UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 26: Overflow Detection in ARM and Floating Point (IEEE 754)

Taylor Johnson

Announcements and Outline

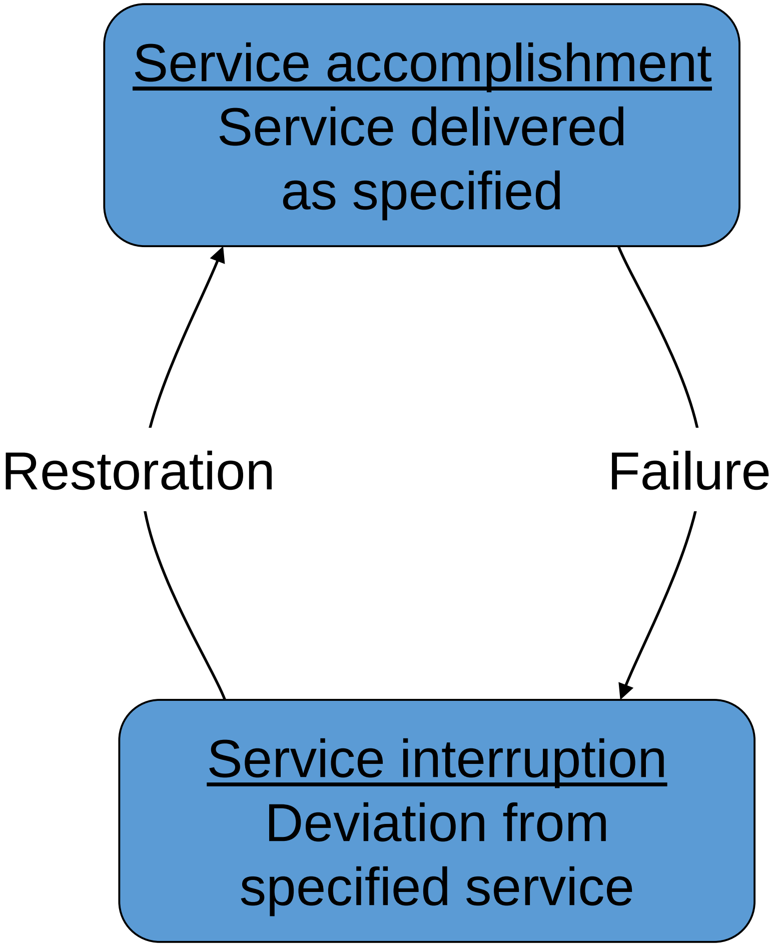
- Programming assignment 3 assigned, due 11/25 by midnight
- Quiz 4 assigned, due by Friday 11/21 by midnight

- Review Dependable memory (briefly)
- Detecting Overflow in ARM (useful for PA3)
- Floating Point

Dependable Memory

Dependability Measures, Error Correcting Codes, RAID, …

Dependability



- Fault: failure of a component
  - May or may not lead to system failure

Dependability Measures

- Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Error Detection – Error Correction

- Memory data can get corrupted, due to things like:
  - Voltage spikes.
  - Cosmic rays.
- The goal in **error detection** is to come up with ways to tell if some data has been corrupted or not.
- The goal in **error correction** is to not only detect errors, but also be able to correct them.
- Both error detection and error correction work by attaching additional bits to each memory word.
- Fewer extra bits are needed for error detection, more for error correction.

Encoding, Decoding, Codewords

- Error detection and error correction work as follows:
- Encoding stage:
  - Break up original data into m-bit words.
  - Each m-bit original word is converted to an n-bit **codeword.**
- Decoding stage:
  - Break up encoded data into n-bit codewords.
  - By examining each n-bit codeword:
    - Deduce if an error has occurred.
    - Correct the error if possible.
    - Produce the original m-bit word.

Parity Bit

- Suppose that we have an *m*-bit word.
- Suppose we want a way to tell if a single error has occurred (i.e., a single bit has been corrupted).
  - No error detection/correction can catch an unlimited number of errors.
- Solution: represent each *m*-bit word using an *(m+1)*-bit codeword.
  - The extra bit is called **parity bit**.
- Every time the word changes, the parity bit is set so as to make sure that the number of 1 bits is even.
  - This is just a convention, enforcing an odd number of 1 bits would also work, and is also used.

Parity Bits - Examples

- Size of original word: *m* = 8.

Original Word (8 bits)	Number of 1s in Original Word	Codeword (9 bits): Original Word + Parity Bit
01101101		
00110000		
11100001		
01011110		

Parity Bits - Examples

- Size of original word: *m* = 8.

Original Word (8 bits)	Number of 1s in Original Word	Codeword (9 bits): Original Word + Parity Bit
01101101	5	011011011
00110000	2	001100000
11100001	4	111000010
01011110	5	010111101

Parity Bit: Detecting A 1-Bit Error

- Suppose now that indeed the memory work has been corrupted in a single bit.
- How can we use the parity bit to detect that?

Parity Bit: Detecting A 1-Bit Error

- Suppose now that indeed the memory work has been corrupted in a single bit.
- How can we use the parity bit to detect that?
- How can a single bit be corrupted?

Parity Bit: Detecting A 1-Bit Error

- Suppose now that indeed the memory work has been corrupted in a single bit.
- How can we use the parity bit to detect that?
- How can a single bit be corrupted?
  - Either it was a 1 that turned to a 0.
  - Or it was a 0 that turned to a 1.
- Either way, the number of 1-bits either increases by 1 or decreases by 1, and **becomes odd**.
- The error detection code just has to check if the number of 1-bits is even.

Error Detection Example

- Size of original word: *m* = 8.
- Suppose that the error detection algorithm gets as input one of the bit patterns on the left column. What will be the output?

Input: Codeword (9 bits): Original Word + Parity Bit	Number of 1s	Error?
011001011		
001100000		
100001010		
010111110		

Error Detection Example

- Size of original word: *m* = 8.
- Suppose that the error detection algorithm gets as input one of the bit patterns on the left colum. What will be the output?

Input: Original Word + Parity Bit (9 bits)	Number of 1s	Error?
011001011	5	yes
001100000	2	no
100001010	3	yes
010111110	6	no

Parity Bit and Multi-Bit Errors

- What if two bits get corrupted?
- The number of 1-bits can:
  - remain the same, or
  - increase by 2, or
  - decrease by 2.
- In all cases, the number of 1-bits remains even.
- The error detection algorithm will not catch this error.
- That is to be expected, a single parity bit is only good for detecting a single-bit error.

The Hamming Distance

- Suppose we have two codewords A and B.
- Each codeword is an n-bit binary pattern.
- We define the distance between A and B to be the number of bit positions where A and B differ.
- This is called the **Hamming distance**.
- One way to compute the Hamming distance:
  - Let C = EXCLUSIVE OR(A, B).
  - Hamming Distance(A, B) = number of 1-bits in C.
- Given a code (i.e., the set of legal codewords), we can find the pair of codewords with the smallest distance.
- We call this minimum distance the **distance of the code.**

Hamming Distance: Example

- What is the Hamming distance between these two patterns?

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

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

- How can we measure this distance?

Hamming Distance: Example

- What is the Hamming distance between these two patterns?

```
1 0 1 1 0 1 0 0 1 0 0 0
0 0 1 1 0 1 0 1 1 0 1 0
```

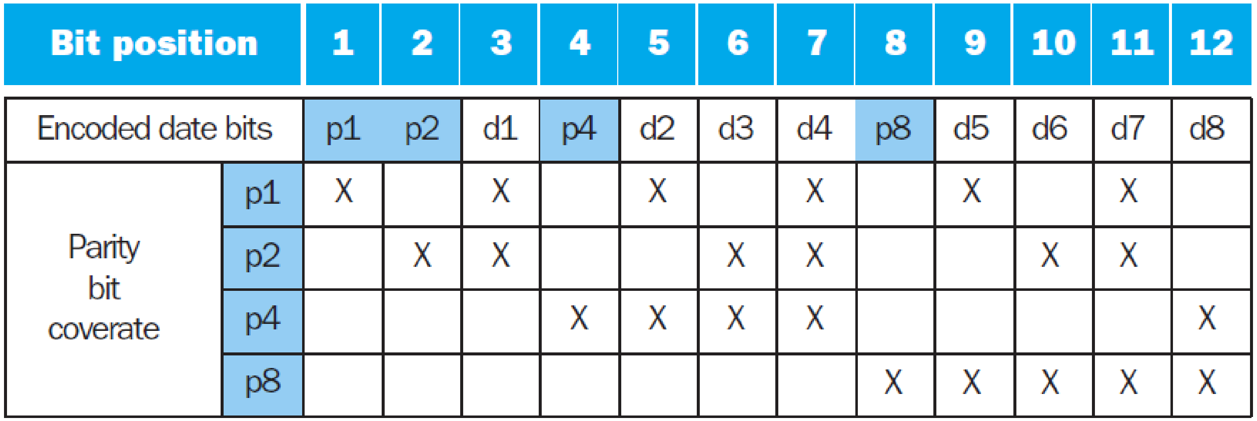
- How can we measure this distance?
  - Find all positions where the two bit patterns differ.
  - Count all those positions.
- Answer: the Hamming distance in the example above is 3.

The Hamming SEC Code

- Hamming distance
  - Number of bits that are different between two bit patterns
- Minimum distance = 2 provides single bit error detection
  - E.g. parity code
- Minimum distance = 3 provides single error correction, 2 bit error detection

Encoding SEC

- To calculate Hamming code:
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - Each parity bit checks certain data bits:



Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverate	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

Decoding SEC

- Value of parity bits indicates which bits are in error
  - Use numbering from encoding procedure
  - E.g.
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 indicates bit 10 was flipped

SEC/DEC Code

- Add an additional parity bit for the whole word (p_n)
- Make Hamming distance = 4
- Decoding:
  - Let H = SEC parity bits
    - H even, p_n even, no error
    - H odd, p_n odd, correctable single bit error
    - H even, p_n odd, error in p_n bit
    - H odd, p_n even, double error occurred
- Note: ECC DRAM uses SEC/DEC with 8 bits protecting each 64 bits

Example: 1-Bit Error Correction

- Size of original word: *m = 3*.
- Number of redundant bits: *r = 3.*
- Size of codeword: *n = 6.*
- Construction:
  - 1 parity bit for bits 1, 2.
  - 1 parity bit for bits 1, 3.
  - 1 parity bit for bits 2, 3.
- You can manually verify that you cannot find any two codewords with Hamming distance 2 (just need to manually check 28 pairs).
- This is a code with distance 3.
- Any 1-bit error can be corrected.

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Example: 1-Bit Error Correction

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?	Most Similar Codeword	Output (original word)
110101			
101000			
110011			
011110			
000010			
101101			
001111			
000110			

- Suppose that the error detection algorithm takes as input bit patterns as shown on the right table.
- What will be the output? How is it determined?

Example: 1-Bit Error Correction

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?	Most Similar Codeword	Output (original word)
110101	Yes	010101	010
101000	Yes	111000	111
110011	No	110011	110
011110	No	011110	011
000010	Yes	000000	000
101101	No	101101	101
001111	Yes	001011	001
000110	Yes	100110	100

- The error detection algorithm:
  - Finds the legal codeword that is most similar to the input.
  - If that legal codeword is not equal to the input, there was an error!
  - Outputs the original word that corresponds to that legal codeword.

Example: 1-Bit Error Correction

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?	Most Similar Codewords	Output (original word)
001100			

- What happens in this case?

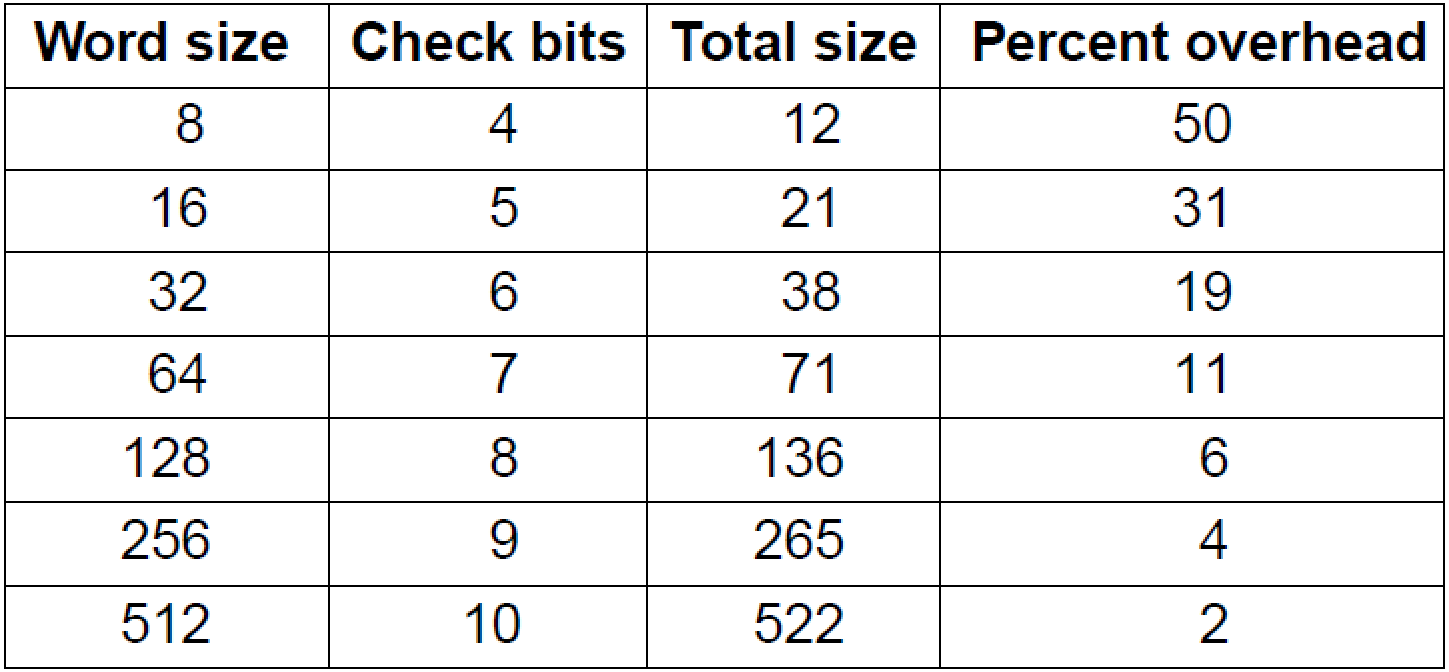
Example: 1-Bit Error Correction

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?	Most Similar Codewords	Output (original word)
001100	Yes	000000 011110 101101	More than 1 bit corrupted, cannot correct!

- No legal codeword is within distance 1 of the input codeword.
- 3 legal codewords are within distance 2 of the input codeword.
- More than 1 bit have been corrupted, the error has been detected, but cannot be corrected.

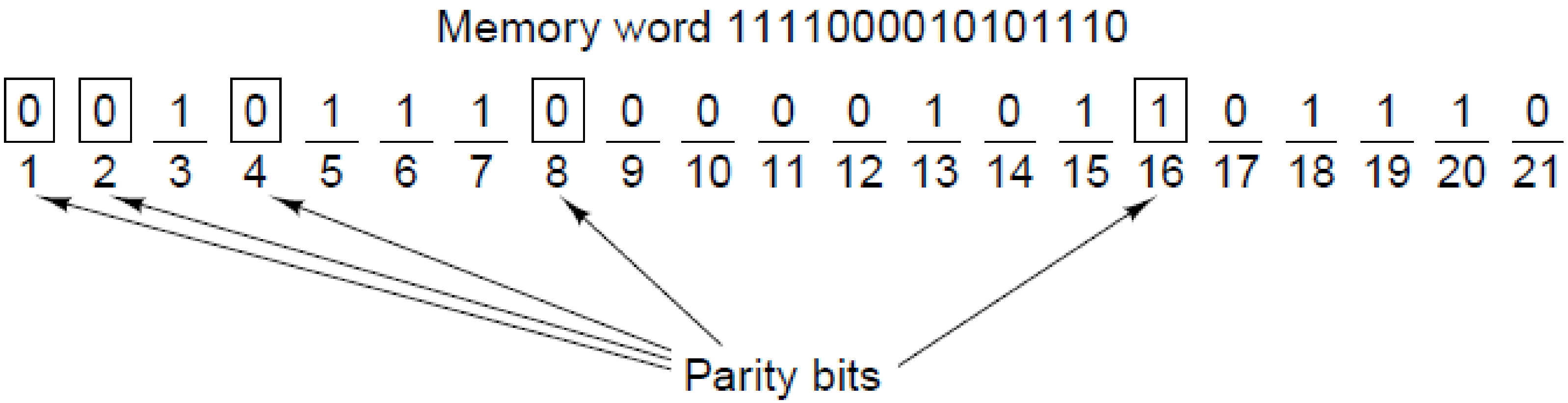
Table of Bits Needed



Word size	Check bits	Total size	Percent overhead
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Number of check bits for a code
that can correct a single error.

An Example Codeword



Memory word 1111000010101110

0	0	1	0	1	1	1	0	0	0	0	0	1	0	1	1	0	1	1	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Parity bits

Construction of the Hamming code
for the memory word 1111000010101110 by
adding 5 check bits to the 16 data bits.

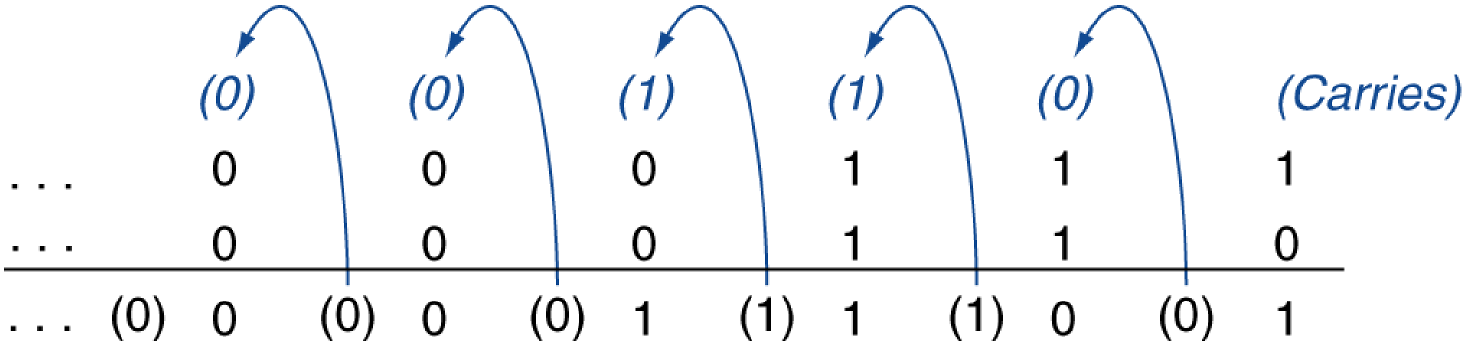
Overflow

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Integer Addition

- Example: 7 + 6



		(0)		(0)		(1)		(1)		(0)		(Carries)
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

$$\begin{array}{ll} +7: & 0000\ 0000\ \ldots\ 0000\ 0111 \\ -6: & 1111\ 1111\ \ldots\ 1111\ 1010 \\ \hline +1: & 0000\ 0000\ \ldots\ 0000\ 0001 \end{array}$$

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Binary Arithmetic

Addition: suppose `r1 = 0x00000005`

```
adds r0, r1, #5
r0 = r1 + #5
r0 = 0x00000005 + #5 (sign extension)
r0 = 0x00000005 + 0x00000005
r0 = 0x0000000A
```

What does the trailing `s` after `add` do?

Update register we use for condition codes

ALU Status Flags

- Application program status register (APSR)
- APSR contains the following ALU status flags
  - N: Set to 1 when the result of the operation is negative, cleared to 0 otherwise
  - Z: Set to 1 when the result of the operation is zero, cleared to 0 otherwise
  - C: Set to 1 when the operation results in a carry, or when a subtraction results in no borrow, cleared to 0 otherwise
  - V: Set to 1 when the operation causes overflow, cleared to 0 otherwise

ARM Condition Codes

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	***C set***	***Carry set / Higher or same (unsigned >=)***
CC or LO	***C clear***	***Carry clear / Lower (unsigned <)***
MI	N set	Negative
PL	N clear	Positive or zero
VS	***V set***	***Overflow (overflow set)***
VC	***V clear***	***No overflow (overflow clear)***

Note: Most instructions update status flags ***only if the S suffix is specified***. CMP, CMN, TEQ, TST ***always*** update condition code flags

ARM Condition Codes (cont)

Suffix	Flags	Meaning
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
HI	C set and Z clear	Higher (unsigned >)

ALU Status Flags

- C is set in one of the following ways:
  - For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an ***unsigned overflow***), and to 0 otherwise
  - For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an ***unsigned underflow***), and to 1 otherwise
  - For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter
  - For other non-addition/subtractions, C is normally left unchanged, but see the individual instruction descriptions for any special cases
- Overflow occurs if the result of a ***signed*** add, subtract, or compare is greater than or equal to 2^{31}, or less than -2^{31}

Conditional Execution

- We’ve already used several types
  - `beq label`
  - `blt label`
  - Etc
- Conditional execution: instruction is executed if condition code is true
  - Example
  - `cmp r0, #0`
  - `moveq r0, #1`
- Same idea as we’ve seen with branch: branch only executed if condition code is true
  - Here, `mov` only executed if `r0 = #0`
- Programming assignment: look at `bvs`, `bvc`, `bcs`, etc.

Back to Arithmetic

Addition: suppose r1 = 0xFFFFFFFF

```
adds r0, r1, #1
r0 = r1 + #1
r0 = 0xFFFFFFFF + #1 (sign extension)
r0 = 0xFFFFFFFF + 0x00000001
r0 = 0x00000000
```

Recall: 0xFFFFFFFF

= b**1**111 1111 1111 1111 1111 1111 1111 1111

Question: does V (overflow of PSR) get set?

No: -1 + 1 = 0, although carry C does get set, and Z is also set (since result is 0)

Back to Arithmetic

Addition: suppose `r1 = 0x7FFFFFFF, r2 = 0x7FFFFFFF`

```
adds r0, r1, r2
r0 = r1 + r2
r0 = 0x7FFFFFFF + 0x7FFFFFFF
r0 = 0xFFFFFFFE
```

Question: does V (overflow of PSR) get set?

Yes: 2*2,147,483,647 > 2^31

Result is: positive + positive = negative number

Floating Point

Representing Fractional Numbers

- Seen several ways to encode information using binary numbers
  - Unsigned integers as binary representation
  - Signed integers using two’s complement
  - Letters using ASCII
  - Etc.

- How can we represent fractional (non-whole) numbers?
  - Fixed-point
  - Floating-point

Fixed-Point

- Suppose we have 16-bits to represent a fractional number
  - Use upper 8 bits to represent whole (integer) portion
  - Use lower 8 bits to represent fractional (non-whole) portion

Whole Part	Decimal Point (.)	Fractional Part
8 bits	.	8 bits
0010 0000	.	0000 0001
20	.	1/256
20	.	0.00390625

- Number of bits reserved for fractional part determines significance of each fractional part
- Here, we have 8 bits, so each fractional part is 1/256, since 2^8 = 256

Why Not Fixed-Point?

- Hard to represent very larger or very small numbers
- Smallest number representable using 64 bits, supposing we keep 32 bits for whole part and 32 bits for fractional part, is:

    1/(2^32) =
0.00000000023283064365386962890625…

- Largest number is still 2^32
- What if we need to represent larger or small numbers?
    - Utilize idea of significant digits
    - If a number is very large, a small deviation results in a small error
    - If a number if very small, a small deviation may result in a large error
    - Utilize relative (percentage) error as opposed to absolute error

Floating Point

- System for representing number where the range of expressible numbers if *independent* of the number of significant digits
- Represent number n in scientific notation:

$$n = f * 10^{e}$$

  - n: number being represented
  - f: fraction (mantissa)
  - e: positive or negative integer
- Examples
  - 3.14 = 0.314 * 10^1 = 3.14 * 10^0
  - 0.000001 = 0.1 * 10^-5 = 1.0 * 10^-6
  - 1941 = 0.1941 * 10^4 = 1.941 * 10^3

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

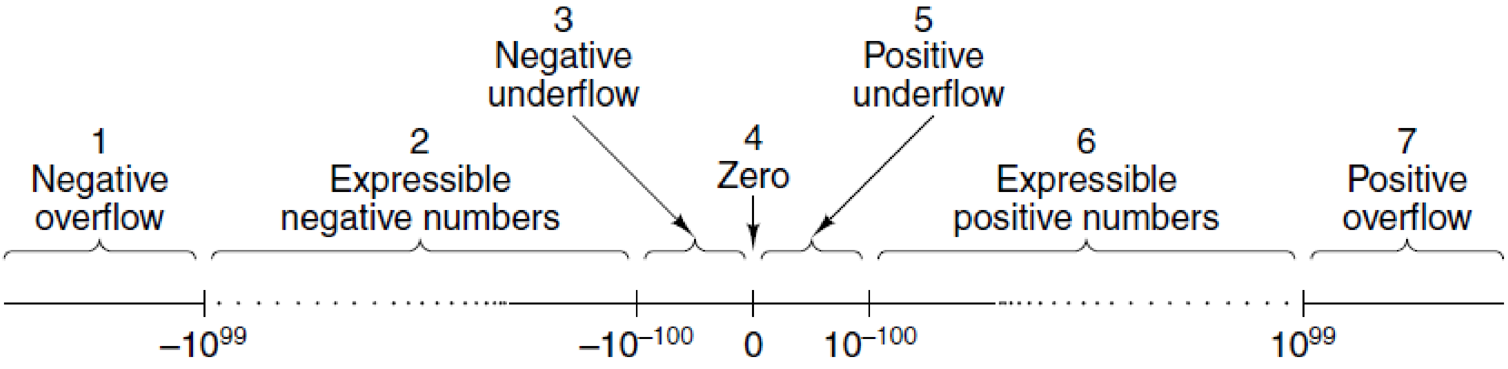
normalized

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Real Number Line Regions

- Divided real number line into seven regions:
  - Large negative numbers less than –0. 999 × 10^{99}
  - Negative between –0.999 × 10^{99} and –0.100×10^{-99}
  - Small negative, magnitudes less than 0.100×10^{-99}
  - Zero
  - Small positive, magnitudes less than 0.100×10^{-99}
  - Positive between 0.100×10^{-99} and 0.999×10^{99}
  - Large positive numbers greater than 0.999×10^{99}



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)

IEEE 754 Floating-Point Format

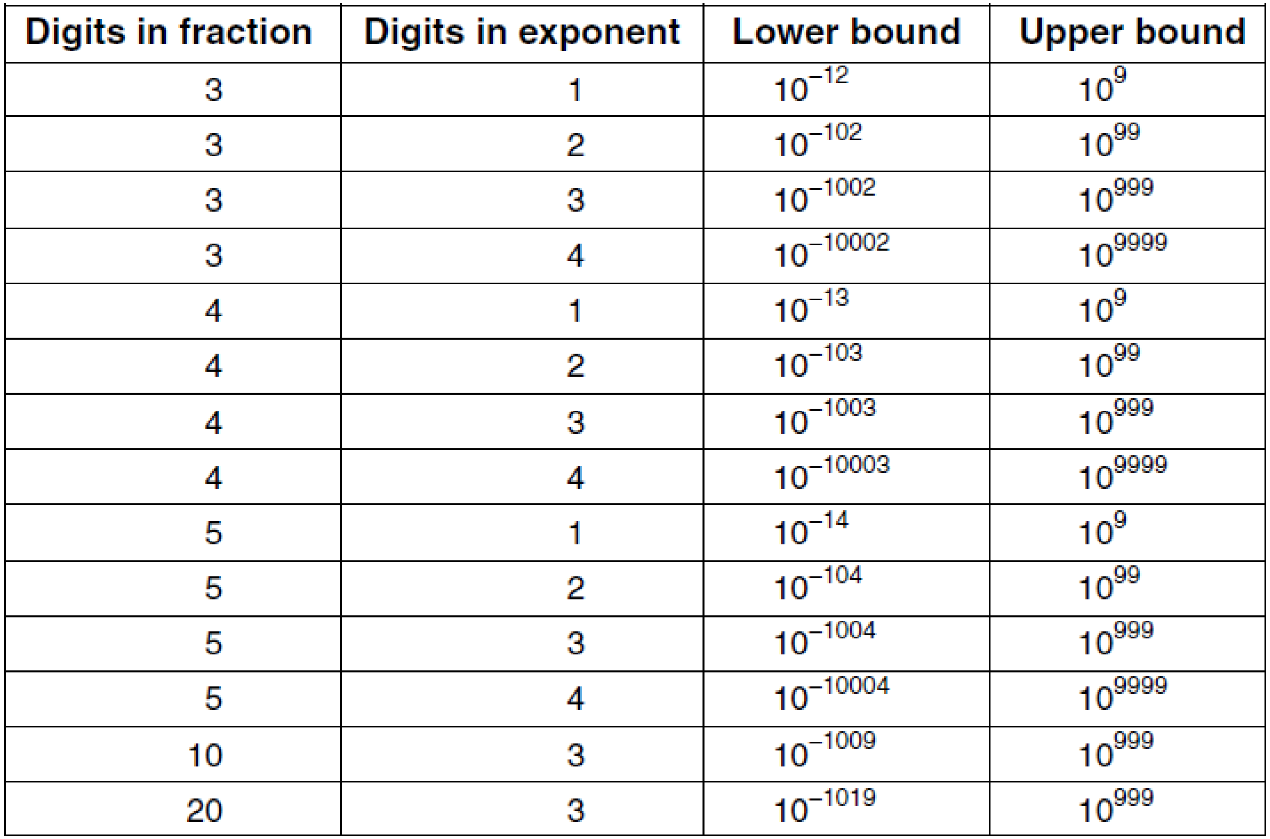
	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
  - 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

Expressible Numbers

- Approximate lower and upper bounds of expressible (unnormalized) floating-point decimal numbers

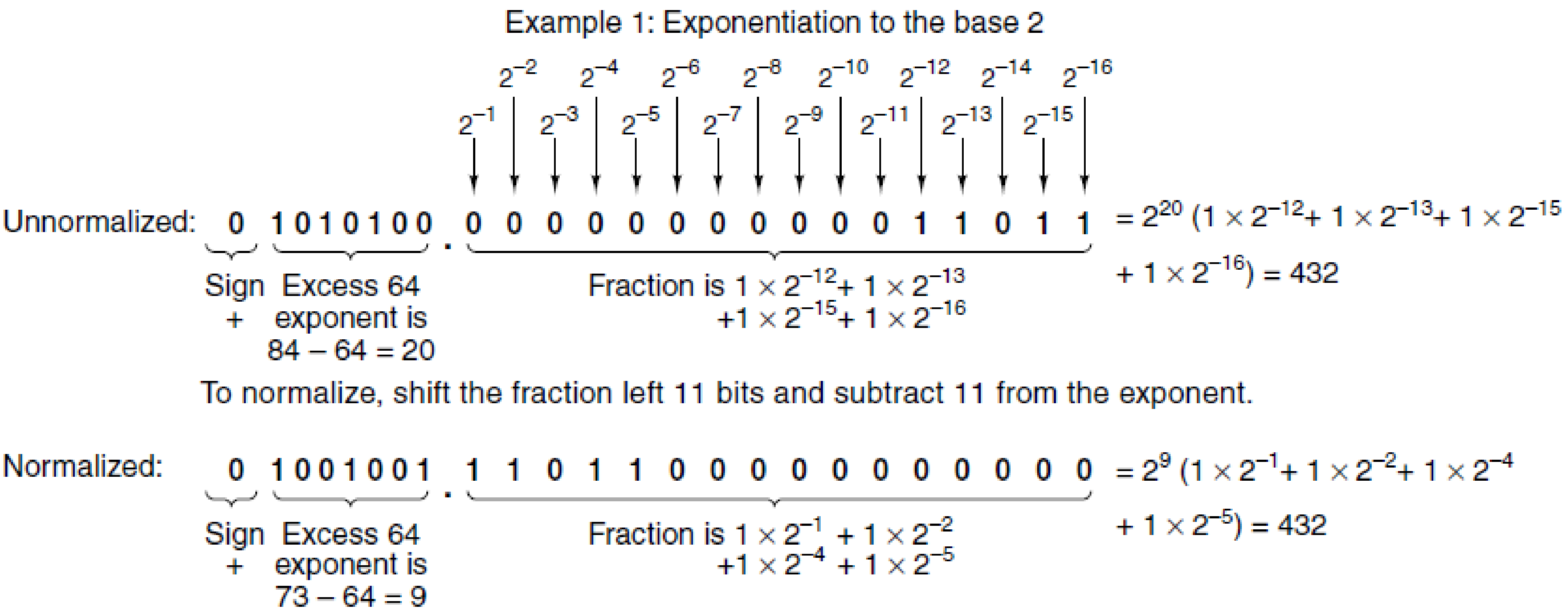


Digits in fraction	Digits in exponent	Lower bound	Upper bound
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Normalization

- Problem: many equivalent representation of same number using the exponent/fraction notation
- Example:
  - 0.5: exponent = -1, fraction = 5: $10^{-1} * 5 = 0.5$
  - 0.5: exponent = -2, fraction = 50: $10^{-2} * 50 = 0.5$
- Binary normalization
  - If leftmost bit is zero, shift all fractional bits left by one and decrease exponent by 1 (assuming no underflow)
  - Fraction with leftmost nonzero bit is normalized
- Benefit: only one normalized representation
  - Simplifies equality comparisons, etc.

Normalization in Binary



Example 1: Exponentiation to the base 2

Unnormalized: $\underbrace{0}_{\text{Sign}}\ \underbrace{1010100}_{\text{Excess 64}} . \underbrace{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1}$ $= 2^{20}(1 \times 2^{-12} + 1 \times 2^{-13} + 1 \times 2^{-15} + 1 \times 2^{-16}) = 432$

Sign +

Excess 64 exponent is $84 - 64 = 20$

Fraction is $1 \times 2^{-12} + 1 \times 2^{-13} + 1 \times 2^{-15} + 1 \times 2^{-16}$

To normalize, shift the fraction left 11 bits and subtract 11 from the exponent.

Normalized: $\underbrace{0}_{\text{Sign}}\ \underbrace{1001001}_{\text{Excess 64}} . \underbrace{1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0}$ $= 2^{9}(1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-4} + 1 \times 2^{-5}) = 432$

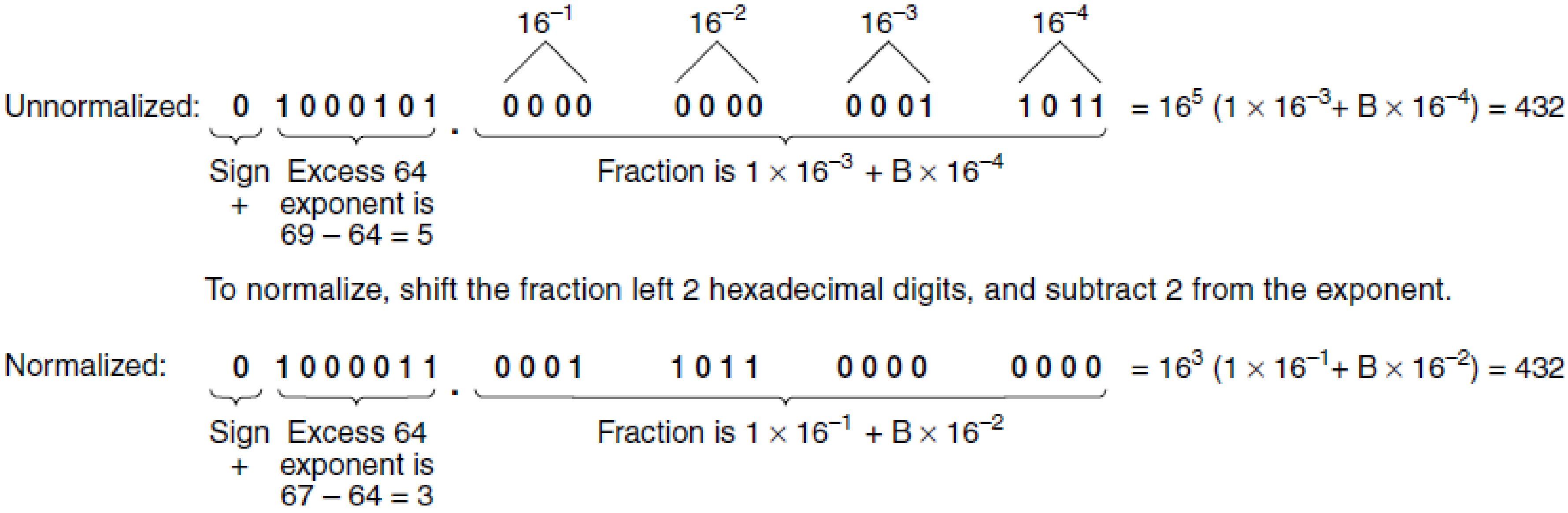
Sign +

Excess 64 exponent is $73 - 64 = 9$

Fraction is $1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-4} + 1 \times 2^{-5}$

Normalization in Hex

Example 2: Exponentiation to the base 16



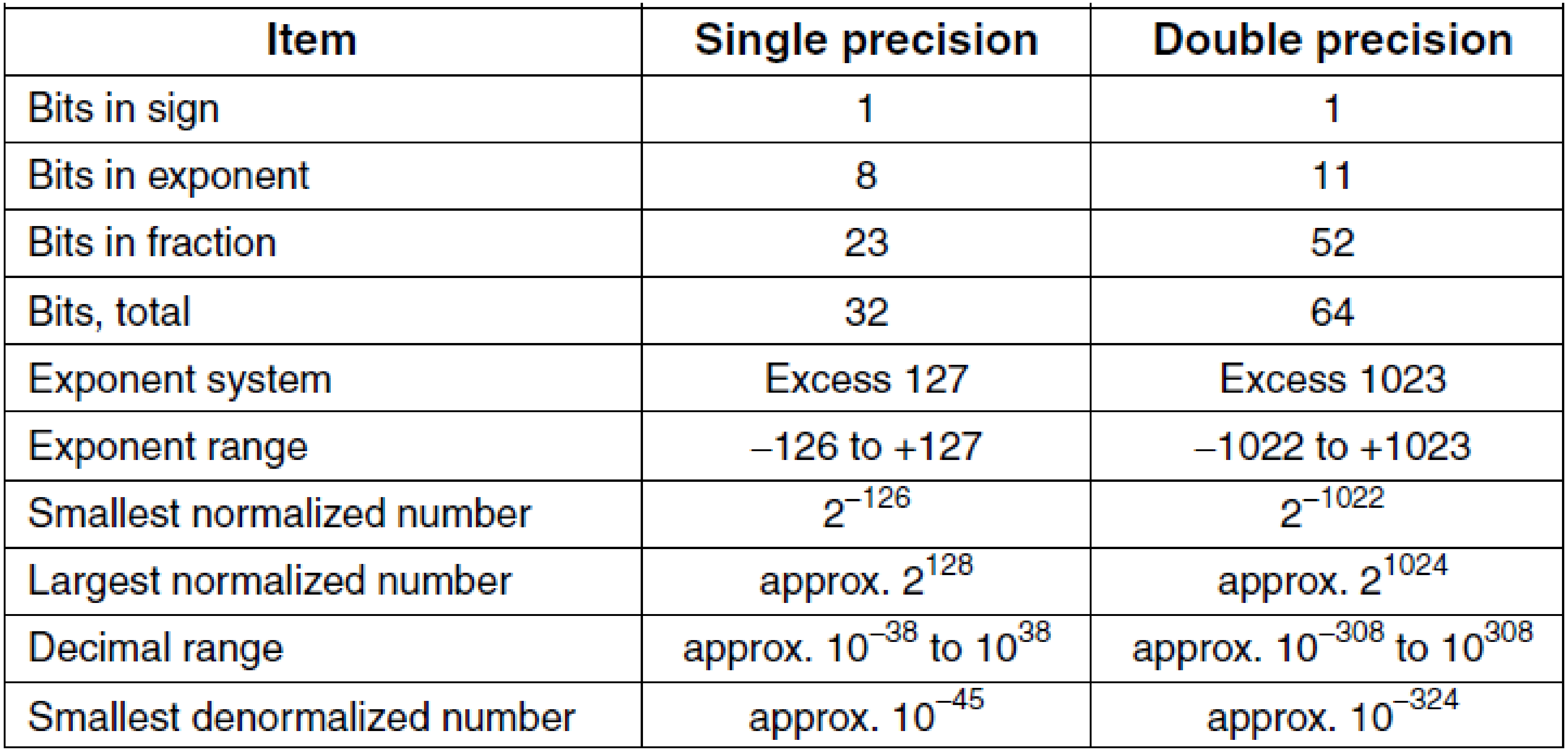
16^{-1} 16^{-2} 16^{-3} 16^{-4}

Unnormalized: $\underbrace{0}_{\text{Sign} +}\ \underbrace{1000101}_{\text{Excess 64 exponent is } 69-64=5} . \underbrace{0000\ 0000\ 0001\ 1011}_{\text{Fraction is } 1 \times 16^{-3} + B \times 16^{-4}}$ $= 16^{5} (1 \times 16^{-3} + B \times 16^{-4}) = 432$

To normalize, shift the fraction left 2 hexadecimal digits, and subtract 2 from the exponent.

Normalized: $\underbrace{0}_{\text{Sign} +}\ \underbrace{1000011}_{\text{Excess 64 exponent is } 67-64=3} . \underbrace{0001\ 1011\ 0000\ 0000}_{\text{Fraction is } 1 \times 16^{-1} + B \times 16^{-2}}$ $= 16^{3} (1 \times 16^{-1} + B \times 16^{-2}) = 432$

IEEE Floating-Point Types



Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	–126 to +127	–1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

IEEE Numerical Types

Normalized	±	0 < Exp < Max	Any bit pattern
Denormalized	±	0	Any nonzero bit pattern
Zero	±	0	0
Infinity	±	1 1 1…1	0
Not a number	±	1 1 1…1	Any nonzero bit pattern

Sign bit

IEEE 754 Example

- $n = sign * 2^e * f$
- 9 = b1.001 * 2^3 = 1.125 * 2^3 = 1.125 * 8 = 9
- Multiply by 2^3 is shift right by 3

Sign	Exponent	Fraction
0	1000 0010	00100000000000000000000

- e = exponent – 127 (biasing)
- f = 1.fraction

IEEE 754 Example

- $n = sign * 2^e * f$
- 5/4 = 1.25 = (-1)^0 * 2^0 * 1.25 = b1.01 = 1 + 1^-2

Sign	Exponent	Fraction
1	0111 1111	01000000000000000000000
-1	127-127=0	1.25

- e = exponent – 127 (biasing)
- f = 1.fraction

IEEE 754 Example

- $n = sign * 2^{e} * f$
- -0.15625 = -5/32 = -1*b1.01 * 2^-3 = b0.00101
- Multiply by 2^-3 is shift left by 3

Sign	Exponent	Fraction
1	0111 1100	01000000000000000000000
-1	124-127=-3	1.25

- e = exponent – 127 (biasing)
- f = 1.fraction
- -5/32 = -0.15625 = -1.25 / 2^3 = -1.25 / 8 = -5/(4*8)

ARM Floating Point

- Instructions prefixed with v, suffixed with, e.g., .f32
- Registers are s0 through s31 and d0 through d15

```
foperandA: .float 3.14
foperandB: .float 2.5
vldr.f32  s1, foperandA    @ s0 = mem[foperandA]
vldr.f32  s2, foperandB    @ s1 = mem[foperandB]
vadd.f32  s0, s1, s2
```

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001 ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110 ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ 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: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 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 = $01000…00_2$
  - Fxponent = 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

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
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (i.e., $0.5 + -0.4375$)
- 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

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

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - “My bank balance is out by 0.0002¢!” ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*
  - Cost hundreds of millions of dollars

Floating-Point Summary

- Floating-point
  - Decimal point moves due to exponents (bit shifting)
  - Positive / negative zeros
- Fixed-point
  - Decimal point remains at fixed point (e.g., after bit 8)
- Spacing between these numbers and real numbers