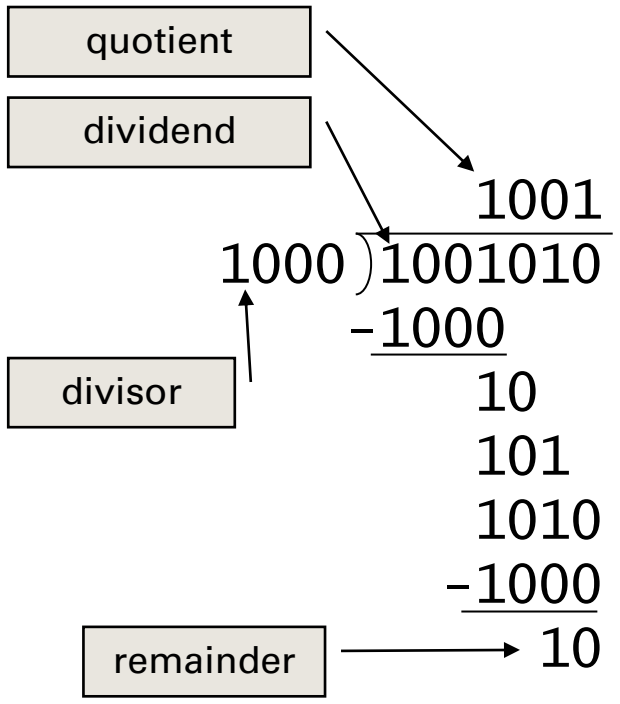
Courtesy of webpage; RV32M, RV64M Instructions

Division

- Exam if divisor is equal to 0

- Long division approach
  - If bit len(divisor) ≤ len(dividend)
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division (for HW impl.)
  - Do the subtract, and if remainder goes < 0, add divisor back

- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required



n-bit operands yield *n*-bit quotient and remainder

RISC-V Division Inst.

- Four instructions for signed integers and unsigned integers
  - ➢div (divide), rem (remainder): for signed division and reminder
  - ➢divu (divide unsigned), remu (remainder unsigned): for unsigned division and reminder

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
	Set if less than	slt x5, x6, x7	x5 = 1 if x5 < x6, else 0	Compare two registers
	Set if less than, unsigned	sltu x5, x6, x7	x5 = 1 if x5 < x6, else 0	Compare two registers
	Set if less than, immediate	slti x5, x6, x7	x5 = 1 if x5 < x6, else 0	Comparison with immediate
	Set if less than immediate, unsigned	sltiu x5, x6, x7	x5 = 1 if x5 < x6, else 0	Comparison with immediate
	Multiply	mul x5, x6, x7	x5 = x6 × x7	Lower 32 bits of 64-bit product
	Multiply high	mulh x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit signed product
	Multiply high, unsigned	mulhu x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit unsigned product
	Multiply high, signed-unsigned	mulhsu x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit signed-unsigned product
	Divide	div x5, x6, x7	x5 = x6 / x7	Divide signed 32-bit numbers
	Divide unsigned	divu x5, x6, x7	x5 = x6 / x7	Divide unsigned 32-bit numbers
	Remainder	rem x5, x6, x7	x5 = x6 % x7	Remainder of signed 32-bit division
	Remainder unsigned	remu x5, x6, x7	x5 = x6 % x7	Remainder of unsigned 32-bit division
Data transfer	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Uns. byte halfword from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
	Add upper immediate to PC	auipc x5, 0x12345	x5 = PC + 0x12345000	Used for PC-relative data addressing
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greatr/eq, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
Uncondit-ional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

FIGURE 3.12 RISC-V core architecture. RISC-V machine language is listed in the RISC-V Reference Data Card at the front of this book.

Floating Point

- Representation for non-integral numbers
  - Numbers with fractions, called *reals* in mathematics
  - Including very small and very large numbers
  - E.g., 3.14159... (pi), 2.71828... (*e*)
  - Computer arithmetic that supports such numbers is called *floating point*

- Scientific notation is an alternative notation for numbers
  - The notation has a single digit to the left of the decimal point
  - -2.34×10^{6}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

normalized — A number in floating-point notation that has no leading 0s

not normalized

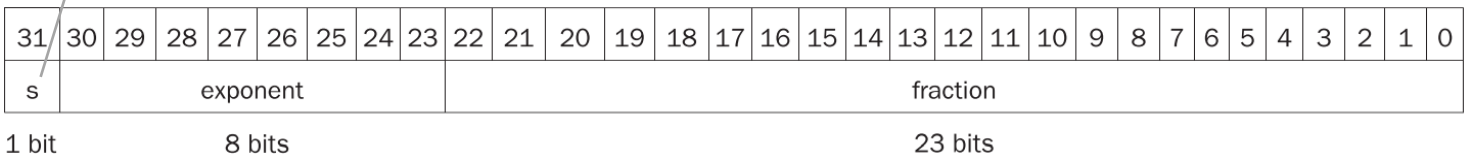
- In binary (base 2), the form is as below with a single nonzero digit to the left of the binary point
  - $\pm 1.xxxxxxx_{two} \times 2^{yyyy}$
  - $1.0_{two} \times 2^{-1}$

- Data types `float` and `double` in C

Floating-Point Representation

- A floating-point representation must find a compromise between
  the size of the fraction and the size of the exponent,
  given *a fixed bit-width* register
  - Fraction (or mantissa) is the value, generally between 0 and 1, placed in the fraction field
  - Exponent is the value placed in the *exponent* field in the numerical representation system of floating-point arithmetic

- A floating-point numbers is often of the form:
  $(-1)^S \times F \times 2^E$



An example of a RISC-V floating-point number (32-bit);
s is the sign of the floating-point number (1 meaning negative), *exponent* is the value of the 8-bit exponent field (including the sign of the exponent), and *fraction* is the 23-bit number

Overflow: A situation in which a *positive* exponent becomes too large to fit in the exponent field
Underflow: A situation in which a *negative* exponent becomes too large to fit in the exponent field

IEEE 754 Floating-Point Standard

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

significand

- S: sign bit
  - 0 ⇒ non-negative, 1 ⇒ negative
- Fraction
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (but it is a hidden bit in IEEE 754)
    It is 23 bits in single precision and 52 bits in double precision
  - Normalize significand: $1.0 \le |\text{significand}| < 2.0$; e.g., $1.0_{\text{two}} \times 2^{-1}$
  - Significand = 1 + Fraction
    It is actually 24 bits in single precision and 53 bits long in double precision
- Exponent
  - An actual exponent = Exponent-Bias
  - Ensures Exponent is unsigned
  - Bias for single precision: 127, for double precision: 1203
  - E.g., an actual exponent of -1 is represented by the bit pattern of the value -1 (Exponent) $+ 127_{\text{ten}}$, or $126_{\text{ten}} = 0111\ 1110_{\text{two}}$
- If we number the bits of the fraction from left to right s1, s2, s3, ..., then the value can be computed via the formula

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E$$

Range of Single Precision Numbers

- Exponents 00000000 and 11111111 are reserved

- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - I.e., $\pm 1.00000000000000000000000_{two} \times 2^{-126}$
    → $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - I.e., $\pm 1.11111111111111111111111_{two} \times 2^{-126}$
    → $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Range of Double Precision Numbers

- Exponents 0000…00 and 1111…11 are reserved

- Smallest value
  - Exponent: ...00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$

- Largest value
  - Exponent: ...11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Encoding of IEEE 754 Numbers

- Different bit patterns serve for different purposes
  - E.g., to represent the value of 0 (Use IEEE 754 format to represent a Zero)
  - E.g., to indicate the result of invalid operations via NaN, such as 0/0 or subtracting infinity from infinity
- The reserved data values are used for indicating the special values

- A new, revised standard IEEE 754-2008 is released in 2008
  - It includes nearly all the IEEE 754-1985 and adds a 16-bit format ("half precision") and a 128-bit format ("quadruple precision")
  - Half precision has a 1-bit sign, 5-bit exponent (with a bias of 15), and a 10-bit fraction
  - Quadruple precision has a 1-bit sign, a 15-bit exponent (with a bias of 262,143), and a 112-bit fraction

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FIGURE 3.13 IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the Elaboration on page 233. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book

Floating-Point Number Example

- The IEEE 754 binary representation of -0.75_{ten} ($-3/4_{ten}$) in single precision
  - -0.75
  - $= -0.11_{two} \times 2^{0} = -0.11_{two} = 1 \times 2^{-1} + 1 \times 2^{-2}$
  - $= -1.1_{two} \times 2^{-1}$ (in normalized scientific notation)
  - $= (-1)^{1} \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(126-127)}$
    - significand
  - ➢ S = 1
  - ➢ Fraction = $1000\cdots00_2$
  - ➢ Exponent = –1 (actual exponent) + Bias = 126
    - ❖ Single: $-1 + 127 = 126 = 01111110_2$

- The bit form of -0.75 in 32-bit register: 10111111010000…00
- Please check the bit form of -0.75 in double precision by yourself

Special Number (Infinities, NaNs, etc.)

- Exponent = 111...1, and Fraction = 000...0
  - ± ∞ (Infinity)
  - This encoding scheme indicates that the number can be used in subsequent calculations, avoiding need for overflow check
  - E.g., F+(+∞) = + ∞, F/∞=0

- Exponent = 111...1, and Fraction ≠ 000...0 (nonzero)
  - It indicates Not-a-Number (NaN)
  - NaN is illegal or undefined result
    - E.g., 0.0 / 0.0
  - Can be used in subsequent calculations

- Exponent = 000...0, and Fraction ≠ 000...0 (nonzero)
  - ± denormalized numbers
  - Please refer to the Elaboration section in page 233 for special numbers

Floating-Point Addition Procedure

1. Match the exponents of the two numbers
   - By adjusting the significand of the number with the smaller exponent to be the same with the larger exponent
2. Add the two significands
3. Normalize the result and check for overflow or underflow
   - The test for overflow and underflow depends on the precision of the operands
   - Refer to Fig. 3.13 for bit patterns
   - In single precision, -126 ≦ Exponent ≦ 127
4. Round the number
   - Round up: truncates the digit if 0 ≦ the digit to the right of desired digit ≦ 4, and
   - Round down: add 1 to the digit if 5 ≦ the number to the right ≦ 9
   - Perform Step 3 after rounding, if necessary, since the data is no longer normalized

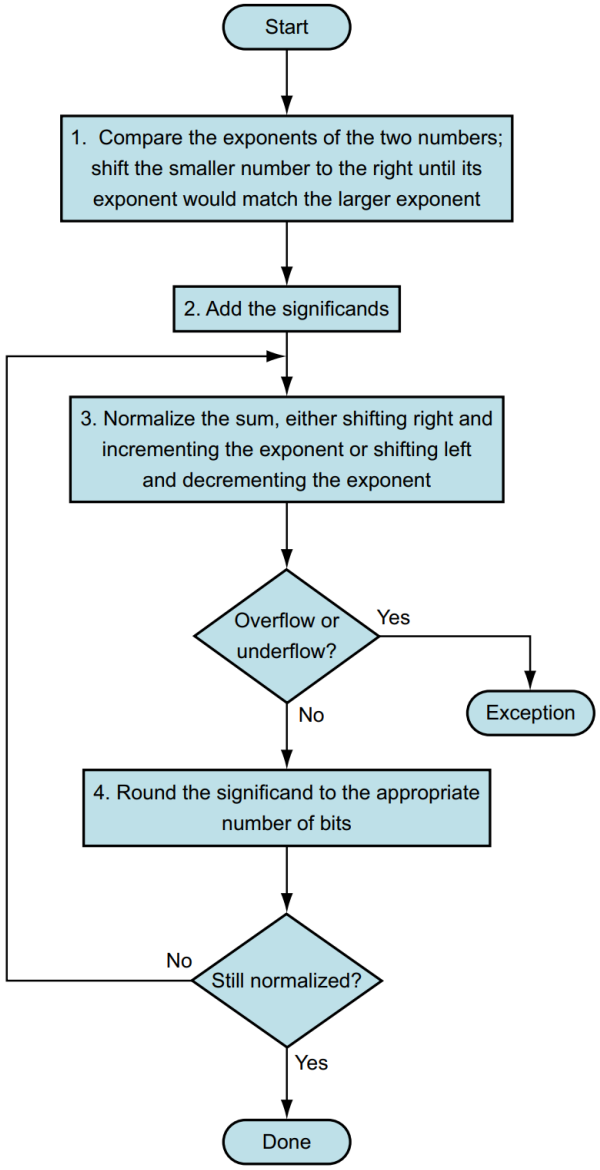


FIGURE 3.14 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3

Floating-Point Addition Example 1

- Now consider a 4-digit binary example (p. 216 in textbook)
  - To obtain the sum for the following two numbers
  - 0.5_{ten} : $1.000_2 \times 2^{-1}$
  - -0.4375_{ten} : $-1.110_2 \times 2^{-2}$

- 1. Match exponents
  - Shift number with smaller exponent (i.e., $-1.110_2 \times 2^{-2}$)
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow

- 4. Round (into 4 bits) and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) $(= 0.0625_{ten})$

Floating-Point Addition Example 2

- Consider a 4-digit decimal example (p. 215 in textbook)
  - Four decimal digits of the significand and two decimal digits of the exponent
  - $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$

1. Align decimal points
   - Shift number with smaller exponent
   - $9.999 \times 10^{1} + 0.016 \times 10^{1}$

2. Add significands
   - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$

3. Normalize result & check for over/underflow
   - 1.0015×10^{2}

4. Round and renormalize if necessary
   - 1.002×10^{2}

Floating-Point Adder

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

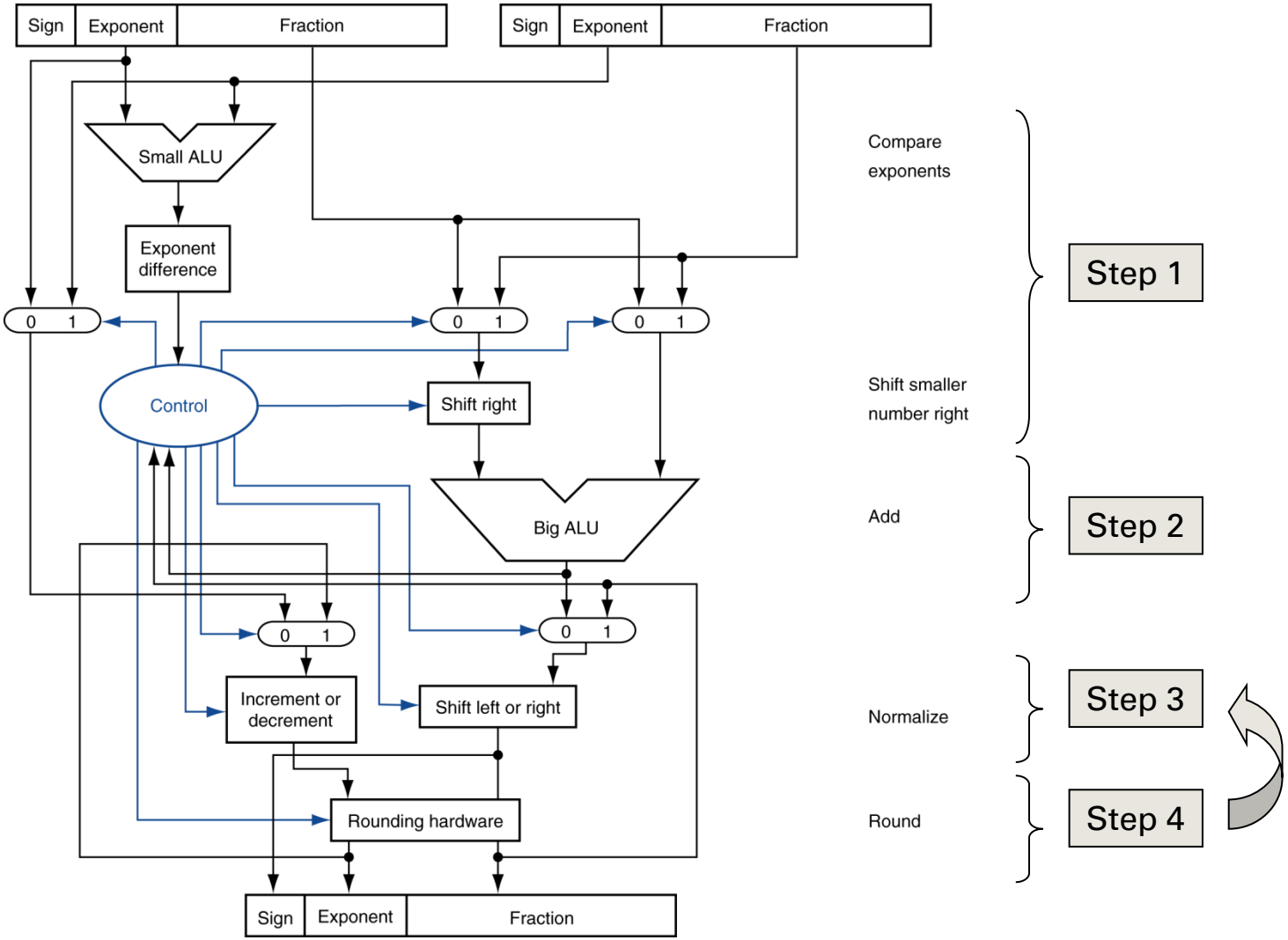


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the actual final result

Floating-Point Multiplication Procedure

1. Calculate the exponent of the product
   - ➢by adding the *exponents* directly if the operands together
   - ➢*Or, you can calculate with the *biased exponents*
2. Multiplication of the significands
   - ➢Be aware of the placement of the binary point of the product
3. Normalize the product
   - ➢Also check for overflow and underflow
4. Round the number
   - ➢to fit the desired digits length (e.g., 4 digits)
5. Use proper sign of the product
   - ➢If the original operands are both the same sign, the sign of the product is positive; otherwise, it's negative

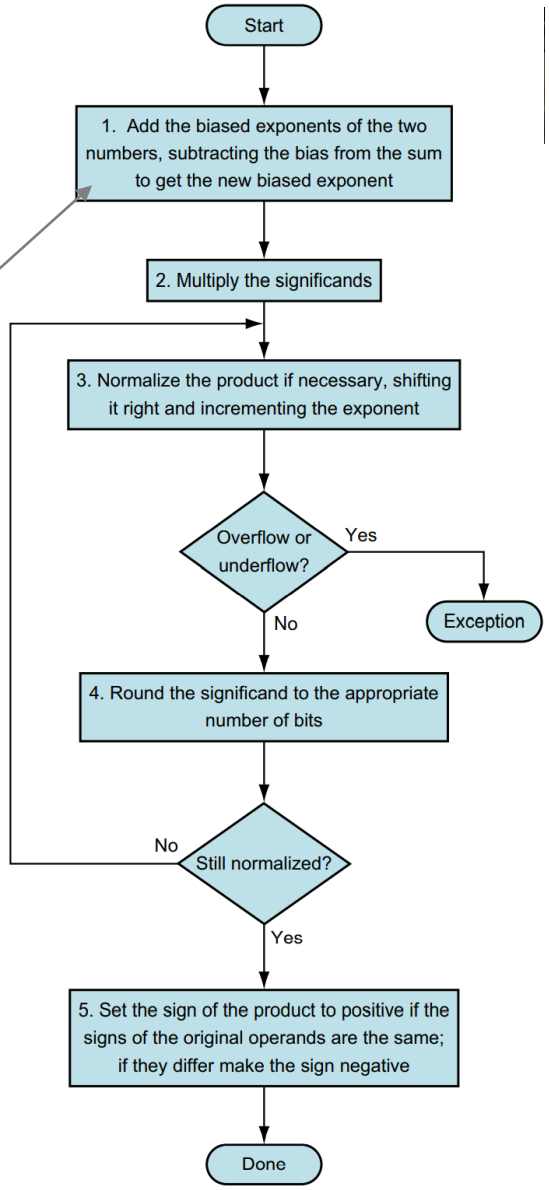


FIGURE 3.16 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3

Floating-Point Multiplication Example 1

- Now consider a 4-digit binary example
  - The multiplication of the two numbers
  - 0.5_{ten} : $1.000_2 \times 2^{-1}$
  - -0.4375_{ten} : $-1.110_2 \times 2^{-2}$

1. Add exponents
   - Unbiased: $-1 + -2 = -3$
   - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127 = 124$

2. Multiply significands
   - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$ (4 digits)

3. Normalize result & check for over/underflow
   - $1.110000_2 \times 2^{-3}$ (normalized; no change)
   - No over/underflow since $-126 \leqq -3 \leqq 127$
   - Check with biased exponent: $1 \leqq 124 \leqq 254$

4. Round and renormalize if necessary
   - $1.110_2 \times 2^{-3}$ (no change; done in step 2)

5. Determine sign
   - The signs of the operands differ: +ve × –ve ⇒ –ve
   - $-1.110_2 \times 2^{-3}$ $(= -0.21875_{ten})$

$$
\begin{array}{r}
1.000_{two} \\
\times \quad 1.110_{two} \\
\hline
0000 \\
1000 \\
1000 \\
1000 \\
\hline
1110000_{two}
\end{array}
$$

Floating-Point Multiplication Example 2

- Consider a 4-digit decimal example (p. 218 in textbook)
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$

1. Add exponents
   - For biased exponents, subtract bias from sum
   - New exponent = 10 + –5 = 5

2. Multiply significands
   - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$

3. Normalize result & check for over/underflow
   - 1.0212×10^{6}

4. Round and renormalize if necessary
   - 1.021×10^{6}

5. Determine sign of result from signs of operands
   - $+1.021 \times 10^{6}$

Floating-Point (FP) Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined
- The RISC-V designers decided to add separate floating-point registers (32 registers)

Floating-Point Instructions

- A separate set of FP registers: f0, …, f31
  - A single precision register is just the lower half of a double-precision register
  - I.e., single-precision values stored in the lower 32 bits
  - NOTE: f0 is not hard-wired to the constant 0
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
  - Include FP load/store and arithmetic instructions
- FP load and store instructions
  - Single-precision load/store: `flw`, `fsw`
  - Double-precision load/store: `fld`, `fsd`

The benefits of introducing another set of FP regs are 1) having twice as many registers without using up more bits in the instruction format, 2) having twice the register bandwidth by having separate integer and floating-point register sets, and 3) being able to customize registers to floating point

Floating-Point Arithmetic Instructions with IEEE 754 Format

- Single-precision arithmetic
  - fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s
    - e.g., fadd.s f2, f4, f6
- Double-precision arithmetic
  - fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d
    - e.g., fadd.d f2, f4, f6
- Single- and double-precision comparison
  - feq.s, flt.s, fle.s
  - feq.d, flt.d, fle.d
  - Result is 0 or 1 in integer destination register
    - Use beq, bne to branch on comparison result
- Branch on FP condition code true or false
  - B.cond

FP Instructions

RISC-V floating-point operands

32 floating-point registers	f0-f31	An *f*-register can hold either a single-precision floating-point number or a double-precision floating-point number.
2^{30} memory words	Memory[0], Memory[4], …, Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

RISC-V floating-point assembly language

R-type format for arithmetic inst.

Arithmetic	FP add single	fadd.s f0, f1, f2	f0 = f1 + f2	FP add (single precision)
	FP subtract single	fsub.s f0, f1, f2	f0 = f1 - f2	FP subtract (single precision)
	FP multiply single	fmul.s f0, f1, f2	f0 = f1 * f2	FP multiply (single precision)
	FP divide single	fdiv.s f0, f1, f2	f0 = f1 / f2	FP divide (single precision)
	FP square root single	fsqrt.s f0, f1	f0 = √f1	FP square root (single precision)
	FP add double	fadd.d f0, f1, f2	f0 = f1 + f2	FP add (double precision)
	FP subtract double	fsub.d f0, f1, f2	f0 = f1 - f2	FP subtract (double precision)
	FP multiply double	fmul.d f0, f1, f2	f0 = f1 * f2	FP multiply (double precision)
	FP divide double	fdiv.d f0, f1, f2	f0 = f1 / f2	FP divide (double precision)
	FP square root double	fsqrt.d f0, f1	f0 = √f1	FP square root (double precision)
Comparison	FP equality single	feq.s x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (single precision)
	FP less than single	flt.s x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (single precision)
	FP less than or equals single	fle.s x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (single precision)
	FP equality double	feq.d x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (double precision)
	FP less than double	flt.d x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (double precision)
	FP less than or equals double	fle.d x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (double precision)
Data transfer	FP load word	flw f0, 4(x5)	f0 = Memory[x5 + 4]	Load single-precision from memory
	FP load doubleword	fld f0, 8(x5)	f0 = Memory[x5 + 8]	Load double-precision from memory
	FP store word	fsw f0, 4(x5)	Memory[x5 + 4] = f0	Store single-precision from memory
	FP store doubleword	fsd f0, 8(x5)	Memory[x5 + 8] = f0	Store double-precision from memory

I-type format for loads

S-type format for stores

FIGURE 3.17 RISC-V floating-point architecture revealed thus far. This information is also found in column 2 of the RISC-V Reference Data Card at the front of this book

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits (in circuitry) of precision (guard, round, sticky)
  - Choice of four rounding modes
    1) always round up (toward $+\infty$), 2) always round down (toward $-\infty$), 3) truncate, and 4) round to nearest *even*
  - Allows programmer to fine-tune numerical behavior of a computation

- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
  - Java only supports the fourth mode above

- Trade-off between hardware complexity, performance, and market requirements

Guard: The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy

Round: Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format. It is also the name of the second of two extra bits kept on the right during intermediate floating-point calculations, which improves rounding accuracy

Sticky bit: A bit used in rounding (for indicating the status) in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit; e.g., to distinguish $0.50 \ldots 00_{ten}$ from $0.50 \ldots 01_{ten}$

Courtesy of link

Floating-Point C and Assembly Code Example

Convert a temperature in Fahrenheit to Celsius

- C code

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr in f10, result in f10 to return, constants in global memory space (base address in x3)

- Compiled RISC-V code:

```
f2c:
  flw    f0,const5(x3)    // f0 = 5.0f
  flw    f1,const9(x3)    // f1 = 9.0f
  fdiv.s f0, f0, f1       // f0 = 5.0f / 9.0f
  flw    f1,const32(x3)   // f1 = 32.0f
  fsub.s f10,f10,f1       // f10 = fahr – 32.0
  fmul.s f10,f0,f10       // f10 = (5.0f/9.0f) * (fahr–32.0f)
  jalr   x0,0(x1)         // return
```

Accurate Arithmetic (Cont'd)

- Example of the effect of guard and round
  - $2.3400_{ten} + 0.0256_{ten}$ (assume the result should be three significant decimal digits)
  - With guard/round
    $0.0256_{ten} + 2.3400_{ten} = 2.3656_{ten} => 2.37_{ten}$ (round up)
  - Without guard/round
    $0.02_{ten} + 2.34_{ten} = 2.36_{ten}$
  - 2.36_{ten} is off by 1 (1 ulp) in the last digit from 2.37_{ten}

- Measure accuracy with "units in the last place" (ulp)
  - The number of bits in error in the least significant bits of the significand between the actual number and the number that can be represented
  - Please refer to p. 230 in textbook (IEEE 754 guarantees ½ ulp)

The BIG Picture
Bit Patterns vs. Instructions

- Bit patterns have no inherent meaning
  - They may represent signed integers, unsigned integers, floating-point numbers, instructions, character strings, and so on
- What is represented depends on the instruction that operates on the bits in the word

- The major difference between computer numbers and numbers in the real world is that computer numbers have limited size and hence limited precision
- Programmers must remember these limits and write programs accordingly

C type	Java type	Data transfers	Operations
int	int	lw, sw	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
unsigned int	–	lw, sw	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
char	–	lb, sb	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
short	char	lh, sh	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
float	float	flw, fsw	fadd.s, fsub.s, fmul.s, fdiv.s, feq.s, flt.s, fle.s
double	double	fld, fsd	fadd.d, fsub.d, fmul.d, fdiv.d, feq.d, flt.d, fle.d

Please read through…

C procedure with 2D matrices (p. 226~229)

Compiling Floating-Point C Procedure with Two-Dimensional Matrices into RISC-V

Row-major vs. column major

The ulp for IEEE 754 (p. 230)

IEEE 754 guarantees ½ ulp

Subword Parallelism

- Many graphics and audio applications would perform the same operation on vectors of these data
- By partitioning a 128-bit adder, a processor could use parallelism to perform simultaneous operations on
  - short vectors of sixteen 8-bit operands (additions),
  - eight 16-bit operands,
  - four 32-bit operands, or
  - two 64-bit operands
- The cost of such partitioned adders was small yet the speedups could be large
- As the parallelism occurs within a wide word, the extensions are classified as subword parallelism
  - Also called data-level parallelism,
  - vector parallelism, or
  - Single Instruction, Multiple Data (SIMD)
  - SIMD will be introduced in Sec. 6.6, but you can read Sec. 3.7~3.8 at this moment

Fallacy
Right Shift and Division

- Is a right shift the same as an integer division by a power of 2?
  - Just as a left shift instruction can replace an integer multiply by a power of 2
- Yes, but the statements only holds for unsigned integers

- For signed integers
  - E.g., –5 / 4 = –1
  - Right shift: the sign bit (shifted bits) is filled with 0
    - $11111011_2 >>> 2 = 00111110_2 = +62$
  - Arithmetic right shift: replicate the sign bit
    - $11111011_2 >> 2 = 11111110_2 = -2$ (wrong answer)

Pitfall

- Floating-point addition is not associative!
  - ➢because floating-point numbers are approximations of real numbers and
  - ➢because computer arithmetic has limited precision

- Parallel programs may interleave operations in unexpected orders
  - ➢Assumptions of associativity may fail
- Need to validate parallel programs under varying degrees of parallelism

- An example
  - ➢(x+y)+z != x+(y+z)
  - ➢(x+y)+z = 1.0
  - ➢x+(y+z) = 0.0

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

Fallacy
Parallel Execution Strategies on Integer and Floating-Point Data Types

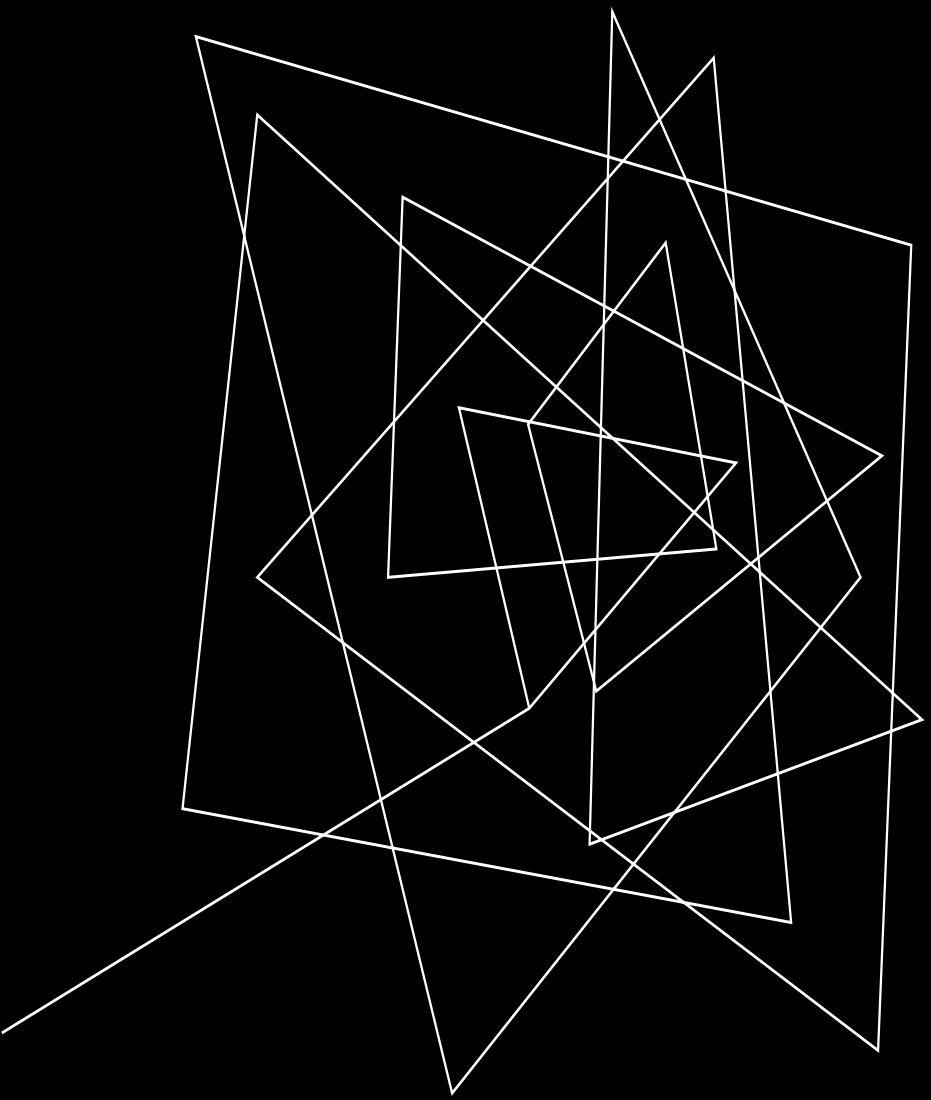
- Are parallel execution strategies that work for integer data types also work for floating-point data types?
  - If the answer is *no*, you presume there is a bug in the parallel version that you need to track down
- Two different dimensions
  - Sequential vs. Parallel
  - Integer vs. Floating-Point

- In a parallel computer, the operating system scheduler may use a different number of processors depending on what other programs are running on a parallel computer
  - since the varying number of processors from each run would cause the floating-point sums to be calculated in different orders

- Programmers who write parallel code with floating-point numbers need to verify whether the results are credible
  - even if they don't give the exact same answer as the sequential code

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs
- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow



Questions?