Floating Point Arithmetic

Introduction

- **Objective: To provide hardware support for floating point arithmetic. To understand how to represent floating point numbers in the computer and how to perform arithmetic with them. Also to learn how to use floating point arithmetic in MIPS.**
- **Approximate arithmetic**
  - **Finite Range**
  - **Limited Precision**
- **Topics**
  - **IEEE format for single and double precision floating point numbers**
  - **Floating point addition and multiplication**
  - **Support for floating point computation in MIPS**

Floating Point

- **An IEEE floating point representation consists of**
  - **A Sign Bit (no surprise)**
  - **An Exponent ("times 2 to the what?")**
  - **Mantissa ("Significand"), which is assumed to be 1.xxxxx (thus, one bit of the mantissa is implied as 1)**
  - **This is called a normalized representation**
- **So a mantissa = 0 really is interpreted to be 1.0, and a mantissa of all 1111 is interpreted to be 1.1111**
- **Special cases are used to represent 0, infinity and NaN.**

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 Floating-Point Format

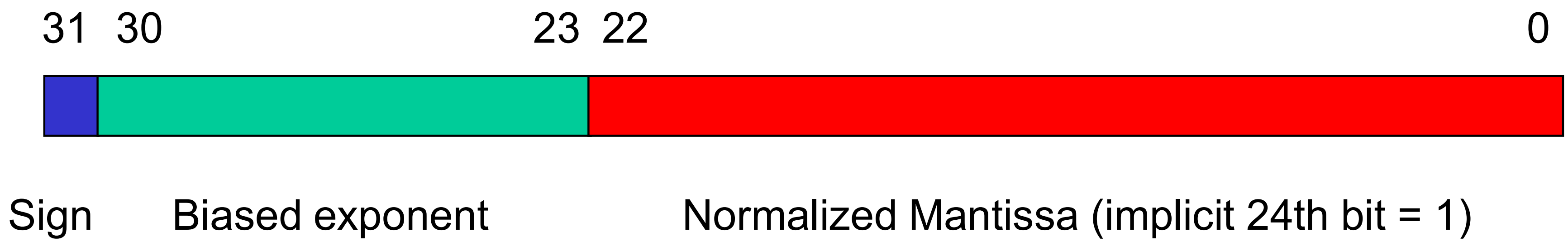
	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction/Mantissa

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

- **S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)**
- **Normalize significand: 1.0 ≤ |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 = 1023**

Representation of Floating Point Numbers

- **IEEE 754 single precision**



$$(-1)^s \times F \times 2^{E-127}$$

Exponent	Mantissa	Object Represented
0	0	0
1-254	anything	FP number
255	0	infinity
255	non-zero	NaN

Basic Technique

- **Represent the decimal in the form +/- $1.xxx_b \times 2^y$**
- **And "fill in the fields"**
  - **Remember biased exponent and implicit "1." mantissa!**
- **Examples:**
  - 0.0: 0 00000000 00000000000000000000000
  - 1.0 (1.0 x 2^0): 0 01111111 00000000000000000000000
  - 0.5 (0.1 binary = 1.0 x 2^-1): 0 01111110 00000000000000000000000
  - 0.75 (0.11 binary = 1.1 x 2^-1): 0 01111110 10000000000000000000000
  - 3.0 (11 binary = 1.1*2^1): 0 10000000 10000000000000000000000
  - -0.375 (-0.011 binary = -1.1*2^-2): 1 01111101 10000000000000000000000
  - 1 10000011 01000000000000000000000 = - 1.01 * 2^4 = -20.0

http://www.math-cs.gordon.edu/courses/cs311/lectures-2003/binary.html

Floating-Point Example

- **What number is represented by the single-precision float**

  11000000101000…00

  - **S = 1**
  - **Fraction = $01000\ldots00_2$**
  - **Fxponent = 10000001_2 = 129**

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

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

  $= -5.0$

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: 1011111101000…00**
- **Double: 1011111111101000…00**

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

Single-Precision Range

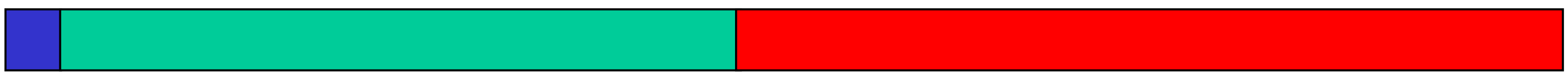
- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
  - **Exponent: 00000001**
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - **Fraction: 000…00 $\Rightarrow$ significand = 1.0**
  - **$\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$**
- **Largest value**
  - **exponent: 11111110**
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - **Fraction: 111…11 $\Rightarrow$ significand ≈ 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
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Representation of Floating Point Numbers

64 63 53 52 0



Sign Biased exponent Normalized Mantissa (implicit 53rd bit)

$$(-1)^s \times F \times 2^{E-1023}$$

Exponent	Mantissa	Object Represented
0	0	0
1-2046	anything	FP number
2047	0	pm infinity
2047	non-zero	NaN

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 Addition

Assuming that the operands are already in the IEEE 754 format, performing floating point addition: Result = X + Y = $(Xm \times 2^{Xe}) + (Ym \times 2^{Ye})$

involves the following steps:

(1) Align binary point:

- **Initial result exponent: the larger of Xe, Ye**
- **Compute exponent difference: Ye - Xe**
- **If Ye > Xe Right shift Xm that many positions to form $Xm\ 2^{Xe-Ye}$**
- **If Xe > Ye Right shift Ym that many positions to form $Ym\ 2^{Ye-Xe}$**

(2) Compute sum of aligned *mantissas*:

i.e $Xm2^{Xe-Ye} + Ym$ or $Xm + Xm2^{Ye-Xe}$

(3) If normalization of result is needed, then a normalization step follows:

- **Left shift result, decrement result exponent (e.g., if result is 0.001xx…) or**
- **Right shift result, increment result exponent (e.g., if result is 10.1xx…)**

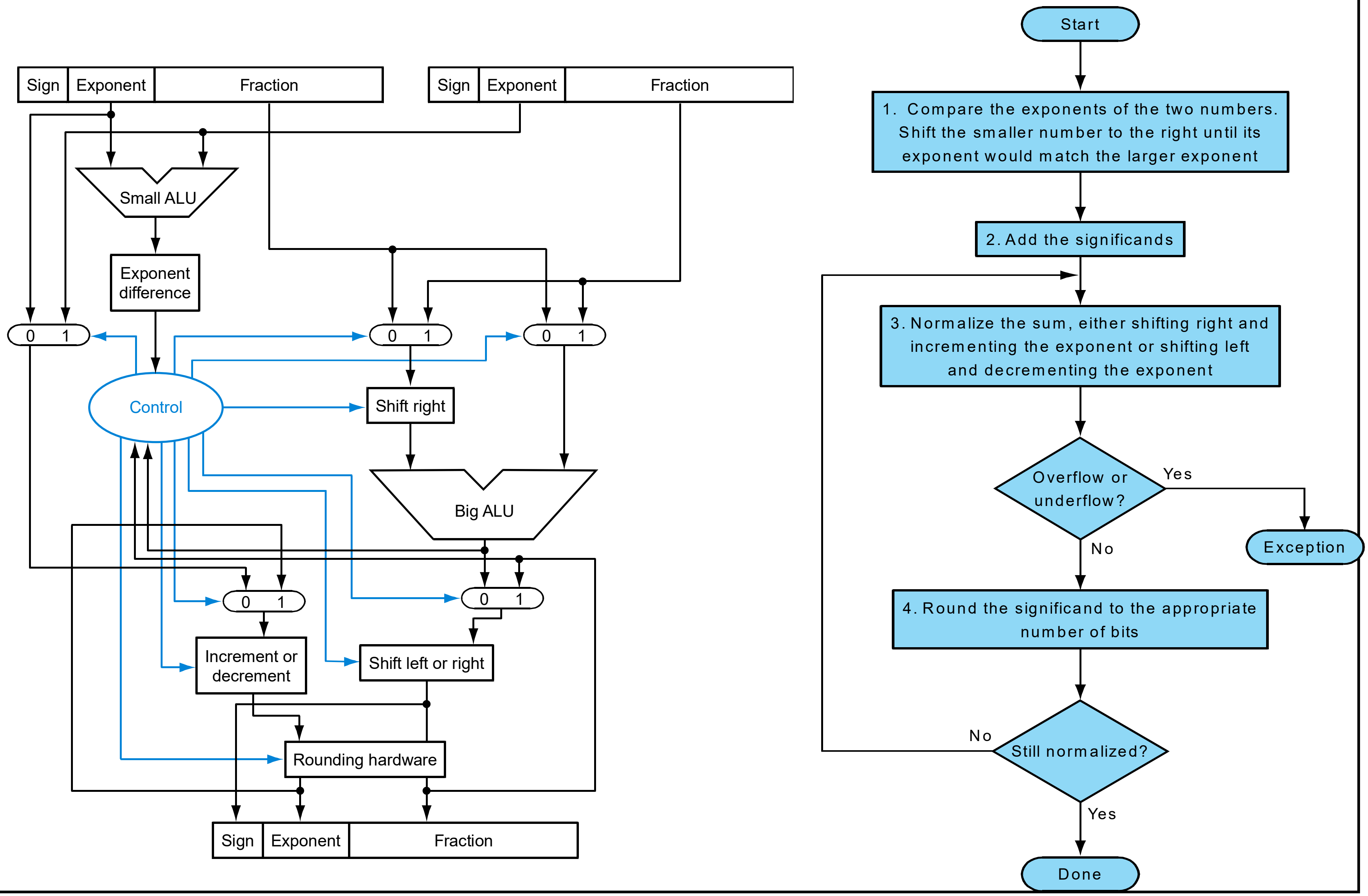
Continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

(4) Check result exponent:

- **If larger than maximum exponent allowed return exponent overflow**
- **If smaller than minimum exponent allowed return exponent underflow**

(5) If result mantissa is 0, may need to set the exponent to zero by a special step to return a proper zero.

Floating point addition



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}

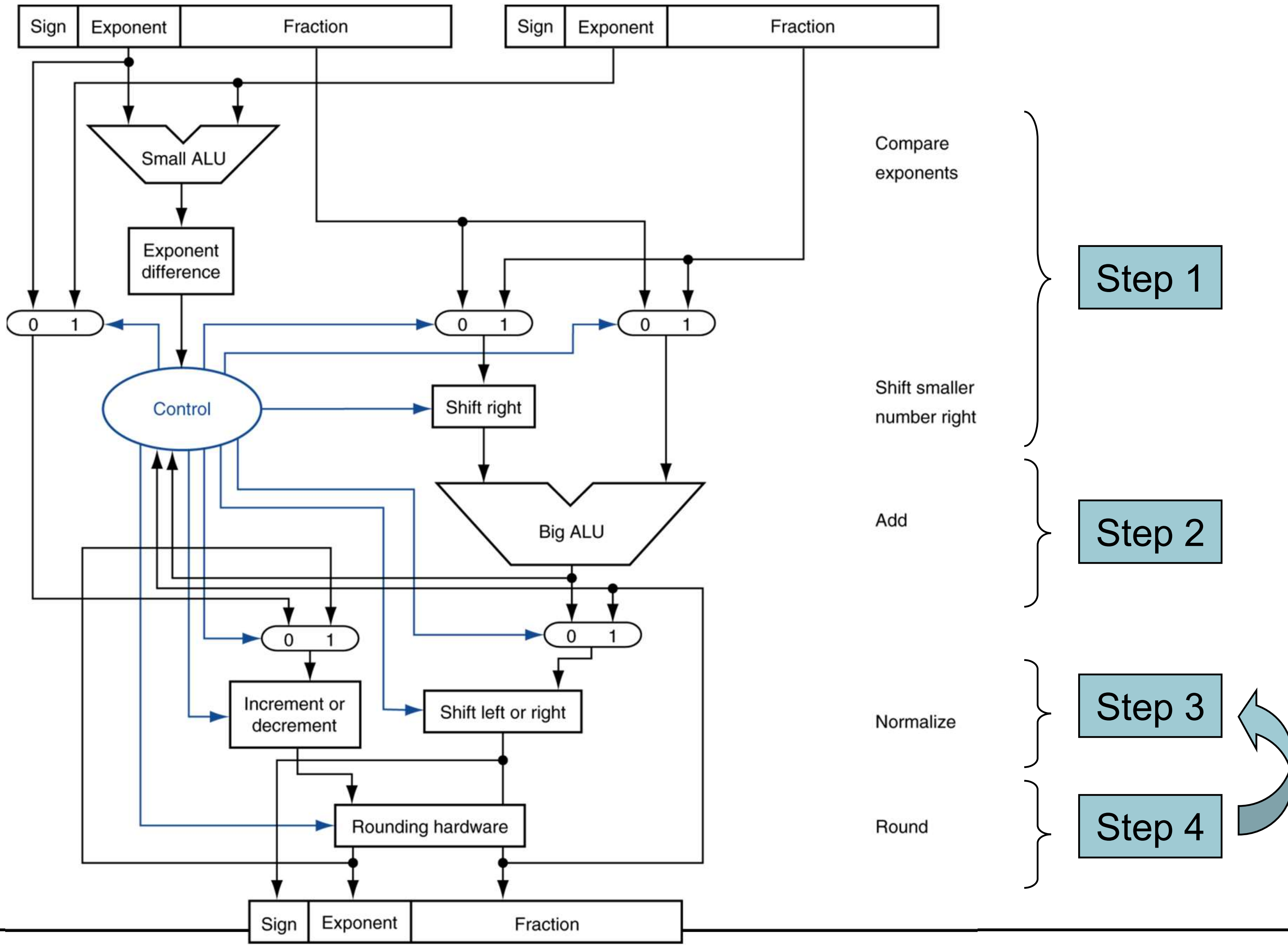
Floating-Point Addition

- 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



Floating Point Multiplication

Assuming that the operands are already in the IEEE 754 format, performing floating point multiplication:

$$\text{Result} = R = X * Y = (-1)^{Xs} (Xm \times 2^{Xe}) * (-1)^{Ys} (Ym \times 2^{Ye})$$

involves the following steps:

(1) If one or both operands is equal to zero, return the result as zero, otherwise:

(2) Compute the sign of the result Xs XOR Ys

(3) Compute the mantissa of the result:

- **Multiply the mantissas: Xm * Ym**
- **Round the result to the allowed number of mantissa bits**

(4) Compute the exponent of the result:

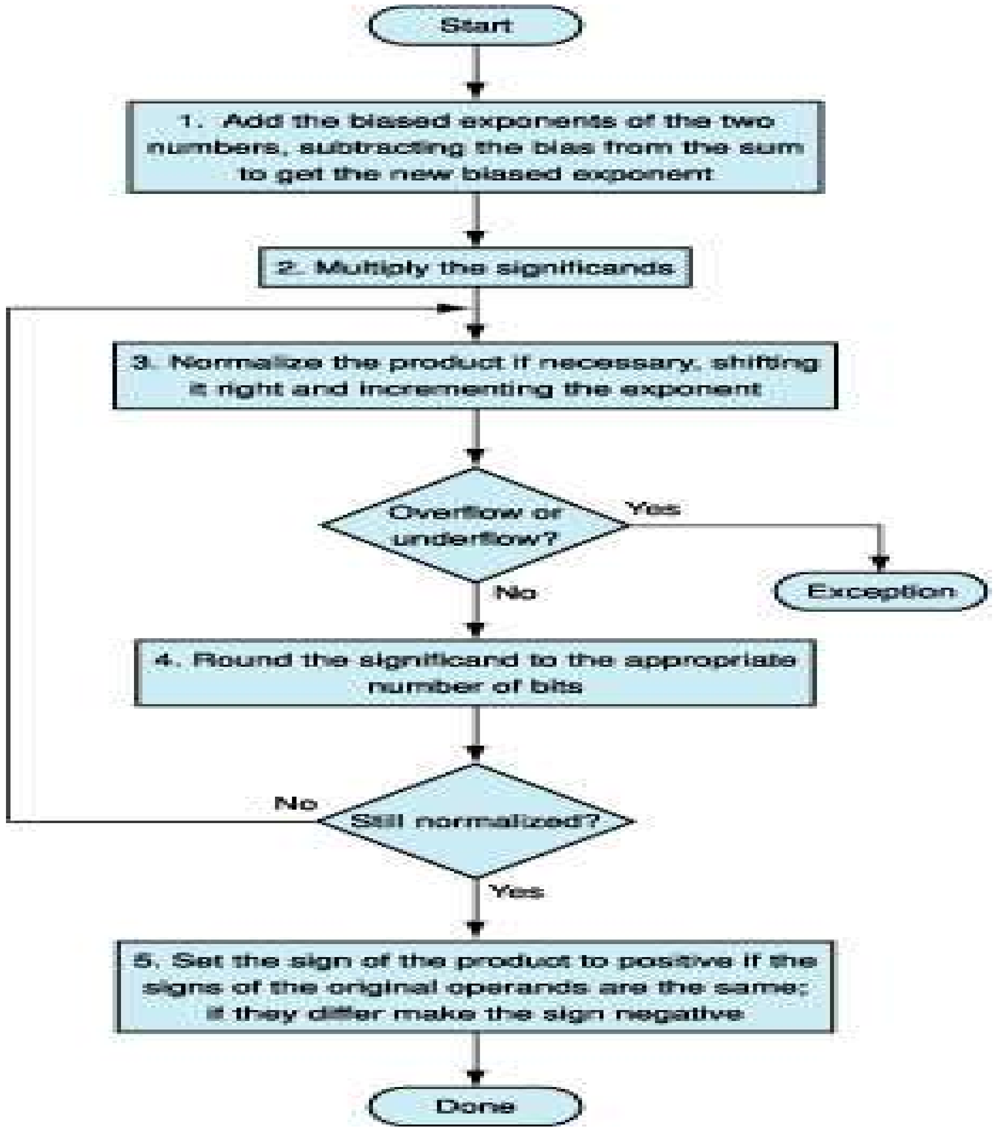
Result exponent = biased exponent (X) + biased exponent (Y) - bias

(5) Normalize if needed, by shifting mantissa right, incrementing result exponent.

(6) Check result exponent for overflow/underflow:

- **If larger than maximum exponent allowed return exponent overflow**
- **If smaller than minimum exponent allowed return exponent underflow**

Floating Point Multiplication Algorithm



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
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 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: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

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

Advantages of IEEE 754 Standard

Used predominantly by the industry

Encoding of exponent and fraction simplifies comparison

- Integer comparator used to compare magnitude of FP numbers

Includes special exceptional values: NaN and ±∞

- Special rules are used such as:
  - 0/0 is NaN, *sqrt*(–1) is NaN, 1/0 is ∞, and 1/∞ is 0
- Computation may continue in the face of exceptional conditions

FP Instructions in MIPS

- Floating point operations are slower than integer operations
- Data is rarely converted from integers to float within the same procedure
- 1980's solution – place FP processing unit in a separate chip
- Today's solution – imbed FP processing unit in processor chip
- Co-processor 1 features:
  - Contains 32 single precision floating point registers: $f0, $f1, … $f31
  - These registers can also act as 16 double precision registers: $f0/$f1, $f2/$f3, … , $f30/$f31 (only the first one is specified in the instructions)
  - Uses special floating point instructions, which are similar (in format) to integer instructions but have .s or .d attached to signify that they work on fp numbers
  - Several special instructions to move between "regular" registers and the co-processor registers

MIPS Floating Point Coprocessor

Called **Coprocessor 1** or the **Floating Point Unit (FPU)**

32 separate floating point registers: $f0, $f1, …, $f31

FP registers are 32 bits for single precision numbers

Even-odd register pair form a double precision register

Use the even number for double precision registers

- $f0, $f2, $f4, …, $f30 are used for double precision

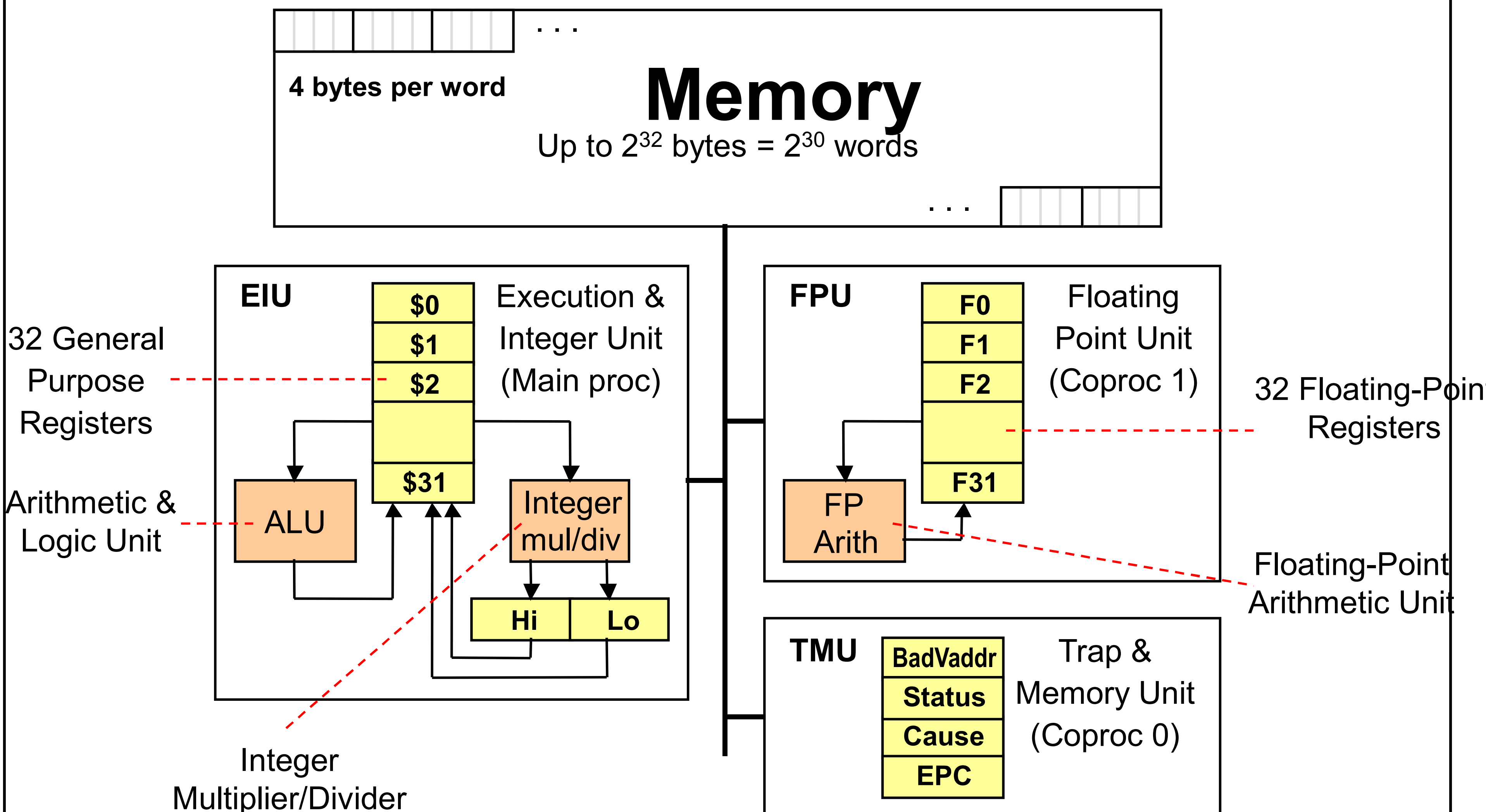
Separate FP instructions for single/double precision

- Single precision: add.s, sub.s, mul.s, div.s (.s extension)
- Double precision: add.d, sub.d, mul.d, div.d (.d extension)

FP instructions are more complex than the integer ones

- Take more cycles to execute

The MIPS Processor



Coprocessor Instruction Set

- Load word to coprocessor
- Store word from coprocessor
- Move to coprocessor
- Move from coprocessor
- Move control to coprocessor
- Move control from coprocessor
- Coprocessor operation
- Branch on coprocessor true
- Branch on coprocessor false

FP Arithmetic Instructions

Instruction	Meaning	Format					
add.s fd, fs, ft	(fd) = (fs) + (ft)	0x11	0	ft^5	fs^5	fd^5	0
add.d fd, fs, ft	(fd) = (fs) + (ft)	0x11	1	ft^5	fs^5	fd^5	0
sub.s fd, fs, ft	(fd) = (fs) – (ft)	0x11	0	ft^5	fs^5	fd^5	1
sub.d fd, fs, ft	(fd) = (fs) – (ft)	0x11	1	ft^5	fs^5	fd^5	1
mul.s fd, fs, ft	(fd) = (fs) × (ft)	0x11	0	ft^5	fs^5	fd^5	2
mul.d fd, fs, ft	(fd) = (fs) × (ft)	0x11	1	ft^5	fs^5	fd^5	2
div.s fd, fs, ft	(fd) = (fs) / (ft)	0x11	0	ft^5	fs^5	fd^5	3
div.d fd, fs, ft	(fd) = (fs) / (ft)	0x11	1	ft^5	fs^5	fd^5	3
sqrt.s fd, fs	(fd) = sqrt (fs)	0x11	0	0	fs^5	fd^5	4
sqrt.d fd, fs	(fd) = sqrt (fs)	0x11	1	0	fs^5	fd^5	4
abs.s fd, fs	(fd) = abs (fs)	0x11	0	0	fs^5	fd^5	5
abs.d fd, fs	(fd) = abs (fs)	0x11	1	0	fs^5	fd^5	5
neg.s fd, fs	(fd) = – (fs)	0x11	0	0	fs^5	fd^5	7
neg.d fd, fs	(fd) = – (fs)	0x11	1	0	fs^5	fd^5	7

FP Load/Store Instructions

- Separate floating point load/store instructions
  - lwc1: load word coprocessor 1
  - ldc1: load double coprocessor 1
  - swc1: store word coprocessor 1
  - sdc1: store double coprocessor 1

General purpose register is used as the base register

Instruction	Meaning	Format			
lwc1 $f2, 40($t0)	($f2) = Mem[($t0)+40]	0x31	$t0	$f2	$im^{16} = 40$
ldc1 $f2, 40($t0)	($f2) = Mem[($t0)+40]	0x35	$t0	$f2	$im^{16} = 40$
swc1 $f2, 40($t0)	Mem[($t0)+40] = ($f2)	0x39	$t0	$f2	$im^{16} = 40$
sdc1 $f2, 40($t0)	Mem[($t0)+40] = ($f2)	0x3d	$t0	$f2	$im^{16} = 40$

- Better names can be used for the above instructions
  - l.s = lwc1 (load FP single), l.d = ldc1 (load FP double)
  - s.s = swc1 (store FP single), s.d = sdc1 (store FP double)

FP Data Movement Instructions

- Moving data between general purpose and FP registers
  - mfc1: move from coprocessor 1 (to general purpose register)
  - mtc1: move to coprocessor 1 (from general purpose register)
- Moving data between FP registers
  - mov.s: move single precision float
  - mov.d: move double precision float = even/odd pair of registers

Instruction	Meaning	Format					
mfc1 $t0, $f2	($t0) = ($f2)	0x11	0	$t0	$f2	0	0
mtc1 $t0, $f2	($f2) = ($t0)	0x11	4	$t0	$f2	0	0
mov.s $f4, $f2	($f4) = ($f2)	0x11	0	0	$f2	$f4	6
mov.d $f4, $f2	($f4) = ($f2)	0x11	1	0	$f2	$f4	6

FP Convert Instructions

- Convert instruction: cvt.x.y
  - Convert to destination format x from source format y
- Supported formats
  - Single precision float = .s (single precision float in FP register)
  - Double precision float = .d (double float in even-odd FP register)
  - Signed integer word = .w (signed integer in FP register)

Instruction	Meaning	Format					
cvt.s.w fd, fs	to single from integer	0x11	0	0	fs^5	fd^5	0x20
cvt.s.d fd, fs	to single from double	0x11	1	0	fs^5	fd^5	0x20
cvt.d.w fd, fs	to double from integer	0x11	0	0	fs^5	fd^5	0x21
cvt.d.s fd, fs	to double from single	0x11	1	0	fs^5	fd^5	0x21
cvt.w.s fd, fs	to integer from single	0x11	0	0	fs^5	fd^5	0x24
cvt.w.d fd, fs	to integer from double	0x11	1	0	fs^5	fd^5	0x24

FP Compare and Branch Instructions

- FP unit (co-processor 1) has a condition flag
  - Set to 0 (false) or 1 (true) by any comparison instruction
- Three comparisons: equal, less than, less than or equal
- Two branch instructions based on the condition flag

Instruction	Meaning	Format					
c.eq.s fs, ft	cflag = ((fs) == (ft))	0x11	0	ft^5	fs^5	0	0x32
c.eq.d fs, ft	cflag = ((fs) == (ft))	0x11	1	ft^5	fs^5	0	0x32
c.lt.s fs, ft	cflag = ((fs) < (ft))	0x11	0	ft^5	fs^5	0	0x3c
c.lt.d fs, ft	cflag = ((fs) < (ft))	0x11	1	ft^5	fs^5	0	0x3c
c.le.s fs, ft	cflag = ((fs) <= (ft))	0x11	0	ft^5	fs^5	0	0x3e
c.le.d fs, ft	cflag = ((fs) <= (ft))	0x11	1	ft^5	fs^5	0	0x3e
bc1f Label	branch if (cflag == 0)	0x11	8	0	im^{16}		
bc1t Label	branch if (cflag == 1)	0x11	8	1	im^{16}		

FP Data Directives

.FLOAT Directive

Stores the listed values as single-precision floating point

.DOUBLE Directive

Stores the listed values as double-precision floating point

Examples

```
var1:  .FLOAT    12.3, -0.1
var2:  .DOUBLE   1.5e-10
pi:    .DOUBLE   3.14159265358979
```

Syscall Services

Service	$v0	Arguments / Result
Print Integer	1	$a0 = integer value to print
Print Float	2	$f12 = float value to print
Print Double	3	$f12 = double value to print
Print String	4	$a0 = address of null-terminated string
Read Integer	5	$v0 = integer read
Read Float	6	$f0 = float read
Read Double	7	$f0 = double read
Read String	8	$a0 = address of input buffer $a1 = maximum number of characters to read
Exit Program	10	
Print Char	11	$a0 = character to print
Read Char	12	$a0 = character read

Supported by MARS

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
```