CHAPTER 3

Arithmetic for Computers

3.1 Introduction 178

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations x

3.2 Addition and Subtraction 178

- Example: 7 + 6 Binary Addition

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

  - Figure 3.1 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

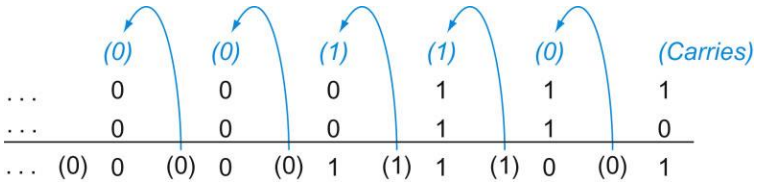


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 1 1 1 1. This generates a 0 for this sum bit and a carry out of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

- Example: 7 - 6 Binary Subtraction

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

  - Remember that **c – a = c + (-a)** because we subtract by negating the second operation than add. Therefore, 7 – 6 = 7 + (-6)

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

- Figure 3.2 shows the combination of operations, operands, and results that indicate an 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

FIGURE 3.2 Overflow conditions for addition and subtraction.

- Dealing with Overflow
  - Some languages (e.g., **C** and **Java**) **ignore** overflow
    - Use MIPS addu, addui, subu instructions
    - Because C ignores overflows, the MIPS C compiler will always generate the unsigned version of the arithmetic instruction addu, addiu, and subu, no matter what type of the variables.
  - Other languages (e.g., **Ada** and **Fortran**) require raising an exception
    - Use MIPS add, addi, sub instructions
    - Languages like Ada and Fortran require the program be notified. The programmer or the programming environment must then decide what to do when overflow occurs.
    - On overflow, invoke exception handler
      - Save PC in exception program counter (EPC) register
      - Jump to predefined handler address
      - mfc0 (move from **coprocessor 0** register) instruction can retrieve EPC value, to return after corrective action

3.3 Multiplication 183

- The length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is **n + m** bit long.

$$
\begin{array}{lll}
\text{Multiplicand} & & 1000_{ten} \\
\text{Multiplier} & \times & 1001_{ten} \\
\hline
 & & 1000 \\
 & & 0000 \\
 & & 0000 \\
 & & 1000 \\
\hline
\text{Product} & & 1001000_{ten}
\end{array}
$$

Sequential Version of the Multiplication Algorithm and Hardware

- The 64-bit Product register is initialized to 0
- It is clear that we will need to move the multiplicand left one digit each step, as it may be added to the intermediate products.
- If each step took a clock cycle, this algorithm would require almost **100** clock cycles to multiply two 32-bit numbers.

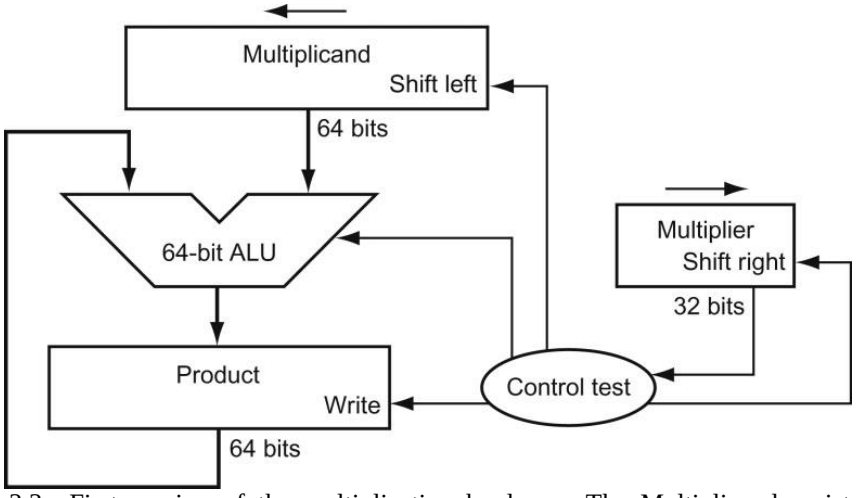


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all **64** bits wide, with only the Multiplier register containing **32** bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register 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.

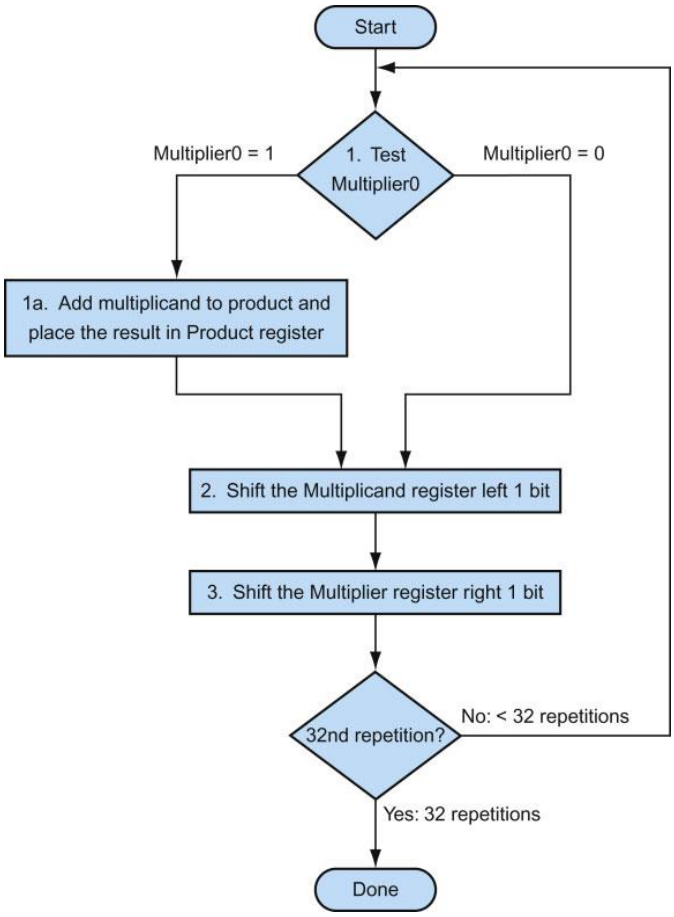


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are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

- Example: (A multiply Algorithm) Use 4-bit numbers to multiple 2_{10} X 3_{10}, or 0010_2 X 0011_2

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001(1)	0000 0010	0000 0000
1	1a: 1 ⇒ 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 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000(0)	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000(0)	0001 0000	0000 0110
4	1: 0 ⇒ 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

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

Signed Multiplication

- Signed Multiplication: Convert the multiplier and multiplicand to **positive** numbers and then **remember** the original signed.
- The algorithm should then be run for 31 iterations, leaving the signs out of the calculation.

Faster Multiplication

- Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.
- Uses multiple adders: Cost / Performance tradeoff

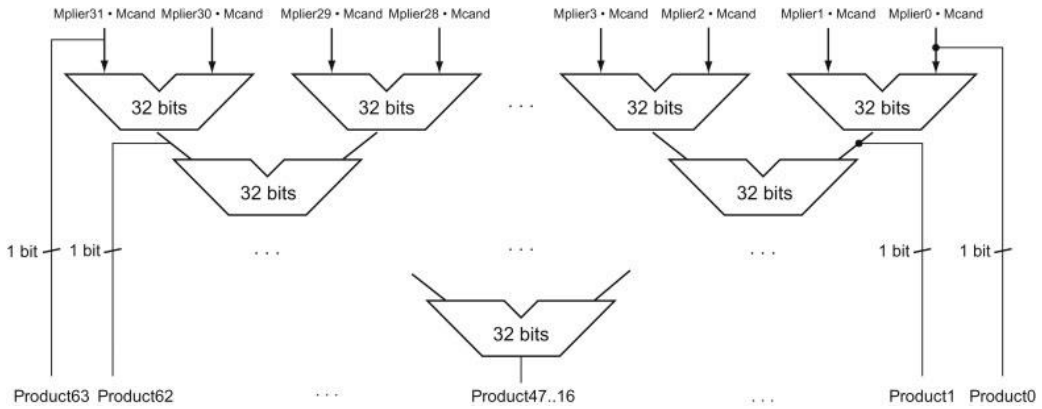


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware "**unrolls the loop**" to use **31 adders** and then organizes them to minimize delay.

Multiply in MIPS

- MIPS provides a separate pair of 32-bit register to contain the 64-bit product, called **Hi** and **Lo**.
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- MIPS Instructions
  - mult rs, rt / multu rs, rt
    - 64-bit product in HI / LO
  - mfhi rd / mflo rd
    - Move from HI / LO to rd
    - Can test HI value to see if product overflows 32 bits

Summary

- Multiplication hardware simply **shifts** and **adds**.
- Compiler even use shift instructions for multiplication by powers of 2.
- With much more hardware we can do the adds in parallel, and do them much faster.

3.4 Division 189

- Divide's two operands, called the dividend and divisor, and result, called the quotient, are accompanied by a second result, called the remainder.
- Here is another way to express the relationship between the components:

  Dividend = Quotient X Divisor + Remainder

$$
\begin{array}{rll}
 & 1001_{ten} & \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)\,1001010_{ten}} & \text{Dividend} \\
 & \underline{-1000} & \\
 & 10 & \\
 & 101 & \\
 & 1010 & \\
 & \underline{-1000} & \\
 & 10_{ten} & \text{Remainder}
\end{array}
$$

A Division Algorithm and Hardware

- In Figure 3.8
  - Quotient: 32-bit Quotient register set 0, shift it left 1 bit each step
  - Divisor: Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with divisor placed in the **left half** of the 64-bit Divisor register and shift it right 1 bit each step to align it with dividend.
  - Remainder register is initialized with the **dividend**.

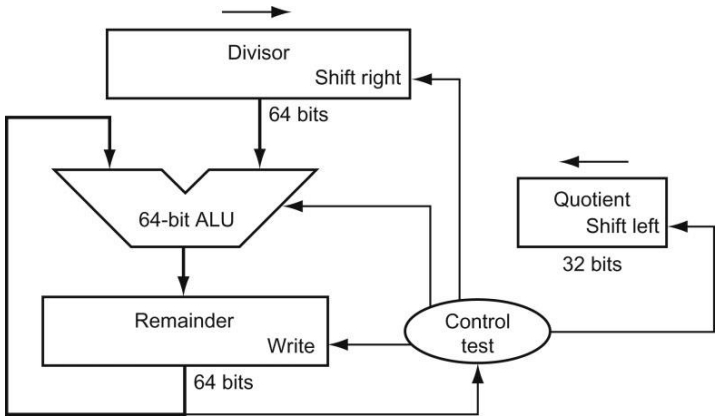


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

- In Figure 3.9, A division algorithm
  - It must first subtract the divisor in step 1
  - If the result is positive, the divisor was smaller or equal to the dividend, so we generate a **1** in the quotient (step 2a).
  - If the result is negative, the next step is to restore the original value by add the divisor back to the remainder and generate a **0** in the quotient (step 2b).
  - The divisor is shifted right and then we iterate again.

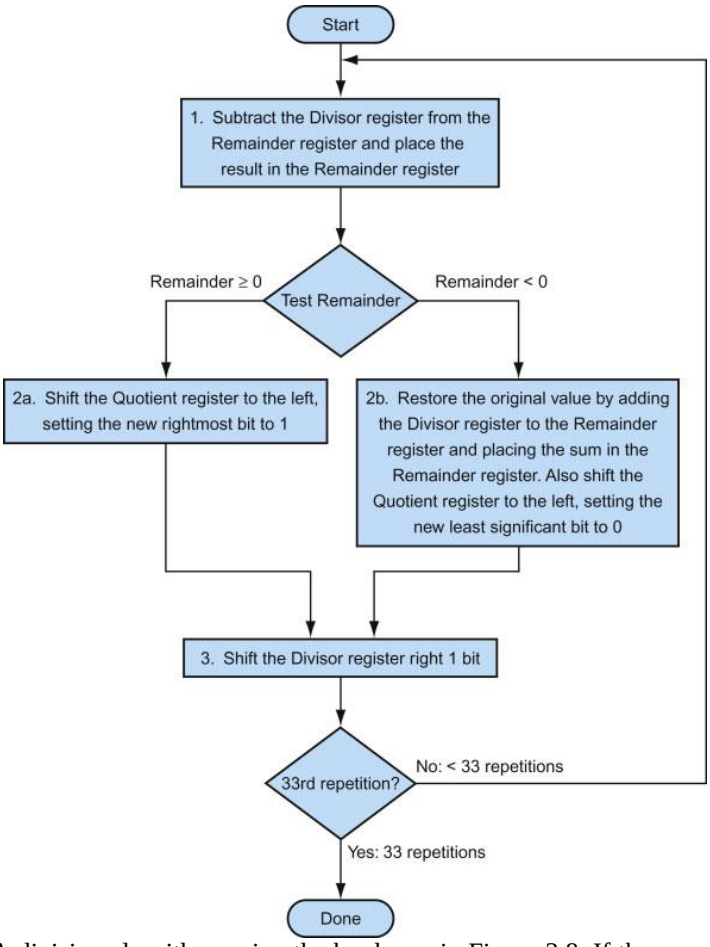


FIGURE 3.9 A division algorithm, using the hardware in Figure 3.8. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

- Example: (A Divide Algorithm) Using a 4-bit version of the algorithm to save pages, let's try dividing 7 by 2 or 0000 0111 by 0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 ⇒ +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	①111 0111
	2b: Rem < 0 ⇒ +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	①111 1111
	2b: Rem < 0 ⇒ +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	⓪000 0011
	2a: Rem ≥ 0 ⇒ 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	⓪000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

Signed Division

- The rule: the dividend and remainder must have the **same** signs, no matter what the signs of the divisor and quotient.
- Signed division algorithm: **Negates** the quotient if signs of the operands are opposite and makes the sign of the nonzero remainder **match** the dividend

Divide in MIPS

- MIPS provides a separate pair of 32-bit Hi and 32-bit Lo registers for both multiply and divide and.
  - **Hi**: 32-bit remainder
  - **Lo**: 32-bit quotient
- MIPS Instructions
  - div rs, rt / divu rs, rt
    - 64-bit product in HI / LO
  - mfhi rd / mflo rd
    - Move from HI / LO to rd

Summary

- The common hardware support for multiply and divide allow MIPS to provide a single pair of 32-bit registers (**Hi** and **Lo**) that are used both for multiply and divide.

- Figure 3.12 summarizes the enhancements to the MIPS architecture.

MIPS assembly language

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 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $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
Data transfer	load word	lw	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw	$s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half unsigned	lhu	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh	$s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte unsigned	lbu	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb	$s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store conditional word	sc	$s1,20($s2)	Memory[$s2+20]=$s1; $s1=0 or 1	Store word as 2nd half atomic swap
	load upper immediate	lui	$s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	AND	AND	$s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	OR	OR	$s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	NOR	NOR	$s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	AND immediate	ANDi	$s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND with constant
	OR immediate	ORi	$s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR with constant
	shift left logical	sll	$s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl	$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Condi-tional branch	branch on equal	beq	$s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne	$s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Compare < constant; two's complement
	set less than unsigned	sltu	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Compare less than; natural numbers
	set less than immediate unsigned	sltiu	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
Uncondi-tional jump	jump	j	2500	go to 10000	Jump to target address
	jump register	jr	$ra	go to $ra	For switch, procedure return
	jump and link	jal	2500	$ra = PC + 4; go to 10000	For procedure call

FIGURE 3.12 MIPS core architecture. The memory and registers of the MIPS architecture are not included for space reasons, but this section added the Hi and Lo registers to support multiply and divide. MIPS machine language is listed in the MIPS Reference Data Card at the front of this book.

3.5 Floating Point 196

- Representation for non-integral numbers
- Including very small and very large numbers
- Scientific notation: A single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a normalized number.
  - Normalized: -2.34×10^{56}
  - Not normalized: $+0.002 \times 10^{-4}$ and $+987.02 \times 10^{9}$
- The programming language C use data type names: float and double
- Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is:

$$\pm 1.xxxxxxx_2 \times 2^{yyyy}$$

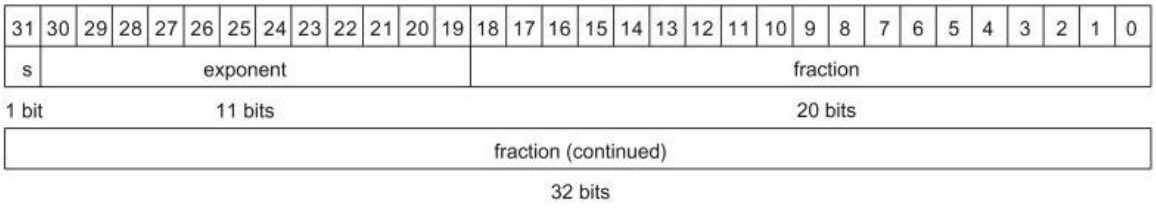
Floating-Point Representation

- IEEE 754 Floating Point Standard
  - Single precision floating point (32-bit)
  - Double precision floating point (64-bit)
- In general, floating-point numbers are of the form

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

  - S: sign bit ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)
  - Normalize significand: $1.0 \le |\text{significand}| < 2.0$
    - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
    - Significand is Fraction with the "1." restored
  - Exponent: excess representation: actual exponent + Bias
    - Exponent is unsigned
    - Single: Bias = **127**; Double: Bias = **1203**

Single precision floating point (32-bit)



Double precision floating point (64-bit)

- Single Precision Range
  - Exponents 0000 0000 and 1111 1111 reserved
  - Smallest value
    - Exponent: 0000 0001 ⇒ actual exponent = 1 – 127 = –126
    - Fraction: 000…00 (23bits) ⇒ significand = 1.0
    - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
  - Largest value
    - exponent: 1111 1110 ⇒ actual exponent = 254 – 127 = +127
    - Fraction: 111…11 (23bits) ⇒ significand ≈ 2.0
    - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$
- Double Precision Range
  - Exponents 000 0000 0000 and 111 1111 1111 reserved
  - Smallest value
    - Exponent: 000 0000 0001 ⇒ actual exponent = 1 – 1023 = –1022
    - Fraction: 000…00 (52bits) ⇒ significand = 1.0
    - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
  - Largest value
    - Exponent: 111 1111 1110 ⇒ actual exponent = 2046 – 1023 = +1023
    - Fraction: 111…11(52 bits) ⇒ significand ≈ 2.0
    - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$
- IEEE 754 makes the leading 1-bit of normalized binary numbers implicit. Hence, the number is actually **24** bits long in in single precision (implied 1 and a 23-bit fraction), and **53** bit long in double precision (1 + 52).

- Figure 3.13 shows IEEE 754 encoding of single and double precision numbers

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 EEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 222. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

- Example: (Floating-Point Representation) Show the IEEE 754 binary representation of the number –0.75 in single and double precision
  - The number –0.75 is also
    $-3_{ten} / 4_{ten}$ or $-3_{ten} / 2^2_{ten}$
  - It is also represented by the binary fraction
    $-11_{two}/2^2_{ten}$ or -0.11_{two}
  - In scientific notation, the value, it is
    $-1.1_{two} \times 2^{-1}$
  - The general form for single and double precision numbers
    $$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$
  - Therefore, the value is
    $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = $0111\ 1110_2$
    - Double: –1 + 1023 = 1022 = $011\ 1111\ 1110_2$
  - Single precision binary representation of -0.75 is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

  - Double precision binary representation of -0.75 is:

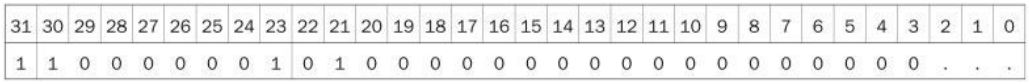
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

- Example: (Converting Binary to Decimal Floating Point) What decimal number is represented by this single precision float:



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

  - The general form for single and double precision numbers

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

    - S = 1
    - Fraction = $01000\ldots00_2$
    - Exponent = $1000\ 0001_2 = 129$

  - x $= (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$
    $= (-1) \times 1.25 \times 2^2$
    $= \mathbf{-5.0}$

- Denormal Number
  - Exponent = 000...0 $\Rightarrow$ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

  - Smaller than normal numbers
    - allow for gradual underflow, with diminishing precision
- Infinities and NaNs
  - Exponent = 111...1, Fraction = 000...0
    - ±Infinity
    - Can be used in subsequent calculations, avoiding need for overflow check
  - Exponent = 111...1, Fraction ≠ 000...0
    - Not-a-Number (NaN)
    - Indicates illegal or undefined result (e.g., 0.0 / 0.0)
    - Can be used in subsequent calculations

Floating-Point Addition

- Example: (Binary Floating-Point Addition) Add the number 0.5 and -04375 in binary.
  - Now consider a 4-digit binary example
    - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
  - 1. Align binary points
    - Shift the smaller number to right until its exponent would match the larger exponent
    - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
  - 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 overflow / underflow
  - 4. Round and renormalize if necessary
    - **$1.000_2 \times 2^{-4}$** (no change) = **0.0625**

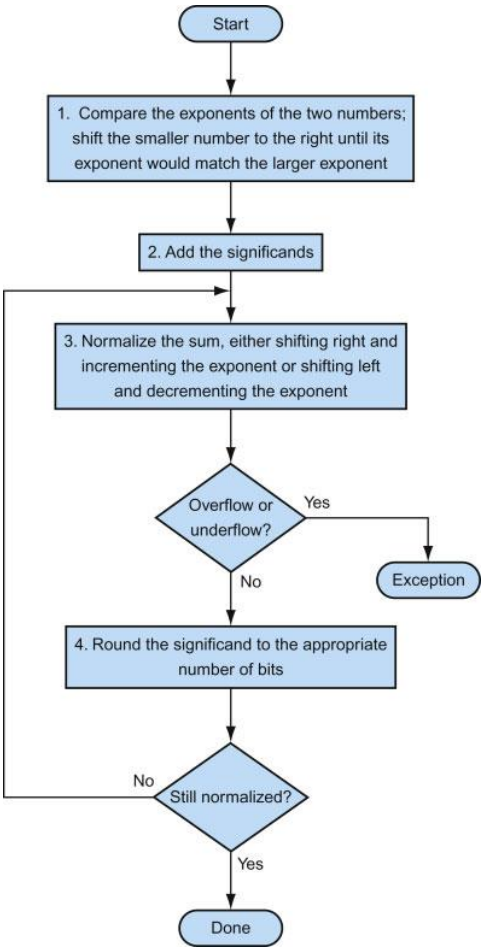


FIGURE 3.14 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Floating-Point Multiplication

- Example: (Binary Floating-Point Multiplication) Multiply the number 0.5 and -04375 in binary.
  - Now consider a 4-digit binary example
    - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
  - 1. Add exponents
    - Unbiased: $-1 + -2 = -3$
    - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127 = 124$
  - 2. Multiply significands
    - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
  - 3. Normalize result & check for overflow / underflow
    - $1.110_2 \times 2^{-3}$ (no change) with no overflow /underflow
  - 4. Round and renormalize if necessary
    - $1.110_2 \times 2^{-3}$ (no change)
  - 5. Determine sign: positive × negative ⇒ negative
    - $\mathbf{-1.110_2 \times 2^{-3} = -0.21875}$

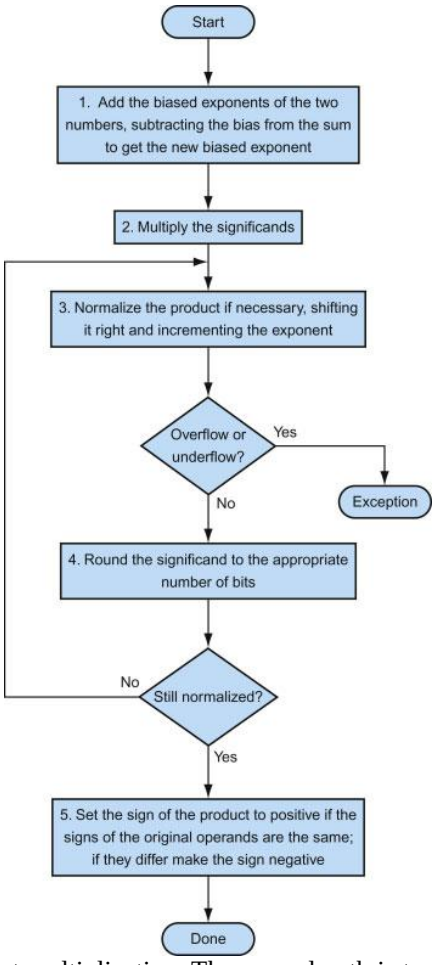


FIGURE 3.16 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Floating-Point Instructions in MIPS

- MIPS supports the IEEE 754 single precision and double precision formats with these instructions:
  - Floating-point addition: single (add.s) and double (add.d)
    - e.g., add.s $f0, $f4, $f6 # $f2 = $f4 + $f6
  - Floating-point subtraction: single (sub.s) and double (sub.d)
    - e.g., sub.d $f2, $f4, $f6 # $f2 = $f4 - $f6
  - Floating-point multiplication: single (mul.s) and double (mul.d)
    - e.g., mul.s $f2, $f4, $f6 # $f2 = $f4 X $f6
  - Floating-point division: single (div.s) and double (div.d)
    - e.g., div.d $f2, $f4, $f6 # $f2 = $f4 / $f6
  - Floating-point comparison: single (c.x.s) and double (c.x.d)
    Where x may be equal (eq), not equal (neq), less than (lt), less than or equal (le), greater than (gt), greater than or equal (qe)
    - e.g., c.lt.s $f2, $f4 # if ($f2 < $f4) cond = 1; else cond = 0
  - Floating-point branch: true (bclt) and false (bclf)
    - e.g., bclt 25 # if (cond == 1) go to PC + 4 + 100

- Floating-point hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate floating-point registers
  - 32 single-precision: $f0, $f1, …, $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …, $f30/$f31
- Floating-point load and store instructions
  - Load word coprocessor 1 (lwc1), store word coprocessor 1 (swc1)
    - e.g., lwc1 $f1, 100($s2) # $f1 = Memory [$s2 + 100]

Summary

- IEEE 754 standard floating-point representation

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

- It almost always an **approximation** of the real number.

3.10 Concluding Remarks 232

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
AND	AND	R	move from system control (EPC)	mfc0	R
AND immediate	ANDi	I	floating-point add single	add.s	R
OR	OR	R	floating-point add double	add.d	R
OR immediate	ORi	I	floating-point subtract single	sub.s	R
NOR	NOR	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
load linked (*atomic update*)	ll	I	branch on floating-point false	bc1f	I
store cond. (*atomic update*)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

FIGURE 3.26 The MIPS instruction set. This book concentrates on the instructions in the left column. This information is also found in columns 1 and 2 of the MIPS Reference Data Card at the front of this book.

Remaining MIPS-32	Name	Format	Pseudo MIPS	Name	Format
exclusive or (*rs* ⊕ *rt*)	xor	R	absolute value	abs	rd,rs
exclusive or immediate	xori	I	negate *(signed or unsigned)*	negs	rd,rs
shift right arithmetic	sra	R	rotate left	rol	rd,rs,rt
shift left logical variable	sllv	R	rotate right	ror	rd,rs,rt
shift right logical variable	srlv	R	multiply and don't check oflw *(signed or uns.)*	muls	rd,rs,rt
shift right arithmetic variable	srav	R	multiply and check oflw *(signed or uns.)*	mulos	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder *(signed or unsigned)*	rems	rd,rs,rt
load byte	lb	I	load immediate	li	rd,imm
load word left *(unaligned)*	lwl	I	load address	la	rd,addr
load word right *(unaligned)*	lwr	I	load double	ld	rd,addr
store word left *(unaligned)*	swl	I	store double	sd	rd,addr
store word right *(unaligned)*	swr	I	unaligned load word	ulw	rd,addr
load linked (*atomic update*)	ll	I	unaligned store word	usw	rd,addr
store cond. (*atomic update*)	sc	I	unaligned load halfword *(signed or uns.)*	ulhs	rd,addr
move if zero	movz	R	unaligned store halfword	ush	rd,addr
move if not zero	movn	R	branch	b	Label
multiply and add (S or uns.)	madds	R	branch on equal zero	beqz	rs,L
multiply and subtract (S or uns.)	msubs	I	branch on compare *(signed or unsigned)*	bxs	rs,rt,L
branch on ≥ zero and link	bgezal	I	(x = lt, le, gt, ge)		
branch on < zero and link	bltzal	I	set equal	seq	rd,rs,rt
jump and link register	jalr	R	set not equal	sne	rd,rs,rt
branch compare to zero	bxz	I	set on compare *(signed or unsigned)*	sxs	rd,rs,rt
branch compare to zero likely	bxzl	I	(x = lt, le, gt, ge)		
(x = lt, le, gt, ge)			load to floating point (s or d)	l.f	rd,addr
branch compare reg likely	bxl	I	store from floating point (s or d)	s.f	rd,addr
trap if compare reg	tx	R			
trap if compare immediate	txi	I			
(x = eq, neq, lt, le, gt, ge)					
return from exception	rfe	R			
system call	syscall	I			
break *(cause exception)*	break	I			
move from FP to integer	mfc1	R			
move to FP from integer	mtc1	R			
FP move (s or d)	mov.f	R			
FP move if zero (s or d)	movz.f	R			
FP move if not zero (s or d)	movn.f	R			
FP square root (s or d)	sqrt.f	R			
FP absolute value (s or d)	abs.f	R			
FP negate (s or d)	neg.f	R			
FP convert (w, s, or d)	cvt.f.f	R			
FP compare un (s or d)	c.xn.f	R			

FIGURE 3.27 Remaining MIPS-32 and Pseudo MIPS instruction sets. *f* means single (s) or double (d) precision floating-point instructions, and *s* means signed and unsigned (u) versions. MIPS-32 also has FP instructions for multiply and add/sub (madd.f/ msub.*f*), ceiling (ceil.*f*), truncate (trunc.*f*), round (round.*f*), and reciprocal (recip.*f*). The underscore represents the letter to include to represent that datatype.