Floating-Point Numbers

Floating Point

Representation for non-integral numbers

- Including very small and very large numbers

Like scientific notation

-2.34×10^{56} normalized

$+0.002 \times 10^{-4}$ not normalized

$+987.02 \times 10^{9}$

In binary

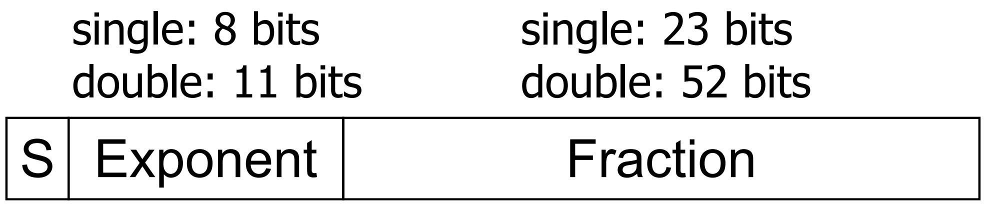
$\pm 1.xxxxxxx_2 \times 2^{yyyy}$

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)

  Types **float** and **double** in C

IEEE Floating-Point Format



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

- 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 = 1203

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: approximately 2^{-23}
  - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- Double: 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: 1011111101000…00

Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float 11000000101000…00

  S = 1

  Fraction = $01000…00_2$

  Exponent = $10000001_2 = 129$

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

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

  $= -5.0$

Infinities and NaNs

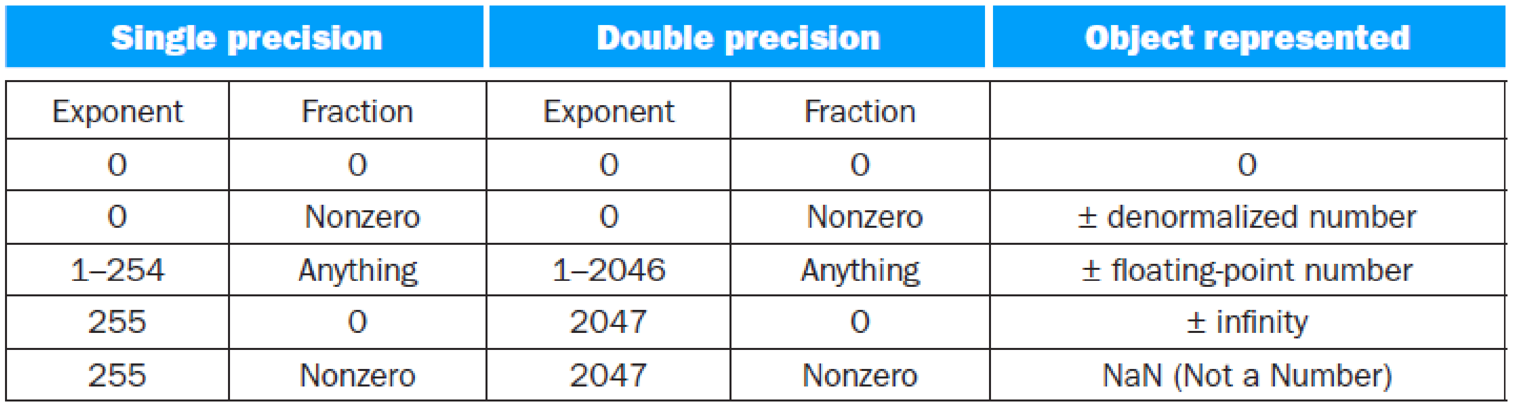
Exponent = 111...1, Fraction = 000...0

- ±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

Figure 3.13



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)

$0.X_{\text{ten}} = 0.\ b\ b\ b\ b\ b\ b \ldots_{\text{two}}$

Let's multiply by 2 on both sides.
Remember multiplying a number in binary by 2 is achieved by shift left one bit.

RHS = b.b b b b b b…

How do we know the b to the left of the binary point is 1 or 0?
It is 1 if the product is >= 1, and is 0 otherwise.
We can found it out by looking at the LHS:

0.X x 2 = **1.X'**, or **0.X** x 2 = **0.X'**

E.g., 0.**75** x 2 = 1.5
X = "75", and X' = "5"

E.g., 0.**45** x 2 = 0.9
X = "45", and X' = "9"

If **0.X** x 2 = **1.X'**, subtract 1 on both side.
Repeat the above process to determine all binary bits on the RHS.

$0.1_{\text{ten}} = 0.\text{bbbb}\ldots_{\text{two}}$?

Converting a fraction number in decimal to binary

```
input: x                                          // a fraction number in decimal
output: binary representation of x        // put in array b

int i = 0
while (x) {
  x = 2 * x
  if (x >= 1) then
                    b[i] = 1
                    x = x -1
  else
                    b[i] = 0
  i++
}
```

Floating-Point Addition (decimal)

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

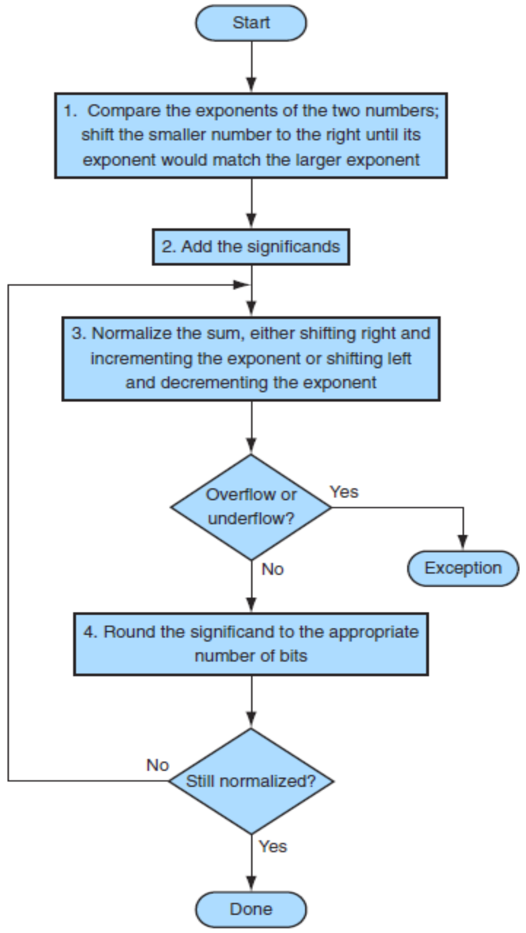

Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

FIGURE 3.14 Floating-point addition. 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.

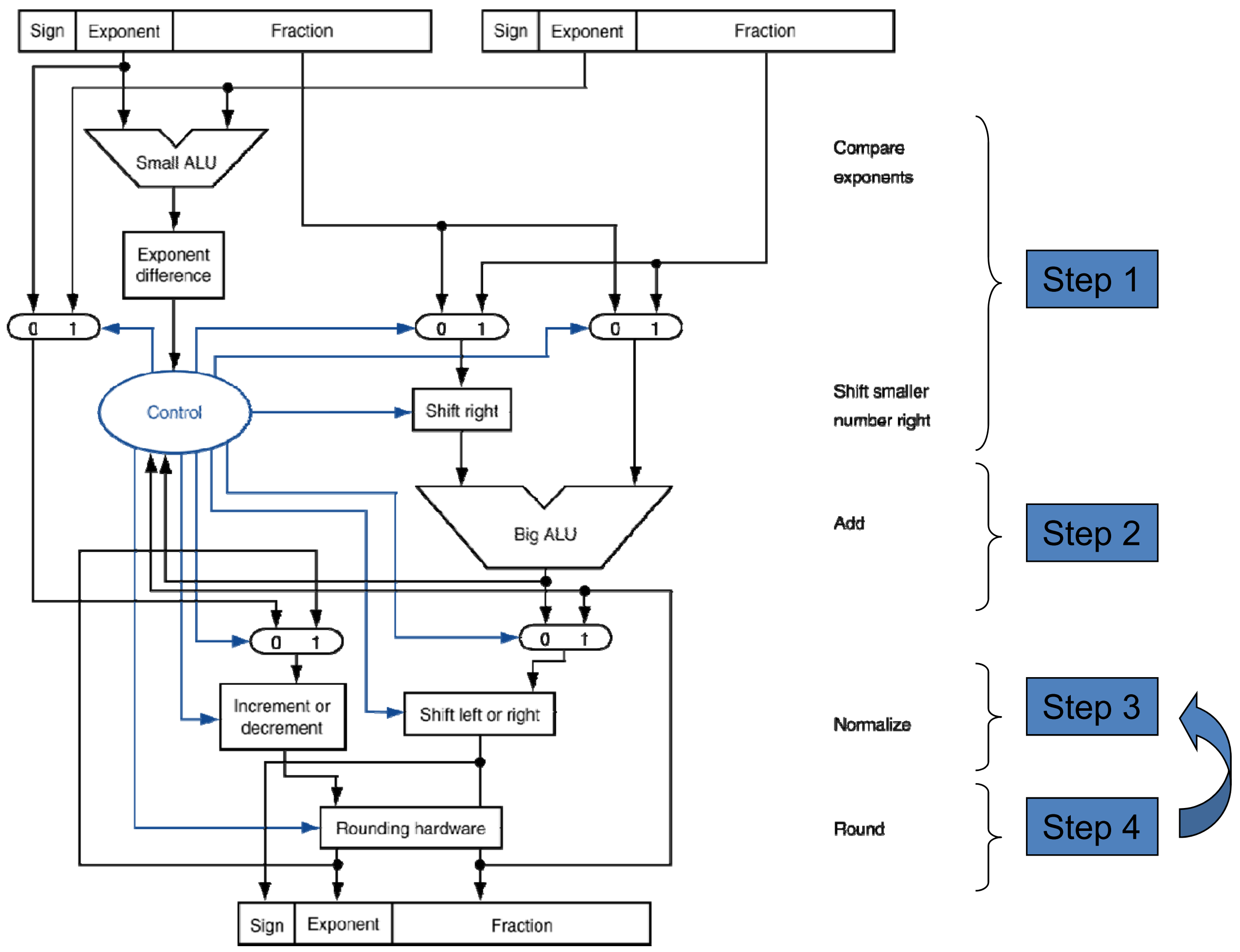
Floating-Point Addition (binary)

- Now consider a 4-digit binary example
  $1.000_2 \times 2^{-1} + (-1.110_2 \times 2^{-2})$, which is $(0.5 + -0.4375)$
- 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 (decimal)

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 (binary)

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

   $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$

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 $\Rightarrow$ –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

FP Instructions in ARM

FP hardware is coprocessor 1
- Adjunct processor that extends the ISA

32 Separate floating-point registers
- s0, s1, … , s31
- Paired for double-precision:
  d0 = (s0, s1); d1 = (s2, s3); …; d15 = (s30, s31)

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: FLDS and FSTS
- e.g.,

```
FLDS  s4, [sp, #0]
FLDS  s6, [sp, #4]
FADDS s2, s4, s6
FSTS  s2, [sp, #8]
```

FP Instructions in ARM

Single-precision arithmetic

- `FADDS`, `FSUBS`, `FMULS`, FDIVS
  - e.g., `FADDS s2, s4, s6`

Double-precision arithmetic

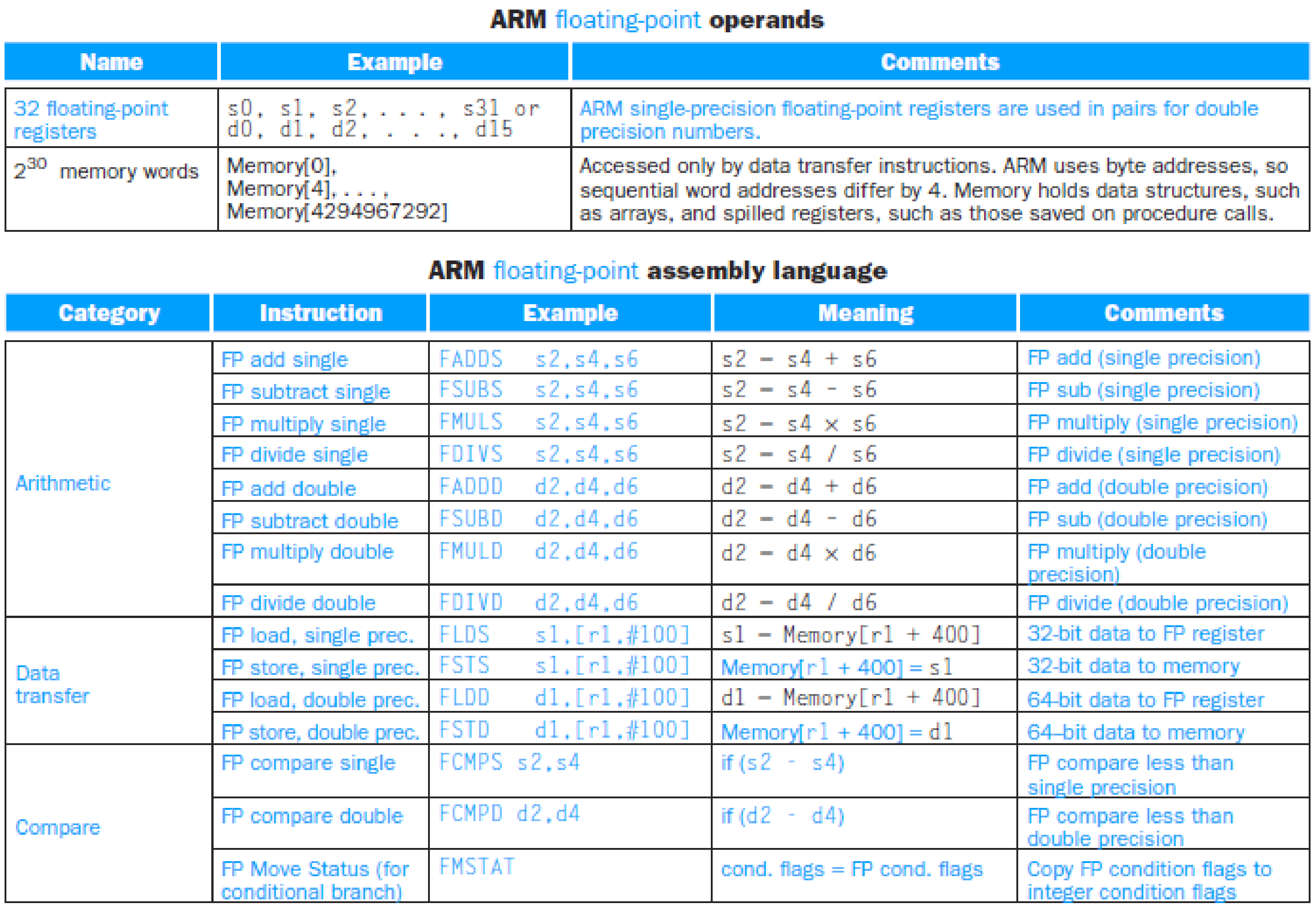
- `FADDD`, `FSUBD`, `FMULD`, `FDIVD`
  - e.g., `FMULD d2, d4, d6`

Data Transfer

- FLDS, FSTS, FLDD, FSTD

Single- and double-precision comparison

- `FCMPS s2, s4`
- `FCMPD d2, d4`
- `FMSTAT`



ARM floating-point operands

Name	Example	Comments
32 floating-point registers	s0, s1, s2, . . . , s31 or d0, d1, d2, . . . , d15	ARM single-precision floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. ARM uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

ARM floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`FADDS s2,s4,s6`	`s2 = s4 + s6`	FP add (single precision)
	FP subtract single	`FSUBS s2,s4,s6`	`s2 = s4 - s6`	FP sub (single precision)
	FP multiply single	`FMULS s2,s4,s6`	`s2 = s4 × s6`	FP multiply (single precision)
	FP divide single	`FDIVS s2,s4,s6`	`s2 = s4 / s6`	FP divide (single precision)
	FP add double	`FADDD d2,d4,d6`	`d2 = d4 + d6`	FP add (double precision)
	FP subtract double	`FSUBD d2,d4,d6`	`d2 = d4 - d6`	FP sub (double precision)
	FP multiply double	`FMULD d2,d4,d6`	`d2 = d4 × d6`	FP multiply (double precision)
	FP divide double	`FDIVD d2,d4,d6`	`d2 = d4 / d6`	FP divide (double precision)
Data transfer	FP load, single prec.	`FLDS s1,[r1,#100]`	`s1 = Memory[r1 + 400]`	32-bit data to FP register
	FP store, single prec.	`FSTS s1,[r1,#100]`	Memory[r1 + 400] = s1	32-bit data to memory
	FP load, double prec.	`FLDD d1,[r1,#100]`	`d1 = Memory[r1 + 400]`	64-bit data to FP register
	FP store, double prec.	`FSTD d1,[r1,#100]`	Memory[r1 + 400] = d1	64–bit data to memory
Compare	FP compare single	`FCMPS s2,s4`	if (s2 - s4)	FP compare less than single precision
	FP compare double	`FCMPD d2,d4`	if (d2 - d4)	FP compare less than double precision
	FP Move Status (for conditional branch)	`FMSTAT`	cond. flags = FP cond. flags	Copy FP condition flags to integer condition flags

FIGURE 3.17 ARM floating-point architecture revealed thus far.

FP Example: convert from °F to °C

- C code:

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

  - `fahr` is in s12, and the literals are in global memory space with offsets from the base address in r12 .
  - Put the result in s0,
- Compiled ARM code:

```
f2c: FLDS  s16, [r12, const5]
     FLDS  s18, [r12, const9]
     FDIVS s16, s16, s18
     FLDS  s18, [r12, const32]
     FSUBS s18, s12, s18
     FMULS s0,  s16, s18
     MOV   pc, lr
```

Accurate Arithmetic

IEEE Std 754 specifies additional rounding control

- Extra bits of precision (guard, round, sticky)
- Choice of rounding modes
- Allows programmer to fine-tune numerical behavior of a computation

Not all FP units implement all options

- Most programming languages and FP libraries just use defaults

Trade-off between hardware complexity, performance, and market requirements

Rounding with Guard Digits

Add $2.56_{ten} \times 10^0$ to $2.34_{ten} \times 10^2$, assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

First we must shift the smaller number to the right to align the exponents, so $2.56_{ten} \times 10^0$ becomes $0.0256_{ten} \times 10^2$. Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents. The guard digit holds 5 and the round digit holds 6. The sum is

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

Thus the sum is $2.3656_{ten} \times 10^2$. Since we have two digits to round, we want values 0 to 49 to round down and 51 to 99 to round up, with 50 being the tiebreaker. Rounding the sum up with three significant digits yields $2.37_{ten} \times 10^2$.

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

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

The answer is $2.36_{ten} \times 10^2$, off by 1 in the last digit from the sum above.

Clicker question

What will be printed by the following java code?

```
double x1 = 0.3;
double x2 = 0.1 + 0.1 + 0.1;
StdOut.println(x1 == x2);
```

A. True
B. False

0.1 + 0.1 + 0.1 = 0.30000000000000004

Clicker Question

What will be printed by the following java code?

```
double x1 = 0.5;
double x2 = 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
StdOut.println(x1 == x2);
```

A. True
B. False

Newton’s method to compute square root

```
#include <stdio.h>

int main(){
            double m;
            double xi;
            printf("type a positive number\n");
             scanf("%lf", &m);
             xi = m;
             while(1) {
                        xi = (xi + m/xi)/2.0;
                        if(xi*xi <= m) break;
            }
            printf("sqrt of %lf = %lf\n", m, xi);
             return 1;
}
```

Clicker question

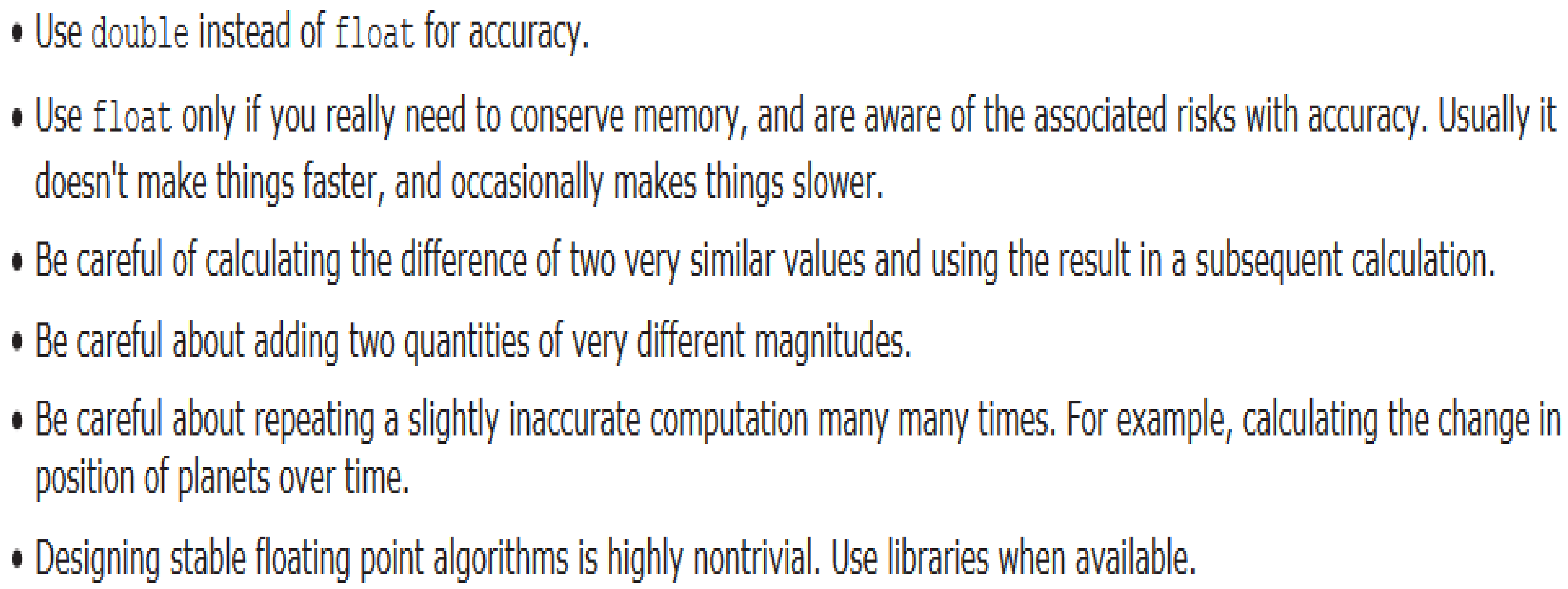
```
double q[8];
...
int x = 0;
while (x < 8) {
  if ( q[x] >= 0 ) return true;
  if ( q[x] < 0 ) ++x;
}
return false;
```

Which answer is wrong regarding the possible outcome of running the above code?

A. Return true
B. Return false
C. Return either true or false
D. Runs into infinite loop
E. None of the above

when you declare the array, you didn't assign numbers to it, so it contains whatever left in that address. There's a possibility that what is there is not a number, remember when the exponent part has all 1s and the fraction has not all 0s, it is not a number, and you cannot get into either if condition, and that's how you can get into infinite loop

Tips for floating-point number accuracy



- Use `double` instead of `float` for accuracy.
- Use `float` only if you really need to conserve memory, and are aware of the associated risks with accuracy. Usually it doesn't make things faster, and occasionally makes things slower.
- Be careful of calculating the difference of two very similar values and using the result in a subsequent calculation.
- Be careful about adding two quantities of very different magnitudes.
- Be careful about repeating a slightly inaccurate computation many many times. For example, calculating the change in position of planets over time.
- Designing stable floating point algorithms is highly nontrivial. Use libraries when available.