Floating Point

COE 301 / ICS 233

Computer Organization

Dr. Muhamed Mudawar

College of Computer Sciences and Engineering

King Fahd University of Petroleum and Minerals

Presentation Outline

- **Floating-Point Numbers**
- **The IEEE 754 Floating-Point Standard**
- Floating-Point Comparison, Addition and Subtraction
- Floating-Point Multiplication
- MIPS Floating-Point Instructions and Examples

The World is Not Just Integers

- Programming languages support numbers with fraction
  - Called floating-point numbers
  - Examples:

    3.14159265… (π)

    2.71828… (e)

    1.0×10^{-9} (seconds in a nanosecond)

    8.64×10^{13} (nanoseconds in a day)

    The last number is a large integer that cannot fit in a 32-bit register
- We use a scientific notation to represent
  - Very small numbers (e.g. 1.0×10^{-9})
  - Very large numbers (e.g. 8.64×10^{13})
  - Scientific notation: $\pm\, d\,.\,\textit{fraction} \times 10^{\pm\,\textit{exponent}}$

Floating-Point Numbers

- Examples of floating-point numbers in base 10

  -5.341×10^{3} , 2.013×10^{-1}

  ↑ ***decimal point***

- Examples of floating-point numbers in base 2

  -1.00101×2^{23} , 1.101101×2^{-3}

  ↑ ***binary point***

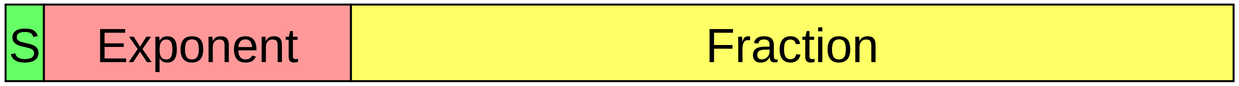
  - Exponents are kept in decimal for clarity

- Floating-point numbers should be **normalized**
  - Exactly **one non-zero digit** should appear before the point
    - In a decimal number, this digit can be from **1 to 9**
    - In a binary number, this digit should be **1**
  - Normalized: -5.341×10^{3} and 1.101101×2^{-3}
  - NOT Normalized: -0.05341×10^{5} and 1101.101×2^{-6}

Floating-Point Representation

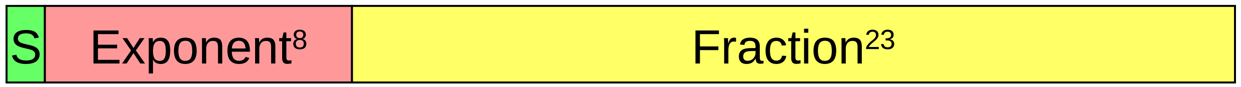
A floating-point number is represented by the triple

- Sign bit (0 is positive and 1 is negative)
  - Representation is called sign and magnitude
- Exponent field (signed value)
  - Very large numbers have large positive exponents
  - Very small close-to-zero numbers have negative exponents
  - More bits in exponent field increases range of values
- Fraction field (fraction after binary point)
  - More bits in fraction field improves the precision of FP numbers

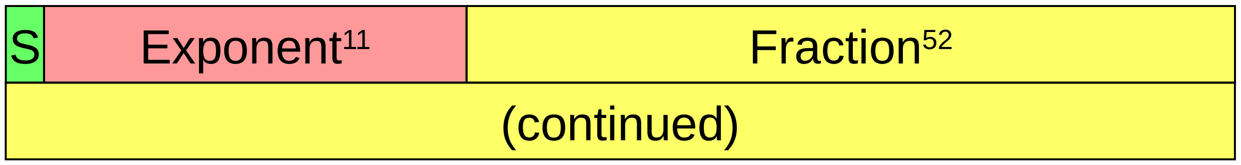


IEEE 754 Floating-Point Standard

- Found in virtually every computer invented since 1980
  - Simplified porting of floating-point numbers
  - Unified the development of floating-point algorithms
  - Increased the accuracy of floating-point numbers
- Single Precision Floating Point Numbers (32 bits)
  - 1-bit sign + 8-bit exponent + 23-bit fraction

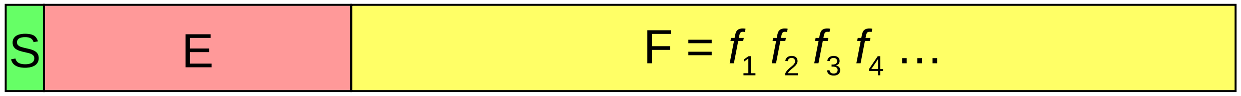


- Double Precision Floating Point Numbers (64 bits)
  - 1-bit sign + 11-bit exponent + 52-bit fraction



Normalized Floating Point Numbers

- For a normalized floating point number (*S*, *E*, *F*)



- Significand is equal to $(1.F)_2 = (1.f_1f_2f_3f_4\ldots)_2$
  - IEEE 754 assumes hidden **1.** (not stored) for normalized numbers
  - Significand is 1 bit longer than fraction
- Value of a Normalized Floating Point Number:

$$\boxed{\begin{array}{l} \pm\,(1.F)_2 \times 2^{exponent\_value} \\ \pm\,(1.f_1f_2f_3f_4\ldots)_2 \times 2^{exponent\_value} \\ \pm\,(1 + f_1\times2^{-1} + f_2\times2^{-2} + f_3\times2^{-3} + f_4\times2^{-4}\ \ldots)_2 \times 2^{exponent\_value} \end{array}}$$

S = 0 is positive, S = 1 is negative

Biased Exponent Representation

- How to represent a signed exponent? Choices are …
  - Sign + magnitude representation for the exponent
  - Two's complement representation
  - Biased representation
- IEEE 754 uses biased representation for the exponent
  - Exponent Value = *E* – Bias (Bias is a constant)
- The exponent field is 8 bits for single precision
  - *E* can be in the range 0 to 255
  - *E* = 0 and *E* = 255 are reserved for special use (discussed later)
  - *E* = 1 to 254 are used for normalized floating point numbers
  - Bias = 127 (half of 254)
  - Exponent value = ***E* – 127** Range: **-126 to +127**

Biased Exponent – Cont'd

- For double precision, the exponent field is 11 bits
  - E can be in the range 0 to 2047
  - $E = 0$ and $E = 2047$ are reserved for special use
  - $E = 1$ to 2046 are used for normalized floating point numbers
  - Bias = 1023 (half of 2046)
  - Exponent value = ***E* – 1023**   Range: **-1022 to +1023**
- Value of a Normalized Floating Point Number is

$$\boxed{\begin{array}{l} \pm (1.F)_2 \times 2^{(E\text{-}Bias)} \\ \pm (1.f_1 f_2 f_3 f_4 \ldots)_2 \times 2^{(E\text{-}Bias)} \\ \pm (1 + f_1\times 2^{-1} + f_2\times 2^{-2} + f_3\times 2^{-3} + f_4\times 2^{-4} \ldots)_2 \times 2^{(E\text{-}Bias)} \end{array}}$$

S = 0 is positive,   S = 1 is negative

Examples of Single Precision Float

❖ What is the decimal value of this Single Precision float?

1	0	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

❖ Solution:

- ◇ Sign = 1 is negative
- ◇ E = $(01111100)_2$ = 124, *E* – bias = 124 – 127 = –3
- ◇ Significand = $(\mathbf{1.}0100 \ldots 0)_2 = 1 + 2^{-2}$ = 1**.**25 (**1.** is implicit)
- ◇ Value in decimal = -1.25×2^{-3} = –0**.**15625

❖ What is the decimal value of?

0	1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

❖ Solution:

implicit ↴

- ◇ Value in decimal = $+(\mathbf{1.}01001100 \ldots 0)_2 \times 2^{130-127}$ =

  $(\mathbf{1.}01001100 \ldots 0)_2 \times 2^3 = (\mathbf{1}010\mathbf{.}01100 \ldots 0)_2$ = 10**.**375

Examples of Double Precision Float

- What is the decimal value of this Double Precision float ?

0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Solution:
  - Value of exponent = $(10000000101)_2$ – Bias = 1029 – 1023 = 6
  - Value of double = $(1.00101010 \ldots 0)_2 \times 2^6$ (1. is implicit) =

    $(1001010.10 \ldots 0)_2$ = 74.5

- What is the decimal value of ?

1	0	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Do it yourself! (answer should be $-1.5 \times 2^{-7} = -0.01171875$)

Decimal to Binary Floating-Point

- Convert –0.8125 to single and double-precision floating-point
- Solution:
  - Fraction bits can be obtained using multiplication by 2
    - 0.8125 × 2 = 1.625
    - 0.625 × 2 = 1.25
    - 0.25 × 2 = 0.5
    - 0.5 × 2 = 1.0
    - Stop when fractional part is 0, or after computing all required fraction bits

$$0.8125 = (0.1101)_2 = ½ + ¼ + 1/16 = 13/16$$

  - Fraction = $(0.1101)_2 = (1.101)_2 \times 2^{-1}$ (Normalized)
  - Exponent = –1 + Bias = 126 (single precision) and 1022 (double)

1	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Single Precision**

1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Double Precision**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Largest Normalized Float

- What is the Largest normalized float?
- Solution for Single Precision:

0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

  - E – bias = 254 – 127 = +127 (largest exponent for SP)
  - Significand = $(1.111 \ldots 1)_2$ = 1.99999988 = almost 2
  - Value in decimal $\approx 2 \times 2^{+127} \approx 2^{+128} \approx 3.4028 \ldots \times 10^{+38}$
- Solution for Double Precision:

0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

  - Value in decimal $\approx 2 \times 2^{+1023} \approx 2^{+1024} \approx 1.79769 \ldots \times 10^{+308}$
- Overflow: exponent is too large to fit in the exponent field

Smallest Normalized Float

- What is the smallest (in absolute value) normalized float?
- Solution for Single Precision:

0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

  - Exponent – bias = 1 – 127 = -126 (smallest exponent for SP)
  - Significand = $(1.000 \ldots 0)_2 = 1$
  - Value in decimal = $1 \times 2^{-126} = 1.17549 \ldots \times 10^{-38}$
- Solution for Double Precision:

0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

  - Value in decimal = $1 \times 2^{-1022} = 2.22507 \ldots \times 10^{-308}$
- Underflow: exponent is too small to fit in exponent field

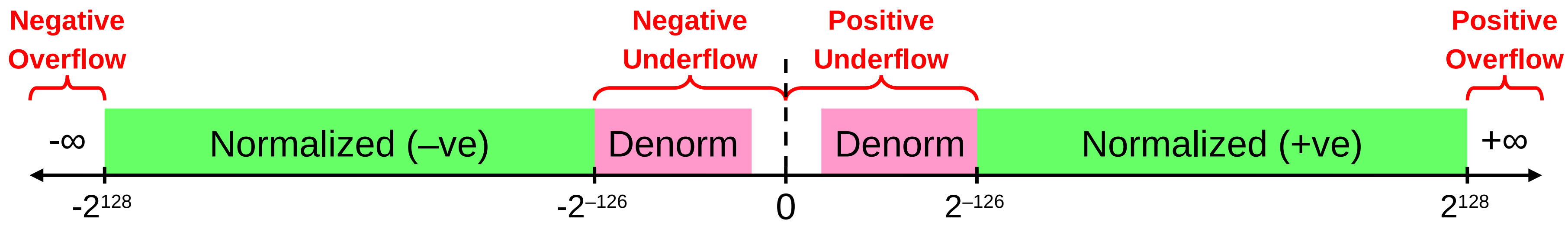
Zero, Infinity, and NaN

- Zero
  - Exponent field $E = 0$ and fraction $F = 0$
  - `+0` and `-0` are both possible according to sign bit S
- Infinity
  - Infinity is a special value represented with maximum E and $F = 0$
    - For single precision with 8-bit exponent: maximum $E = 255$
    - For double precision with 11-bit exponent: maximum $E = 2047$
  - Infinity can result from overflow or division by zero
  - $+\infty$ and $-\infty$ are both possible according to sign bit S
- NaN (Not a Number)
  - NaN is a special value represented with maximum E and $F \neq 0$
  - **0 / 0 → NaN, 0 × ∞ → NaN, sqrt(-1) → NaN**
  - Operation on a NaN is typically a NaN: **Op(*X*, NaN) → NaN**

Denormalized Numbers

- IEEE standard uses denormalized numbers to …
  - Fill the gap between 0 and the smallest normalized float
  - Provide gradual underflow to zero
- Denormalized: exponent field E is 0 and fraction $F \neq 0$
  - The Implicit **1.** before the fraction now becomes **0.** (denormalized)
- Value of denormalized number (S, 0, F)

Single precision:	$\pm\,(0.F)_2 \times 2^{-126}$
Double precision:	$\pm\,(0.F)_2 \times 2^{-1022}$



Summary of IEEE 754 Encoding

Single-Precision	Exponent = 8	Fraction = 23	Value
Normalized Number	1 to 254	Anything	$\pm (1.F)_2 \times 2^{E-127}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-126}$
Zero	0	0	± 0
Infinity	255	0	$\pm \infty$
NaN	255	nonzero	NaN

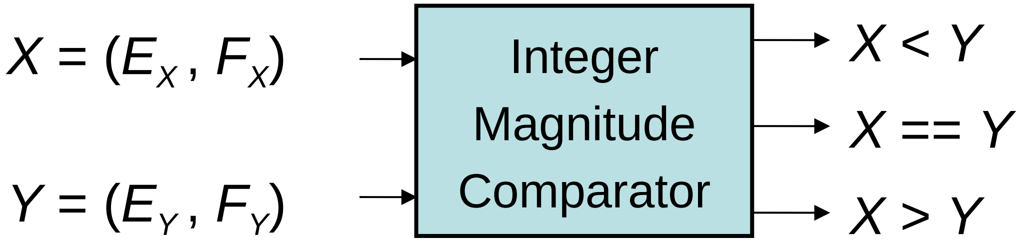
Double-Precision	Exponent = 11	Fraction = 52	Value
Normalized Number	1 to 2046	Anything	$\pm (1.F)_2 \times 2^{E-1023}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-1022}$
Zero	0	0	± 0
Infinity	2047	0	$\pm \infty$
NaN	2047	nonzero	NaN

Next . . .

- ❖ Floating-Point Numbers
- ❖ The IEEE 754 Floating-Point Standard
- ❖ Floating-Point Comparison, Addition and Subtraction
- ❖ Floating-Point Multiplication
- ❖ MIPS Floating-Point Instructions and Examples

Floating-Point Comparison

- IEEE 754 floating point numbers are ordered (except NaN)
  - Because the exponent uses a biased representation …
    - Exponent value and its binary representation have same ordering
  - Placing exponent before the fraction field orders the magnitude
    - Larger exponent ⇒ larger magnitude
    - For equal exponents, Larger fraction ⇒ larger magnitude
    - $0 < (0.F)_2 \times 2^{Emin} < (1.F)_2 \times 2^{E-Bias} < \infty$ $(E_{\text{min}} = 1 - Bias)$
  - Sign bit provides a quick test for signed <
- Integer comparator can compare the magnitudes



Floating Point Addition

- Consider Adding Single-Precision Floats:

$$\begin{array}{rl} & 1.11100100000000000000010_2 \times 2^4 \\ + & 1.10000000000000110000101_2 \times 2^2 \end{array}$$

- Cannot add significands … Why?
  - Because exponents are not equal
- How to make exponents equal?
  - Shift the significand of the lesser exponent right
  - Difference between the two exponents = 4 – 2 = 2
  - So, shift right second number by 2 bits and increment exponent

$$\begin{array}{rl} & 1.10000000000000110000101_2 \times 2^2 \\ = & 0.01100000000000001100001\ 01_2 \times 2^4 \end{array}$$

Floating-Point Addition – cont'd

- Now, ADD the Significands:

$$\begin{array}{rll}
 & 1.11100100000000000000010 & \times 2^{4} \\
+ & 1.10000000000000110000101 & \times 2^{2} \\
\hline
 & 1.11100100000000000000010 & \times 2^{4} \\
+ & 0.01100000000000001100001\ 01 & \times 2^{4}\ \text{(shift right)} \\
\hline
 & 10.01000100000000001100011\ 01 & \times 2^{4}\ \text{(result)}
\end{array}$$

- Addition produces a carry bit, result is NOT normalized
- Normalize Result (shift right and increment exponent):

$$\begin{array}{rll}
 & 10.01000100000000001100011\ 01 & \times 2^{4} \\
\hline
= & 1.00100010000000000110001\ 101 & \times 2^{5}\ \text{(normalized)}
\end{array}$$

Rounding

- Single-precision requires only 23 fraction bits
- However, Normalized result can contain additional bits

1.00100010000000000110001 | 1 01 × 2⁵

***Round Bit:* R = 1** ***Sticky Bit:* S = 1**

- Two extra bits are used for rounding
  - **Round bit:** appears just after the normalized result
  - **Sticky bit:** appears after the round bit (OR of all additional bits)
- Since **RS = 11**, increment fraction to **round to nearest**

```
  1.00100010000000000110001 × 2⁵
                         +1
  1.00100010000000000110010 × 2⁵  (Rounded)
```

Floating-Point Subtraction

- Addition is used when operands have the same sign
- Addition becomes a subtraction when sign bits are different
- Consider adding floating-point numbers with different signs:

$$\begin{array}{lllll}
+ & \texttt{1.00000000101100010001101} & & \times\ 2^{-6} & \\
- & \texttt{1.00000000000000010011010} & & \times\ 2^{-1} & \\
\hline
+ & \texttt{0.00001000000001011000100} & \texttt{01101} & \times\ 2^{-1} & \text{(shift right 5 bits)} \\
- & \texttt{1.00000000000000010011010} & & \times\ 2^{-1} & \\
\hline
0 & \texttt{0.00001000000001011000100} & \texttt{01101} & \times\ 2^{-1} & \\
1 & \texttt{0.11111111111111101100110} & & \times\ 2^{-1} & \text{(2's complement)} \\
\hline
1 & \texttt{1.00001000000001000101010} & \texttt{01101} & \times\ 2^{-1} & \text{(Negative result)} \\
\hline
- & \texttt{0.11110111111110111010101} & \texttt{10011} & \times\ 2^{-1} & \text{(Sign Magnitude)}
\end{array}$$

- 2's complement of result is required if result is negative

Floating-Point Subtraction – cont'd

```
+ 1.00000000101100010001101 × 2^-6
- 1.00000000000000010011010 × 2^-1
  ________________________________________
- 0.11110111111110111010101 10011 × 2^-1 (Sign Magnitude)
```

- Result should be normalized (unless it is equal to zero)
  - For subtraction, we can have leading zeros. To normalize, count the number of leading zeros, then shift result left and decrement the exponent accordingly.

```
                                      Guard bit
- 0.11110111111110111010101 1 0011 × 2^-1
  ________________________________________
- 1.11101111111101110101011   0011 × 2^-2   (Normalized)
```

- Guard bit: guards against loss of a fraction bit
  - Needed for subtraction only, when result has a leading zero and should be normalized.

Floating-Point Subtraction – cont'd

- Next, the normalized result should be rounded

```
                                        Guard bit
- 0.11110111111110111010101 1 0 011  ×  2^-1
───────────────────────────────────────────────────────────
- 1.11101111111101110101011   0 011  ×  2^-2 (Normalized)
         Round bit: R=0 - -        - - Sticky bit: S = 1
```

- Since **R = 0**, it is more accurate to truncate the result even though **S = 1**. We simply discard the extra bits.

```
- 1.11101111111101110101011    0 011  ×  2^-2 (Normalized)
───────────────────────────────────────────────────────────
- 1.11101111111101110101011    ×  2^-2 (Rounded to nearest)
```

- IEEE 754 Representation of Result

1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1	0	1	0	1	0	1	1

Rounding to Nearest Even

- Normalized result has the form: **1. $f_1\ f_2\ \ldots\ f_l$ R S**
  - The round bit **R** appears immediately after the last fraction bit f_l
  - The sticky bit **S** is the OR of all remaining additional bits
- Round to Nearest Even: default rounding mode
- Four cases for **RS**:
  - **RS = 00** 🖧 Result is Exact, no need for rounding
  - **RS = 01** 🖧 **Truncate** result by discarding **RS**
  - **RS = 11** 🖧 **Increment** result: ADD 1 to last fraction bit
  - **RS = 10** 🖧 Tie Case (either truncate or increment result)
    - **Check Last fraction bit f_l** (f_{23} for single-precision or f_{52} for double)
    - **If f_l is 0 then truncate result to keep fraction even**
    - **If f_l is 1 then increment result to make fraction even**

Additional Rounding Modes

❖ IEEE 754 standard includes other rounding modes:

1. Round to Nearest Even: described in previous slide
2. Round toward +Infinity: result is rounded up

   Increment result if sign is positive and **R** or **S** = **1**

3. Round toward -Infinity: result is rounded down

   Increment result if sign is negative and **R** or **S** = **1**

4. Round toward 0: always truncate result

❖ Rounding or Incrementing result might generate a carry

- This occurs only when all fraction bits are **1**
- Re-Normalize after Rounding step is required only in this case

Example on Rounding

- Round following result using IEEE 754 rounding modes:

  $-1.11111111111111111111111\ 1\ 0 \times 2^{-7}$

  Round Bit *Sticky Bit*

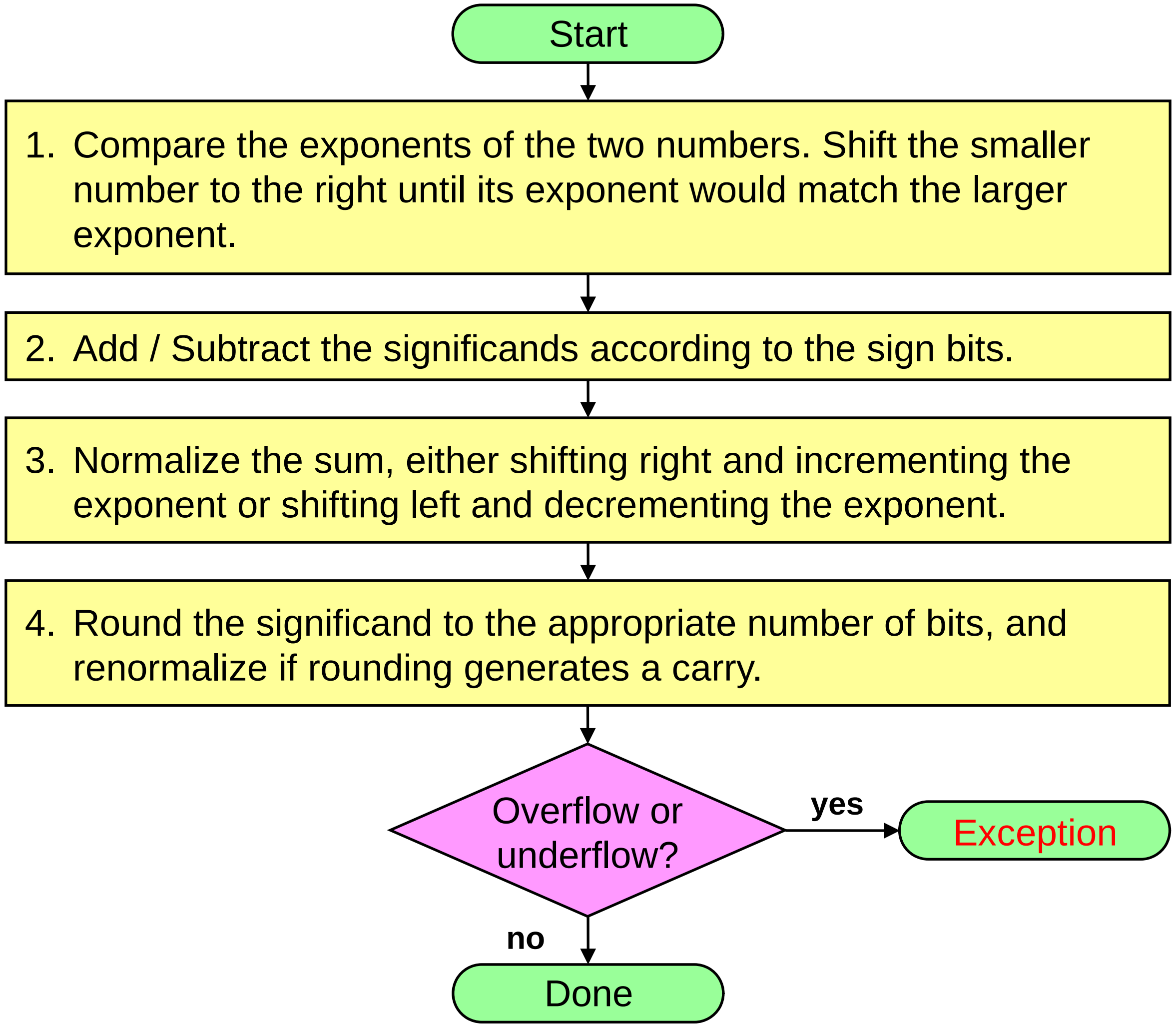
- Round to Nearest Even:
  - Increment result since **RS = 10 and f_{23} = 1**
  - Incremented result: $-10.00000000000000000000000 \times 2^{-7}$
  - Renormalize and increment exponent (because of carry)
  - Final rounded result: $-1.00000000000000000000000 \times 2^{-6}$
- Round towards +∞: Truncate result since negative
  - Truncated Result: $-1.11111111111111111111111 \times 2^{-7}$
- Round towards –∞: Increment since negative and **R = 1**
  - Final rounded result: $-1.00000000000000000000000 \times 2^{-6}$
- Round towards 0: Truncate always

Accuracy can be a Big Problem

Value1	Value2	Value3	Value4	Sum
1.0E+30	-1.0E+30	9.5	-2.3	7.2
1.0E+30	9.5	-1.0E+30	-2.3	-2.3
1.0E+30	9.5	-2.3	-1.0E+30	0

- Adding double-precision floating-point numbers (Excel)
- Floating-Point addition is NOT associative
- Produces different sums for the same data values
- Rounding errors when the difference in exponent is large

Floating Point Addition / Subtraction


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 / Subtract the significands according to the sign bits.
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.
4. Round the significand to the appropriate number of bits, and renormalize if rounding generates a carry.
Overflow or underflow?
yes
Exception
no
Done

Shift significand right by $d = | E_X - E_Y |$

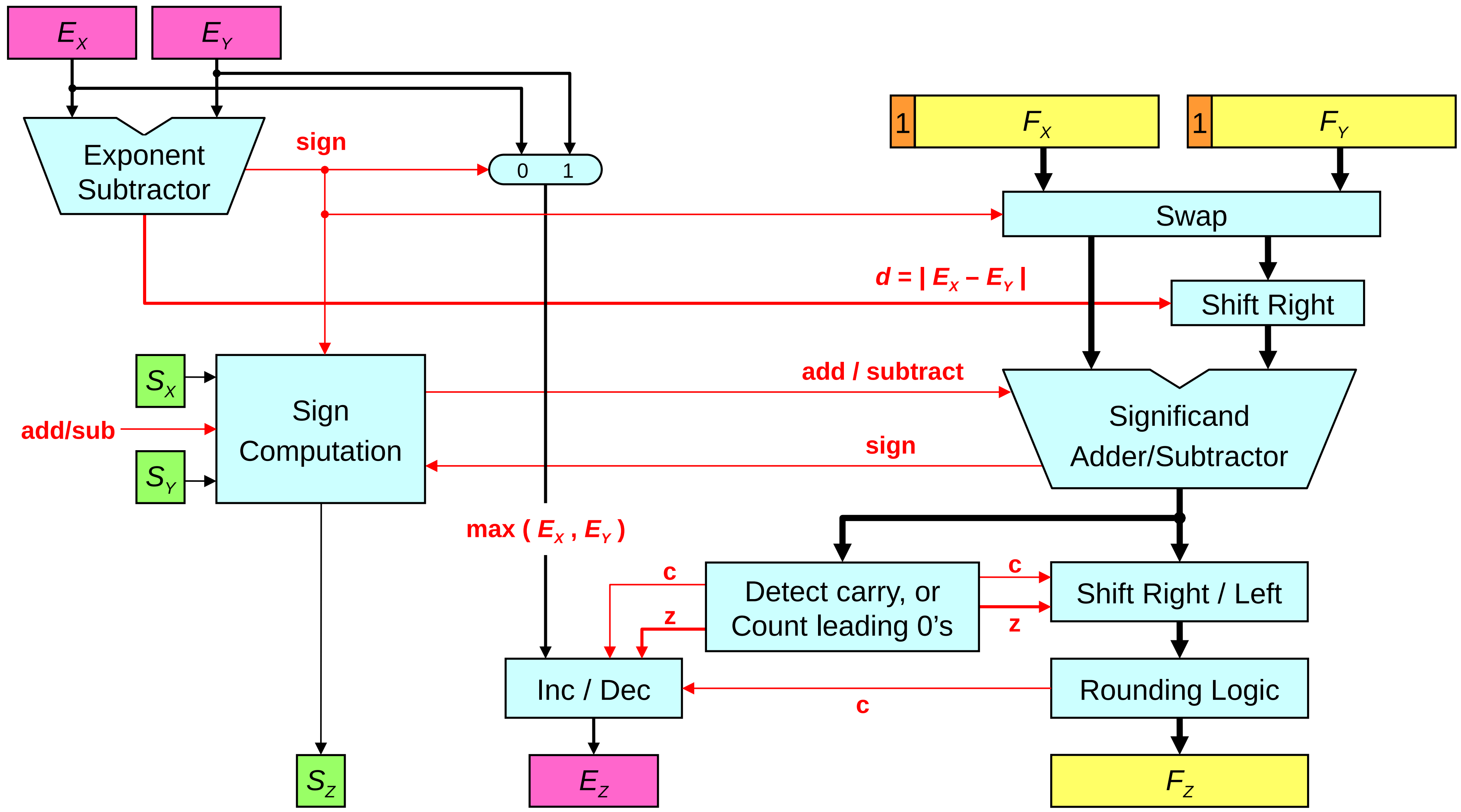
Add significands when signs of *X* and *Y* are identical, Subtract when different.

Convert negative result from 2's complement to sign-magnitude.

Normalization shifts right by 1 if there is a carry, or shifts left by the number of leading zeros in the case of subtraction.

Rounding either truncates fraction, or adds a 1 to least significant fraction bit.

Floating Point Adder Block Diagram



Next . . .

- Floating-Point Numbers
- The IEEE 754 Floating-Point Standard
- Floating-Point Comparison, Addition and Subtraction
- **Floating-Point Multiplication**
- MIPS Floating-Point Instructions and Examples

Floating Point Multiplication Example

- Consider multiplying:

```
   -1.110 1000 0100 0000 1010 0001₂ × 2⁻⁴
×   1.100 0000 0001 0000 0000 0000₂ × 2⁻²
```

- Unlike addition, we add the exponents of the operands
  - Result exponent value = (–4) + (–2) = –6
- Using the biased representation: $E_Z = E_X + E_Y - Bias$
  - E_X = (–4) + 127 = 123 (*Bias* = 127 for single precision)
  - E_Y = (–2) + 127 = 125
  - E_Z = 123 + 125 – 127 = 121 (exponent value = –6)
- Sign bit of product can be computed independently
- Sign bit of product = Sign_X **XOR** Sign_Y = **1** (negative)

Floating-Point Multiplication, cont'd

- Now multiply the significands:

```
            (Multiplicand)
1.11010000100000010100001
            (Multiplier)     ×
1.10000000100000000000000

                    11101000010000000101000001
        11101000010000000101000001
     1.11010000100000010100001
10.10111000[illegible]000000000000
```

- 24 bits × 24 bits → 48 bits (double number of bits)
- Multiplicand × 0 = 0 Zero rows are eliminated
- Multiplicand × 1 = Multiplicand (shifted left)

Floating-Point Multiplication, cont'd

- Normalize Product:

  -10.1011100011111011111110011001... $\times 2^{-6}$

  Shift right and increment exponent because of carry bit

  = **-1.0101110001111101111111001100...** $\times 2^{-5}$

- Round to Nearest Even: (keep only 23 fraction bits)

  -1.01011100011111011111100 | $\textcircled{1}$ $\boxed{\texttt{100...}}$ $\times 2^{-5}$

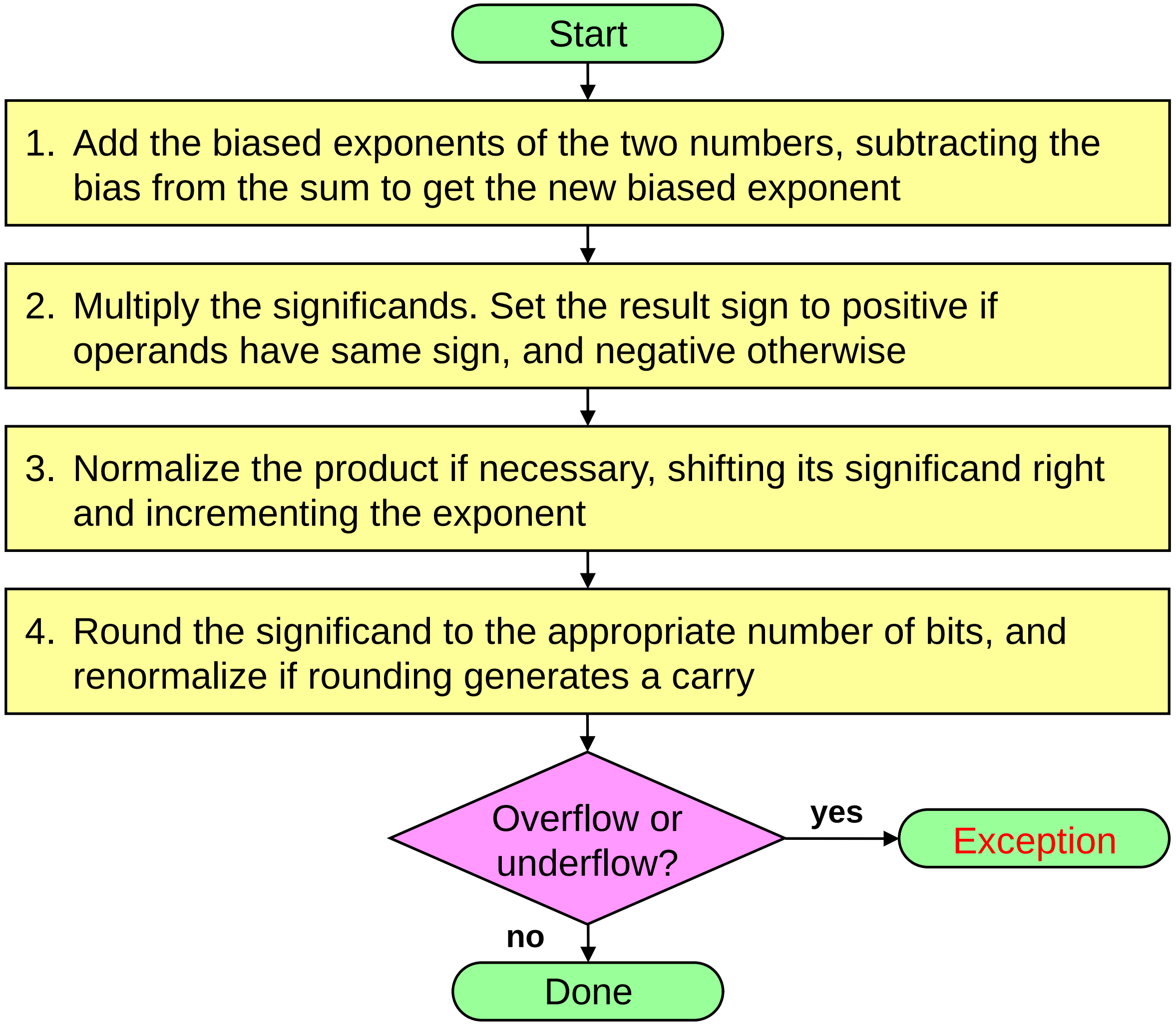
  Round bit = **1**, Sticky bit = **1**, so increment fraction

  Final result = **-1.01011100011111011111101** $\times 2^{-5}$

- IEEE 754 Representation

1	0	1	1	1	1	0	1	0	0	1	0	1	1	1	0	0	0	1	1	1	1	1	0	1	1	1	1	1	1	0	1

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. Set the result sign to positive if operands have same sign, and negative otherwise
3. Normalize the product if necessary, shifting its significand right and incrementing the exponent
4. Round the significand to the appropriate number of bits, and renormalize if rounding generates a carry

Biased Exponent Addition

$E_Z = E_X + E_Y - Bias$

Result sign $S_Z = S_X$ **xor** S_Y can be computed independently

Since the operand significands $1.F_X$ and $1.F_Y$ are ≥ 1 and < 2, their product is ≥ 1 and < 4.

To normalize product, we need to shift right at most by 1 bit and increment exponent

Rounding either truncates fraction, or adds a 1 to least significant fraction bit

Extra Bits to Maintain Precision

- Floating-point numbers are approximations for …
  - Real numbers that they cannot represent
- Infinite real numbers exist between 1.0 and 2.0
  - However, exactly 2^{23} fractions represented in Single Precision
  - Exactly 2^{52} fractions can be represented in Double Precision
- Extra bits are generated in intermediate results when …
  - Shifting and adding/subtracting a *p*-bit significand
  - Multiplying two *p*-bit significands (product is 2*p* bits)
- But when packing result fraction, extra bits are discarded
- Few extra bits are needed: guard, round, and sticky bits
- Minimize hardware but without compromising accuracy

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
- Denormalized numbers to fill the gap
  - Between smallest normalized number $1.0 \times 2^{E_{min}}$ and zero
  - Denormalized numbers , values $0.F \times 2^{E_{min}}$, are closer to zero
  - Gradual underflow to zero

Floating Point Complexities

- Operations are somewhat more complicated
- In addition to overflow we can have underflow
- Accuracy can be a big problem
  - Extra bits to maintain precision: guard, round, and sticky
  - Four rounding modes
  - Division by zero yields Infinity
  - Zero divide by zero yields Not-a-Number
  - Other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse

Next . . .

- Floating-Point Numbers
- The IEEE 754 Floating-Point Standard
- Floating-Point Comparison, Addition and Subtraction
- Floating-Point Multiplication
- **MIPS Floating-Point Instructions and Examples**

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

Floating-Point Arithmetic Instructions

Instruction	Meaning	Op^6	fmt^5	ft^5	fs^5	fd^5	func^6
`add.s $f5,$f3,$f4`	`$f5 = $f3 + $f4`	0x11	0x10	$f4	$f3	$f5	0
`sub.s $f5,$f3,$f4`	`$f5 = $f3 – $f4`	0x11	0x10	$f4	$f3	$f5	1
`mul.s $f5,$f3,$f4`	`$f5 = $f3 × $f4`	0x11	0x10	$f4	$f3	$f5	2
`div.s $f5,$f3,$f4`	`$f5 = $f3 / $f4`	0x11	0x10	$f4	$f3	$f5	3
`sqrt.s $f5,$f3`	`$f5 = sqrt($f3)`	0x11	0x10	0	$f3	$f5	4
`abs.s $f5,$f3`	`$f5 = abs($f3)`	0x11	0x10	0	$f3	$f5	5
`neg.s $f5,$f3`	`$f5 = -($f3)`	0x11	0x10	0	$f3	$f5	7
`add.d $f6,$f2,$f4`	`$f6,7 = $f2,3 + $f4,5`	0x11	0x11	$f4	$f2	$f6	0
`sub.d $f6,$f2,$f4`	`$f6,7 = $f2,3 – $f4,5`	0x11	0x11	$f4	$f2	$f6	1
`mul.d $f6,$f2,$f4`	`$f6,7 = $f2,3 × $f4,5`	0x11	0x11	$f4	$f2	$f6	2
`div.d $f6,$f2,$f4`	`$f6,7 = $f2,3 / $f4,5`	0x11	0x11	$f4	$f2	$f6	3
`sqrt.d $f6,$f2`	`$f6,7 = sqrt($f2,3)`	0x11	0x11	0	$f2	$f6	4
`abs.d $f6,$f2`	`$f6,7 = abs($f2,3)`	0x11	0x11	0	$f2	$f6	5
`neg.d $f6,$f2`	`$f6,7 = -($f2,3)`	0x11	0x11	0	$f2	$f6	7

Floating-Point Load and Store

- Separate floating-point load and 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 **address** register

Instruction	**Meaning**	Op^6	rs^5	ft^5	Immediate^{16}
lwc1 $f2, 8($t0)	$f2 ⚡$_4$ Mem[$t0+8]	0x31	$t0	$f2	8
swc1 $f2, 8($t0)	$f2 🖧$_4$ Mem[$t0+8]	0x39	$t0	$f2	8
ldc1 $f2, 8($t0)	$f2,3 ⚡$_8$ Mem[$t0+8]	0x35	$t0	$f2	8
sdc1 $f2,	$f2,3 🖧$_8$ Mem[$t0+8]	0x3d	$t0	$f2	8

Data Movement Instructions

- Moving data between general purpose and FP registers
  - `mfc1:` move from coprocessor 1 (to a general purpose register)
  - `mtc1:` move to coprocessor 1 (from a 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**	Op^6	fmt^5	rt^5	fs^5	fd^5	func
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	0x10	0	$f2	$f4	6
mov.d $f4, $f2	$f4,5 = $f2,3	0x11	0x11	0	$f2	$f4	6

Convert Instructions

- Convert instruction: `cvt.x.y`
  - Convert the **source** format **y** into **destination** format **x**
- Supported Formats:
  - Single-precision float = `.s`
  - Double-precision float = `.d`
  - Signed integer word = `.w` (in a floating-point register)

Instruction	**Meaning**	Op⁶	fmt⁵		fs⁵	fd⁵	func
`cvt.s.w $f2,$f4`	`$f2 = W2S($f4)`	0x11	0x14	0	$f4	$f2	0x20
`cvt.s.d $f2,$f4`	`$f2 = D2P($f4,5)`	0x11	0x11	0	$f4	$f2	0x20
`cvt.d.w $f2,$f4`	`$f2,3 = W2D($f4)`	0x11	0x14	0	$f4	$f2	0x21
`cvt.d.s $f2,$f4`	`$f2,3 = S2D($f4)`	0x11	0x10	0	$f4	$f2	0x21
`cvt.w.s $f2,$f4`	`$f2 = S2W($f4)`	0x11	0x10	0	$f4	$f2	0x24
`cvt.w.d $f2,$f4`	`$f2 = D2W($f4,5)`	0x11	0x11	0	$f4	$f2	0x24

Floating-Point Compare and Branch

- Floating-Point unit has eight condition code **cc** flags
  - Set to 0 (false) or 1 (true) by any comparison instruction
- Three comparisons: **eq** (equal), **lt** (less than), **le** (less or equal)
- Two branch instructions based on the condition flag

Instruction	Meaning	Op^6	fmt^5	ft^5	fs^5		func
c.eq.s cc $f2,$f4	cc = ($f2 == $f4)	0x11	0x10	$f4	$f2	cc	0x32
c.eq.d cc $f2,$f4	cc = ($f2,3 == $f4,5)	0x11	0x11	$f4	$f2	cc	0x32
c.lt.s cc $f2,$f4	cc = ($f2 < $f4)	0x11	0x10	$f4	$f2	cc	0x3c
c.lt.d cc $f2,$f4	cc = ($f2,3 < $f4,5)	0x11	0x11	$f4	$f2	cc	0x3c
c.le.s cc $f2,$f4	cc = ($f2 <= $f4)	0x11	0x10	$f4	$f2	cc	0x3e
c.le.d cc $f2,$f4	cc = ($f2,3 <= $f4,5)	0x11	0x11	$f4	$f2	cc	0x3e
bc1f cc Label	branch if (cc == 0)	0x11	8	cc,0	16-bit Offset		
bc1t cc Label	branch if (cc == 1)	0x11	8	cc,1	16-bit Offset		

Example 1: Area of a Circle

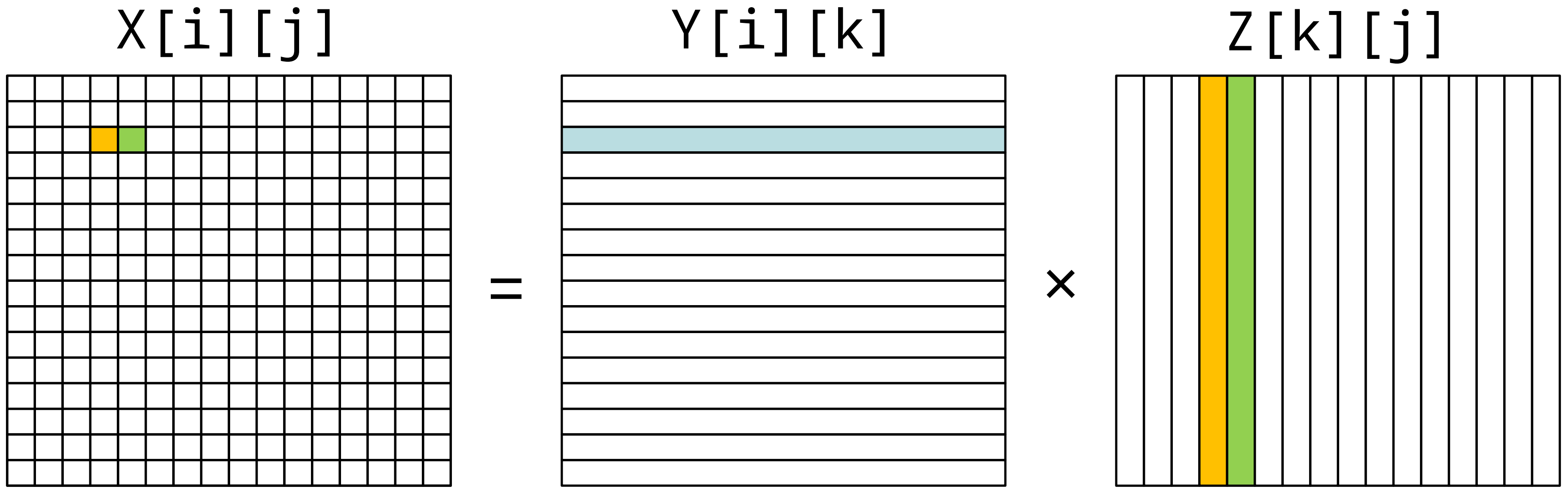
```
.data
  pi:   .double    3.141592653589793
  msg:  .asciiz    "Circle Area = "
.text
main:
  ldc1  $f2, pi    # $f2,3 = pi
  li    $v0, 7     # read double (radius)
  syscall          # $f0,1 = radius
  mul.d $f12, $f0, $f0   # $f12,13 = radius*radius
  mul.d $f12, $f2, $f12  # $f12,13 = area
  la    $a0, msg
  li    $v0, 4     # print string (msg)
  syscall
  li    $v0, 3     # print double (area)
  syscall          # print $f12,13
```

Example 2: Matrix Multiplication

```
void mm (int n, float X[n][n], Y[n][n], Z[n][n]) {
  for (int i=0; i!=n; i=i+1) {
    for (int j=0; j!=n; j=j+1) {
        float sum = 0.0;
      for (int k=0; k!=n; k=k+1) {
        sum = sum + Y[i][k] * Z[k][j];
      }
        X[i][j] = sum;
    }
  }
}
```

- Matrix size is passed in `$a0 = n`
- Matrix addresses in `$a1 = &X`, `$a2 = &Y`, and `$a3 = &Z`
- What is the MIPS assembly code for the procedure?

Access Pattern for Matrix Multiply



Matrix X is accessed by row.

Matrix Y is accessed by row.

Matrix Z accessed by column.

```
&X[i][j] = &X + (i*n + j)*4 = &X[i][j-1] + 4
&Y[i][k] = &Y + (i*n + k)*4 = &Y[i][k-1] + 4
&Z[k][j] = &Z + (k*n + j)*4 = &Z[k-1][j] + 4*n
```

Matrix Multiplication Procedure (1 of 3)

```
# arguments $a0=n, $a1=&X, $a2=&Y, $a3=&Z
mm:     sll  $t0, $a0, 2      # $t0 = n*4 (row size)
        li   $t1, 0     # $t1 = i = 0

# Outer for (i = . . . )  loop starts here
L1:     li   $t2, 0     # $t2 = j = 0

# Middle for (j = . . . ) loop starts here
L2:     li   $t3, 0     # $t3 = k = 0
        move $t4, $a2   # $t4 = &Y[i][0]
        sll  $t5, $t2, 2      # $t5 = j*4
        addu $t5, $a3, $t5    # $t5 = &Z[0][j]
        mtc1 $zero, $f0       # $f0 = sum = 0.0
```

Matrix Multiplication Procedure (2 of 3)

```
# Inner for (k = . . . ) loop starts here
# $t3 = k, $t4 = &Y[i][k], $t5 = &Z[k][j]
L3:    lwc1 $f1, 0($t4)    # load $f1 = Y[i][k]
       lwc1 $f2, 0($t5)    # load $f2 = Z[k][j]
       mul.s     $f3, $f1, $f2  # $f3 = Y[i]
  [k]*Z[k][j]
       add.s     $f0, $f0, $f3  # sum = sum + $f3
       addiu     $t3, $t3, 1    # k = k + 1
       addiu     $t4, $t4, 4    # $t4 = &Y[i][k]
       addu $t5, $t5, $t0  # $t5 = &Z[k][j]
       bne       $t3, $a0, L3   # loop back if (k
  != n)
```

Matrix Multiplication Procedure (3 of 3)

```
        swc1 $f0, 0($a1)    # store X[i][j] = sum
        addiu     $a1, $a1, 4    # $a1 = &X[i][j]

        addiu     $t2, $t2, 1    # j = j + 1
        bne  $t2, $a0, L2   # loop L2 if (j != n)
# End of middle for loop

        addu $a2, $a2, $t0  # $a2 = &Y[i][0]
        addiu     $t1, $t1, 1    # i = i + 1
        bne  $t1, $a0, L1   # loop L1 if (i != n)
# End of outer for loop

        jr   $ra  # return to caller
```