ECE-4300: Chapter-3: Arithmetic for Computers FPU

Prof: Mohamed El-Hadedy

Email: mealy@cpp.edu

Office: 909-869-2594

Floating Point

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called reals in mathematics.

Examples for reals:

1) pi: 3.14159265 (decimal)
2) e: 2.71828
3) 0.000000001 → 1.0 x 10^{-9}
4) 3,155,760,000 or 3.15576 x 10^{9}

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

normalized

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types float and double in C

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)

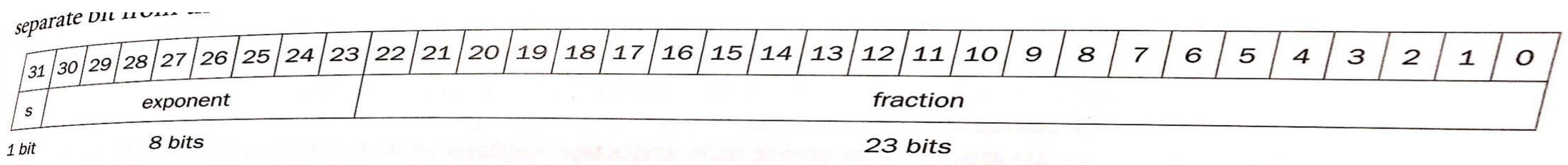
Floating-Point Representation

Precision	Base	Sign	Exponent	Significand
Single Precision	2	1	8	23+1
Double Precision	2	1	11	52+1

A finite number can also represented by four integers components, a sign (s), a base (b), a significand (m), and an exponent (e). Then the numerical value of the number is evaluated as

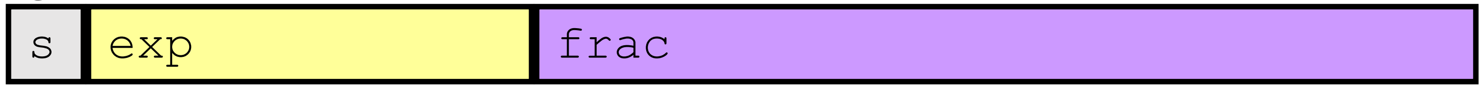
$(-1)^s \times m \times b^e$ ——— Where $m < |b|$

Single-Precision



Floating Point Representation

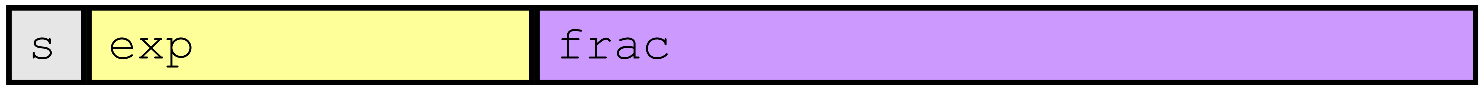
- Numerical Form
  - $-1^{s} M \; 2^{E}$
    - Sign bit s determines whether number is negative or positive
    - Significand M normally a fractional value in range [1.0,2.0).
    - Exponent E weights value by power of two
- Encoding



  - MSB is sign bit
  - exp field encodes E
  - frac field encodes M

Floating Point Precisions

- Encoding



  - MSB is sign bit
  - exp field encodes *E*
  - frac field encodes *M*
- Sizes
  - Single precision: 8 exp bits, 23 frac bits
    - 32 bits total
  - Double precision: 11 exp bits, 52 frac bits
    - 64 bits total
  - Extended precision: 15 exp bits, 63 frac bits
    - Only found in Intel-compatible machines
    - Stored in 80 bits
      - 1 bit wasted

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

- 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
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

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: 101111110100000000000000000000000
- Double: 101111111110100000000000000000000

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: 10111111010000000000000000000000000

- Double: 1011111111110100000000000000000000000

Floating-Point Example (Converting Binary to Decimal Floating Point)

- What number is represented by the single-precision float

  1100000001010000…00

  - S = 1
  - Fraction = $01000…00_2$
  - Exponent = 10000001_2 = 129
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

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

  $= -5.0$

“Normalized” Numeric Values

- Condition
  - exp ≠ 000…0 and exp ≠ 111…1
- Exponent coded as *biased* value

  $E = Exp - Bias$
    - *Exp* : unsigned value denoted by exp
    - *Bias* : Bias value
      - Single precision: 127 (*Exp*: 1…254, *E*: -126…127)
      - Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)
      - in general: *Bias* = 2^{e-1} - 1, where e is number of exponent bits
- Significand coded with implied leading 1

  $M = 1.\text{xxx}\ldots\text{x}_2$
    - xxx…x: bits of frac
    - Minimum when 000…0 (*M* = 1.0)
    - Maximum when 111…1 (*M* = 2.0 – □)
    - Get extra leading bit for “free”

Normalized Encoding Example

- Value

  Float F = 15213.0;

  - $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \text{ X } 2^{13}$
- Significand

  M = $1.\underline{1101101101101}_2$

  frac = $\underline{1101101101101}0000000000_2$
- Exponent

  E = 13

  $Bias$ = 127

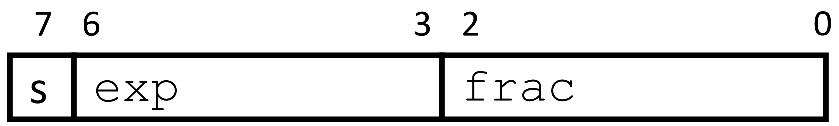
  Exp = 140 = 10001100_2

Floating Point Representation (Class 02):

Hex:	4 6 6 D B 4 0 0
Binary:	0100 0110 0110 1101 1011 0100 0000 0000
140:	100 0110 0
15213:	1110 1101 1011 01

Tiny Floating Point Example

- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit.
  - the next four bits are the exponent, with a bias of 7.
  - the last three bits are the frac
- Same General Form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity



Denormal Numbers

- 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
- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 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}

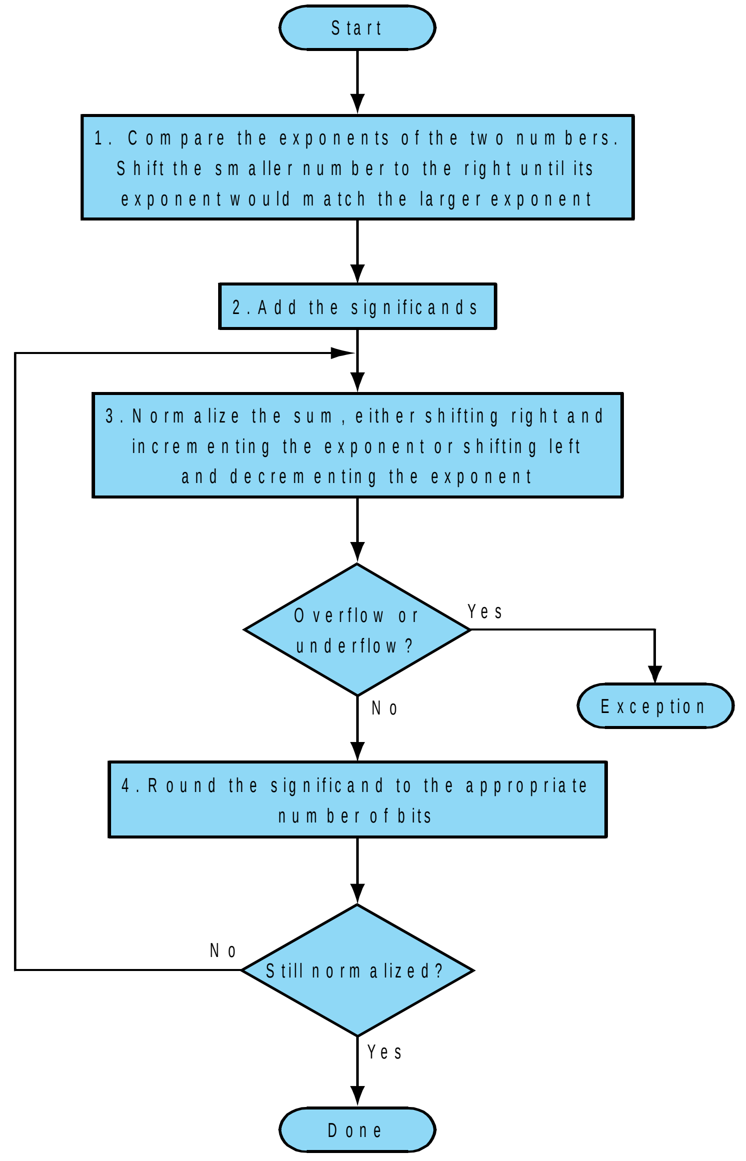
Floating-Point Addition

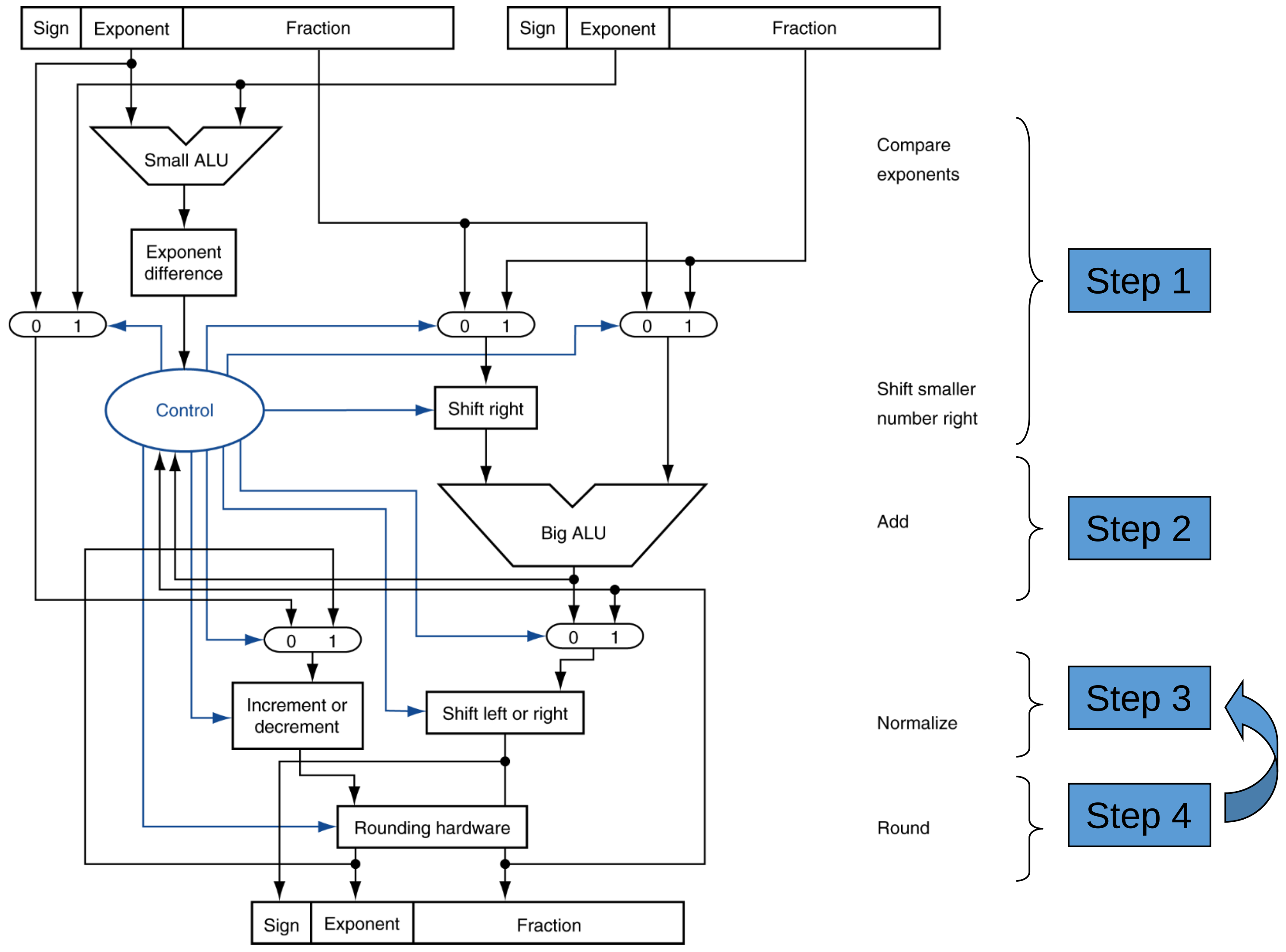
- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) $= 0.0625$

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

FP Adder Hardware


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done



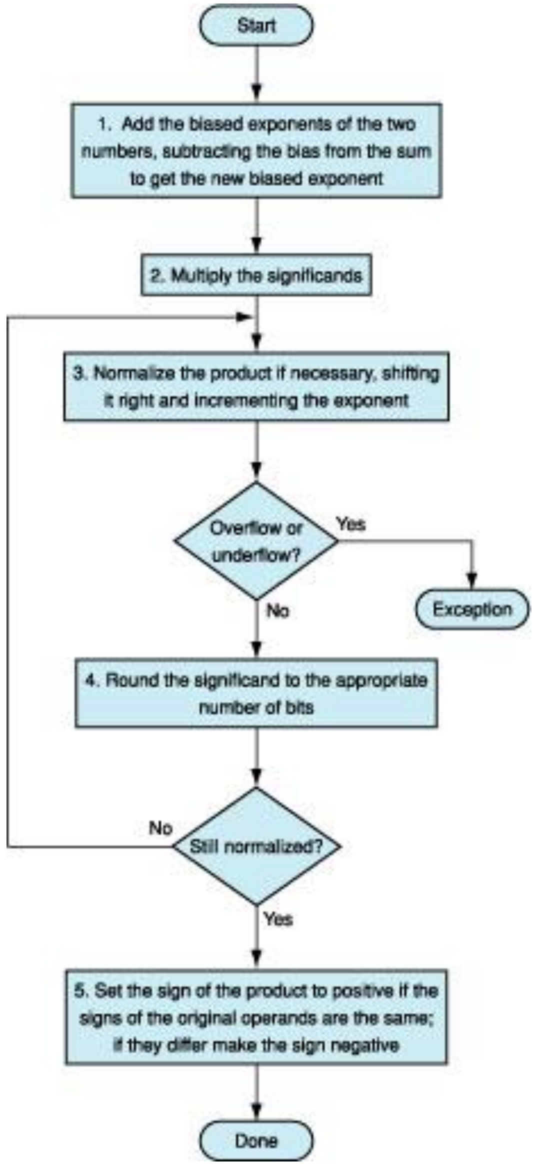
Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 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 Multiplication

- 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: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: $+\text{ve} \times -\text{ve} \Rightarrow -\text{ve}$
  - $-1.110_2 \times 2^{-3} = -0.21875$

Floating-Point Multiplication



Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

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

Differences between FP and Integer Instructions

- # o Separate set of registers. (f0-f31)
- # o Separate instructions to operate on those registers ..
- # ... for arithmetic (add) ...
- # ... and loads and stores.
- # o Need for format conversion.

Differences between FP and Integer Instructions

## **Registers**

#

## **MIPS Registers**

#

# MIPS has four sets of coprocessor registers.

# The integer (GPR) registers are NOT one of the four sets.

# Each set has 32 registers.

#

# Co-processor 0: Processor and system control.

# Co-processor 1: MIPS-32 floating-point

# Co-processor 2: Reserved for special-purpose designs.

# Co-processor 3: MIPS-64 floating-point

#

Differences between FP and Integer Instructions

## **Coprocessor 1 and The Floating-Point Registers**

#

# Floating point handled by co-processor 1, one of 4 co-processors.

#

# MIPS floating point registers also called co-processor 1 registers.

# MIPS floating point instructions called co-processor 1 instructions.

#

# Registers named f0-f31.

# Each register is 32 bits. (For MIPS-32)

## Coprocessor 1 Instructions

Coprocessor 1 instructions use coprocessor 1 (FP) registers.

# This includes instructions that do FP arithmetic ..

# .. and other types of instructions.

Many coprocessor 1 instructions have "c1" in their names ..

# .. for example, lwc1.

Coprocessor 1 *arithmetic* instructions *do not* have c1 in their names ..

# .. but they include a /completer/ that indicates data type ..

# .. for example, add.d, where ".d" is the completer.

Completers:

".s" Single-Precision Floating Point (32 bits)

# ".d" Double-Precision Floating Point (64 bits)

# ".w" Integer (32 bits)

## Double-Precision Operands

```
#
# In MIPS-32 (including MIPS-I) FP registers are 32 bits.
# Instructions with ".d" operands get each operand from a pair of regs.
# Register numbers must be even.
# :Example:
            add.d $f0, $f2, $f4 # {$f0,$f1} = { $f2, $f3 } + { $f4, $f5 }
            add.d $f0, $f2, $f5 # ILLEGAL in MIPS 32, because f5 is odd.
```

## Immediate Operands and Constant Registers

```
#
# MIPS FP instructions do not take immediate values.
      addi.s $f0, $f1, 2.3 # ILLEGAL, no immediate FP instructions.
# There is no counterpart of integer register 0 ($0 aka r0 aka $zero).
      add.s $f0, $f1, $f2            # f0 has the sum,
```

## Arithmetic Operations

#

# Add double-precision (64-bit) operands.

#

# MIPS: add.d $f0, $f2, $f4

#

## Load and Store

#

# Load double (eight bytes into two consecutive registers).

#

# MIPS: ldc1 $f0, 8($t0)

#

#

## Move Between Register Files (E.g., integer to FP)

```
#
# MIPS:      mtc1 $t0, $f0
## Format Conversion
#
# Convert from one format to another, e.g., integer to double.
#
# MIPS: cvt.d.w $f0, $f2
```

## Floating Point Condition Code Setting

```
#
# Compare and set condition code.
#
# MIPS:      c.gt.d $f0, $f2
#
#
```

## Conditional Branch

```
#
# Branch on floating-point condition.
#
# MIPS: bc1f TARGET  # Branch coprocessor 1 [condition code] false.
```

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

```
## FP Load and Store
# MIPS
#
# Load word in to coprocessor 1
lwc1 $f0, 4($t4)        # $f0 = Mem[ $t4 + 4 ]
#
# Load double in to coprocessor 1
ldc1 $f0, 0($t4)        # $f0 = Mem[ $t4 + 0 ]; $f1 = Mem[ $t4 + 4 ]
#
# Store word from coprocessor 1.
swc1 $f0, 4($t4)      # $f0 = Mem[ $t4 + 4 ]
#
# Store double from coprocessor 1.
sdc1 $f0, 0($t4)      # Mem[ $t4 + 0 ] = $f0; Mem[ $t4 + 4 ] = $f1
```

FP Instructions in MIPS

```
## Move Instructions
# MIPS
#
# Move to coprocessor 1
mtc1 $t0, $f0        # f0 = t0 Note that destination is *second* reg.
#
# Move from coprocessor 1.
mfc1 $t0, $f0
```

FP Instructions in MIPS

```
#
# Data Type Conversion
# Convert between floating-point and integer formats.
# NOTE: Values don't convert automatically, need to use these insn.
# MIPS
#
# To: s, d, w; From: s, d, w
#
# cvt.TO.FROM fd, fs
#
cvt.d.w $f0, $f2              # $f0 = convert_from_int_to_double( $f2 )
```

FP Instructions in MIPS

```
## Setting Condition Codes
# In preparation for a branch, set cond code based on FP comparison.
# MIPS
#
# Compare: fs COND ft
# COND: eq, gt, lt, le, ge
# FMT: s, d
#
# c.COND.FMT fs, ft
# Sets condition code to true or false.
# Condition is false if either operand is not a number.
#
c.lt.d $f0, $f2         # CC = $f0 < $f2
bc1t TARG # Branch if $f0 < $f2
nop
c.ge.d $f0, $f2         # CC = $f0 >= $f2
bc1t TARG2              # Branch if $f0 < $f2
nop
# Reachable?
```

```
## FP Branches
# MIPS
#
# Branch insn specifies whether CC register true or
false.
#
bc1t TARG
nop
# bc1f TARG
nop
```

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
```

FP Example: Array Multiplication

- X = X + Y × 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]
    l.d  $f4, 0($t2)    # $f4 = 8 bytes of x[i][j]
L3: 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
    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
```

Interpretation of Data

The BIG Picture

- 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

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(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

- Need to validate parallel programs under varying degrees of parallelism

x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

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

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 double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
  - c.f. 11111011_2 >>> 2 = 00111110_2 = +62

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

Reference:

- **Book:** D. A. Patterson and J. L. Hennessy, **Computer Organization and Design: The Hardware/ Software Interface**, 5th Edition, San Mateo, CA: Morgan and Kaufmann. ISBN: 1-55860-604-1
- https://www.mips.com/
- https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html
- Professor El-Naga ECE-425 notes