Today

- How’s Lab 1 going?
  - Some of these are tricky (for us too)
  - Riley has office hours today 12-1pm in CSE 002
  - Section Friday after lecture
  - I’ll be in my office much of the afternoon today, office hours Friday afternoon
- General feedback has been helpful to me; keep it coming.

- Today:
  - integer review
  - floating point (briefly)
  - machine programming?
- Textbook shirt!

4-bit Unsigned vs. Two’s Complement

1 0 1 1

2³ x 1 + 2² x 0 + 2¹ x 1 + 2⁰ x 1

-2³ x 1 + 2² x 0 + 2¹ x 1 + 2⁰ x 1

11 → (math) difference = 16 = 2⁴ → -5
8-bit representations

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001001</td>
<td>10000001</td>
</tr>
<tr>
<td>11111111</td>
<td>00100111</td>
</tr>
</tbody>
</table>

C: casting between unsigned and signed just reinterprets the same bits.

Sign Extension

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>000000010</td>
</tr>
<tr>
<td>1100</td>
<td>11000</td>
</tr>
</tbody>
</table>

00001100  8-bit 12

10001100  8-bit -116
Sign Extension

0 0 1 0  4-bit 2
0 0 0 0 0 0 1 0  8-bit 2

1 1 0 0  4-bit -4
1 1 1 1 1 1 0 0  8-bit -4

Overflow/Wrapping: Unsigned

addition: drop the carry bit

15  1 1 1 1
+ 2  + 0 0 1 0
17  1 0 0 0 1
1

Overflow/Wrapping: Two’s Complement

addition: drop the carry bit

-1  1 1 1 1
+ 2  + 0 0 1 0
1  1 0 0 0 1

6  0 1 1 0
+ 3  + 0 0 1 1
9  1 0 0 1

Modular Arithmetic

Shifting and Arithmetic

x = 27;
y = x << 2;
y == 108

x*2^n

logical shift left:
shift in zeros from the right

overflow

x/2^n

logical shift right:
shift in zeros from the left

unsigned
x = 237;
y = x >> 2;
y == 59
Shifting and Arithmetic

\[ x = -101; \quad y = x << 2; \quad y = 108 \]
\[ x = -19; \quad y = x >> 2; \quad y = -5 \]

\[ x*2^n \quad \text{logical shift left:} \quad 10011011 \quad \text{shift in zeros from the right} \]
\[ x/2^n \quad \text{arithmetic shift right:} \quad 11101101 \quad \text{shift in copies of most significant bit from the left} \]

 clarification from Mon.: shifts by \( n < 0 \) or \( n \geq \) word size are undefined

Floating point topics

- Background: fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover.
  - It’s a 58-page standard...

Roadmap

Data & addressing
Integers & floats
Machine code & C
x86 assembly
programming
Procedures &
stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:

Windows 8

Computer system:

Machine code:

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:

```
get_mpq:
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret
```

Fractional Binary Numbers

- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number: \[ \sum_{k=-j}^{i} b_k \cdot 2^k \]
**Fractional Binary Numbers**

- **Value** | **Representation**
  - 5 and 3/4 | $101.11_2$
  - 2 and 7/8 | $10.111_2$
  - 47/64 | $0.1011111_2$

- **Observations**
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form $0.111111..._2$ are just below 1.0

- **Limitations**
  - Exact representation possible only for numbers of the form $x \times 2^y$
  - Other rational numbers have repeating bit representations
    - $1/3 = 0.333333..._{10} = 0.01010101[01]..._2$

**Fixed Point Representation**

- **Implied binary point.** Examples:
  - #1: the binary point is between bits 2 and 3
    - $b_7 b_6 b_5 b_4 [.] b_3 b_2 b_1 b_0$
  - #2: the binary point is between bits 4 and 5
    - $b_7 b_6 b_5 [.] b_4 b_3 b_2 b_1 b_0$

- **Same hardware as for integer arithmetic.**
  - #3: integers! the binary point is after bit 0
    - $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 [.]$

- **Fixed point = fixed range and fixed precision**
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers

**IEEE Floating Point**

- Analogous to scientific notation
  - 12000000 | $1.2 \times 10^7$ | C: 1.2e7
  - 0.0000012 | $1.2 \times 10^{-6}$ | C: 1.2e-6

- **IEEE Standard 754 used by all major CPUs today**

- **Driven by numerical concerns**
  - Rounding, overflow, underflow
  - Numerically well-behaved, but hard to make fast in hardware

**Floating Point Representation**

- **Numerical form:**
  - $V_{10} = (-1)^s \times M \times 2^E$

  - Sign bit $s$ determines whether number is negative or positive
  - Significand (mantissa) $M$ normally a fractional value in range [1.0,2.0)
  - Exponent $E$ weights value by a (possibly negative) power of two
Floating Point Representation

- **Numerical form:**
  \[ V_{10} = (-1)^s \times M \times 2^E \]
  - Sign bit \( s \) determines whether number is negative or positive
  - Significand (mantissa) \( M \) normally a fractional value in range \([1.0, 2.0)\)
  - Exponent \( E \) weights value by a (possibly negative) power of two

- **Representation in memory:**
  - MSB \( a \) is sign bit \( s \)
  - \( \text{exp} \) field encodes \( E \) (but is \textit{not equal} to \( E \))
  - \( \text{frac} \) field encodes \( M \) (but is \textit{not equal} to \( M \))

\[
\begin{array}{c|c|c}
\text{s} & \text{exp} & \text{frac} \\
\end{array}
\]

Precisions

- **Single precision:** 32 bits
  \[
  \begin{array}{c|c|c}
  s & \text{exp} & \text{frac} \\
  \hline
  1 & 8 & 23 \\
  \end{array}
  \]

- **Double precision:** 64 bits
  \[
  \begin{array}{c|c|c}
  s & \text{exp} & \text{frac} \\
  \hline
  1 & 11 & 52 \\
  \end{array}
  \]

- Finite representation means not all values can be represented exactly. Some will be approximated.

Normalization and Special Values

- **\( V = (-1)^s \times M \times 2^E \)\**

  - "Normalized" = \( M \) has the form 1.xxxxx
    - As in scientific notation, but in binary
    - 0.011 x 2\(^3\) and 1.1 x 2\(^2\) represent the same number, but the latter makes better use of the available bits
    - Since we know the mantissa starts with a 1, we don't bother to store it

- How do we represent 0.0? Or special / undefined values like 1.0/0.0?

\[
\begin{array}{c|c|c}
\text{s} & \text{exp} & \text{frac} \\
\end{array}
\]

Normalization and Special Values

- **\( V = (-1)^s \times M \times 2^E \)\**

  - "Normalized" = \( M \) has the form 1.xxxxx
    - As in scientific notation, but in binary
    - 0.011 x 2\(^3\) and 1.1 x 2\(^2\) represent the same number, but the latter makes better use of the available bits
    - Since we know the mantissa starts with a 1, we don't bother to store it

- **Special values:**
  - \( \text{zero} \):
    \[
    \begin{array}{c|c|c}
    s & \text{exp} & \text{frac} \\
    \hline
    0 & 00...0 & 00...0 \\
    \end{array}
    \]
  - \( +\infty, -\infty \):
    \[
    \begin{array}{c|c|c}
    s & \text{exp} & \text{frac} \\
    \hline
    0 & 11...1 & 00...0 \\
    \end{array}
    \]
  - 1.0/0.0 = -1.0/-0.0 = +\infty, 1.0/-0.0 = -1.0/0.0 = -\infty
  - \( \text{NaN} \) ("Not a Number"):
    \[
    \begin{array}{c|c|c}
    s & \text{exp} & \text{frac} \\
    \hline
    0 & 11...1 & \text{!00...0} \\
    \end{array}
    \]
    Results from operations with undefined result: sqrt(-1), \( \infty - \infty \), \( \infty \times 0 \), etc.
  - note: \text{exp}=11...1 and \text{exp}=00...0 are reserved, limiting \text{exp} range...
Floating Point Operations: Basic Idea

\[ V = (-1)^s \times M \times 2^E \]

- \[ x +_\varepsilon y = \text{Round}(x + y) \]
- \[ x \times_\varepsilon y = \text{Round}(x \times y) \]

- Basic idea for floating point operations:
  - First, compute the exact result
  - Then, round the result to make it fit into desired precision:
    - Possibly overflow if exponent too large
    - Possibly drop least-significant bits of significand to fit into \( \text{frac} \)

Mathematical Properties of FP Operations

- Exponent overflow yields \(+\infty\) or \(-\infty\)

- Floats with value \(+\infty\), \(-\infty\), and NaN can be used in operations
  - Result usually still \(+\infty\), \(-\infty\), or NaN; sometimes intuitive, sometimes not

- Floating point operations are not always associative or distributive, due to rounding!
  - \((3.14 + 1e10) - 1e10 \neq 3.14 + (1e10 - 1e10)\)
  - \(1e20 * (1e20 - 1e20) \neq (1e20 * 1e20) - (1e20 * 1e20)\)

Floating Point in C

- C offers two levels of precision
  - \float\: single precision (32-bit)
  - \double\: double precision (64-bit)

- \#include <math.h> to get INFINITY and NAN constants

- Equality (\(==\)) comparisons between floating point numbers are tricky, and often return unexpected results
  - Just avoid them!

Conversions between data types:

- Casting between \int, \float, and \double changes the bit representation.
  - \int \rightarrow \float
    - May be rounded; overflow not possible
  - \int \rightarrow \double or \float \rightarrow \double
    - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
  - \long \int \rightarrow \double
    - Rounded or exact, depending on word size
  - \double or \float \rightarrow \int
    - Truncates fractional part (rounded toward zero)
    - Not defined when out of range or NaN: generally sets to Tmin
Number Representation Really Matters

- 1991: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- 1996: Ariane 5 rocket exploded (§1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- 2000: Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- 2038: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to Tmin in 2038
- other related bugs
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown "smart" warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)

Many more details for the curious...

- Exponent bias
- Denormalized values – to get finer precision near zero
- Distribution of representable values
- Floating point multiplication & addition algorithms
- Rounding strategies

We won’t be using or testing you on any of these extras in 351.

Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”
- Mathematically equivalent ways of writing an expression may compute different results
  - Violates associativity/distributivity
- Never test floating point values for equality!
- Careful when converting between ints and floats!

HW 1 will be posted this afternoon

- due Wednesday, July 10.
- Integers
- Floats (simple stuff)
- Disassembly
Normalized Values

\[ V = (-1)^s \cdot M \cdot 2^E \]

- **Condition:** `exp ≠ 000...0` and `exp ≠ 111...1`
- **Exponent coded as biased value:** \( E = \text{exp} - \text{Bias} \)
  - `exp` is an unsigned value ranging from 1 to \(2^{k-1} - 1\)
  - `Bias = 2^{k-1} - 1`
    - Single precision: 127 (so `exp`: 1...254, \(E\): -126...127)
    - Double precision: 1023 (so `exp`: 1...2046, \(E\): -1022...1023)
  - These enable negative values for \(E\), for representing very small values

- **Significand coded with implied leading 1:** \( M = 1 . \text{xxx...x}_2 \)
  - \( \text{xxx...x}\): the \(n\) bits of `frac`
  - Minimum when \(000...0\) (\(M = 1.0\))
  - Maximum when \(111...1\) (\(M = 2.0 - \epsilon\))
  - Get extra leading bit for “free”

Denormalized Values

- **Condition:** `exp = 000...0`

- **Exponent value:** \( E = \text{exp} - \text{Bias} + 1 \) (instead of \(E = \text{exp} - \text{Bias}\))
- **Significand coded with implied leading 0:** \( M = 0 . \text{xxx...x}_2 \)
  - \( \text{xxx...x}\): bits of `frac`

- **Cases**
  - `exp = 000...0, frac = 000...0`
    - Represents value 0
    - Note distinct values: +0 and –0 (why?)
  - `exp = 000...0, frac ≠ 000...0`
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced

Normalized Encoding Example

\[ V = (-1)^s \cdot M \cdot 2^E \]

- **Value:** `float f = 12345.0;`
  - \(12345_{10} = 1100000011_2\)
    - \(1.100000011_2 \times 2^{13}\) (normalized form)
- **Significant:**
  - \(M = 1.10000001111001_2\)
  - `frac` = \(1000001110010000000000000_2\)
- **Exponent:** \(E = \text{exp} - \text{Bias}\), so `exp = E + Bias`
  - \(E\) = 13
  - `Bias` = 127
  - `exp` = 140 = 10001100_2

- **Result:**
  - \(s\) \(\text{exp}\) \(\text{frac}\)

Special Values

- **Condition:** `exp = 111...1`

- **Case:** `exp = 111...1, frac = 000...0`
  - Represents value \(\infty\) (infinity)
  - Operation that overflows
  - Both positive and negative
    - E.g., \(1.0/0.0 = -1.0/-0.0 = +\infty\), \(1.0/-0.0 = -1.0/0.0 = -\infty\)

- **Case:** `exp = 111...1, frac ≠ 000...0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
    - E.g., `sqrt(-1)`, \(\infty - \infty\), \(\infty \times 0\)`
Visualization: Floating Point Encodings

Denormalized numbers:
-0 0000 000 -6 0 1/8*1/64 = 1/512  closest to zero
-0 0000 001 -6 1/8*1/64 = 1/512
-0 0000 010 -6 2/8*1/64 = 2/512
...
-0 0000 110 -6 6/8*1/64 = 6/512
-0 0000 111 -6 7/8*1/64 = 7/512  largest denorm

Normalized numbers:
-0 0001 000 -6 8/8*1/64 = 8/512  smallest norm
-0 0001 001 -6 9/8*1/64 = 9/512
...
-0 0110 110 -1 14/8*1/2 = 14/16
-0 0110 111 -1 15/8*1/2 = 15/16  closest to 1 below
-0 0111 000 0 8/8*1 = 1
-0 0111 001 0 9/8*1 = 9/8  closest to 1 above
-0 0111 010 0 10/8*1 = 10/8
-0 1110 110 7 14/8*128 = 224
-0 1110 111 7 15/8*128 = 240  largest norm
-0 1111 000 n/a inf

Tiny Floating Point Example

8-bit Floating Point Representation
- the sign bit is in the most significant bit.
- the next four bits are the exponent, with a bias of 7.
- the last three bits are the \[ \text{frac} \]

Same general form as IEEE Format
- normalized, denormalized
- representation of 0, NaN, infinity

Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>00000001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512 closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>00000100</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td>0</td>
<td>00000111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512 largest denorm</td>
</tr>
<tr>
<td>0</td>
<td>00001000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512 smallest norm</td>
</tr>
<tr>
<td>0</td>
<td>00001001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00110110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td>0</td>
<td>00110111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16 closest to 1 below</td>
</tr>
<tr>
<td>0</td>
<td>00111000</td>
<td>0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td>0</td>
<td>00111001</td>
<td>0</td>
<td>9/8*1 = 9/8 closest to 1 above</td>
</tr>
<tr>
<td>0</td>
<td>00111010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td>0</td>
<td>01101110</td>
<td>7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td>0</td>
<td>01101111</td>
<td>7</td>
<td>15/8*128 = 240 largest norm</td>
</tr>
<tr>
<td>0</td>
<td>01110000</td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>

Distribution of Values

6-bit IEEE-like format
- e = 3 exponent bits
- f = 2 fraction bits
- Bias is 2^{3-1} = 3

Notice how the distribution gets denser toward zero.
Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Special Properties of Encoding

- Floating point zero (0*) exactly the same bits as integer zero
  - All bits = 0

- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider 0* = 0t = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>00...00</td>
<td>00...00</td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00...00</td>
<td>11...11</td>
<td>(1.0 – ε) * 2⁻¹²８</td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>1.0 * 2⁻¹²８</td>
</tr>
<tr>
<td>Smallest Pos. Norm.</td>
<td>00...01</td>
<td>00...00</td>
<td>2⁻¹²³</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11...10</td>
<td>11...11</td>
<td>(2.0 – ε) * 2¹２³</td>
</tr>
</tbody>
</table>

Floating Point Multiplication

\((-1)^{x1} M_1 \ 2^{E_1} \ast (-1)^{x2} M_2 \ 2^{E_2}\)

- Exact Result: \((-1)^x M \ 2^E\)
  - Sign s:
    - s1 ^ s2 \ // xor of s1 and s2
  - Significand M:
    - M1 ^ M2
  - Exponent E:
    - E1 + E2

- Fixing
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit frac precision
Floating Point Addition

\[(-1)^{s_1} M_1 \times 2^{E_1} + (-1)^{s_2} M_2 \times 2^{E_2}\]

Assume \(E_1 > E_2\)

- **Exact Result:** \((-1)^s M \times 2^E\)
  - Sign \(s\), significand \(M\):
    - Result of signed align & add
  - Exponent \(E\): \(E_1\)

- **Fixing**
  - If \(M \geq 2\), shift \(M\) right, increment \(E\)
  - If \(M < 1\), shift \(M\) left \(k\) positions, decrement \(E\) by \(k\)
  - Overflow if \(E\) out of range
  - Round \(M\) to fit frac precision

Rounding Binary Numbers

- **Binary Fractional Numbers**
  - “Half way” when bits to right of rounding position = 100...2

- **Examples**
  - Round to nearest 1/4 (2 bits right of binary point)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3/32</td>
<td>10.000112</td>
<td>10.002</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.001102</td>
<td>10.012</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.111002</td>
<td>11.002</td>
<td>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.101002</td>
<td>10.102</td>
<td>(1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>

Closer Look at Round-To-Even

- **Default Rounding Mode**
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- **Applying to Other Decimal Places / Bit Positions**
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth
    - 1.2349999 1.23 (Less than half way)
    - 1.2350001 1.24 (Greater than half way)
    - 1.2350000 1.24 (Half way—round up)
    - 1.2450000 1.24 (Half way—round down)

Floating Point and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for ( i=0; i<10; i++ ) {
        f2 += 1.0/10.0;
    }
    printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.8f\n", f1);
    printf("f2 = %10.8f\n", f2);

    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no");
    return 0;
}
```
Memory Referencing Bug

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) –> 3.14
fun(1) –> 3.14
fun(2) –> 3.1399998664856
fun(3) –> 2.00000061035156
fun(4) –> 3.14, then segmentation fault

Explanation:

Saved State

<table>
<thead>
<tr>
<th></th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>3</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>2</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)

Memory Referencing Bug (Revisited)

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) –> 3.14
fun(1) –> 3.14
fun(2) –> 3.1399998664856
fun(3) –> 2.00000061035156
fun(4) –> 3.14, then segmentation fault

Saved State

<table>
<thead>
<tr>
<th></th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>3</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>2</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)

Representing 3.14 as a Double FP Number

- $1073741824 = 0100 0000 0000 0000 0000 0000 0000 0000$
- $3.14 = 11.0010 0011 1101 0111 0000 1010 0000...$
- $(-1)^{e} M 2^{E}$
  - $S = 0$ encoded as 0
  - $M = 1.1001 0001 1110 1011 1000 0101 000...$ (leading 1 left out)
  - $E = 1$ encoded as 1024 (with bias)

<table>
<thead>
<tr>
<th>exp (11)</th>
<th>frac (first 20 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100 0000 0000 1001 0011 1110 1011 1000</td>
</tr>
</tbody>
</table>

frac (the other 32 bits)

0101 0000 ...

Memory Referencing Bug (Revisited)

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) –> 3.14
fun(1) –> 3.14
fun(2) –> 3.1399998664856
fun(3) –> 2.00000061035156
fun(4) –> 3.14, then segmentation fault

Saved State

<table>
<thead>
<tr>
<th></th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>3</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>2</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)
Memory Referencing Bug (Revisited)

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) -> 3.14
fun(1) -> 3.14
fun(2) -> 3.1399998664856
fun(3) -> 2.00000061035156
fun(4) -> 3.14, then segmentation fault

Saved State

\[
\begin{array}{cccc}
\text{d7} & \cdots & \text{d4} & \text{d3} & \cdots & \text{d0} \\
0100 & 0000 & 0000 & 0000 & 0000 & 0000 \\
0101 & 0000 & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

Location accessed by fun(i)