Computer Architecture

Chapter 3

Fall 2005

Department of Computer Science

Kent State University

Objectives

- Signed and Unsigned Numbers
- Addition and Subtraction
- Multiplication and Division
- Floating Point

The Binary Numbering System

- A computer's internal storage techniques are different from the way humans represent information in daily lives

Humans

- Decimal numbering system to rep real numbers
  - Base-10
  - Each position is a power of 10

$$3052 = 3 \times 10^3 + 0 \times 10^2 + 5 \times 10^1 + 2 \times 10^0$$

Binary Representation of Numbers

- Information inside a digital computer is stored as a collection of "binary data"

- Binary numbering system
  - Base-2
  - Built from ones and zeros
  - Each position is a power of 2

    $1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  - Digits 0,1 are called bits (binary digits)

Binary Representation of Numbers

- 6-Digit Binary Number (111001)
  - $111001 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
    $= 32 + 16 + 8 + 0 + 0 + 1$
    $= 57$
- 5-Digit Binary Number (10111)
  - $10111 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
    $= 16 + 0 + 4 + 2 + 1$
    $= 23$

Binary Representation of Numbers

- Computers use finite number of Bits for Integer Storage

Size ("word")	Max **Unsigned** Number Allowed
– 16	$1x2^{15} + 1x2^{14} + \ldots + 1x2^{1} + 1x2^{0}$
– MIPS-32	$1x2^{31} + 1x2^{30} + \ldots + 1x2^{1} + 1x2^{0}$

Otherwise **Arithmetic Overflow**

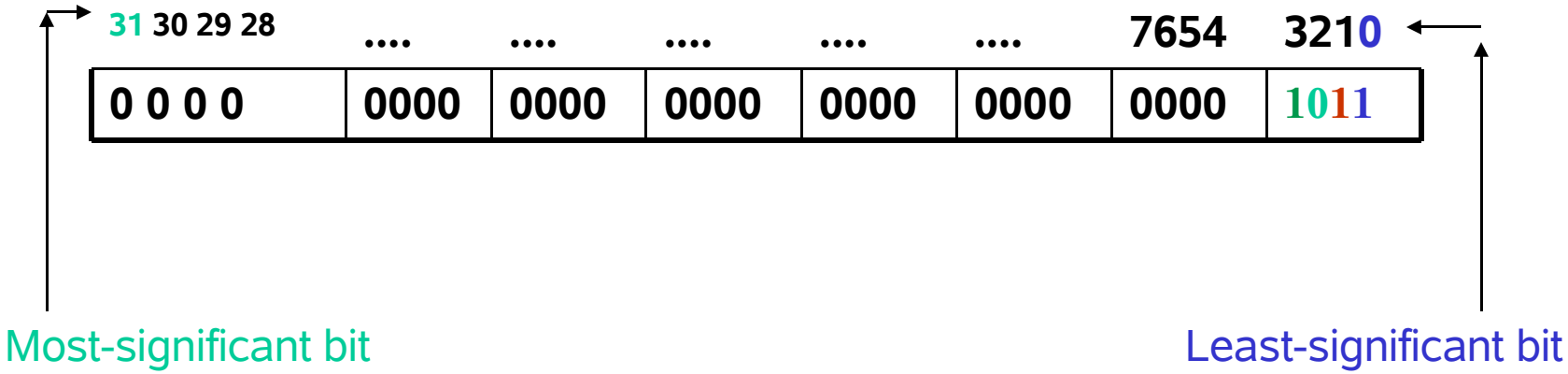
Number Representation

MIPS word

Example: how to translate 11_{ten} into binary?

$11_{ten} = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$= 1011_{two}$



How many (unsigned) binary numbers can 32 bits represent?

How to represent negative numbers?

- You have a budget of 32 bits to represent positive numbers and negative numbers. In other words, you need to map any 32-bit code to a (binary) number
- You need to make some (simple) rules so that in your system, you will be able to recognize/separate positive numbers and negative numbers very easily
- Questions
  - In your system, how many positive number and negative number you can express?
  - In your system, how to perform add and sub operation?

What is a good coding?

- Balance
  - Ideally, half positive, half negative, is it possible?
- Number of Zeros
- Easy of operations
- Easy of recognization
- …

Signed-Magnitude

- Explicit sign bit
- Remaining bits encode unsigned magnitude
- Two representations for zero (+0 and -0)
- Addition and subtraction are more complicated

Representation	Value
000	+0
001	+1
010	+2
011	+3
100	-0
101	-1
110	-2
111	-3

Biased

- Add a bias to the signed number in order to make it unsigned
- Subtract the bias to return the original value
- Typically the bias is 2^{k-1} for a k-bit representation

Representation	Value
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

Two's Complement

- Most significant bit has a negative weight
- Implicit sign bit
- One negative number that has no positive
- Handles overflow well

Representation	Value
000	0
001	+1
010	+2
011	+3
100	-4
101	-3
110	-2
111	-1

Signed Number Representation

Two's Complement Notation

- Leading 0s mean +ve
- Leading 1s mean -ve

1 0 0 0	0000	0000	0000	0000	0000	0111	0001

$1 \times -2^{31} + 0 \times 2^{30} + \ldots 1\times2^{6} + 1\times2^{5} + 1\times2^{4} + 0\times2^{3} + 0\times2^{2} + 0\times2^{1} + 1\times2^{0}$

= -2,147,483,648 + 64 + 32 +16 +1
= -2,147,483,535

Compare with sign/magnitude representation for -49

cf: Sign Magnitude/ Two's Complement Notations

Up Close

Sign Magnitude	Two's Complement
000 = +0	000 = +0
001 = +1	001 = +1
010 = +2	010 = +2
011 = +3	011 = +3
100 = -0	100 = -4
101 = -1	101 = -3
110 = -2	110 = -2
111 = -3	111 = -1

MIPS

- 32 bit signed numbers:

Two's Complement Representation	Value
0000 0000 0000 0000 0000 0000 0000 0000	= 0
0000 0000 0000 0000 0000 0000 0000 0001	= + 1
0000 0000 0000 0000 0000 0000 0000 0010	= + 2
...	
0111 1111 1111 1111 1111 1111 1111 1110	= + 2,147,483,646
0111 1111 1111 1111 1111 1111 1111 1111	= + 2,147,483,647
1000 0000 0000 0000 0000 0000 0000 0000	= - 2,147,483,648
1000 0000 0000 0000 0000 0000 0000 0001	= - 2,147,483,647
1000 0000 0000 0000 0000 0000 0000 0010	= - 2,147,483,646
...	
1111 1111 1111 1111 1111 1111 1111 1101	= - 3
1111 1111 1111 1111 1111 1111 1111 1110	= - 2
1111 1111 1111 1111 1111 1111 1111 1111	= - 1

Some basic questions

- Consider you have a number (52, -52) in decimal, how do transform it into the Two's complement binary representation?
- How to perform add or sub operation in such a system?

Review

- What's is two's complement notation? Sign/magnitude?
- 1011, 0011 → decimal (assume we only have 4 bits)
- Express -3 and 3 in two's complement notation (8 bits)

Two's Complement Operation

- **To Negate a Two's complement number:**
  - First invert all bits then
  - Add 1 to the inverted bits
  - Let's work on some examples (-2→2 ， -2← 2)
- To Convert n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the LHS half of the word

```
0010  -> 0000 0010
1010  -> 1111 1010
```

Addition and Subtraction

- Addition (carries 1s)

```
0000 0000 0000 0000 0000 0000 0000 0011  =  + 3
0000 0000 0000 0000 0000 0000 0000 0010  =  + 2

0000 0000 0000 0000 0000 0000 0000 0101  =  + 5
```

- Subtraction: use addition of negative numbers

```
0000 0000 0000 0000 0000 0000 0000 0011  =  + 3
1111 1111 1111 1111 1111 1111 1111 1110  =  – 2

0000 0000 0000 0000 0000 0000 0000 0001  =  + 1
```

Let's do some excises!

7+6, 7-6

Overflow

- if result too large to fit in the finite computer word of the result register
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001
------
  1000
```

- When the overflow can happen?
  - One positive+one negative?
  - Two positive/Two negative?

Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow

Overflow in MIPS

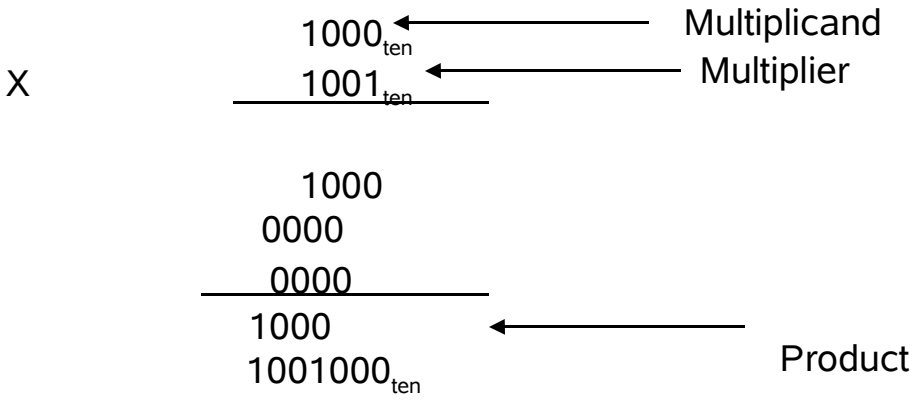
- In MIPS there are two versions of each add and subtract instruction
- Add (**add**), add immediate (**addi**), and subtract (**sub**) cause an *exception* on overflow
- Add unsigned (**addu**), add immediate unsigned (**addiu**), and subtract unsigned (**subu**) ignore overflow
- C++ code always uses the unsigned versions because it ignores overflow

Review

- Using two different methods to get -3 in two's complement notation (4 bits)
- What is (-3)'s two's complementation notation with (8 bits)
- How to do 2+(-3), (-3)+(-2) in two's complement notation?
- What is overflow? How to detect overflow in two's complement notation?

Multiplication

- Recall:



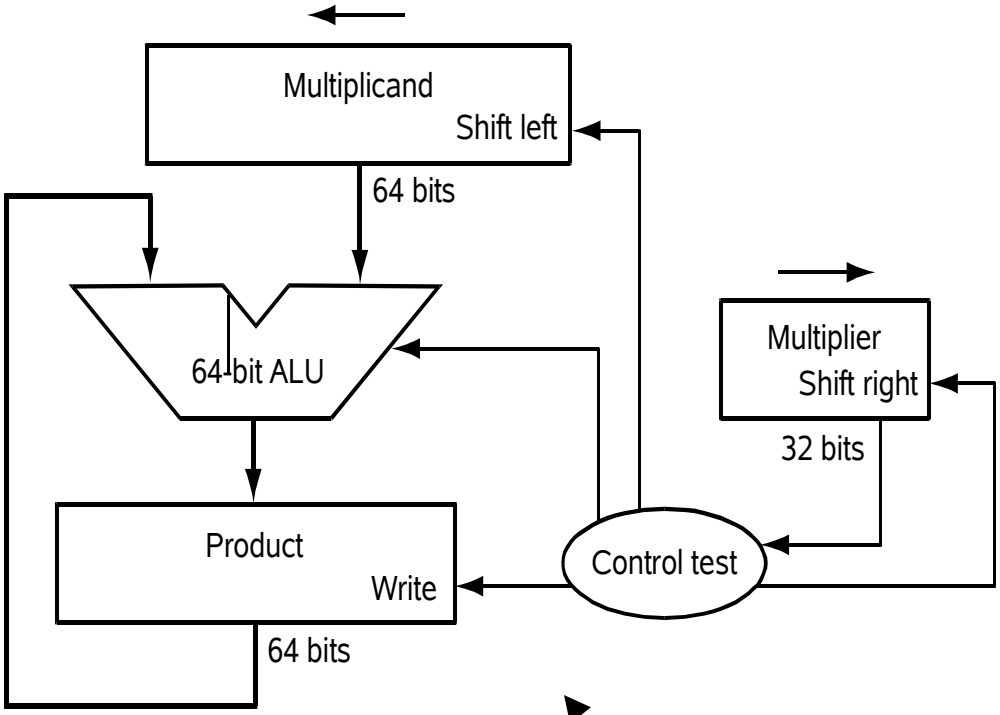
Observations

- More storage required to store the product
- Place copy of multiplicand in proper location if multiplier is a 1
- Place 0 in proper location if multiplier is 0
- Product of *n*-bit Multiplicand and *m*-Multiplier is (*n* + *m*)-bit long
- Number of steps (move digits to LHS) is *n -1*; where n rep the number of digits (1,0)

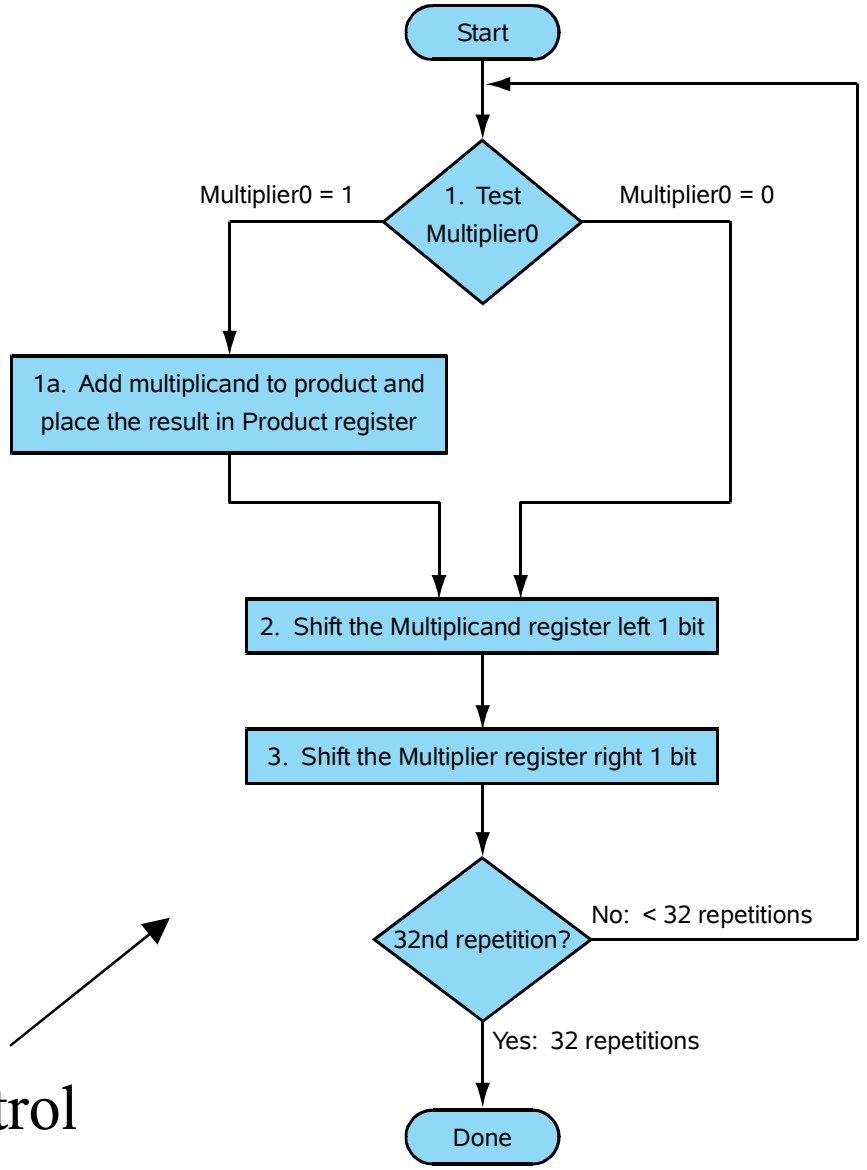
Let's examine 2 versions of multiplication algorithm for binary numbers

Multiplication

Version 1



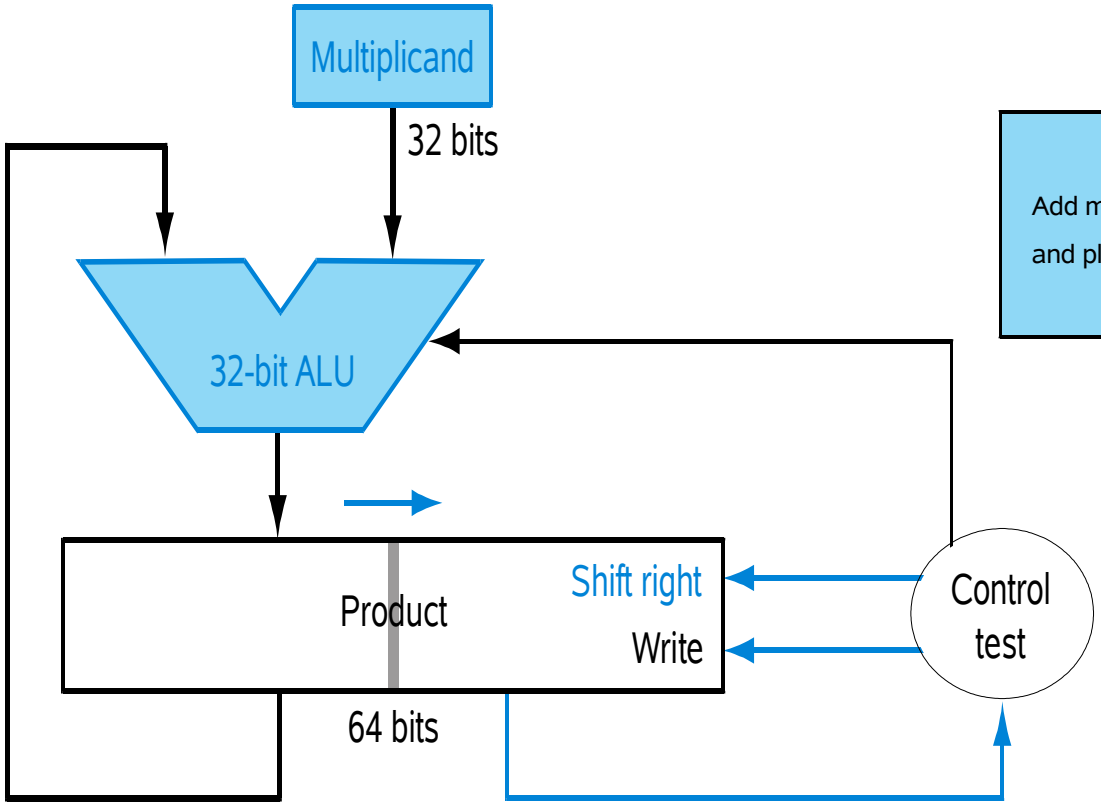
Datapath

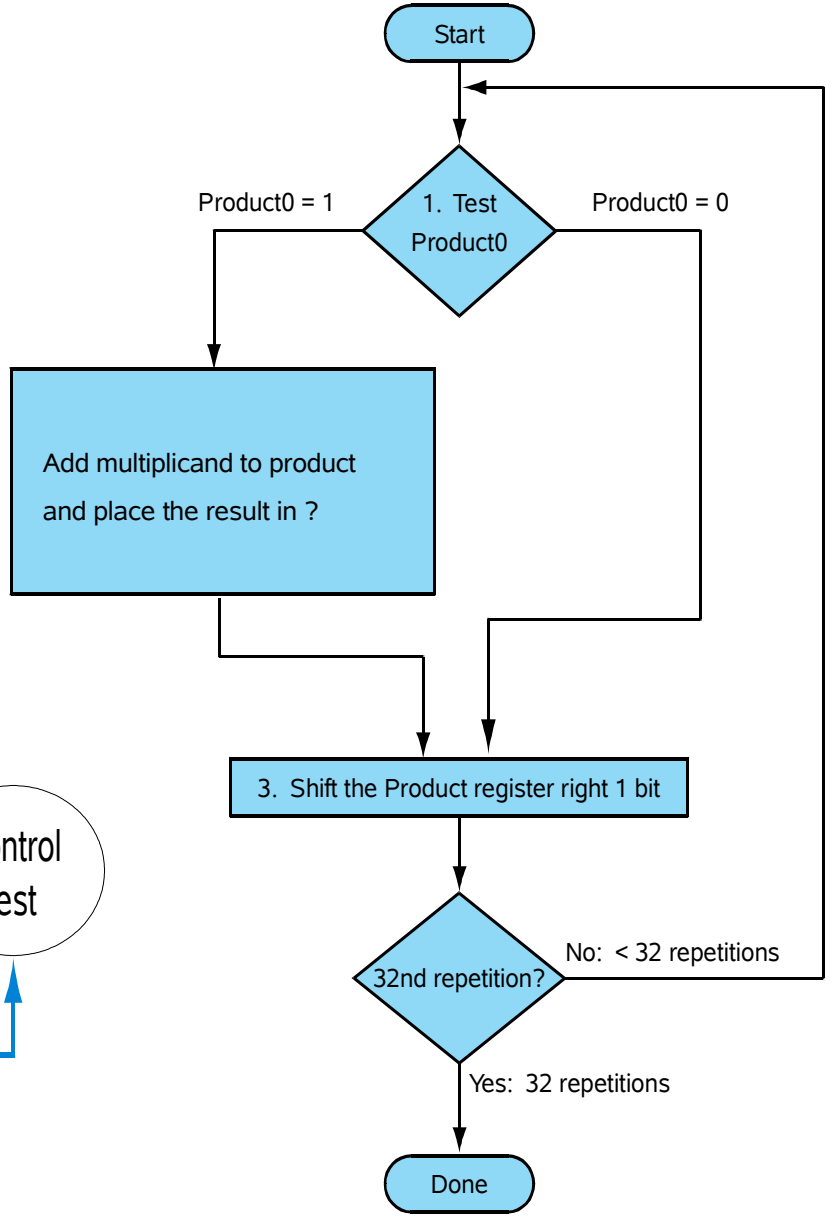


Control

Multiplication

Refined Version





Multiplication

Negative Numbers

- Convert Multiplicand and Multiplier to Positive Numbers
- Run the Multiplication algorithm for 31 iterations (ignoring the sign bit)
- Negate product only if original signs for Multiplicand and Multiplier are different

Multiply and Divide in MIPS

- Instructions in MIPS
  - Multiply (**mult**)
  - Multiply unsigned (**multu**)
  - Divide (**div**)
  - Divide unsigned (**divu**)
- The results are not stored in a general-purpose register; instead they are stored in two special registers called *hi* and *lo*
- Additional instructions move values between *hi* and *lo* and the general-purpose registers
  - mflo, mfhi

Floating Point Puzzles

– For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;

float f = …;

double d = …;
```

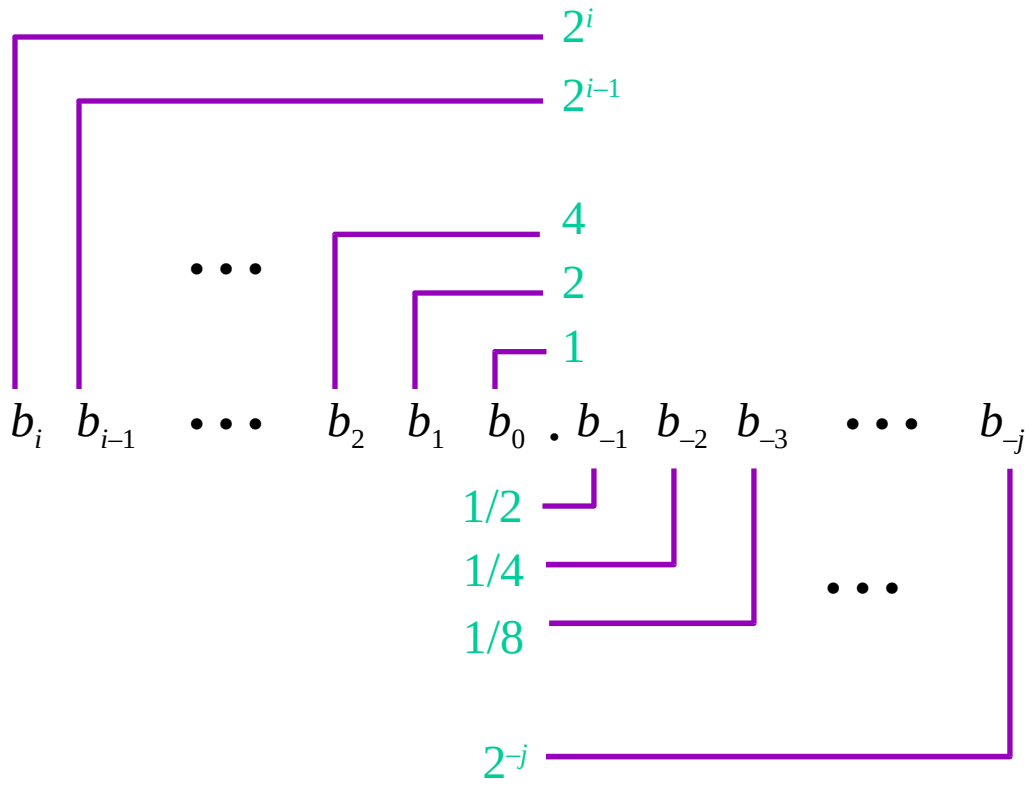
Assume neither `d` nor `f` is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0  ⇒  ((d*2) < 0.0)`
- `d > f  ⇒  -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

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 (What is underflow?)
  - Hard to make go fast
    - Numerical analysts predominated over hardware types in defining standard

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
  - 63/64 0.111111_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 $0.0101010101[01]\ldots_2$

  1/5 $0.001100110011[0011]\ldots_2$

  1/10 $0.0001100110011[0011]\ldots_2$

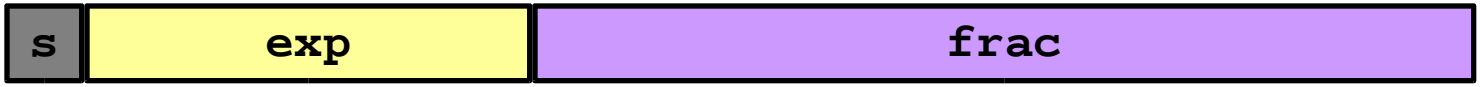
Floating Point Representation

- Numerical Form
  - $-1^s\ M\ \ 2^E$
    - Sign bit s determines whether number is negative or

    - Significand M normally a fractional value in range [1.0,2.0).
    - Exponent E weights value by power of two
- Encoding
  - MSB is sign bit
  - `exp` field encodes E
  - `frac` field encodes M

Floating Point Precisions

- Encoding



  - MSB is sign bit
  - `exp` field encodes *E*
  - `frac` field encodes *M*
- Sizes
  - Single precision: 8 `exp` bits, 23 `frac` bits
    - 32 bits total
  - Double precision: 11 `exp` bits, 52 `frac` bits
    - 64 bits total

“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`** ($M = 2.0 - \varepsilon$)
  - Get extra leading bit for “free”

Normalized Encoding Example

- Value

  `Float F = 15213.0;`

  – $\mathbf{15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}}$

- Significand

  $\boldsymbol{M}$ = $\mathbf{1.\underline{1101101101101}_2}$

  `frac` = $\underline{1101101101101}0000000000_2$

- Exponent

  $\boldsymbol{E}$ = 13

  $\boldsymbol{Bias}$ = 127

  $\boldsymbol{Exp}$ = 140 = 10001100_2

Floating Point Representation (Class 02):

Hex: **4 6 6 D B 4 0 0**

Binary: **0100 0110 0110 1101 1011 0100 0000 0000**

140: **100 0110 0**

15213: ***1*110 1101 1011 01**

Special Numbers

- IEEE FP also defines classes of special numbers
  - Denormalized numbers
  - Zero
  - Infinity
  - Not a Number (NaN)

Underflow

- Underflow occurs when a number is too small in magnitude to be represented
- This occurs when the exponent is less than the minimum representable value
- Be careful not to confuse negative overflow with underflow
- Underflow is unique to floating-point; integer arithmetic can never underflow

Denormalized Numbers

- It is also difficult to represent numbers that are close to zero in normalized form
- Denormalized numbers are stored unnormalized and therefore do not have a hidden bit
- IEEE also uses a special encoding for denormals
  - Biased exponent is zero
  - Fraction is not zero
- Denormals help prevent *underflow*
- Also known as subnormal numbers

Denormalized Values

- Condition
  - `exp` = `000...0`
- Value
  - Exponent value $E = -Bias + 1$
  - Significand value $M =$ `0.xxx...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), ∞ – ∞, Dividing zero by zero

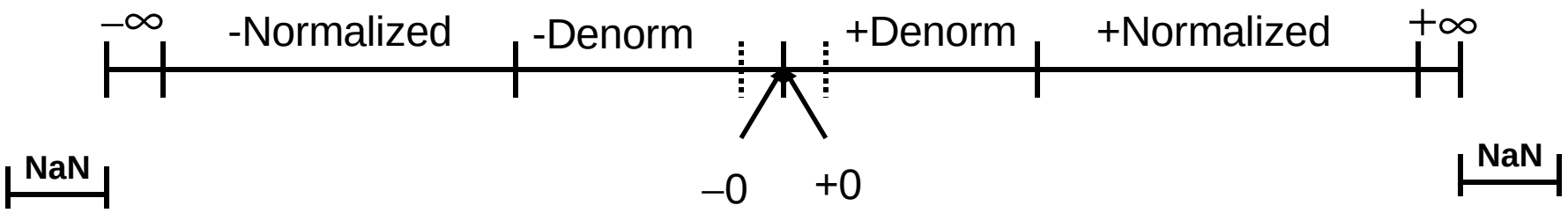
Not a Number (NaN)

- In IEEE an undefined operation results in a special value called Not a Number (NaN)
  - Biased exponent is maximum (255 for single)
  - Fraction is not zero
  - Sign is ignored
- Example of undefined operations
  - Dividing zero by zero
  - Adding infinities of different signs
  - Square root of a negative number
- Any operation on a NaN results in a NaN

Types of Numbers in IEEE

Single		Double		Meaning
Exponent	**Fraction**	**Exponent**	**Fraction**	
0	0	0	0	0
0	nonzero	0	nonzero	Denormalized
1-254	-	1-2046	-	Normalized
255	0	2047		Infinity
255	nonzero	2047	nonzero	Not a Number (NaN)

Summary of Floating Point Real Number Encodings



Answers to Floating Point Puzzles

```
int x = …;

float f = …;

double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` — No: 24 bit significand
- `x == (int)(double) x` — Yes: 53 bit significand
- `f == (float)(double) f` — Yes: increases precision
- `d == (float) d` — No: loses precision
- `f == -(-f);` — Yes: Just change sign bit
- `2/3 == 2/3.0` — No: 2/3 == 0
- `d < 0.0 ⇒ ((d*2) < 0.0)` — Yes!
- `d > f ⇒ -f > -d` — Yes!
- `d * d >= 0.0` — Yes!
- `(d+f)-d == f` — No: Not associative