Roadmap

- Operations on integers
  - Addition and subtraction
  - Multiplication
  - Division
- Floating-point numbers
  - Representation (check the review material)
  - Addition and multiplication

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 ≤ |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 non-negative
  - Bias = 2^(|exp|-1)-1
  - Single: Bias = 127; Double: Bias = 1023

Denormalized Numbers

- Exponent = 000…0 $\Rightarrow$ hidden bit is 0

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

- Smaller than normalized numbers
  - Allow for gradual underflow, with diminishing precision
- Denormalized with fraction = 000…0

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

Two representations of 0.0!

Other Special Patterns

- Exponent = 1…1
  - Fraction = all zero $\Rightarrow$ Infinities
  - Result of computations like X/0
  - Allows operations to continue past overflow situations
    - E.g. X/0 > 10
- Exponent = 1…1
  - Fraction = not all zero $\Rightarrow$ Not a number
  - Result of computations like sqrt(-4) or 0/0
  - Support mixing numerical and symbolic computation or other extensions

Floating-Point Example

- What number is represented by the single-precision float

  1 10000001 01000…00

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

Floating-Point Example

- Order the following floating-point numbers from highest to lowest:

  0 10000001 01000...00

  0 10000001 01111...00

  0 10000111 01000...00

Floating-Point Example

- Represent –0.75 as single/double precision encoding
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$

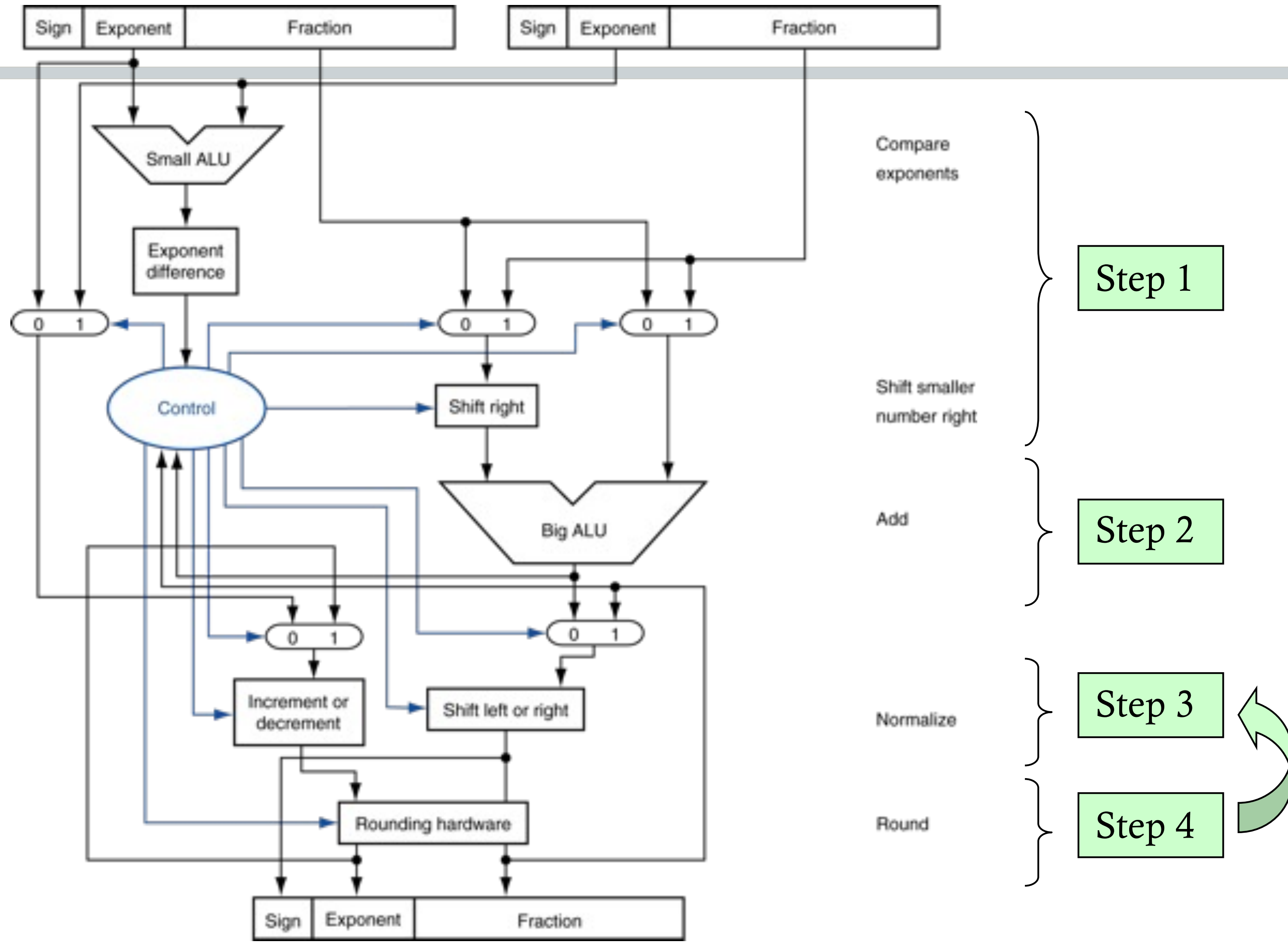
Roadmap

- Operations on integers
  - Addition and subtraction
  - Multiplication
  - Division
- Floating-point real numbers
  - Representation
  - Addition and multiplication

Floating-Point Addition

- 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



Rounding Modes

- Assume that we can only keep two digits to the right of binary point

Value	1.0101	1.0111	-1.0110	1.1010
Round up	1.10	1.10	-1.01	1.11
Round down	1.01	1.01	-1.10	1.10
Truncate (round to zero)	1.01	1.01	-1.01	1.10
Round to nearest even	1.01	1.10	-1.10	1.10

- Round to even
  - If <half, round down; if >half, round up
  - If ==half, use evenness to break the tie

Accurate Arithmetic

- Round accurately requires HW to include extra bits in the calculation
- Extra bits on the right during intermediate calculations – Guard, Round and Sticky bits
- Example:
  - $2.56 \cdot 10^{0} + 2.34 \cdot 10^{2}$
  - $5.03 \cdot 10^{-1} + 2.34 \cdot 10^{2}$
  - assuming 3 significant digits (2 significant digits after decimal point)

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 adjusting by -127 = –3 + 127
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110 \Rightarrow 1.110_2 \times 2^{-3}$
- 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
  - Otherwise needs a long clock cycle time
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor (usually `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
  - Eight condition code (cc) flags
  - `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)
     lwc1  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

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

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

Extra: Booth's Algorithm

- [CS367 exercise] Rewrite the multiplication below using shiftings, additions, and subtractions
  - X * 7
  - X * 15
- Observation: we have a quick way to calculate X * Y if Y is a sequence of contiguous one bits
  - X * $\underbrace{111....1}_{\text{k ones}}$ = X <<k -X

Extra: Booth's Algorithm

0 1 1 1 1 0

End of run (current bit = 0, bit to right = 1)

Middle of run

Beginning of run

(current bit = 1, bit to right = 0)

Booth's Algorithm:

00: Middle of string of 0s ➔ no operation

01: End of string of 1s ➔add (shifted) multiplicand to the left half of the partial product

10: Beginning of the 1s run ➔ subtract (shifted) multiplicand from the left half of the partial product

11: Middle of the 1s run, ➔ no operation

Extra: Booth Example

Step	Multiplcand(m)	Product(P)	Operation
0. init	0010	0000 0111 0	10→sub
1. P = P-m		+1110	
		1110 0111 0	shift P right
2.		1111 0011 1	11→no op, shift
3.		1111 1001 1	11→no op, shift
4.		1111 1100 1	01→add
P = P+m		+0010	
		0001 1100 1	shift
		0000 1110 0	done

2x7 = 0010 x 0111

Extra: Booth Example

Step	Multiplcand(m)	Product(P)	Operation
0. init	0010	0000 1101 0	10→sub
1. P = P-m		+1110	
		1110 1101 0	shift P right
2.		1111 0110 1	01→add
P = P+m		+0010	
		0001 0110 1	shift P right
3.		0000 1011 0	10→sub
P = P-m		+1110	
		1110 1011 0	shift P right
4.		1111 0101 1	11→no op, shift
		1111 1010 1	done

$\boxed{2x(-3) = 0010 \times 1101}$

Extra: Booth's Algorithm

- Efficient: fewer number of addition / subtraction operations
- Deal with both signed and unsigned multiplications