

Chap. 3

Arithmetic for computer (ALU)

臺大電機系
吳安宇教授

2025-03-24 v1



Outline

- 3.1 Introduction
- 3.2 Addition and Subtraction
- 3.3 Multiplication
- 3.4 Division
- 3.5 Floating Point
- 3.6 Parallelism and Computer Arithmetic:
Associativity



Introduction

- Arithmetic algorithms (ALU Operations)
 - 1. Addition/Subtraction
 - 2. Multiplication/Division
- Representation of numbers
 - 1. What is the "largest" and "smallest" number that can be represented by a computer **word**?
 - 2. How to represent ***floating-point*** numbers?
- MIPS instructions for (A)&(B)



Outline

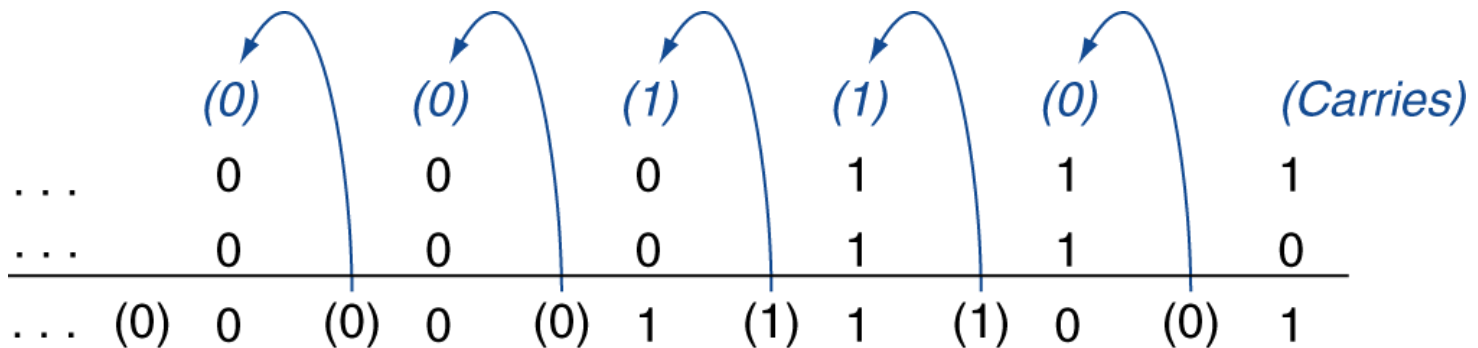
- 3.1 Introduction
- 3.2 Addition and Subtraction
- 3.3 Multiplication
- 3.4 Division
- 3.5 Floating Point
- 3.6 Parallelism and Computer Arithmetic:
Associativity

Addition and Subtraction

- Example
 - Adding 6 to 7

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 + \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\text{two}} = 13_{\text{ten}}
 \end{array}$$

The 4 bits to the right have all the action; Figure 3.2 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.



Addition and Subtraction

- Example
 - Subtracting 6 from 7 (or add 2's complement of 6)

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{r} \textcircled{-} \quad \begin{array}{l} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \end{array} \\ \hline = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}} \end{array}$$

or via addition using the two's complement representation of -6 :

$$\begin{array}{r} \textcircled{+} \quad \begin{array}{l} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{\text{two}} = -6_{\text{ten}} \end{array} \\ \hline = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}} \end{array}$$

-
- | | Bit | 31 | 30 | | 3 | 2 | 1 | 0 | |
|-------|-----|----|----|---|---|---|---|---|-------------------|
| | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | = 2,147,483,647 |
| +) | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 2 |
| <hr/> | | | | | | | | | |
| | | 1 | 0 | | | | | 0 | = - 2,147,483,647 |
- check sign bit (overflow detected)

Overflow condition

- **Exception** Also called **interrupt**. An unscheduled event that disrupts program execution; used to detect **overflow**.
- **Interrupt** An exception that comes from outside of the processor. (Some architectures use the term **interrupt** for all exceptions)
 - Add (**add**), add immediate (**addi**), and subtract (**sub**) cause exceptions on **overflow**.
 - Add unsigned (**addu**), add immediate unsigned (**addiu**), and subtract unsigned (**subu**) do **NOT** cause exceptions on **overflow**.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

- Adding operands of **different signs** → **"no"** overflow can occur



How to handle overflow in HW?

1. MIPS detects overflow with “exception” (interrupt) which is essentially an unplanned procedure call.
2. Steps : (HW interrupt)
 - ① The address of the instruction that caused overflow is saved in a register.
 - ② Jump to the (interrupt) service routine (ISR) for that exception (Exception Vector Interrupt Address of Arithmetic Overview = 8000 0180_hex)
 - ③ Return to the program
3. MIPS uses
 - ① A register called “exception program counter” (EPC) to contain the address of the instruction that causes the exception.
 - ② Instruction “move from system control (MFC0)” to copy EPC into a register so that MIPS software can return to the offending instruction via “jr” instruction.



Unsigned number

- Unsigned number is usually used for memory address and logic operations. In MIPS, the max address number is

$$2^{32} = 4,294,967,296_{10}$$

→ Usually, overflow can be **ignored** in unsigned numbers.



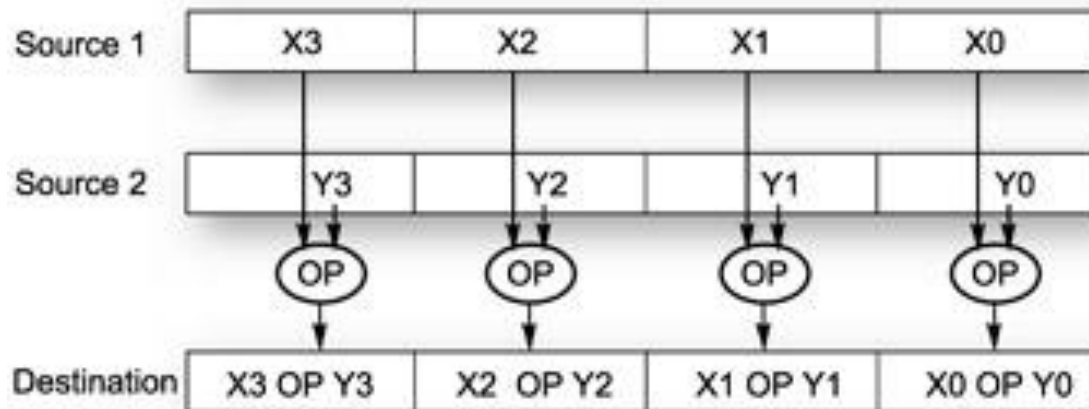
Arithmetic for Multimedia

- By partitioning the carry chains within a 64-bit adder, a processor can perform simultaneous operations on short vectors of *eight 8-bit operands, four 16-bit operands, or two 32-bit operands*.
- The extensions called **vector** or **SIMD (Single Instruction Multiple Datapath)**, e.g., MMX extension in x86 CPU
- **Saturating Operations:** When a calculation overflows, the result is set to the **largest positive** number or most negative number, rather than a modulo calculation as in conventional 2's complement arithmetic.

MMX Technology

- Single instruction, multiple data (SIMD) technology forms an extension to Intel Architecture Processors, starting with the Intel Pentium processor with MMX technology.
- **MMX**由英特爾開發的一種**SIMD**多媒體指令集，共有57條指令。
- 它於1996年整合在英特爾奔騰（Pentium）MMX處理器上，以提高其多媒體資料的處理能力

A typical SIMD instruction achieves higher performance by operating on multiple data elements at the same time.





MMX Technology

- MMX is a Pentium microprocessor from Intel that is designed to run faster when playing multimedia applications. According to Intel, a PC with an MMX microprocessor runs a **multimedia application** up **to 60% faster** than one with a microprocessor having the same clock speed but without MMX.
- In addition, an MMX microprocessor runs other applications about **10% faster**.



Arithmetic for Multimedia

Instruction category	Operands
Unsigned add/subtract	Eight 8-bit or Four 16-bit
Saturating add/subtract	Eight 8-bit or Four 16-bit
Max/min/minimum	Eight 8-bit or Four 16-bit
Average	Eight 8-bit or Four 16-bit
Shift right/left	Eight 8-bit or Four 16-bit

<http://www.globalspec.com/reference/77671/203279/chapter-12-simd-technology>



MMX Technology

A brief history of extending SIMD technology from 8-byte packed integers in the MMX technology to 16-byte packed floating-point numbers and packed integers in the *Streaming SIMD Extensions (SSE, SSE2, and SSE3)*

Technology	First Appeared	Description
MMX technology	Pentium processor with MMX technology	Introduced <u>8-byte packed integers</u> .
SSE	Pentium III processor	Added <u>16-byte packed single-precision floating-point numbers</u> .
SSE2	Pentium 4 processor	Added <u>16-byte packed double-precision floating-point numbers and integers</u> .
SSE3	Pentium 4 processor with Hyper-Threading Technology	Added some instructions to SSE2.
SSE3 on Intel EM64T	Intel EM64T processors	Extended number of SIMD registers <u>from 8 to 16</u> .



Outline

- 3.1 Introduction
- 3.2 Addition and Subtraction
- 3.3 Multiplication (omit)
- 3.4 Division
- 3.5 Floating Point (important)
- 3.6 Parallelism and Computer Arithmetic:
Associativity

Multiplication

- Multiplying $1000_2 \times 1001_2$

Multiplicand \rightarrow 1000 \rightarrow 8

Multiplier \rightarrow x) 1001 \rightarrow 9

1000

00000

000000

1000000

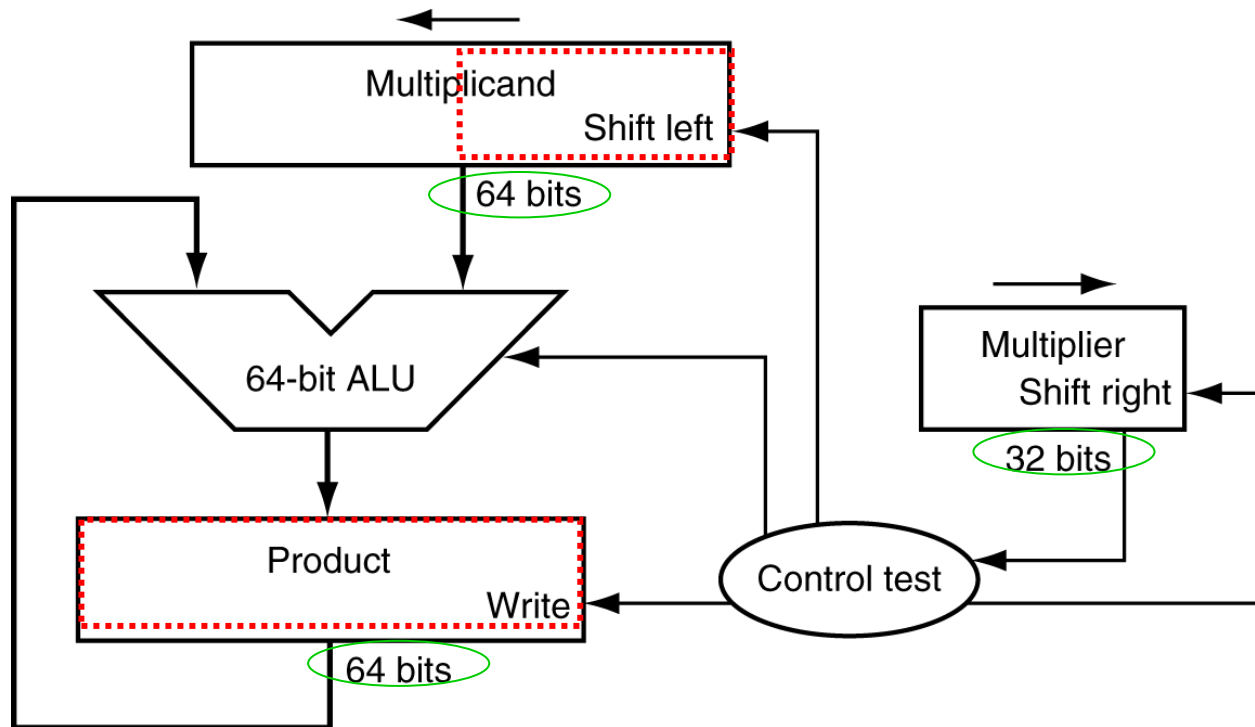
product \rightarrow 1001000 \rightarrow 72

- Steps:

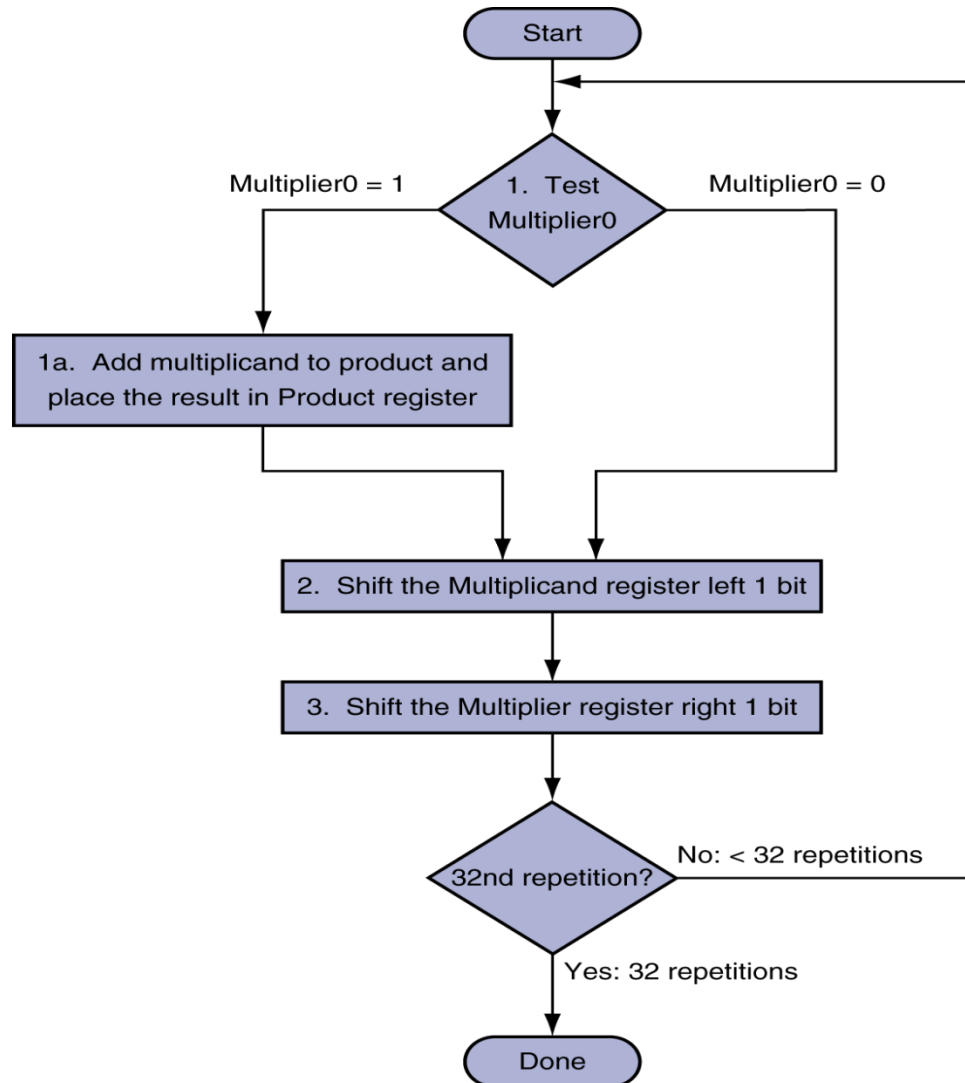
1. If multiplier bit = 0, place 0
= 1, place multiplicand
2. Do summation to get partial product

- n -bit multiplicand \times m -bit multiplier $\rightarrow (n+m)$ -bit product



First version of multiplication HW



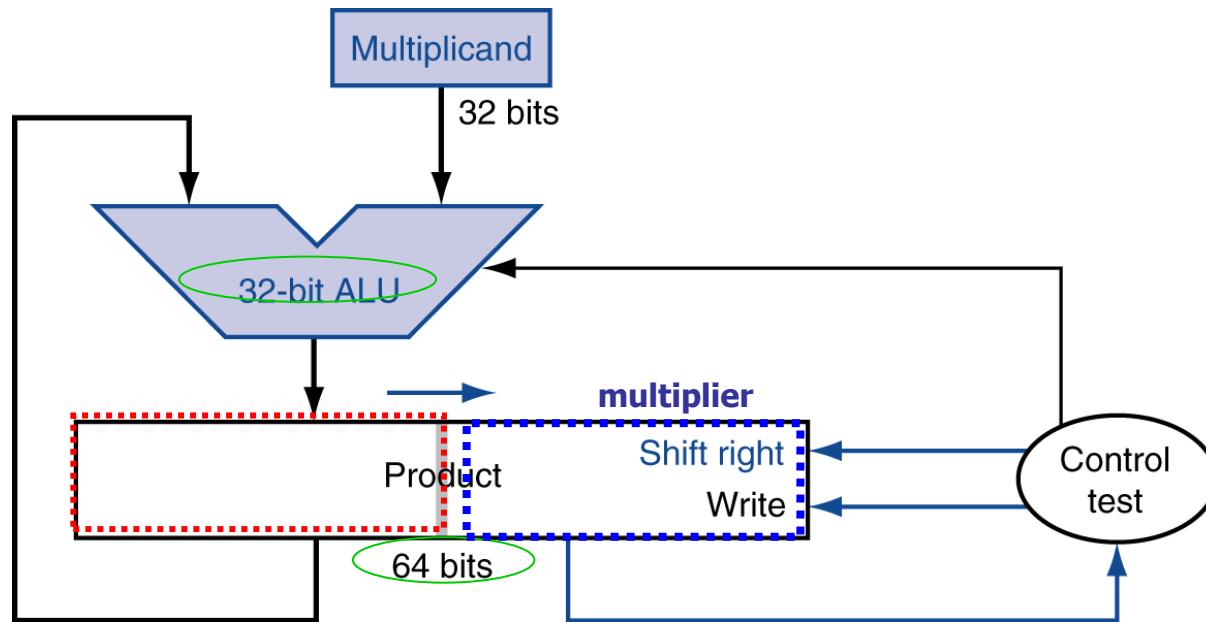
First version of multiplication algorithm



0010 (Multiplicand) x 0011 (Multiplier)

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001 1	0000 0010	0000 0000
1	1a: 1 \Rightarrow Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand 	0011	0000 0100	0000 0010
	3: Shift right Multiplier 	000 1	0000 0100	0000 0010
2	1a: 1 \Rightarrow Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000 1	0000 1000	0000 0110
3	1: 0 \Rightarrow no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000 1	0001 0000	0000 0110
4	1: 0 \Rightarrow no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Second version of multiplication HW



Differences between 1st & 2nd version:

- 64-bit multiplicand → 32 bit
- 64-bit ALU → 32 bit
- Instead of shifting the multiplicand to the left (x2), we shift the “product” to the right (initial at upper half of Product, then div 2)



Multiply in MIPS

- MIPS provides a separate pair of 32-bit registers to contain the **64-bit** product, called *Hi* and *Lo*.
- To produce a properly signed or unsigned product, MIPS has two instructions: **multiply (mult)** and **multiply unsigned (multu)**.
- To fetch the integer 32-bit product, the programmer uses ***move from lo (mflo)***.
- The MIPS assembler generates a **pseudo instruction** for multiply that specifies three general purpose registers, generating ***mflo*** and ***mfhi*** instructions to place the product into registers.

Summary of MIPS Arithmetic Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow detected
	add unsigned	addu \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow undetected
	subtract unsigned	subu \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow undetected
	add immediate unsigned	addiu \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow undetected
	move from coprocessor register	mfc0 \$s1,\$epc	$\$s1 = \epc	Copy Exception PC + special regs
	multiply	mult \$s2,\$s3	$Hi, Lo = \$s2 \times \$s3$	64-bit signed product in Hi, Lo
	multiply unsigned	multu \$s2,\$s3	$Hi, Lo = \$s2 \times \$s3$	64-bit unsigned product in Hi, Lo
	divide	div \$s2,\$s3	$Lo = \$s2 / \$s3,$ $Hi = \$s2 \bmod \$s3$	Lo = quotient, Hi = remainder
	divide unsigned	divu \$s2,\$s3	$Lo = \$s2 / \$s3,$ $Hi = \$s2 \bmod \$s3$	Unsigned quotient and remainder
	move from Hi	mfhi \$s1	$\$s1 = Hi$	Used to get copy of Hi
	move from Lo	mflo \$s1	$\$s1 = Lo$	Used to get copy of Lo



Outline

- 3.1 Introduction
- 3.2 Addition and Subtraction
- 3.3 Multiplication
- 3.4 Division (omit)
- 3.5 Floating Point
- 3.6 Parallelism and Computer Arithmetic:
Associativity

Division

- Dividing 1,001,010 by 1000

$$\begin{array}{r} 1001 \\ 1000 \overline{) 1001010} \\ \underline{-1000} \\ 10 \\ 101 \\ 1010 \\ \underline{-1000} \\ 10 \end{array}$$

Annotations:

- Quotient: 1001
- Dividend (64 bits): 1001010
- Divisor (32 bits): 1000
- Divisor with weighting: 1000 (highlighted in yellow)
- Remainder (from Dividend): 10

- $\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$
where ***Remainder*** < ***Divisor***

First version of Division HW

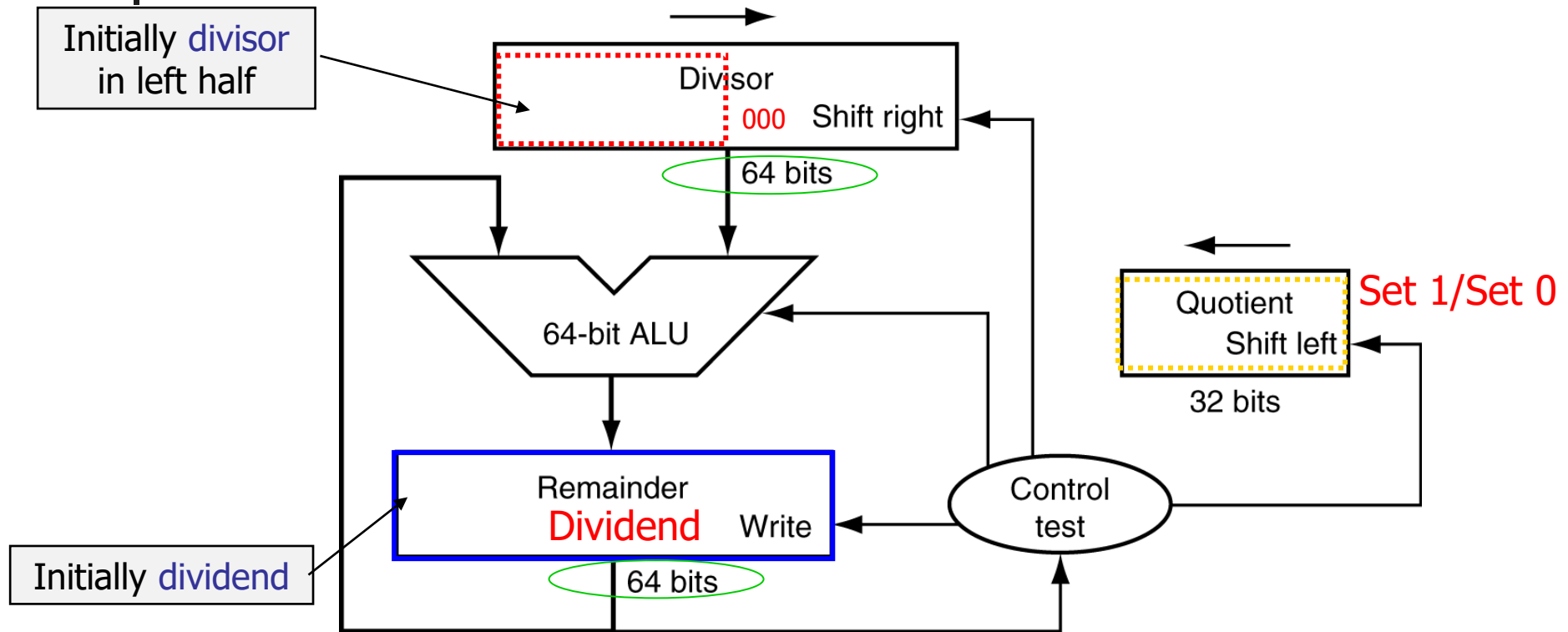


FIGURE 3.8 First version of the division hardware



$$0000\ 0111 \div 0010$$

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	1 110 0111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	1 111 0111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	1 111 1111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	0 000 0011
	2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	0 000 0001
	2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Negative
Restore/set0

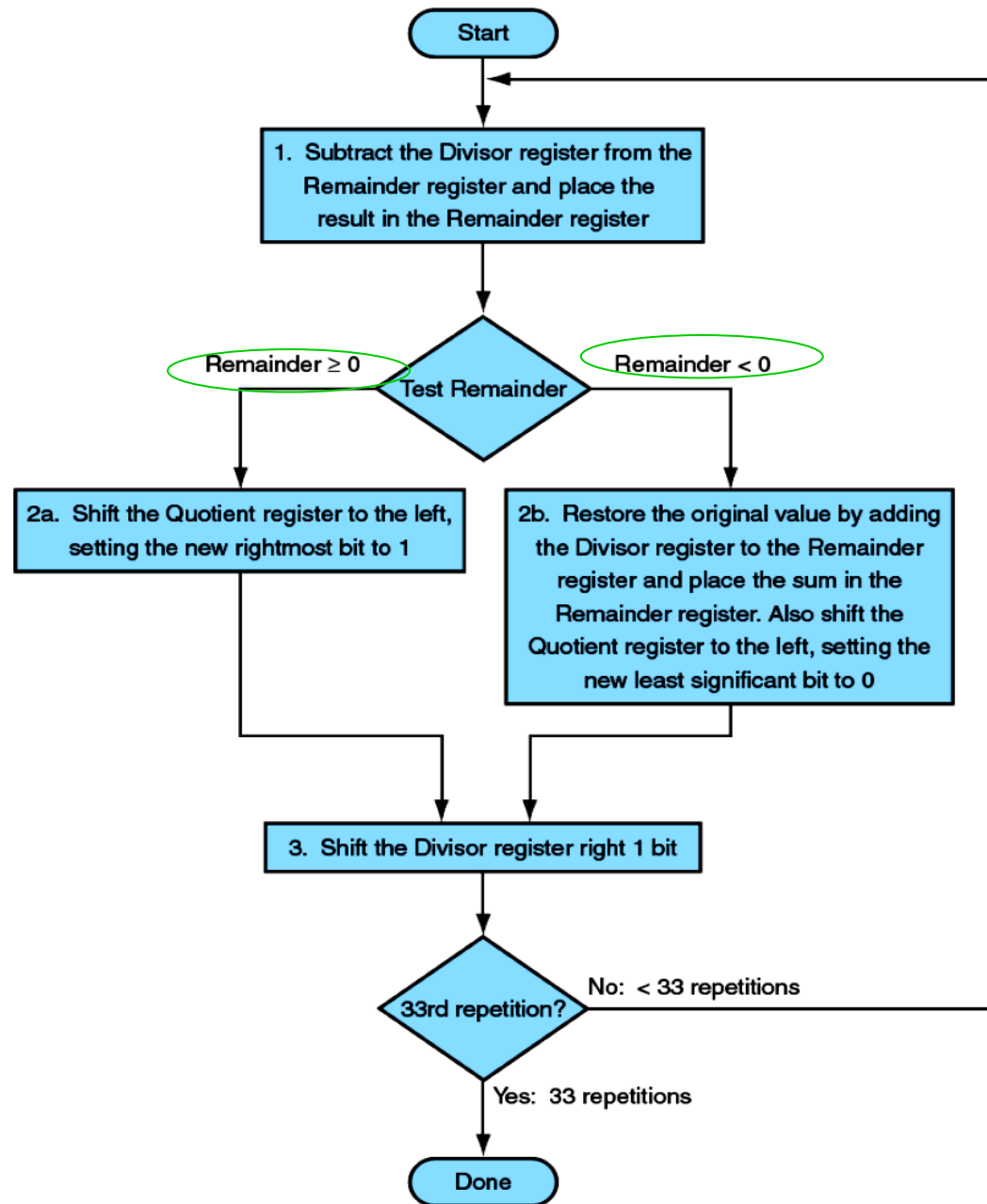
Negative
Restore/set0

Negative
Restore/set0

Positive
set1

Positive
set1

First version of Division algorithm



An improved version of division

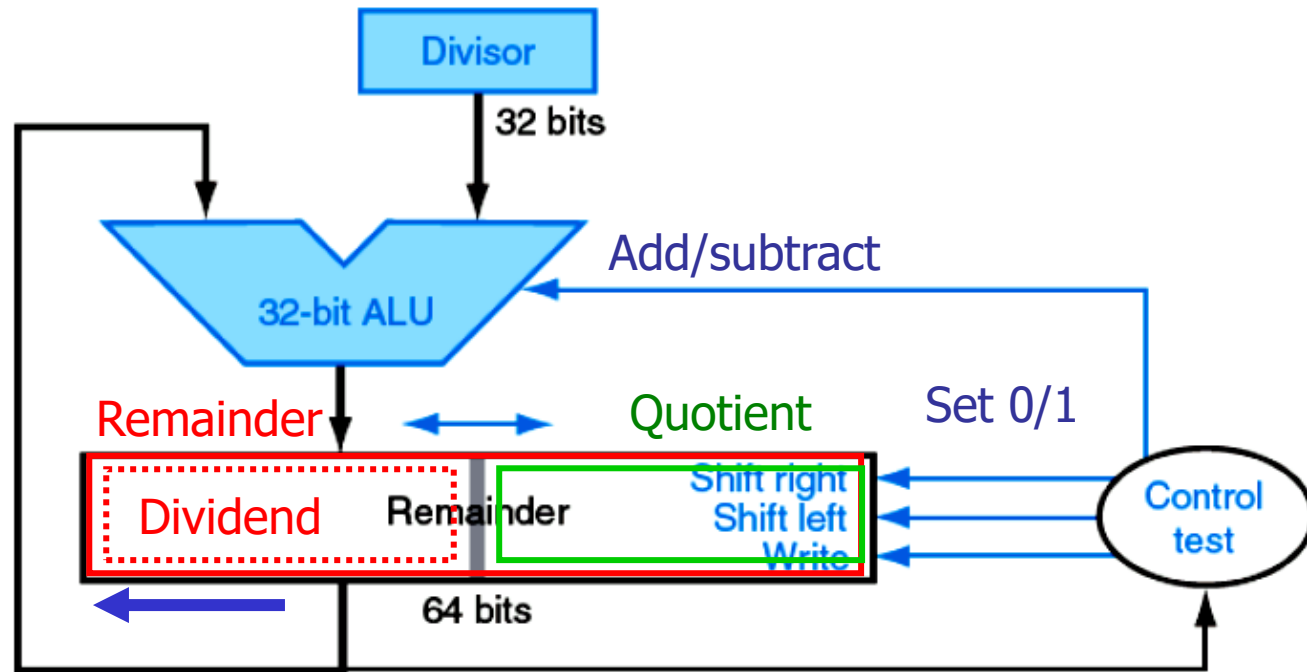


FIGURE 3.11 An improved version of the division hardware



Bidirection for both Multiply & Division



for Division only

c. f. 2nd version of multiplication HW

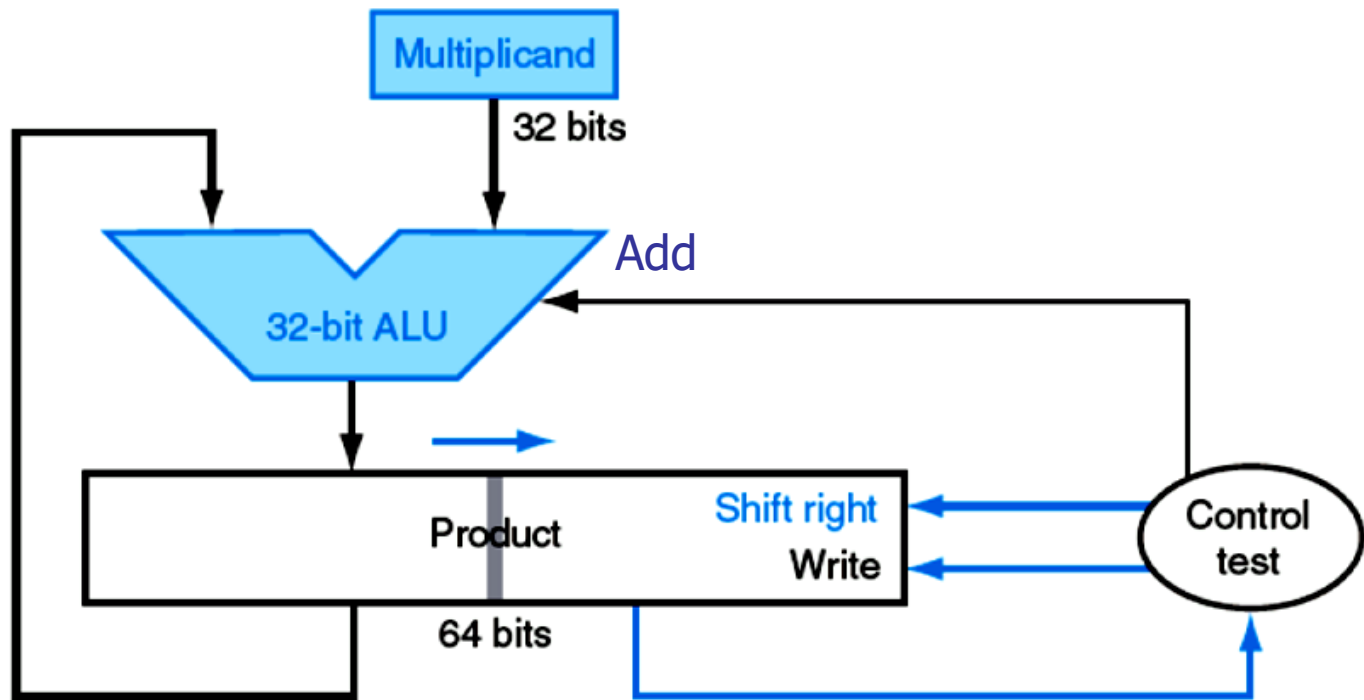


FIGURE 3.5 Refined version of the multiplication hardware



Division in MIPS

- The **same** sequential hardware can be used for both **multiply** and **divide**. The only requirement is a 64-bit register that can *shift left or right* and a 32-bit ALU that *adds or subtracts*.
- MIPS uses the 32-bit **Hi** and **Lo** registers for both multiply and divide.
- MIPS (**Lo = quotient, Hi = Remainder**)
 - div** \$2, \$3 # **Lo**=\$2 / \$3 , **Hi**=\$2 mod \$3 (*signed*)
 - divu** \$2, \$3 # **Lo**=\$2 / \$3 , **Hi**=\$2 mod \$3 (*unsigned*)
 - mflo** \$4 # Move Quotient to \$4
 - mfhi** \$1 # Move Remainder to \$1

Summary of MIPS Arithmetic Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow detected
	add unsigned	addu \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow undetected
	subtract unsigned	subu \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow undetected
	add immediate unsigned	addiu \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow undetected
	move from coprocessor register	mfc0 \$s1,\$epc	$\$s1 = \epc	Copy Exception PC + special regs
	multiply	mult \$s2,\$s3	$Hi, Lo = \$s2 \times \$s3$	64-bit signed product in Hi, Lo
	multiply unsigned	multu \$s2,\$s3	$Hi, Lo = \$s2 \times \$s3$	64-bit unsigned product in Hi, Lo
	divide	div \$s2,\$s3	$Lo = \$s2 / \$s3,$ $Hi = \$s2 \bmod \$s3$	Lo = quotient, Hi = remainder
	divide unsigned	divu \$s2,\$s3	$Lo = \$s2 / \$s3,$ $Hi = \$s2 \bmod \$s3$	Unsigned quotient and remainder
	move from Hi	mfhi \$s1	$\$s1 = Hi$	Used to get copy of Hi
	move from Lo	mflo \$s1	$\$s1 = Lo$	Used to get copy of Lo



Outline

- 3.1 Introduction
- 3.2 Addition and Subtraction
- 3.3 Multiplication (omit)
- 3.4 Division (omit)
- 3.5 Floating Point
- 3.6 Parallelism and Computer Arithmetic:
Associativity



Normalized number

- In addition to signed and unsigned integers, we have “**Real numbers**” in mathematics

$$\pi = 3.14159\dots$$

$$e = 2.71828\dots$$

$$3,155,760,000 = 3.15576 \times 10^9$$

$$0.000\ 000\ 001 = 1.0 \times 10^{-9}$$

- A number in scientific notation that has **no** leading 0 is called a “**normalized**” number

Ex.

1.0×10^{-9} (V) → Normalized scientific notation

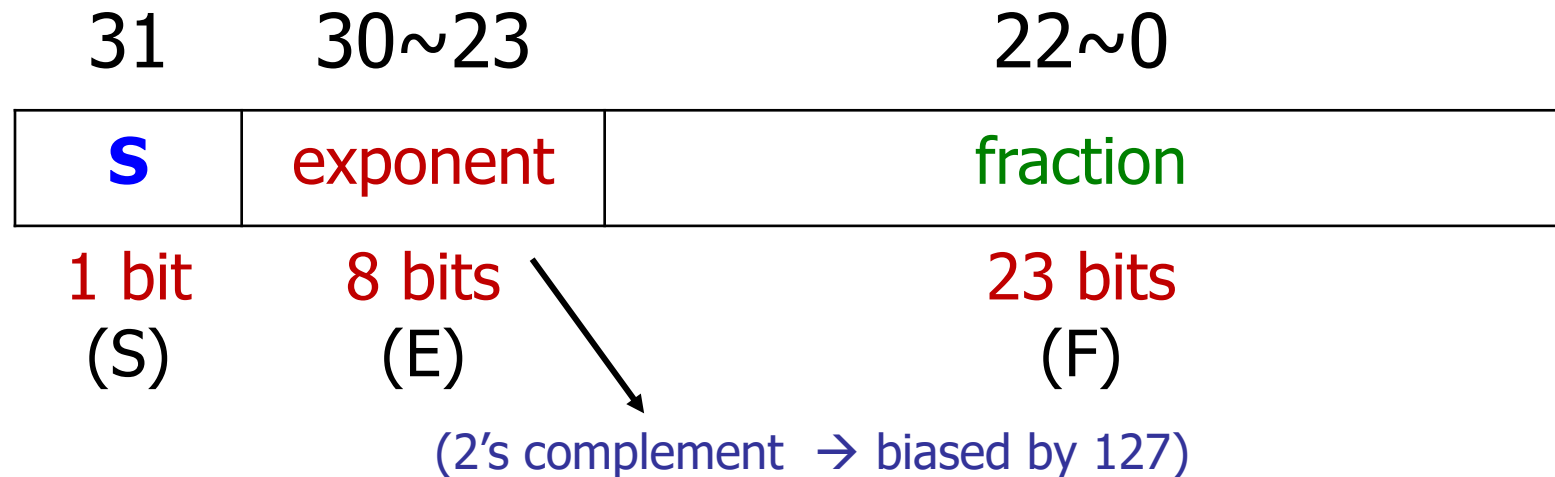
0.1×10^{-10} (X)

10.0×10^{-8} (X)

Representation of Floating-point Number

- In binary digits
- $\pm 1.xxxxxxx_2 \times 2^{yyyy}$ (base 2)
- Representation in MIPS Single Precision

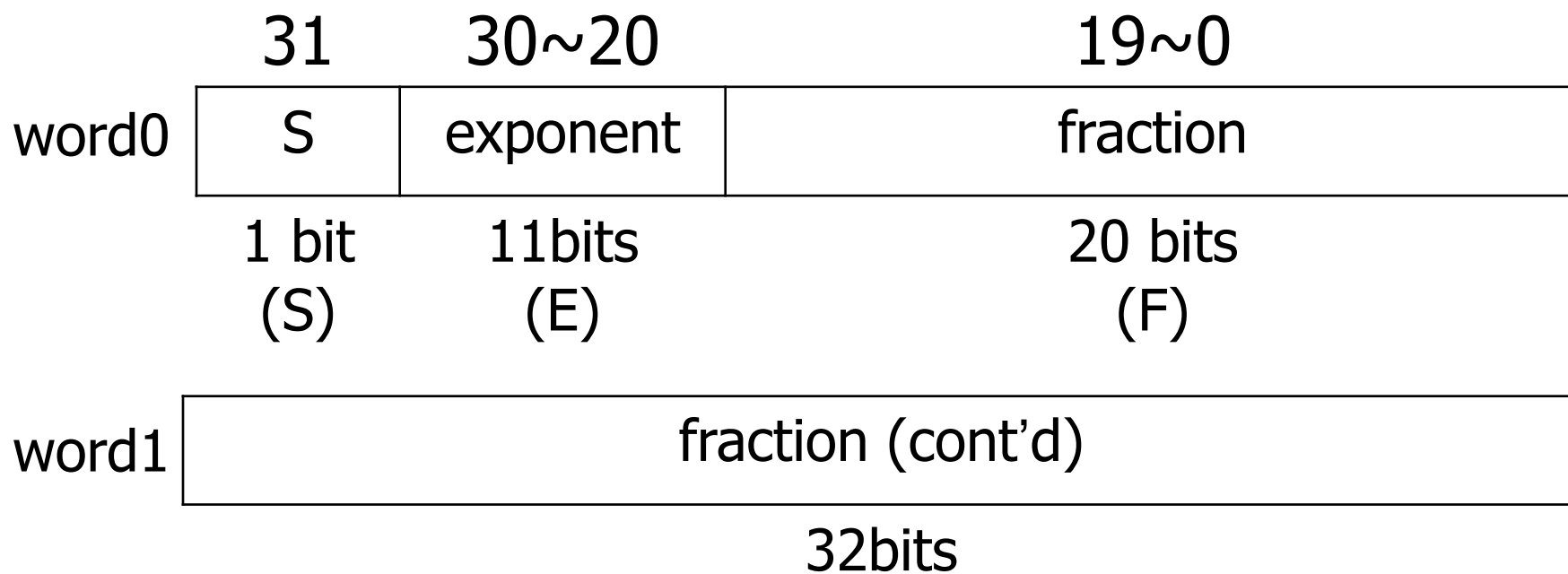
$$N = (-1)^S * F * 2^E$$





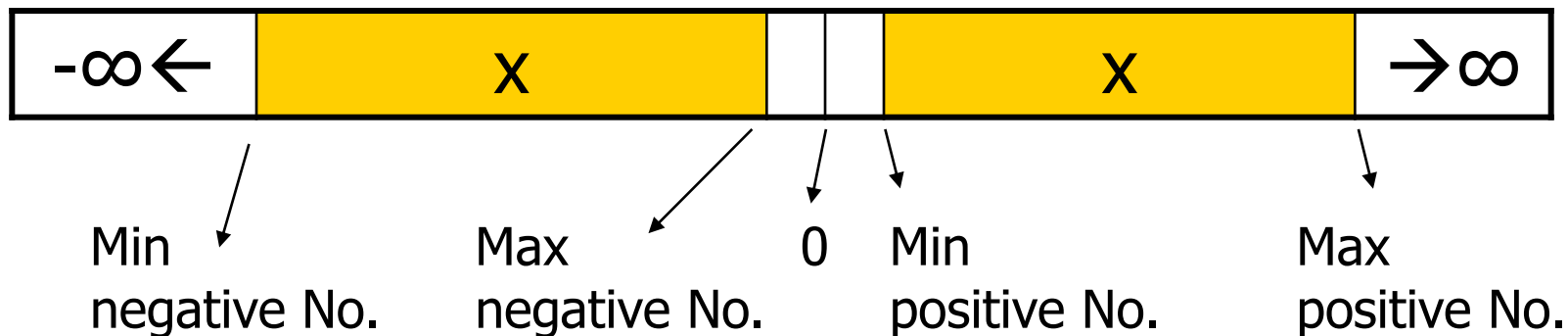
Double precision

- Double precision floating-point number (two **32-bit words**, by IEEE 754 floating-point standard)



Accuracy & Range

- Trade-off between “**accuracy**” and “**range**”
 - Increasing the size of **fraction** enhances **accuracy**.
 - Increasing the size of **exponent** increases the **range**.
- **Overflow**: number is too **large** to be represented by the hardware.
- **Underflow**: number is too **small** to be represented by the hardware.
- Exercise



Representing floating-point numbers

(IEEE 754 floating-point standard)

1. $0 \rightarrow S=E=F=0$
2. **Hidden 1** is added (Recall $N = \pm 1.xxxxxxx_2 \times 2^{yyyy}$)
3. $N = (-1)^S \times [1 + \text{Fraction}] \times 2^E$
 $= (-1)^S \times [1 + (S_1 \times 2^{-1}) + (S_2 \times 2^{-2}) + \dots + \dots] \times 2^E$

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

FIGURE 3.13 IEEE 754 encoding of floating-point numbers.

Representing floating-point numbers

- Unbiased exponent: 2's complement for Exponent
 $+1.0 \times 2^{-1}$ (not good) $+1.0 \times 2^{+1}$ (not good)

0	11111111	000.....000
---	----------	-------------

31 30~23 22~0

0	00000001	000.....000
---	----------	-------------

→ Biased notation

Exponent:

$\therefore N = (-1)^S \times (1 + \text{significand}) \times 2^{(\text{exponent} - \text{bias})}$
 where bias = 127 (for single precision)
 = 1023 (for double precision)

0	→	Reserved for 0 & denormalized
1	→	-126
.....	
127	→	0
128	→	+1
.....	- bias (127) Real meaning
254	→	+127
255	→	Reserved for NaN and Infinity

Representing floating-point numbers

類別	正負號	實際指數	有偏移指數	Exponent	Fraction	數值
最小的非正規數	*	-126	0	0000 0000	000 0000 0000 0000 0000 0001	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
中間大小的非正規數	*	-126	0	0000 0000	100 0000 0000 0000 0000 0000	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
最大的非正規數	*	-126	0	0000 0000	111 1111 1111 1111 1111 1111	$\pm (1 - 2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
最小的正規數	*	-126	1	0000 0001	000 0000 0000 0000 0000 0000	$\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
最大的正規數	*	127	254	1111 1110	111 1111 1111 1111 1111 1111	$\pm (2 - 2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$



Floating Point

- Ex. Represent -0.75_{10} in IEEE 754 standard
$$\begin{aligned}-0.75_{10} &= -0.11_2 \\ &= -0.11 \times 2^0 \text{ (scientific notation)} \\ &= -1.10 \times 2^{-1} \text{ (normalized scientific notation)}\end{aligned}$$

add 127 to exponent

$$\rightarrow (-1)^1 \times (1 + .1000\dots 0) \times 2^{(126)}$$

→

31	30~23 (exp)	22~0 (fraction)
1	01111110	1000.....00



Floating Point

- Ex. What's the number in base 10 ?

31	30~23	22~0
1	10000001	010.....00

Sign bit = 1

exponent = 129 \rightarrow real exponent = $129 - 127 = 2$

significand (fraction) = $2^{-2} = 0.25_{\text{base10}}$

$$\begin{aligned} N &= (-1)^S \times (1 + \text{fraction}) \times 2^{(\text{exponent} - \text{bias})} \\ &= -1^1 \times (1 + 0.25) \times 2^2 \\ &= -1 \times 1.25 \times 4 = -5_{10} \end{aligned}$$



Floating Point Addition

- Ex. $9.999 \times 10^1 + 1.610 \times 10^{-1}$ (with 4 significant digits)
 - Step1. align the decimal point with the larger exponent
(Note: large number is more important !)
 $1.610 \times 10^{-1} \rightarrow 0.016 \times 10^{+1}$
 - Step2. Adding the significand
$$\begin{array}{r} 9.999 \\ +) 0.016 \\ \hline 10.015 \end{array} \quad \text{sum} = 10.015 \times 10^{+1}$$
 - Step3. Write in “**normalized form**”
 $\text{sum} = 1.0015 \times 10^2$
 - Step4. **Round** significand to 4 digits
 $\text{sum} = 1.002 \times 10^2$



FP Adder Hardware

- 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
- **Overflow check:** Exponent (E) (0, 255 are reserved)
 - Single precision: $-126 \leq E \leq 127$
 - Double precision: $-1022 \leq E \leq 1023$

FP Adder Hardware

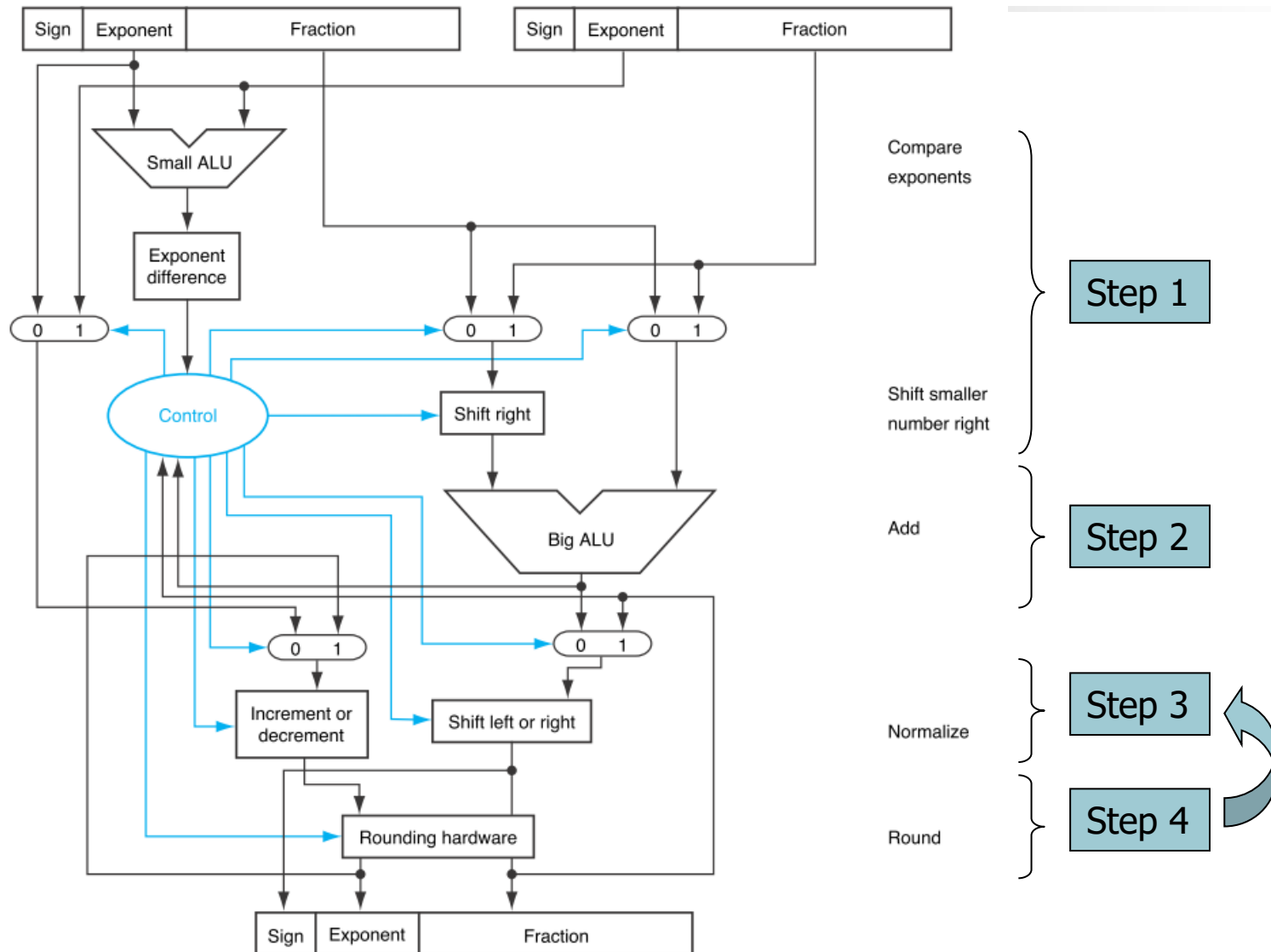


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition

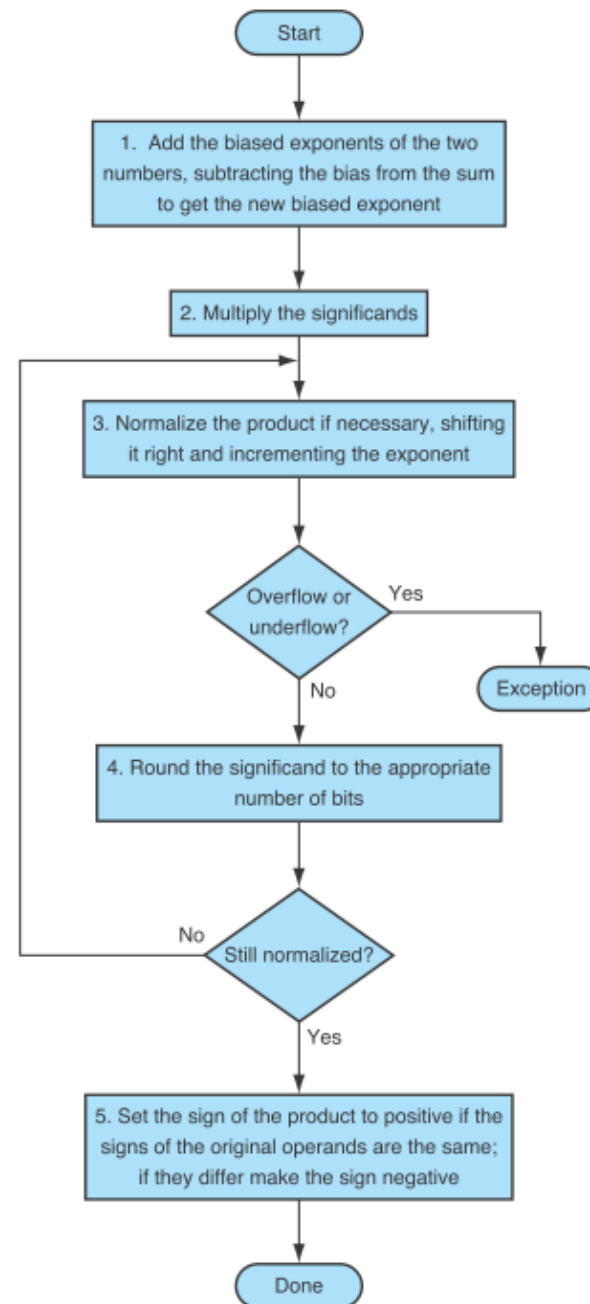


Floating Point Multiplication (skip)

- Ex: $(1.110 \times 10^{10}) \times (9.200 \times 10^{-5})$
(4 digits for significand, 2 digits for exponent)
 - Step1. **Adding exponent**
 $10 + (-5) = 5 \rightarrow$ new exponent
 - Step2. **Multiplying significands**
$$\begin{array}{r} 1.110 \\ \times 9.200 \\ \hline 10.21200_{10} \end{array} \rightarrow \text{product} = 10.21200 \times 10^5$$
 - Step3. **Normalize** the result
 $10.21200 \times 10^5 = 1.0212 \times 10^6$
check **overflow**: exponent is too large
underflow: large negative exponent } $-126 \leq E \leq 127$
 - Step4. **Rounding** significant (to 4 digits): 1.021×10^6
 - Step5. **Add sign**: $+1.021 \times 10^6$



Floating Point Multiplication (Fig. 3.6)





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 \leftrightarrow integer conversion
- Operations usually takes several cycles
 - Can be pipelined



Floating-point Instructions in MIPS

Floating-Point Instructions in MIPS

MIPS supports the IEEE 754 single precision and double precision formats with these instructions:

- Floating-point *addition, single* (add.s) and *addition, double* (add.d)
- Floating-point *subtraction, single* (sub.s) and *subtraction, double* (sub.d)
- Floating-point *multiplication, single* (mul.s) and *multiplication, double* (mul.d)
- Floating-point *division, single* (div.s) and *division, double* (div.d)



Floating-point Registers

The MIPS designers decided to add separate floating-point registers—called `$f0`, `$f1`, `$f2`, ...—used either for single precision or double precision. Hence, they included separate loads and stores for floating-point registers: `lwc1` and `swc1`. The base registers for floating-point data transfers remain integer registers. The MIPS code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
lwc1      $f4,x($sp)  # Load 32-bit F.P. number into F4
lwc1      $f6,y($sp)  # Load 32-bit F.P. number into F6
add.s     $f2,$f4,$f6 # F2 = F4 + F6 single precision
swc1      $f2,z($sp)  # Store 32-bit F.P. number from F2
```

A double precision register is really an even-odd pair of single precision registers, using the even register number as its name. Thus, the pair of single precision registers `$f2` and `$f3` also form the double precision register named `$f2`.

- *`lwc1`: load fp word from memory to coprocessor 1*
- *`swc1`: store fp word from coprocessor 1 to memory*

Floating-Point Inst. format (on Page A-681)

In the actual instructions below, bits 21–26 are 0 for single precision and 1 for double precision. In the pseudoinstructions below, `fdest` is a floating-point register (e.g., `$f2`).

Floating-point absolute value double

<code>abs.d fd, fs</code>	0x11	1	0	fs	fd	5
	6	5	5	5	5	6

Floating-point absolute value single

<code>abs.s fd, fs</code>	0x11	0	0	fs	fd	5
---------------------------	------	---	---	----	----	---

Compute the absolute value of the floating-point double (single) in register `fs` and put it in register `fd`.

Floating-point addition double

<code>add.d fd, fs, ft</code>	0x11	0x11	ft	fs	fd	0
	6	5	5	5	5	6

Summary of MIPS floating-pt Operands

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	\$f0, \$f1, \$f2, . . . , \$f31	MIPS floating-point registers are used in pairs for double precision numbers.
2 ³⁰ memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s \$f2,\$f4,\$f6	\$f2 = \$f4 + \$f6	FP add (single precision)
	FP subtract single	sub.s \$f2,\$f4,\$f6	\$f2 = \$f4 - \$f6	FP sub (single precision)
	FP multiply single	mul.s \$f2,\$f4,\$f6	\$f2 = \$f4 × \$f6	FP multiply (single precision)
	FP divide single	div.s \$f2,\$f4,\$f6	\$f2 = \$f4 / \$f6	FP divide (single precision)
	FP add double	add.d \$f2,\$f4,\$f6	\$f2 = \$f4 + \$f6	FP add (double precision)
	FP subtract double	sub.d \$f2,\$f4,\$f6	\$f2 = \$f4 - \$f6	FP sub (double precision)
	FP multiply double	mul.d \$f2,\$f4,\$f6	\$f2 = \$f4 × \$f6	FP multiply (double precision)
	FP divide double	div.d \$f2,\$f4,\$f6	\$f2 = \$f4 / \$f6	FP divide (double precision)
Data transfer	load word copr. 1	lwcl \$f1,100(\$s2)	\$f1 = Memory[\$s2 + 100]	32-bit data to FP register
	store word copr. 1	swcl \$f1,100(\$s2)	Memory[\$s2 + 100] = \$f1	32-bit data to memory
Conditional branch	branch on FP true	bclt 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bclf 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s \$f2,\$f4	if (\$f2 < \$f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d \$f2,\$f4	if (\$f2 < \$f4) cond = 1; else cond = 0	FP compare less than double precision



Floating-point Brach Inst. in MIPS

- Floating-point *division, single* (`div.s`) and *division, double* (`div.d`)
- Floating-point *comparison, single* (`c.x.s`) and *comparison, double* (`c.x.d`), where *x* may be *equal* (`eq`), *not equal* (`neq`), *less than* (`lt`), *less than or equal* (`le`), *greater than* (`gt`), or *greater than or equal* (`ge`)
- Floating-point *branch, true* (`bc1t`) and *branch, false* (`bc1f`)

Summary of MIPS floating-pt Operands

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	\$f0, \$f1, \$f2, . . . , \$f31	MIPS floating-point registers are used in pairs for double precision numbers.
2 ³⁰ memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s \$f2,\$f4,\$f6	\$f2 = \$f4 + \$f6	FP add (single precision)
	FP subtract single	sub.s \$f2,\$f4,\$f6	\$f2 = \$f4 - \$f6	FP sub (single precision)
	FP multiply single	mul.s \$f2,\$f4,\$f6	\$f2 = \$f4 × \$f6	FP multiply (single precision)
	FP divide single	div.s \$f2,\$f4,\$f6	\$f2 = \$f4 / \$f6	FP divide (single precision)
	FP add double	add.d \$f2,\$f4,\$f6	\$f2 = \$f4 + \$f6	FP add (double precision)
	FP subtract double	sub.d \$f2,\$f4,\$f6	\$f2 = \$f4 - \$f6	FP sub (double precision)
	FP multiply double	mul.d \$f2,\$f4,\$f6	\$f2 = \$f4 × \$f6	FP multiply (double precision)
	FP divide double	div.d \$f2,\$f4,\$f6	\$f2 = \$f4 / \$f6	FP divide (double precision)
Data transfer	load word copr. 1	lwcl \$f1,100(\$s2)	\$f1 = Memory[\$s2 + 100]	32-bit data to FP register
	store word copr. 1	swcl \$f1,100(\$s2)	Memory[\$s2 + 100] = \$f1	32-bit data to memory
Conditional branch	branch on FP true	bclt 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bclf 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s \$f2,\$f4	if (\$f2 < \$f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d \$f2,\$f4	if (\$f2 < \$f4) cond = 1; else cond = 0	FP compare less than double precision

Summary of MIPS fp Machine Language

MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s \$f2,\$f4,\$f6
sub.s	R	17	16	6	4	2	1	sub.s \$f2,\$f4,\$f6
mul.s	R	17	16	6	4	2	2	mul.s \$f2,\$f4,\$f6
div.s	R	17	16	6	4	2	3	div.s \$f2,\$f4,\$f6
add.d	R	17	17	6	4	2	0	add.d \$f2,\$f4,\$f6
sub.d	R	17	17	6	4	2	1	sub.d \$f2,\$f4,\$f6
mul.d	R	17	17	6	4	2	2	mul.d \$f2,\$f4,\$f6
div.d	R	17	17	6	4	2	3	div.d \$f2,\$f4,\$f6
lwc1	I	49	20	2	100			lwc1 \$f2,100(\$s4)
swc1	I	57	20	2	100			swc1 \$f2,100(\$s4)
bclt	I	17	8	1	25			bclt 25
bcltf	I	17	8	0	25			bcltf 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s \$f2,\$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d \$f2,\$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

ft,

fs,

fd



FP Example: °F to °C

- C code:

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

- fahr in \$f12, result in \$f0, literals in global memory space

- Compiled MIPS code:

f2c:

```
lwc1    $f16, const5($gp)    # $f16 = 5.0  
lwc1    $f18, const9($gp)    # $f18 = 9.0  
div.s   $f16, $f16, $f18     # 5.0/9.0  
lwc1    $f18, const32($gp)   # $f18 = 32.0  
sub.s   $f18, $f12, $f18     # $f18 = fahr - 32.0  
mul.s   $f0, $f16, $f18      # $f0 = result  
jr      $ra                  # return
```




FP Example: Array Multiplication (skip: please check by yourself)

- $X = X + Y \times Z$
 - All 32×32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double x[][],  
         double y[][], double z[][]) {  
    int i, j, k;  
    for (i = 0; i != 32; i = i + 1)  
        for (j = 0; j != 32; j = j + 1)  
            for (k = 0; k != 32; k = k + 1)  
                x[i][j] = x[i][j]  
                    + y[i][k] * z[k][j];  
}
```

- Addresses of x, y, z in \$a0, \$a1, \$a2, and
i, j, k in \$s0, \$s1, \$s2



FP Example: Array Multiplication

■ MIPS code:

	li	\$t1, 32	# \$t1 = 32 (row size/loop end)
	li	\$s0, 0	# i = 0; initialize 1st for loop
L1:	li	\$s1, 0	# j = 0; restart 2nd for loop
L2:	li	\$s2, 0	# k = 0; restart 3rd for loop
	sll	\$t2, \$s0, 5	# \$t2 = i * 32 (size of row of x)
	addu	\$t2, \$t2, \$s1	# \$t2 = i * size(row) + j
	sll	\$t2, \$t2, 3	# \$t2 = byte offset of [i][j]
	addu	\$t2, \$a0, \$t2	# \$t2 = byte address of x[i][j]
L3:	l.d	\$f4, 0(\$t2)	# \$f4 = 8 bytes of x[i][j]
	sll	\$t0, \$s2, 5	# \$t0 = k * 32 (size of row of z)
	addu	\$t0, \$t0, \$s1	# \$t0 = k * size(row) + j
	sll	\$t0, \$t0, 3	# \$t0 = byte offset of [k][j]
	addu	\$t0, \$a2, \$t0	# \$t0 = byte address of z[k][j]
	l.d	\$f16, 0(\$t0)	# \$f16 = 8 bytes of z[k][j]

...



FP Example: Array Multiplication

...

sll	\$t0, \$s0, 5	# \$t0 = i*32 (size of row of y)
addu	\$t0, \$t0, \$s2	# \$t0 = i*size(row) + k
sll	\$t0, \$t0, 3	# \$t0 = byte offset of [i][k]
addu	\$t0, \$a1, \$t0	# \$t0 = byte address of y[i][k]
l.d	\$f18, 0(\$t0)	# \$f18 = 8 bytes of y[i][k]
mul.d	\$f16, \$f18, \$f16	# \$f16 = y[i][k] * z[k][j]
add.d	\$f4, \$f4, \$f16	# f4=x[i][j] + y[i][k]*z[k][j]
addiu	\$s2, \$s2, 1	# \$k k + 1
bne	\$s2, \$t1, L3	# if (k != 32) go to L3
s.d	\$f4, 0(\$t2)	# x[i][j] = \$f4 (A-684,pseudo)
addiu	\$s1, \$s1, 1	# \$j = j + 1
bne	\$s1, \$t1, L2	# if (j != 32) go to L2
addiu	\$s0, \$s0, 1	# \$i = i + 1
bne	\$s0, \$t1, L1	# if (i != 32) go to L1



Summary

- Computer arithmetic are essential parts of ALU designs.
- Most modern CPU has supported **floating-point operations** for high-performance computation
- SIMD extension is also popular to enhance CPU's capability for multimedia applications.
- Note the computing power and speed efficiency among arithmetic operations.
- In general, Integer ADD/SUB < (Int) MUL
<< (Int) Div << Floating-point operations (single)
<< Floating-point operations (double)