EECS4030: Computer Architecture

Arithmetic for Computers (II)

Prof. Chung-Ta King

Department of Computer Science

National Tsing Hua University, Taiwan

(Adapted from textbook slides https://www.elsevier.com/books-and-journals/book-companion/9780128122754/lecture-slides)

Outline

- Addition and subtraction (Sec. 3.2)
- Multiplication (Sec. 3.3)
- Division (Sec. 3.4)
- Floating point (Sec. 3.5)
- Parallelism and computer arithmetic: subword parallelism (Sec. 3.6)
- Streaming SIMD extensions and advanced vector extensions in x86 (Sec. 3.7)
- Subword parallelism and matrix multiply (Sec. 3.8)

Floating Point Numbers: Motivation

- What can be represented in N bits?

Unsigned	0	to	$2^n - 1$
2's Complement	-2^{n-1}	to	$2^{n-1} - 1$

- But, what about ...
  - very large numbers?
    9,349,398,989,787,762,244,859,087,678
  - very small number?
    0.0000000000000000000000045691
  - rationals 2/3
  - irrationals $\sqrt{2}$
  - transcendentals e, π
- How to represent them in just N bits?

Floating Point Numbers

- In math, we use scientific notation to representation very small and very large numbers, e.g.,
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

Normalized (exactly one non-zero digit to the left of decimal point)

Non-normalized

- In binary:

Significand (Mantissa)

exponent

$$\mathbf{1.0}_{\text{two}} \times 2^{-1}$$

"**binary point**"

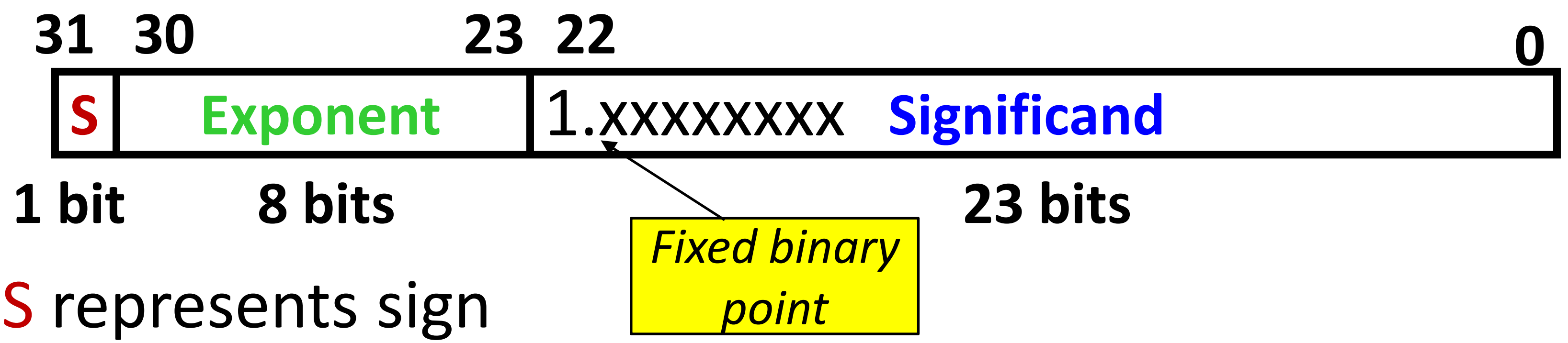
radix (base)

Floating point: binary point is not fixed (vs. fixed-point numbers such as integers)

- Types `float` and `double` in C/C++

Intuitive Floating Point Representation

- Normalized format: $\pm 1.xxxxxxxxxx_{two} \times 2^{\pm yyyy_{two}}$
- Can use 32 bits to represent (*single-precision*):

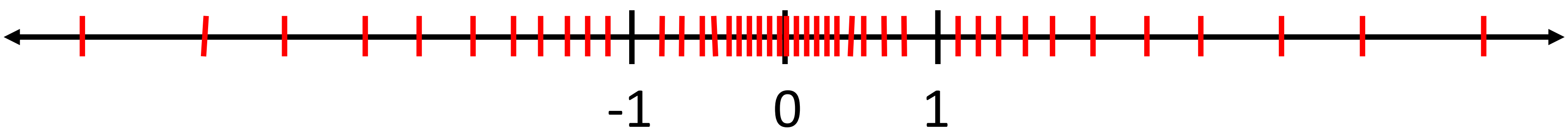


S represents sign

Exponent represents y's (positive or negative)

Significand represents 1.x's

- Represent numbers from 1.0×10^{-38} to 2.0×10^{38}



Intuitive Floating Point Representation

- Can use 64 bits if more precision is needed (*double precision*)

31	30 … 20	19 … 0
S	Exponent	1.xxxxxxxx Significand
1 bit	11 bits	20 bits

Significand (cont'd)
32 bits

- Double precision (vs. single precision)
  – Represent numbers almost as small as 1.0×10^{-308} to almost as large as 2.0×10^{308}
  – Main advantage is greater accuracy due to larger significand

Floating Point Standard

- Defined by IEEE Std 754-1985
  - For portability; more efficient use of bits

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)
  - Normalized significand: 1.0 ≤ |significand| < 2.0
    - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (*hidden bit*) → to pack more bits
    - Significand is Fraction with the "1." restored
  - Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

Floating Point Standard

- Handling exponent:
  - Need to represent positive and negative exponents
  - Also want to compare FP numbers as if they were integers
  - If use 2's complement to represent?
    e.g., 1.0×2^{-1} versus $1.0 \times 2^{+1}$ (1/2 versus 2)

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

  - ***If we use integer comparison for these two words, we would conclude that 1/2 > 2!!!***

Floating Point Standard

- Handling exponent: (cont.)
  - Instead, let notation 0000 0000 be the most negative, and 1111 1111 the most positive → biased notation, where bias is the number subtracted to get the real number
  - IEEE 754 uses bias of 127 (0111 1111) for single precision: (Exponent – 127) to get actual value for exponent

		Most positive (127)	Most negative (-126)
Actual	2's comp.	0111 1111	1000 0010
Rep	biased	1111 1110	0000 0001

00000000 & 11111111 reserved

  - 1023 (011 1111 1111) is bias for double precision

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

Biased Number Representation

Decimal	2's Compl.	Bias-3 (011)	Bias-4 (100)
+3	011	110	111
+2	010	101	110
+1	001	100	101
0	000	011	100
-1	111	010	011
-2	110	001	010
-3	101	000	001
-4	100	111	000

Exponent 0000 and 1111 are reserved

Floating Point Standard

- Summary (single precision):

31	30 23	22 0
S	Exponent	Fraction
1 bit	8 bits	23 bits

$(-1)^S \times (1.\text{Fraction}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023

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
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating Point Precision

- Relative precision
  - All fraction bits are significant
  - Single precision: approximately 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double precision: approximately 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 precision: 1011111101000…00
- Double precision: 1011111111101000…00

Floating Point Example

- A more difficult case: representing 1/3?

  $= 0.33333\ldots_{10} = 0.0101010101\ldots_2 \times 2^0$

  $= 1.0101010101\ldots_2 \times 2^{-2}$

  - Sign: 0
  - Exponent = -2 + 127 = $125_{10} = 01111101_2$
  - Fraction = 0101010101…

0	0111 1101	0101 0101 0101 0101 0101 010

  - This 32-bit number is only an approximation of 1/3!
    If we use this number as 1/3 in subsequent computations, the error will propagate and magnify

Floating Point Example

- What number is represented by the single-precision floating point?

  11000000101000…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 0.01_2) \times 2^{(129-127)}$

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

  $= -5.0$
- Can it also represent the following number?

1	1000 0001	0100 0000 0000 0000 0000 000

01

Special Numbers in IEEE 754 Standard

- So far, we have not used the full range of 32/64 bits in defining floating point numbers
- Consider single precision representation:

Exponent	Fraction	Object
0	0	+/- 0
0	nonzero	???
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

Denormalized Numbers

- Represent denormalized numbers (*denorms*)
  - Exponent: 000...0
  - Fraction: non-zeroes $\Rightarrow$ hidden bit is 0

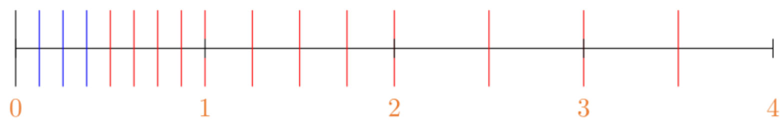
$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-126}$$

Not -127

0	0000 0000	0100 0000 0000 0000 0000 000

$= 0.01_2 \times 2^{-126}$

  - Allow a number to degrade in significance until it become 0 (*gradual underflow*)



  - Smallest normalized number
    - $1.000 \ldots 000 \times 2^{-126}$ = 0 | 0000 0001 | 0000 … 0000
  - Smallest/largest de-normalized number
    - $0.000 \ldots 001 \times 2^{-126}$ $0.111 \ldots 111 \times 2^{-126}$

Gradually smaller

Special Numbers in IEEE 754 Standard

- So far, we have not used the full range of 32/64 bits in defining floating point numbers
- Consider single precision representation:

Exponent	Fraction	Object
0	0	+/- 0
0	nonzero	denorm
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

  – Denormal with fraction = 000...0

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

Two representations of 0.0!

Representation for +/- Infinity

- In FP, divide by zero should produce +/- infinity, not overflow
- Why?
  - OK to do further computations with infinity, e.g., X/0 > Y may be a valid comparison
- IEEE 754 represents +/- infinity
  - Most positive exponent reserved for infinity
  - Fraction is all zeroes

S	**1111 1111**	**0000 0000 0000 0000 0000 000**

Special Numbers in IEEE 754 Standard

- So far, we have not used the full range of 32/64 bits in defining floating point numbers
- Consider single precision representation:

Exponent	Fraction	Object
0	0	+/- 0
0	nonzero	denorm
1-254	anything	+/- floating-point
255	0	+/- infinity
255	nonzero	???

Representation for Not a Number

- What do I get if I calculate sqrt(-4.0) or 0/0?
  - If infinity is not an error, these should not be either
  - They are called *Not a Number* (NaN)
    → Exponent = 255, Fraction nonzero
- Why is this useful?
  - Indicates illegal or undefined result
  - Hope NaNs help with debugging
  - They contaminate: op(NaN, X) = NaN
  - OK if calculate but don't use it

Summary: IEEE 754 FP Standard

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)

Fig. 3.13

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 and 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
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

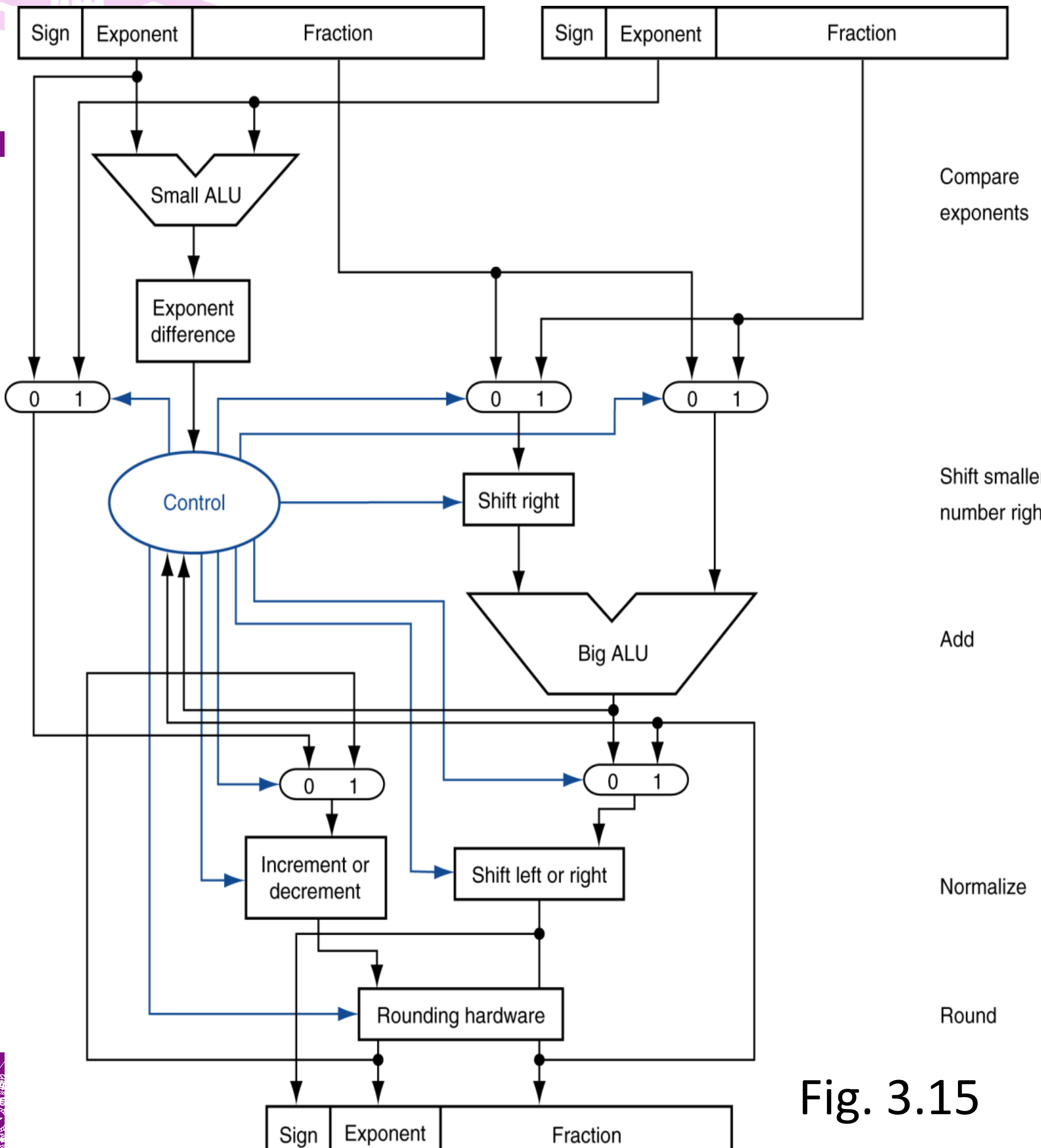
3) Normalize result and check for over/underflow
   - $1.000_2 \times 2^{-4}$, with no over/underflow

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



Step 1

Step 2

Step 3

Step 4

Fig. 3.15

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 and 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

$$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2} \ (0.5 \times -0.4375)$$

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 and 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: $+\text{ve} \times -\text{ve} \Rightarrow -\text{ve}$
   - $-1.110_2 \times 2^{-3} = -0.21875$

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 ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in RISC-V

- Separate FP registers: f0, …, f31
  - double-precision
  - single-precision values stored in the lower 32 bits
- 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
  - Single precision load/store: **`flw`**, **`fsw`**
  - Double precision load/store: **`fld`**, **`fsd`**

FP Instructions in RISC-V F/D Extension

- Single-precision arithmetic
  - **`fadd.s`**, **`fsub.s`**, **`fmul.s`**, **`fdiv.s`**, **`fsqrt.s`**
    - e.g., **`fadd.s f2,f4,f6`**
- Double-precision arithmetic
  - **`fadd.d`**, **`fsub.d`**, **`fmul.d`**, **`fdiv.d`**, **`fsqrt.d`**
    - e.g., **`fadd.d f2,f4,f6`**
- Single- and double-precision comparison
  - **`feq.s`**, **`flt.s`**, **`fle.s`**
  - **`feq.d`**, **`flt.d`**, **`fle.d`**
  - Result is 0 or 1 in integer destination register

```
feq.d  x5,f0,f1
```

FP Example: °F to °C

- C code:

```
float f2c(float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` in `f10`, result in `f10`, literals in global memory space

- Compiled RISC-V code:

```
f2c: flw    f0,const5(x3)   // f0 = 5.0f
     flw    f1,const9(x3)   // f1 = 9.0f
     fdiv.s f0,f0,f1        // f0 = 5.0f/9.0f
     flw    f1,const32(x3)  // f1 = 32.0f
     fsub.s f10,f10,f1      // f10 = fahr−32.0
     fmul.s f10,f0,f10      // f10 = () * ()
     jalr   x0,0(x1)        // return
```

FP Example: Array Multiplication

- C = C + A × B
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double c[][],
         double a[][], double b[][]) {
  size_t 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)
        c[i][j] = c[i][j]
                  + a[i][k] * b[k][j];
}
```

  - Addresses of **c**, **a**, **b** in **x10**, **x11**, **x12**, and **i**, **j**, **k** in **x5**, **x6**, **x7**

FP Example: Array Multiplication

- RISC-V code:

```
    li    x28,32        // x28 = 32 (row size/loop end)
    li    x5,0          // i = 0; initialize 1st for loop
L1: li    x6,0          // j = 0; initialize 2nd for loop
L2: li    x7,0          // k = 0; initialize 3rd for loop
    slli  x30,x5,5      // x30 = i * 2^5 (size of row of c)
    add   x30,x30,x6    // x30 = i * size(row) + j
    slli  x30,x30,3     // x30 = byte offset of [i][j]
    add   x30,x10,x30   // x30 = byte address of c[i][j]
    fld   f0,0(x30)     // f0 = c[i][j]
L3: slli  x29,x7,5      // x29 = k * 2^5 (size of row of b)
    add   x29,x29,x6    // x29 = k * size(row) + j
    slli  x29,x29,3     // x29 = byte offset of [k][j]
    add   x29,x12,x29   // x29 = byte address of b[k][j]
    fld   f1,0(x29)     // f1 = b[k][j]
```

FP Example: Array Multiplication

```
slli   x29,x5,5      // x29 = i * 2^5 (size of row of a)
add    x29,x29,x7    // x29 = i * size(row) + k
slli   x29,x29,3     // x29 = byte offset of [i][k]
add    x29,x11,x29   // x29 = byte address of a[i][k]
fld    f2,0(x29)     // f2 = a[i][k]
fmul.d f1, f2, f1    // f1 = a[i][k] * b[k][j]
fadd.d f0, f0, f1    // f0 = c[i][j] + a[i][k] * b[k][j]
addi   x7,x7,1       // k = k + 1
bltu   x7,x28,L3     // if (k < 32) go to L3
fsd    f0,0(x30)     // c[i][j] = f0
addi   x6,x6,1       // j = j + 1
bltu   x6,x28,L2     // if (j < 32) go to L2
addi   x5,x5,1       // i = i + 1
bltu   x5,x28,L1     // if (i < 32) go to L1
```

Accurate Arithmetic

- Integer arithmetic is accurate, exact
  - Because integers can represent exactly every number between the smallest and largest number
- FP arithmetic is approximate, inexact
  - Because FP numbers are just approximations for the actual number they want to represent
  - The approximation errors widen as FP numbers are operated upon to generate new FP numbers
  - Need to be careful in rounding intermediate results
- IEEE Std 754 specifies additional rounding control
  - Use extra bits during HW calculation to preserve precision (*guard*, *round*, *sticky*) and allow choice of rounding modes

Associativity of FP Operations

- Is FP add/subtract associative?

		(X + Y) + Z	X + (Y + Z)
X	-1.50×10^{38}		-1.50×10^{38}
Y	1.50×10^{38}	0.0	
Z	1.0	1.0	1.50×10^{38}
		1.0	0.0

- FP add/subtract are not associative!
  – Why? FP result approximates real result!
  – Ex: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Outline

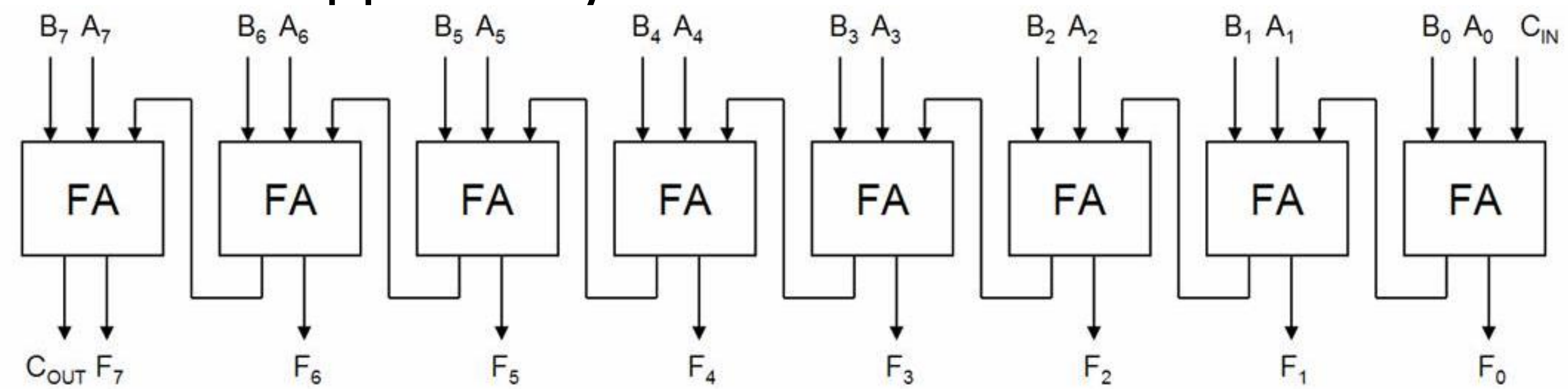
- Addition and subtraction (Sec. 3.2)
- Multiplication (Sec. 3.3)
- Division (Sec. 3.4)
- Floating point (Sec. 3.5)
- Parallelism and computer arithmetic: subword parallelism (Sec. 3.6)
- Streaming SIMD extensions and advanced vector extensions in x86 (Sec. 3.7)
- Subword parallelism and matrix multiply (Sec. 3.8)

Arithmetic for Multimedia

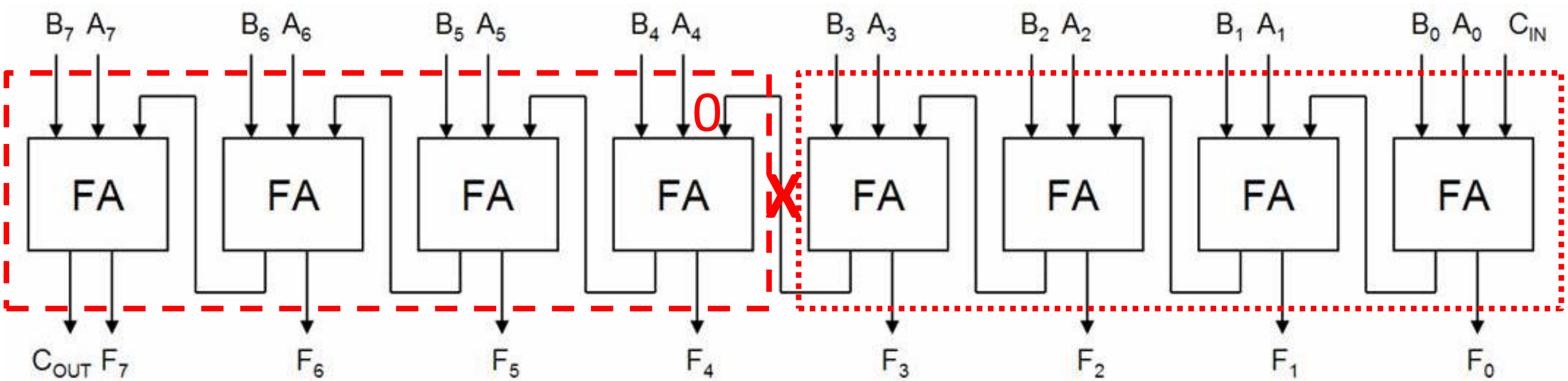
- Graphics and media processing often perform same operations on vectors of 8-bit and 16-bit data
  - These data are often packed into words
- How to operate on these words of short data?
  - Since we have already had 64-bit adder, can we leverage it?
  - Partition the carry chain of the 64-bit adder so that the adder can perform 8 8-bit or 4 16-bit vector add/sub
  - This is called *subword parallelism*
- *Saturating* operations
  - On overflow, result is the largest representable value, e.g., clipping in audio, saturation in video

Subword Parallelism

- An 8-bit ripple-carry adder:



- Turned into two 4-bit ripple-carry adders:



http://ece-research.unm.edu/pollard/classes/338/lademo/LookAheadDemo.htm

Fallacies and Pitfalls

- Left shift by i places → multiplies an integer by 2^i
- Right shift by i places divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    $11111011_2 >> 2 = 11111110_2 = -2$
  - If we only do logic shift right,
    $11111011_2 >> 2 = 00111110_2 = +62$

Summary

- RISC-V arithmetic: successive refinement to final design
  - 64-bit adder and logic unit
  - 64-bit multiplier and divisor
- FP numbers approximate values that we want to use
  - IEEE 754 Standard is most widely accepted representation
  - New RISC-V registers (f0~f31) and instructions:
    - Single-precision (32 bits): **`fadd.s`**, **`fsub.s`**, **`fmul.s`**, **`fdiv.s`**, **`fsqrt.s`** , **`feq.s`**, **`flt.s`**, **`fle.s`**
    - Double-precision (64 bits): **`fadd.d`**, **`fsub.d`**, **`fmul.d`**, **`fdiv.d`**, **`fsqrt.d`**, **`feq.d`**, **`flt.d`**, **`fle.d`**

Concluding Remarks

- 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
- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow