Floating Point Numbers

- We need a way to represent a wide range of numbers
  - numbers with fractions, e.g., 3.1416
  - large number:

    $$976{,}000{,}000{,}000{,}000 = 9.76 \times 10^{14}$$

  - small number:

    $$0.0000000000000976 = 9.76 \times 10^{-14}$$

- Representation:
  - sign, exponent, significand:

    $$(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$$

  - more bits for significand gives more accuracy
  - more bits for exponent increases range

Scientific Notation

- Scientific notation:
  - $0.525 \times 10^5 = 5.25 \times 10^4 = 52.5 \times 10^3$
  - 5.25×10^4 is in *normalized* scientific notation.
    - position of decimal point fixed
    - leading digit non-zero
- Binary numbers
  - $5.25 = 101.01 = 1.0101 \times 2^2$
  - Binary point
    - multiplication by 2 moves the point to the left
    - division by 2 moves the point to the right
  - Known as *floating point format*.

Binary to Decimal Conversion

Binary $(-1)^S (1.b_1b_2b_3b_4) \times 2^E$

Decimal $(-1)^S \times (1 + b_1\times2^{-1} + b_2\times2^{-2} + b_3\times2^{-3} + b_4\times2^{-4}) \times 2^E$

Example: -1.1100×2^{-2} (binary) $= - (1 + 2^{-1} + 2^{-2}) \times2^{-2}$

$= - (1 + 0.5 + 0.25)/4$

$= - 1.75/4$

$= - 0.4375$ (decimal)

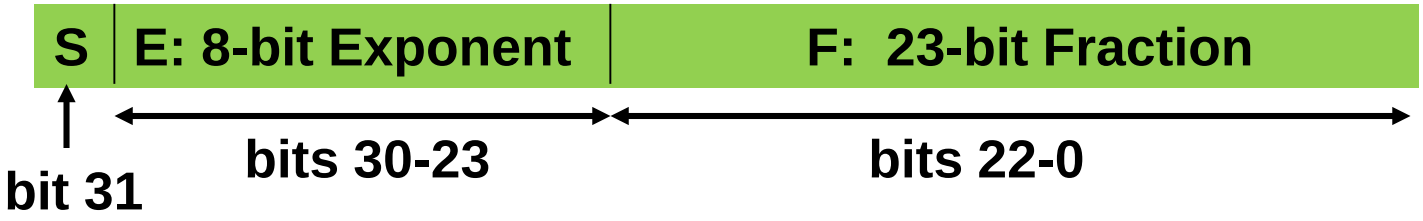
Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

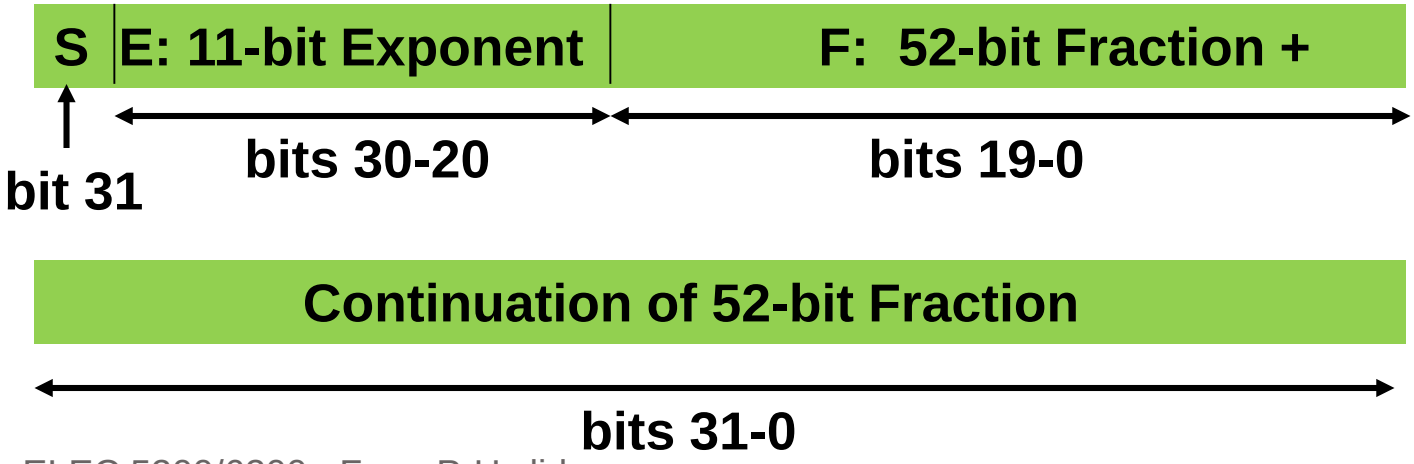
IEEE Std. 754 Floating-Point Format

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

Single-Precision



Double-Precision



IEEE 754 floating-point standard

- Represented value = $(-1)^{sign} \times (1+F) \times 2^{exponent - bias}$
- Exponent is "biased" (excess-K format) to make sorting easier
  - bias of 127 for single precision and 1023 for double precision
  - E values in [1 .. 254] (0 and 255 reserved)
  - Range = 2^{-126} to 2^{+127} (10^{38} to 10^{+38})
- Significand in sign-magnitude, normalized form
  - Significand = (1 + F) = $1.b_{-1}b_{-1}b_{-1}\ldots b_{-23}$
  - Suppress storage of leading 1
- Overflow: Exponent requiring more than 8 bits. Number can be positive or negative.
- Underflow: Fraction requiring more than 23 bits. Number can be positive or negative.

IEEE 754 floating-point standard

- Example:
  - Decimal: -5.75 = - (4 + 1 + ½ + ¼)
  - Binary: -101.11 = -1.0111 x 2^2
  - Floating point: exponent = 129 = 10000001
  - IEEE single precision:
    `11000000101110000000000000000000`

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 10111111001000…00
- Double: 10111111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  1100000010100000...00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Examples

Biased exponent (0-255), bias 127 (01111111) to be subtracted

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}$

$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}$

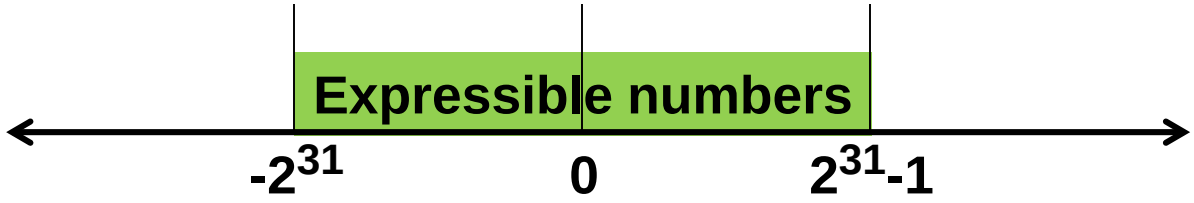
$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}$

$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$

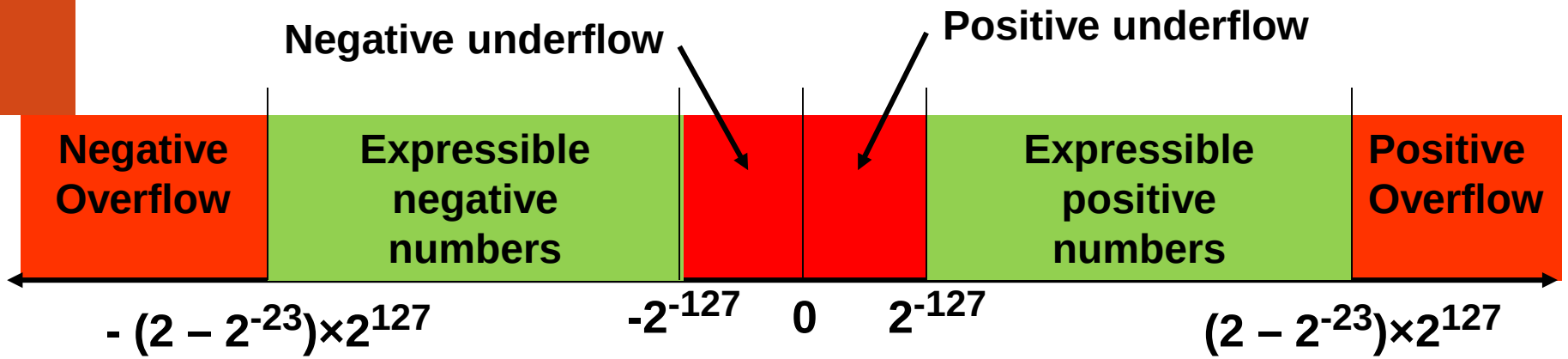
0.5
0.125
0.0078125
0.6328125

Numbers in 32-bit Formats

- Two's complement integers



- Floating point numbers



- Ref: W. Stallings, Computer Organization and Architecture, Sixth Edition, Upper Saddle River, NJ: Prentice-Hall.

IEEE 754 Special Codes

Zero **S 00000000 00000000000000000000000**

- $\pm 1.0 \times 2^{-127}$
- Smallest positive number in single-precision IEEE 754 standard.
- Interpreted as **positive/negative zero.**
- Exponent less than -127 is **positive underflow (**regard as zero).

Infinity **S 11111111 00000000000000000000000**

- $\pm 1.0 \times 2^{128}$
- Largest positive number in single-precision IEEE 754 standard.
- Interpreted as $\pm \infty$
- If true exponent = 128 and fraction ≠ 0, then the number is greater than ∞. It is called **"not a number"** or **NaN** (interpret as ∞).

Addition and Subtraction

- Addition/subtraction of two floating-point numbers:

Example: 2×10^3 + 3×10^4

Align mantissas: 0.2×10^4 + 3×10^4 = 3.2×10^4

- General Case:

$$m_1 \times 2^{e_1} \pm m_2 \times 2^{e_2} = (m_1 \pm m_2 \times 2^{e_2 - e_1}) \times 2^{e_1} \quad \text{for } e_1 > e_2$$

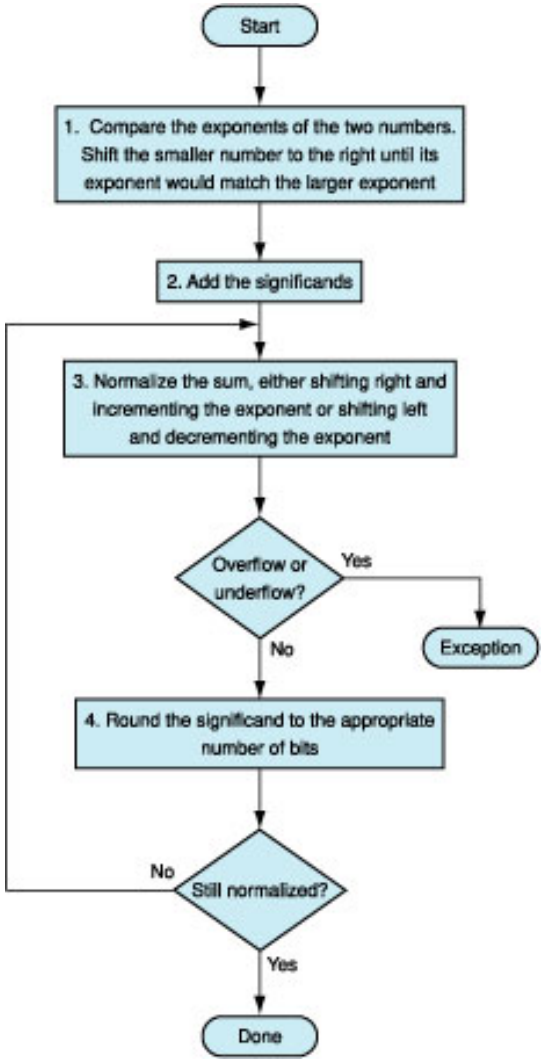
$$= (m_1 \times 2^{e_1 - e_2} \pm m_2) \times 2^{e_2} \quad \text{for } e_2 > e_1$$

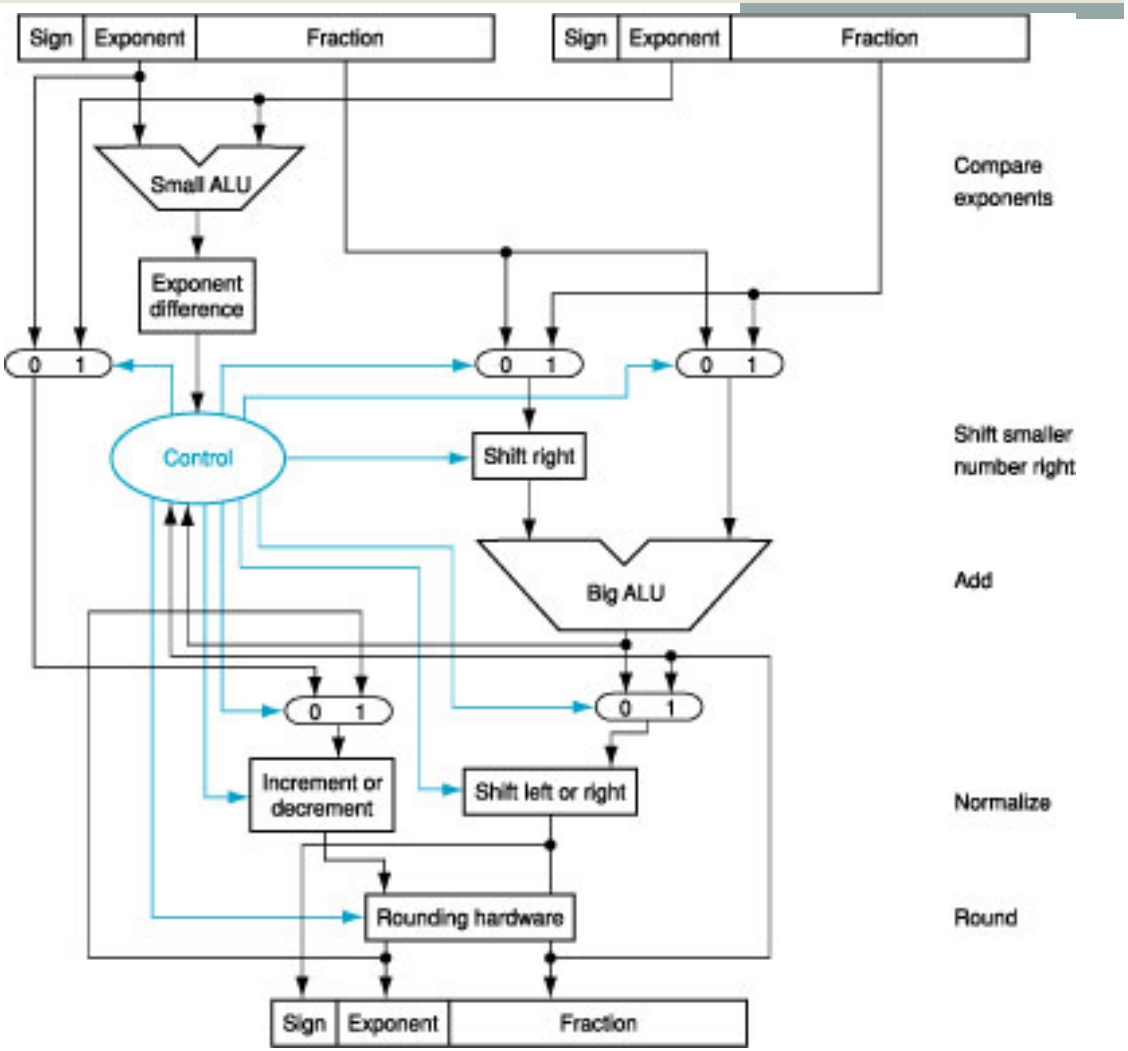
- Shift smaller mantissa right by $| e_1 - e_2 |$ bits to align the mantissas.

Addition/Subtraction Algorithm

0. Zero check
   - Change the sign of subtrahend
   - If either operand is 0, the other is the result
1. Significand alignment: right shift smaller significand until two exponents are identical.
2. Addition: add significands and report exception if overflow occurs.
3. Normalization
   - Shift significand bits to normalize.
   - report overflow or underflow if exponent goes out of range.
4. Rounding

FP Add/Subtract (PH Text Figs. 3.16/17)





Example

- Subtraction: $0.5_{ten} - 0.4375_{ten}$
- Step 0: Floating point numbers to be added

  $1.000_{two} \times 2^{-1}$ and $-1.110_{two} \times 2^{-2}$
- Step 1: Significand of lesser exponent shifted right until exponents match

  $-1.110_{two} \times 2^{-2} \longrightarrow -0.111_{two} \times 2^{-1}$
- Step 2: Subtract significands, $1.000_{two} + (-0.111_{two})$

  Result is $0.001_{two} \times 2^{-1}$
- Step 3: Normalize, $1.000_{two} \times 2^{-4}$

  No overflow/underflow since $127 \geq$ exponent ≥ -126
- Step 4: Rounding, no change since the sum fits in 4 bits.

  $1.000_{two} \times 2^{-4} = (1+0)/16 = 0.0625_{ten}$

FP Multiplication: Basic Idea

$$(m_1 \times 2^{e1}) \times (m_2 \times 2^{e2}) = (m_1 \times m_2) \times 2^{e1+e2}$$

- Separate signs
- Add exponents
- Multiply significands
- Normalize, round, check overflow
- Replace sign

FP Mult. Illustration

- Multiply 0.5_{ten} and -0.4375_{ten} (answer = -0.21875_{ten}) or
- Multiply $1.000_{two} \times 2^{-1}$ and $-1.110_{two} \times 2^{-2}$
- Step 1: Add exponents

$$-1 + (-2) = -3$$

- Step 2: Multiply significands

```
  1.000
 ×1.110
  0000
 1000
1000
1000
1110000        Product is 1.110000
```

* Step 3: Normalization: If necessary, shift significand right and increment exponent.

  Normalized product is 1.110000×2^{-3}

  Check overflow/underflow: 127 ≥ exponent ≥ -126

* Step 4: Rounding: 1.110×2^{-3}
* Step 5: Sign: Operands have opposite signs, Product is -1.110×2^{-3}

  Decimal value = - (1+0.5+0.25)/8 = -0.21875_{ten}

FP Division: Basic Idea

- Separate sign.
- Check for zeros and infinity.
- Subtract exponents.
- Divide significands.
- Normalize/overflow/underflow.
- Rounding.
- Replace sign.

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- 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
- FP load and store instructions
  - lwc1, ldc1, swc1, sdc1
    - e.g., ldc1 $f8, 32($sp)

FP Instructions in MIPS

- Single-precision arithmetic
  - add.s, sub.s, mul.s, div.s
    - e.g., add.s $f0, $f1, $f6
- Double-precision arithmetic
  - add.d, sub.d, mul.d, div.d
    - e.g., mul.d $f4, $f4, $f6
- Single- and double-precision comparison
  - c.*xx*.s, c.*xx*.d (*xx* is eq, lt, le, …)
  - Sets or clears FP condition-code bit
    - e.g., c.lt.s $f3, $f4
- Branch on FP condition code true or false
  - bc1t, bc1f
    - e.g., bc1t TargetLabel

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)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

MIPS Floating Point Instructions

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	lwc1 $f1,100($s2)	$f1 = Memory[$s2 + 100]	32-bit data to FP register
	store word copr. 1	swc1 $f1,100($s2)	Memory[$s2 + 100] = $f1	32-bit data to memory
Condi-tional branch	branch on FP true	bc1t 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bc1f 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

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)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 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

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have "underflow"
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields "infinity"
  - zero divide by zero yields "not a number"
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i)	FIADDP mem/ST(i)	FICOMP	FPATAN
FISTP mem/ST(i)	FISUBRP mem/ST(i)	FIUCOMP	F2XMI
FLDPI	FIMULP mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	FIDIVRP mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FPSIN
	FRNDINT		FYL2X

- Optional variations
  - I: integer operand
  - P: pop operand from stack
  - R: reverse operand order
  - But not all combinations allowed

FSTSW:
Move FP status to integer unit for conditional jump.

Instruction	Operands	Comment
FADD		Both operands in stack; result replaces top of stack.
FADD	ST(i)	One source operand is *i*th register below the top of stack; result replaces the top of stack.
FADD	ST(i), ST	One source operand is the top of stack; result replaces *i*th register below the top of stack.
FADD	mem32	One source operand is a 32-bit location in memory; result replaces the top of stack.
FADD	mem64	One source operand is a 64-bit location in memory; result replaces the top of stack.

FIGURE 3.21 The variations of operands for floating-point add in the x86.

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit single precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

FIGURE 3.22 The SSE/SSE2 floating-point instructions of the x86. xmm means one operand is a 128-bit SSE2 register, and mem/xmm means the other operand is either in memory or it is an SSE2 register. We use the curly brackets {} to show optional variations of the basic operations: {SS} stands for Scalar Single precision floating point, or one 32-bit operand in a 128-bit register; {PS} stands for Packed Single precision floating point, or four 32-bit operands in a 128-bit register; {SD} stands for Scalar Double precision floating point, or one 64-bit operand in a 128-bit register; {PD} stands for Packed Double precision floating point, or two 64-bit operands in a 128-bit register; {A} means the 128-bit operand is aligned in memory; {U} means the 128-bit operand is unaligned in memory; {H} means move the high half of the 128-bit operand; and {L} means move the low half of the 128-bit operand.

Chapter Three Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two's complement
  - IEEE 754 floating point
- Computer instructions determine "meaning" of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines
- Algorithm choice is important and may lead to hardware optimizations for both space and time (e.g., multiplication)
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent