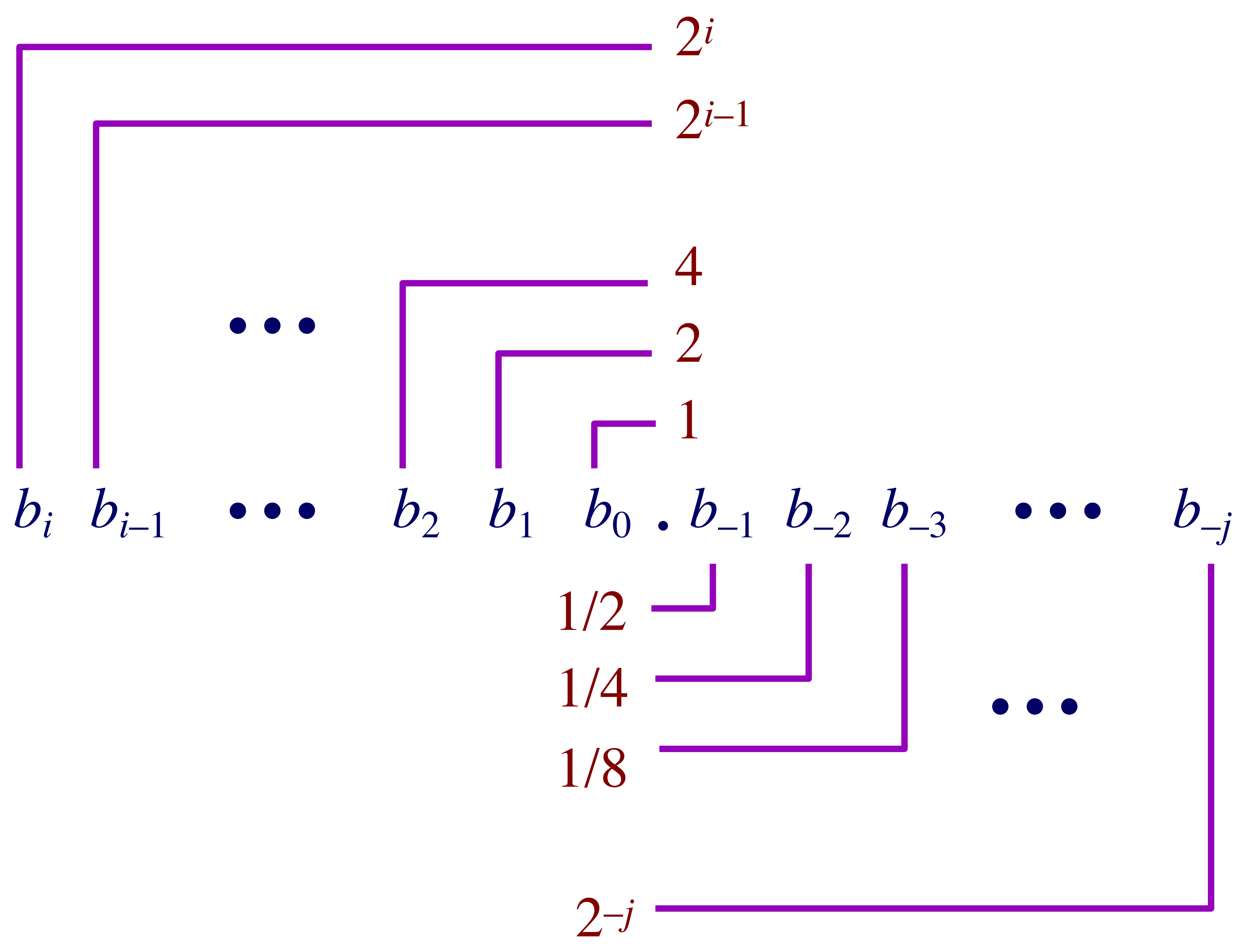
Floating Point

Most slides from CMU 213

Fractional Binary Numbers



Representation

- Bits to right of "binary point" represent fractional powers of 2
- Represents rational number:

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Frac. Binary Number Examples

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2

Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form $0.111111\ldots_2$ just below 1.0
  - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
  - Use notation $1.0 - \varepsilon$

Representable Numbers

Limitation

- Can only exactly represent numbers of the form $x/2^k$
- Other numbers have repeating bit representations

Value	Representation
1/3	$0.0101010101[01]..._2$
1/5	$0.001100110011[0011]..._2$
1/10	$0.0001100110011[0011]..._2$

This is where we need to slightly sacrifice precision when storing these numbers

Floating Point Representation

Numerical Form

- $(-1)^s\, M\ 2^E$
  - Sign bit *s* determines whether number is negative or positive
  - Significand *M* normally a fractional value in range [1.0,2.0).
  - Exponent *E* weights value by power of two

Example:
$15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13}$

Encoding

s	exp	frac

- MSB is sign bit
- `exp` field encodes *E* (but is not equal to E)
- `frac` field encodes *M* (but is not equal to M)

IEEE Floating Point

IEEE Standard 754

- Established in 1985 as uniform standard for floating point arithmetic
  - Before that, many idiosyncratic formats
- Supported by all major CPUs

Driven by Numerical Concerns

- Nice standards for rounding, overflow, underflow
- Hard to make go fast
  - Numerical analysts predominated over hardware types in defining standard

Drawback

- Naturally, it cannot represent all real numbers accuratelyN
  - It cannot store recurring digits in base 10 like 1/3, so these numbers are always rounded down slightly. It allows use to store very small and also large numbers by reducing a little precision.

IEEE Precision Options

Numbers are stored in scientific notation

◊ Single precision: 32 bits —> approximately ± 10 ^ 38

◊ Double precision: 64 bits —> approximately ± 10 ^ 308

“Normalized” Numeric Values

Condition

- `exp` ≠ `000…0` and `exp` ≠ `111…1`

Exponent coded as *biased* value

E = *Exp* – *Bias*

- *Exp* : unsigned value denoted by `exp`
- *Bias* : Bias value
  - » Single precision: 127 (*Exp*: 1…254, *E*: -126…127)
  - » Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)
  - » in general: *Bias* = 2^{e-1} - 1, where e is number of exponent bits

Significand coded with implied leading 1

M = $1.xxx\ldots x_2$

- `xxx…x`: bits of `frac`
- Minimum when `000…0` (*M* = 1.0)
- Maximum when `111…1` (*M* = 2.0 – ε)
- Get extra leading bit for “free”

An Example of a Normalized Float

Value

`Float F = 15213.0;`

- $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \text{ X } 2^{13}$

Significand

$M = \quad 1.\underline{1101101101101}_2$

$\texttt{frac} = \quad \underline{\texttt{1101101101101}}\texttt{0000000000}_2$

Exponent

$E \quad = \mathbf{13}$

$Bias \quad = \mathbf{127}$

$\boldsymbol{E} \quad = \boldsymbol{Exp - Bias}$ → **Exp** $= \mathbf{13 + 127 = 140} = 10001100_2$

s	exp	frac
0	10001100	11011011011010000000000

Denormalized Values

Condition

- `exp` = `000…0`

Value

- Exponent value $E = -Bias + 1$
- Significand value $M =$ $0.xxx\ldots x_2$
  - `xxx…x`: bits of `frac`

Cases

- `exp` = `000…0`, `frac` = `000…0`
  - Represents value 0
  - Note that have distinct values +0 and –0
- `exp` = `000…0`, `frac` ≠ `000…0`
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - “Gradual underflow”

Special Values

Condition

- `exp` = `111…1`

Cases

- `exp` = `111…1`, `frac` = `000…0`
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞
- `exp` = `111…1`, `frac` ≠ `000…0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞

Operations on Floating Points

- Summing floating numbers:
  - It has a relative error, epsilon
    - (a + b) + c != a + (b + c)
  - Designing efficient algorithms for computing a faithfully rounded floating-point is challenging
  - Proposed several efficient parallel algorithms for summing *n* floating point numbers, so as to produce a faithfully rounded floating-point representation of the sum. [Michael T. Goodrich, Ahmed Eldawy 2016]

Operations on Floating Points (cont.)

- Comparison is tricky:
  - If involved equality, Due to rounding errors, it demands special measures
    - Next slide shows you how to handle it

Floating Point Comparison

```
root [0] double capacity = 1.0
(double) 1.0000000

root [1] capacity -= 0.8
(double) 0.20000000

root [2] capacity -= 0.1
(double) 0.10000000

root [3] double item = 0.1
(double) 0.10000000

root [4] item <= capacity
(bool) false

root [5] delta = item - capacity
(double) 5.5511151e-17
```

```
root [6] double epsilon = 1e-6
(double) 1.0000000e-06

root [7] equal = fabs(delta) < epsilon
(bool) true

root [8] item < capacity || equal
(bool) true

root [9] bool equals(double a, double b){
root[10]   return fabs(a - b) < 1e-6;
root[11] }

root[12] item < capacity || equals(...)
(bool true)
```