CSC 3210 Computer organization and programming

Chapter 3

Chunlan Gao

Georgia State University

Review

IEEE Std 754-1985

Bias： Single precision 127， double precision 1203

$$\begin{tikzcd} 1111\,1110 \arrow[d] \arrow[r, "--->"] & 00000001\ ? \\ -126 + 127 --> 1, \text{ so the bias is } 127 \end{tikzcd}$$

normal value：

Exponents from 00000001 to 11111110

$$(-1)^{S} \times (1 + \text{Fraction}) \times 2^{E}$$

Reserved：

Exponents 00000000 and 11111111

Denormal Numbers（Subnormal numbers)

- Exponent = 000...0 ⇒ hidden bit is 0

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

- Smaller than normal numbers (at **≈ 1.4 × 10^{-45}** and go down to zero)
  - 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!

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

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

0.5 + 0.4375

0.5ten= 1/2 = 0.1two= 1.000 $\times 2^{-1}$

0.4375= -7/16 = $-7/2^4$=-0.0111two = -1.110two $\times 2^{-2}$

Step1:

The significand of the number with the lesser exponent (-1.11two $\times 2^{-2}$) is shifted right until its exponent matches the larger number

$$-1.110_{\text{two}} \times 2^{-2} = -0.111_{\text{two}} \times 2^{-1}$$

Step 2. Add the 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: Since $127 \geq -4 \geq -126$, there is no overflow or underflow.

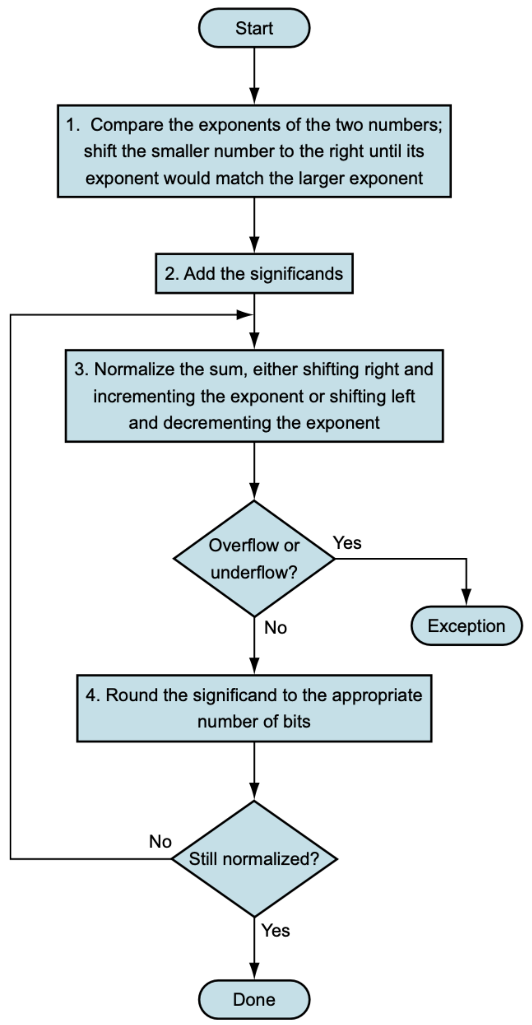
$$0.001_{\text{two}} \times 2^{-1} = 0.010_{\text{two}} \times 2^{-2} = 0.100_{\text{two}} \times 2^{-3} = 1.000_{\text{two}} \times 2^{-4}$$

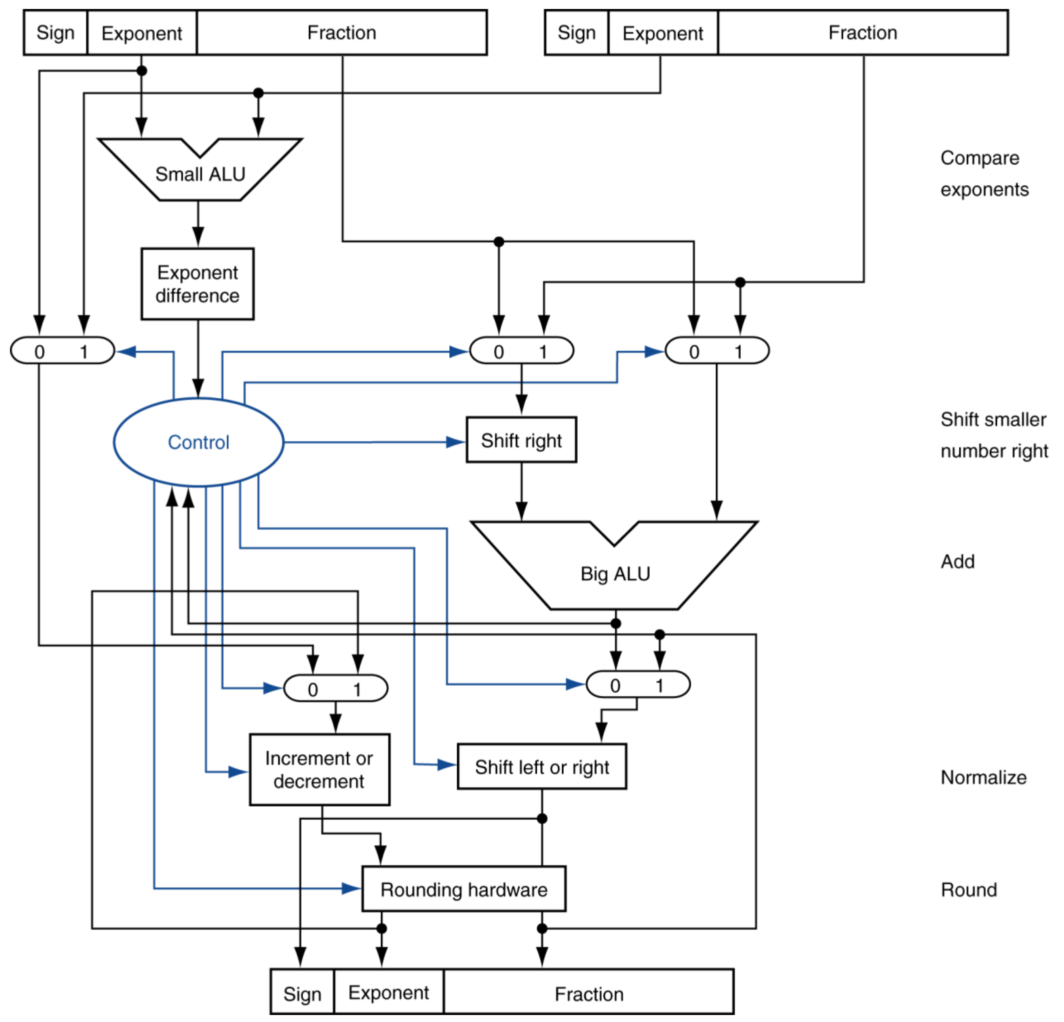
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

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
No
Still normalized?
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
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 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
  - $-1.110_2 \times 2^{-3} = -0.21875$

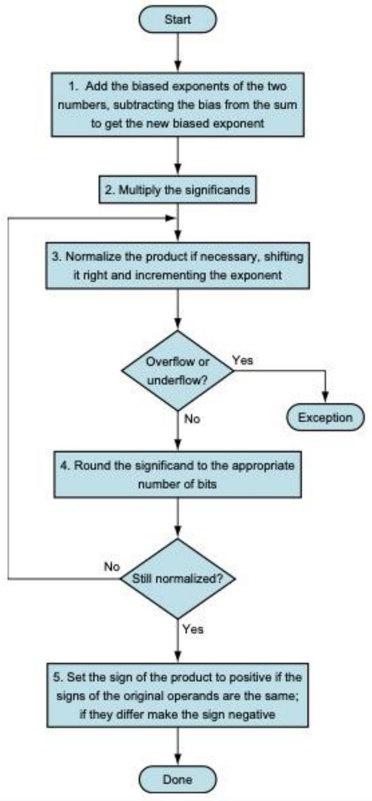

Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to 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

FIGURE 3.16 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

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

FP Instructions in RISC-V

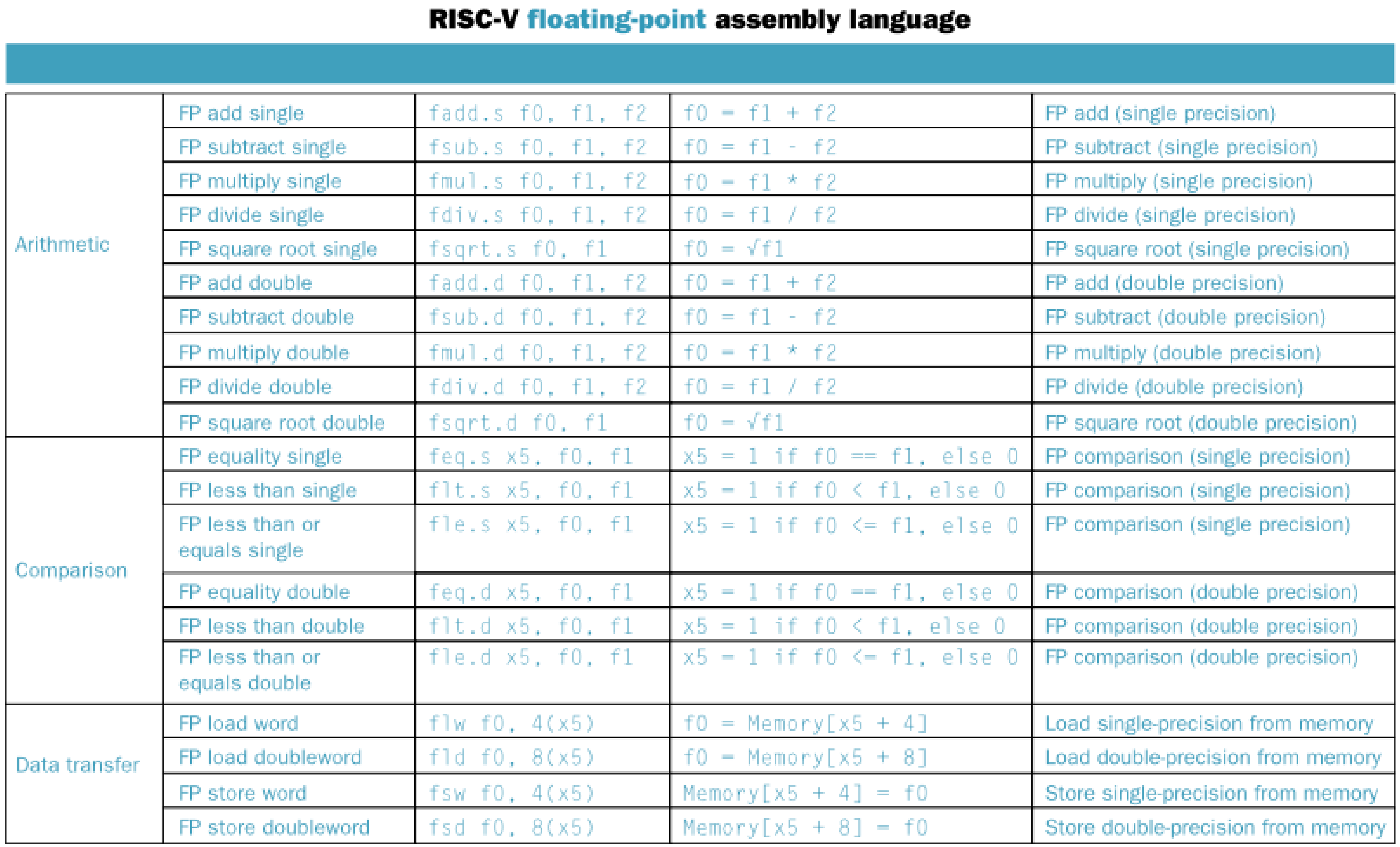
- Separate FP registers: f0, …, f31
    - double-precision
    - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
    - Programs generally don't do integer ops on FP data, or vice versa
    - More registers with minimal code-size impact
- FP load and store instructions
    - `flw, fld`
    - `fsw, fsd`

FP Instructions in RISC-V

- Single-precision arithmetic
  - `fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s`
    - e.g., `fadds.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
    - Use beq, bne to branch on comparison result

- Branch on FP condition code true or false
  - `B.cond`

FL in RISC-V



RISC-V floating-point assembly language

Arithmetic	FP add single	`fadd.s f0, f1, f2`	`f0 = f1 + f2`	FP add (single precision)
	FP subtract single	`fsub.s f0, f1, f2`	`f0 = f1 - f2`	FP subtract (single precision)
	FP multiply single	`fmul.s f0, f1, f2`	`f0 = f1 * f2`	FP multiply (single precision)
	FP divide single	`fdiv.s f0, f1, f2`	`f0 = f1 / f2`	FP divide (single precision)
	FP square root single	`fsqrt.s f0, f1`	`f0 = √f1`	FP square root (single precision)
	FP add double	`fadd.d f0, f1, f2`	`f0 = f1 + f2`	FP add (double precision)
	FP subtract double	`fsub.d f0, f1, f2`	`f0 = f1 - f2`	FP subtract (double precision)
	FP multiply double	`fmul.d f0, f1, f2`	`f0 = f1 * f2`	FP multiply (double precision)
	FP divide double	`fdiv.d f0, f1, f2`	`f0 = f1 / f2`	FP divide (double precision)
	FP square root double	`fsqrt.d f0, f1`	`f0 = √f1`	FP square root (double precision)
Comparison	FP equality single	`feq.s x5, f0, f1`	`x5 = 1 if f0 == f1, else 0`	FP comparison (single precision)
	FP less than single	`flt.s x5, f0, f1`	`x5 = 1 if f0 < f1, else 0`	FP comparison (single precision)
	FP less than or equals single	`fle.s x5, f0, f1`	`x5 = 1 if f0 <= f1, else 0`	FP comparison (single precision)
	FP equality double	`feq.d x5, f0, f1`	`x5 = 1 if f0 == f1, else 0`	FP comparison (double precision)
	FP less than double	`flt.d x5, f0, f1`	`x5 = 1 if f0 < f1, else 0`	FP comparison (double precision)
	FP less than or equals double	`fle.d x5, f0, f1`	`x5 = 1 if f0 <= f1, else 0`	FP comparison (double precision)
Data transfer	FP load word	`flw f0, 4(x5)`	`f0 = Memory[x5 + 4]`	Load single-precision from memory
	FP load doubleword	`fld f0, 8(x5)`	`f0 = Memory[x5 + 8]`	Load double-precision from memory
	FP store word	`fsw f0, 4(x5)`	`Memory[x5 + 4] = f0`	Store single-precision from memory
	FP store doubleword	`fsd f0, 8(x5)`	`Memory[x5 + 8] = f0`	Store double-precision from memory

FIGURE 3.17 RISC-V floating-point architecture revealed thus far. This information is also found in column 2 of the RISC-V Reference Data Card at the front of this book.

FP Example: °F to °C

$$^\circ C = \frac{5}{9} \times (^\circ F - 32)$$

- C code:

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

  - `fahr` in f10, result in f10, literals in global memory space
  - We assume that the compiler places the three floating-point constants in memory within easy reach of register x3

- 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 = (5.0f/9.0f) * (fahr–32.0f)
 jalr   x0,0(x1)    // return
```

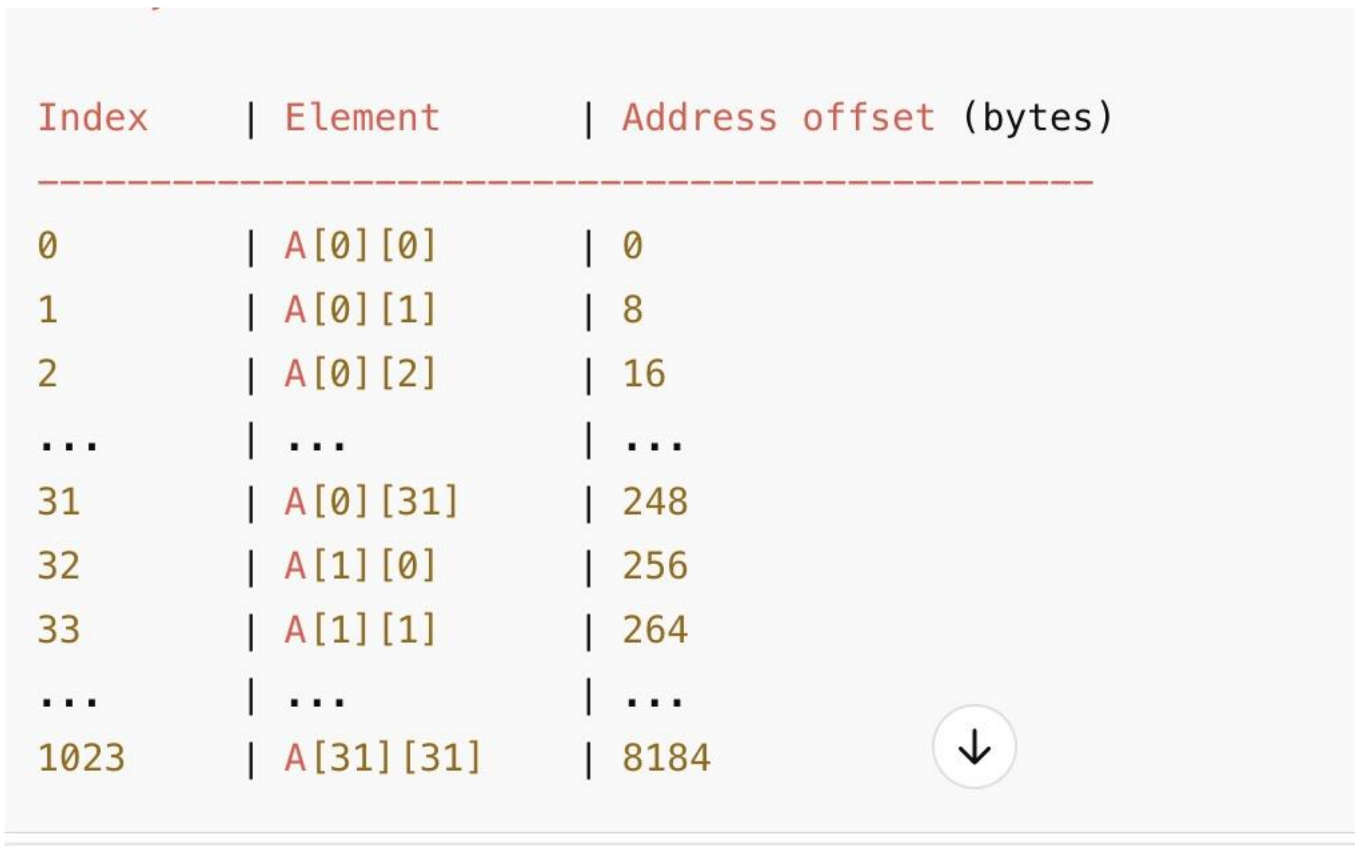
FP Example: Array Multiplication

- 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
  - Assume that the integer variables are in x5, x6, and x7, respectively

Accurate Arithmetic



```
Index      | Element        | Address offset (bytes)
------------------------------------------------
0          | A[0][0]        | 0
1          | A[0][1]        | 8
2          | A[0][2]        | 16
...        | ...            | ...
31         | A[0][31]       | 248
32         | A[1][0]        | 256
33         | A[1][1]        | 264
...        | ...            | ...
1023       | A[31][31]      | 8184
```

FP Example: Array Multiplication

- RISC-V code:

```
 mm:...
          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
# f0 = c[i][j]
               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]
# f1 = b[k][j] and f2= a[k][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,x5,3     // 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 b[i][k]
        add    x29,x11,x29  // x29 = byte address of a[i][k]
        fld    f2,0(x29)    // f2 = a[i][k]
#f[0]= c[i][j] + a[i][k] * b[k][j]
        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]
 # k++, check if k<32
        addi   x7,x7,1      // k = k + 1
        bltu   x7,x28,L3    // if (k < 32) go to L3
 # save f0 to c[i][j]
         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

IEEE 754, therefore, always keeps two extra bits

on the right during intervening additions, called guard and round

Add 2.56 * 10^0 to 2.34 * 10^2, assuming we have three significant decimal digit, (the guard digit hold5 and round digit hold 6,)

$$\begin{array}{r} 2.3400_{\text{ten}} \\ +0.0256_{\text{ten}} \\ \hline 2.3656_{\text{ten}} \end{array}$$

Round to 3 digit 2.37

Without guard and round

the result is 2.36

$$\begin{array}{r} 2.34_{\text{ten}} \\ +0.02_{\text{ten}} \\ \hline 2.36_{\text{ten}} \end{array}$$

Subword Parallellism