Computer Arithmetic And ALU Design II

Instructor: Dmitri A. Gusev

Fall 2007

CS 502: Computers and Communications

Lecture 5, September 19, 2007

More Arithmetic Instructions

Instruction	Example	Meaning	Comments
multiply	mult \$s2,\$s3	Hi,Lo = \$s2*\$s3	64-bit signed product
multiply unsigned	multu \$s2,\$s3	Hi,Lo = \$s2*\$s3	64-bit unsigned product
divide	div \$s2,\$s3	Lo = \$s2/\$s3, Hi = \$s2 mod \$s3	Lo = quotient, Hi = remainder
divide unsigned	divu \$s2,\$s3	Lo = \$s2/\$s3, Hi = \$s2 mod \$s3	Unsigned quotient and remainder
move from Hi	mfhi \$s1	\$s1 = Hi	Used to get copy of Hi
move from Lo	mflo \$s1	\$s2 = Lo	Used to get copy of Lo

Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9
- Representation:
  - sign, exponent, significand: $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
    - the floating point is *binary point*, no longer decimal!
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
  - normalized: $1.\text{xxxxxxxxx}_2 * 2^{\text{yyyy}}$
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

IEEE 754 floating-point standard

- Leading "1" bit of significand is implicit

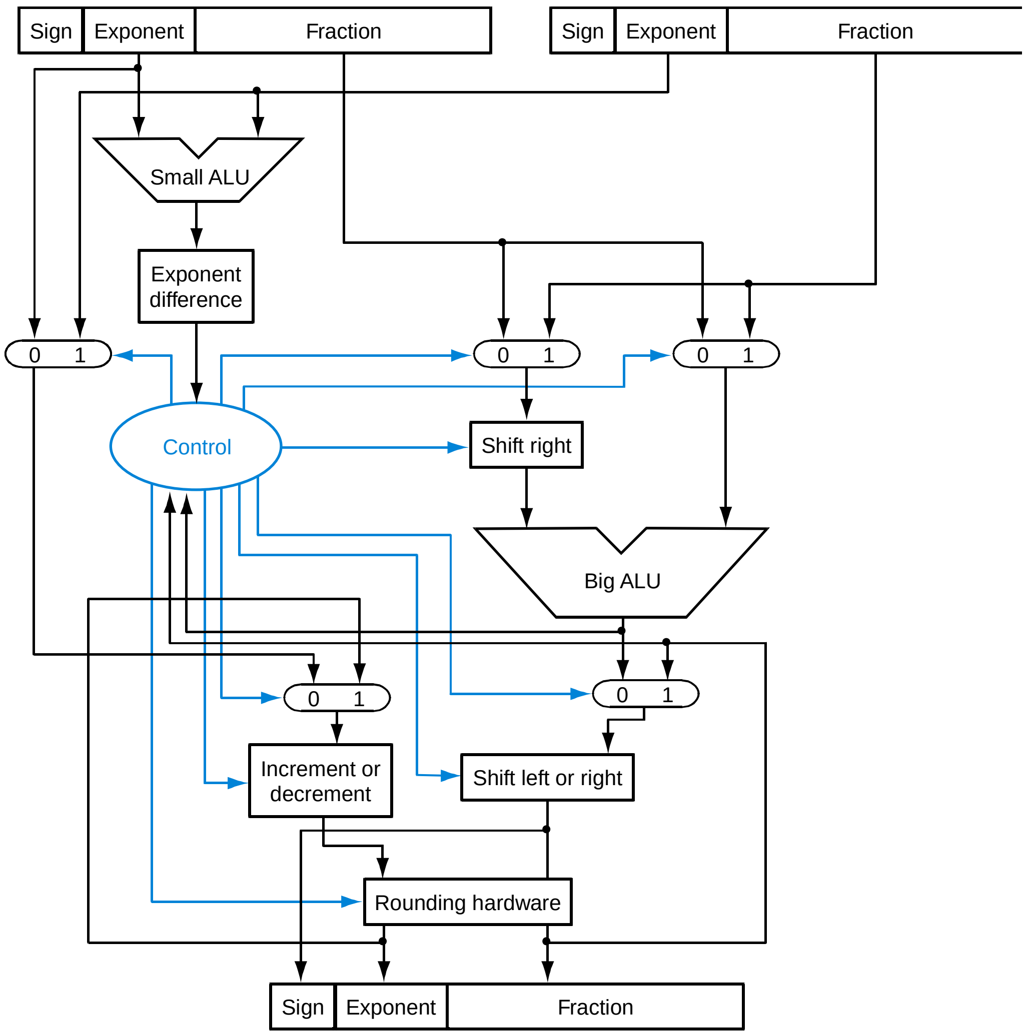
- Exponent is "biased" to make sorting easier
  – all 0s is smallest exponent all 1s is largest
  – bias of 127 for single precision and 1023 for double precision
  – summary: $(-1)^{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}$

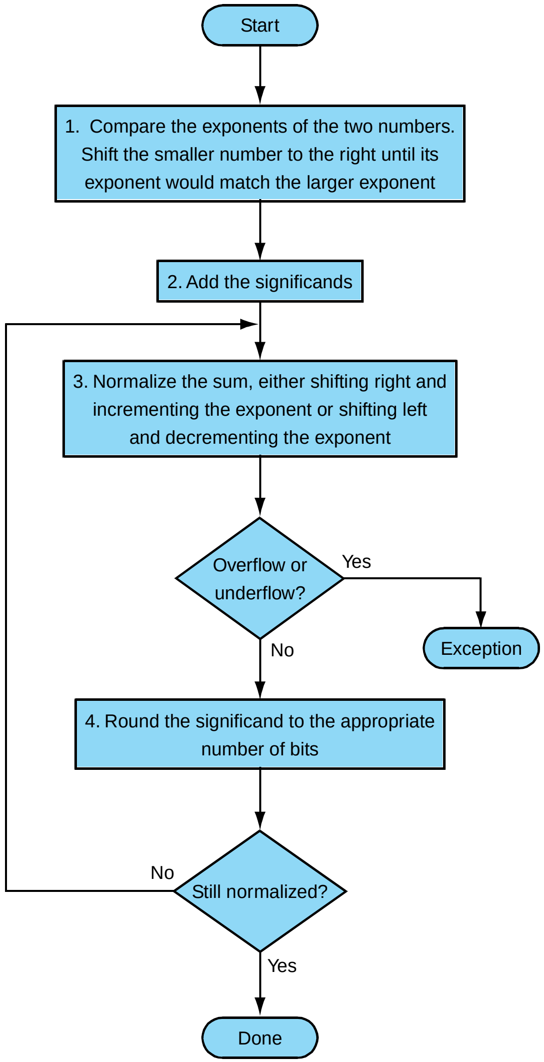
- Example:

  – decimal: -.75 = - (½ + ¼)
  – binary: $-.11 = -1.1 \times 2^{-1}$
  – floating point: exponent = 126 = 01111110

  – IEEE single precision:
    10111111010000000000000000000000

Floating point addition

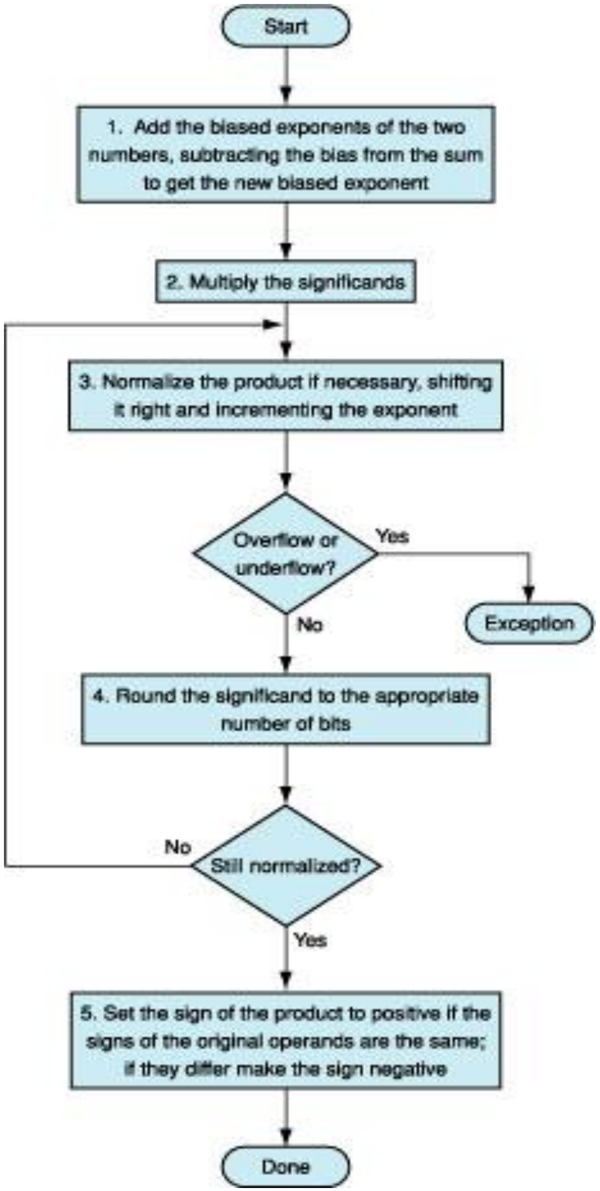
- 



Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Floating-Point Multiplication



Example: $1.110 * 10^{10} * 9.200 * 10^{-5}$

1. (10+bias)+(-5+bias) – bias = 5+bias
2. 1.110*9.200=10.212000 [$*10^{5}$]
3. $1.0212*10^{6}$
4. $1.021*10^{6}$
5. $+1.021*10^{6}$

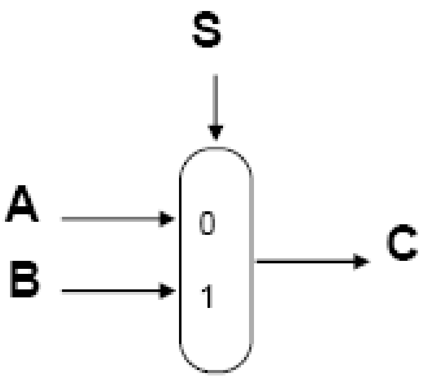
Floating-Point Instructions in MIPS

- 32 floating-point registers: $f0,$f1,…,$f31
- The floating-point registers are used in pairs for double precision numbers
- Instructions:

Floating-Point	single	double
add	add.s	add.d
subtract	sub.s	sub.d
multiply	mul.s	mul.d
divide	div.s	div.d

Multiplexor

- Selects one of the inputs to be the output, based on a control input



note: we call this a 2-input mux
even though it has 3 inputs!

- Lets build our ALU using a MUX:

Use of Multiplexor

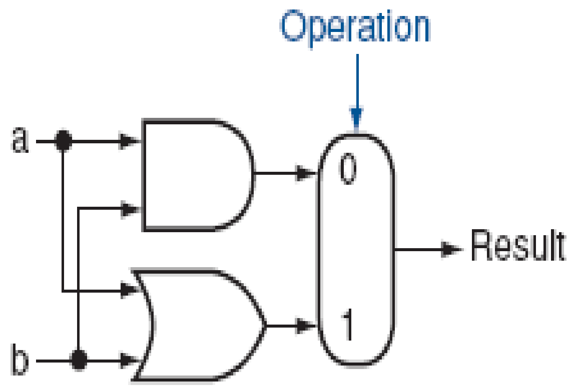
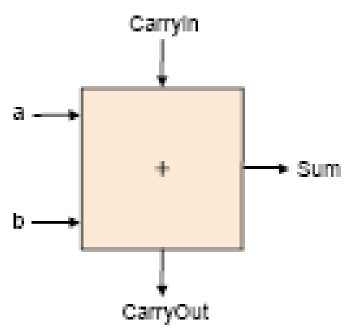


FIGURE B.5.1 The 1-bit logical unit for AND and OR.

Different Implementations

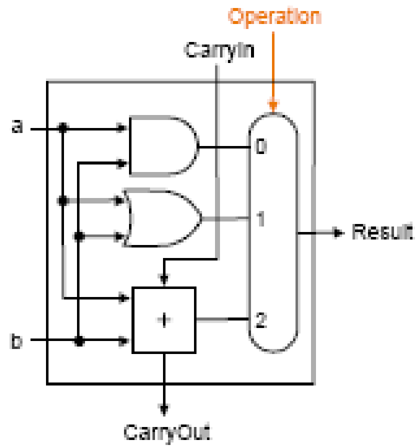
- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate
  - Dont want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:

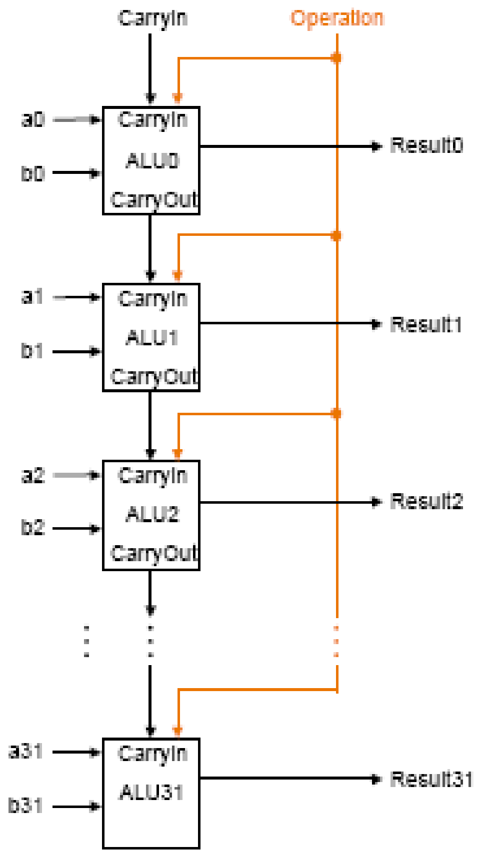


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU





What about subtraction (a – b) ?

- Two's complement approach: just negate b and add.
- How do we negate?

- A very clever solution: