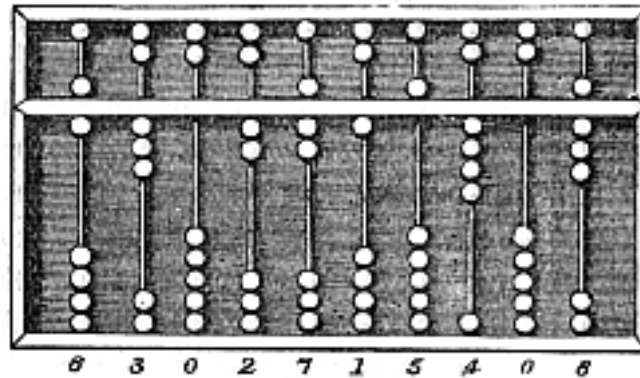
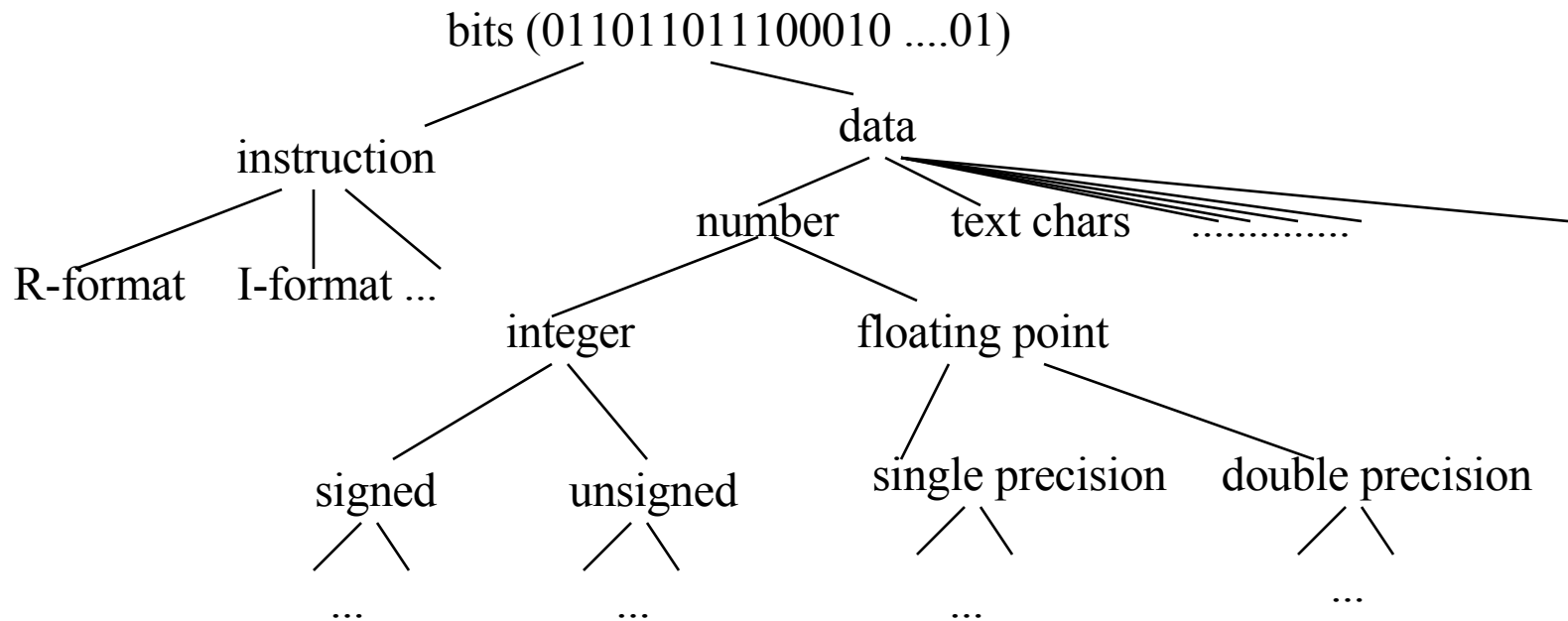


Number Systems and Computer Arithmetic



*Counting to four billion
two fingers at a time*

What do all those bits mean now?



Questions About Numbers

- How do you represent
 - negative numbers?
 - fractions?
 - really large numbers?
 - really small numbers?
- How do you
 - do arithmetic?
 - identify errors (e.g. overflow)?
- What is an ALU and what does it look like?
 - ALU=arithmetic logic unit

Review: Binary Numbers

Consider a 4-bit binary number

Decimal	Binary	Decimal	Binary
0	0000	4	0100
1	0001	5	0101
2	0010	6	0110
3	0011	7	0111

Examples of binary arithmetic:

$$3 + 2 = 5$$

Diagram illustrating the binary addition of 3 (0011) and 2 (0010). The result is 5 (0101). An arrow indicates a carry from the second column (ones place) to the third column (twos place).

$$\begin{array}{rcccc} & & & 1 & \\ & 0 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 0 \\ \hline & 0 & 1 & 0 & 1 \end{array}$$

$$3 + 3 = 6$$

Diagram illustrating the binary addition of 3 (0011) and 3 (0011). The result is 6 (0110). Two arrows indicate carries: one from the second column (ones place) to the third column (twos place), and another from the third column to the fourth column (fours place).

$$\begin{array}{rcccc} & & 1 & & 1 \\ & 0 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

Negative Numbers?

- We would like a *number system* that provides
 - obvious representation of 0,1,2...
 - uses adder for addition
 - single value of 0
 - equal coverage of positive and negative numbers
 - easy detection of sign
 - easy negation

Some Alternatives

- Sign Magnitude -- MSB is sign bit, rest the same

$$-1 == 1001$$

$$-5 == 1101$$

- One's complement -- flip all bits to negate

$$-1 == 1110$$

$$-5 == 1010$$

Two's Complement Representation

- 2's complement representation of negative numbers
 - Take the bitwise inverse and add 1
- Biggest 4-bit Binary Number: 7 Smallest 4-bit Binary Number: -8

<u>Decimal</u>	<u>Two's Complement Binary</u>
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Two's Complement Arithmetic

Decimal	2's Complement Binary	Decimal	2's Complement Binary
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

- Examples: $7 - 6 = 7 + (-6) = 1$

$$\begin{array}{r} 0 \quad 1 \quad 1 \quad 1 \\ + \quad 1 \quad 0 \quad 1 \quad 0 \\ \hline \end{array}$$

$$3 - 5 = 3 + (-5) = -2$$

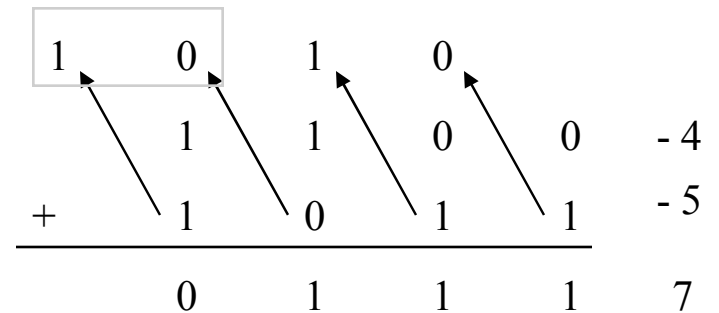
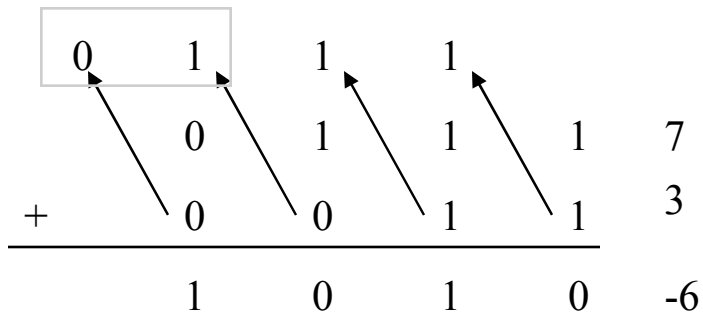
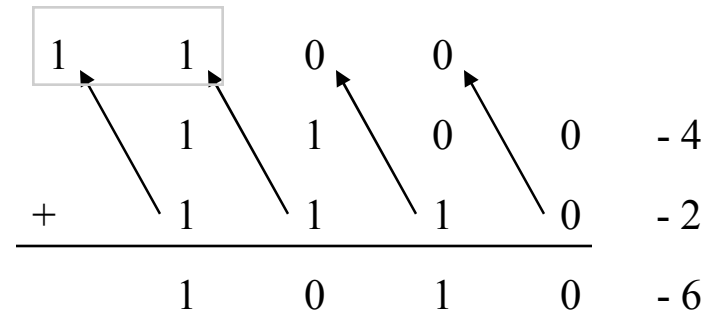
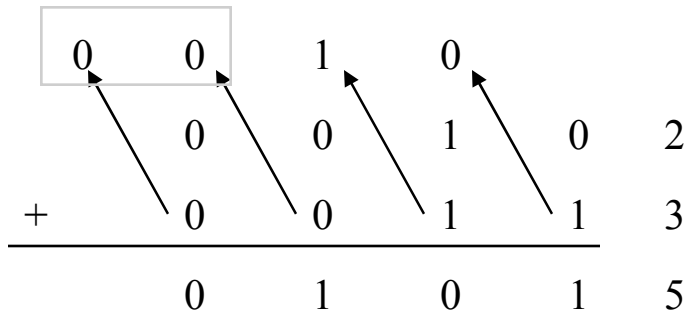
$$\begin{array}{r} 0 \quad 0 \quad 1 \quad 1 \\ + \quad 1 \quad 0 \quad 1 \quad 1 \\ \hline \end{array}$$

Some Things We Want To Know About Our Number System

- negation
- sign extension
 - $+3 \Rightarrow 0011, 00000011, 0000000000000011$
 - $-3 \Rightarrow 1101, 11111101, 1111111111111101$
- overflow detection

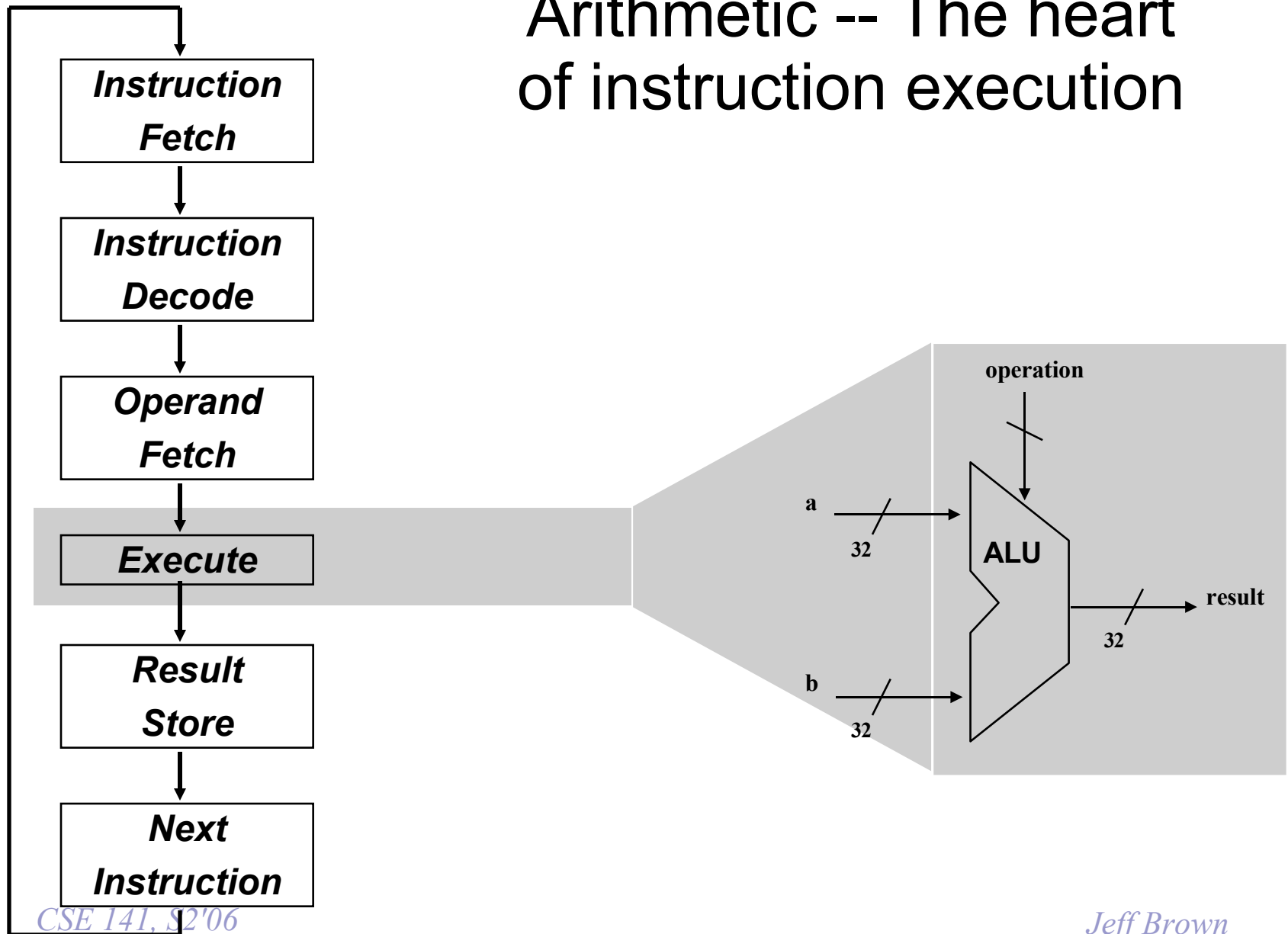
0101	5
+ <u>0110</u>	<u>6</u>

Overflow Detection

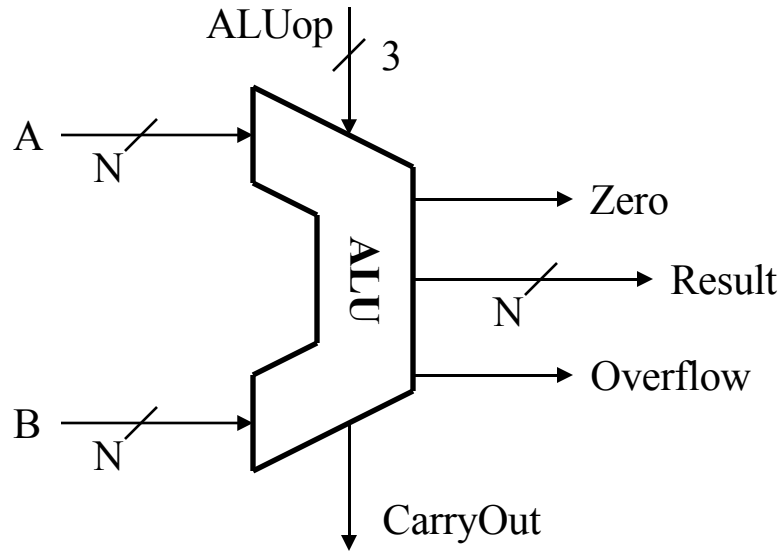


So how do we detect overflow?

Arithmetic -- The heart of instruction execution



Designing an Arithmetic Logic Unit



- **ALU Control Lines (ALUop)**

- 000
- 001
- 010
- 110
- 111

Function

And

Or

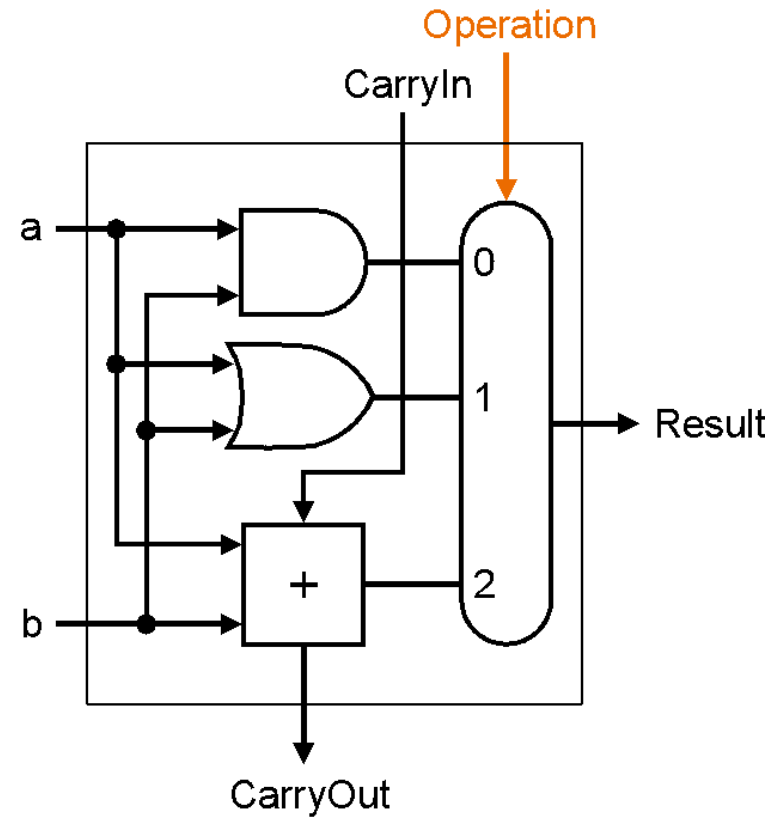
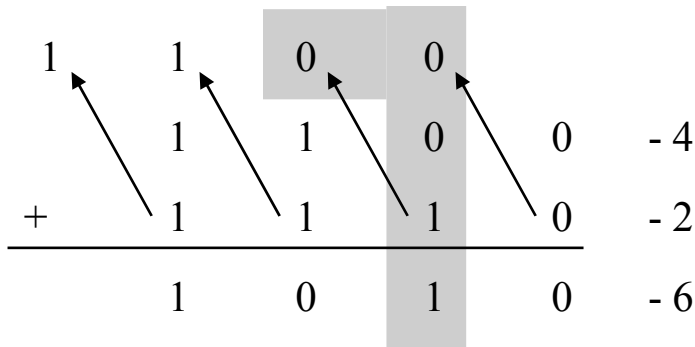
Add

Subtract

Set-on-less-than

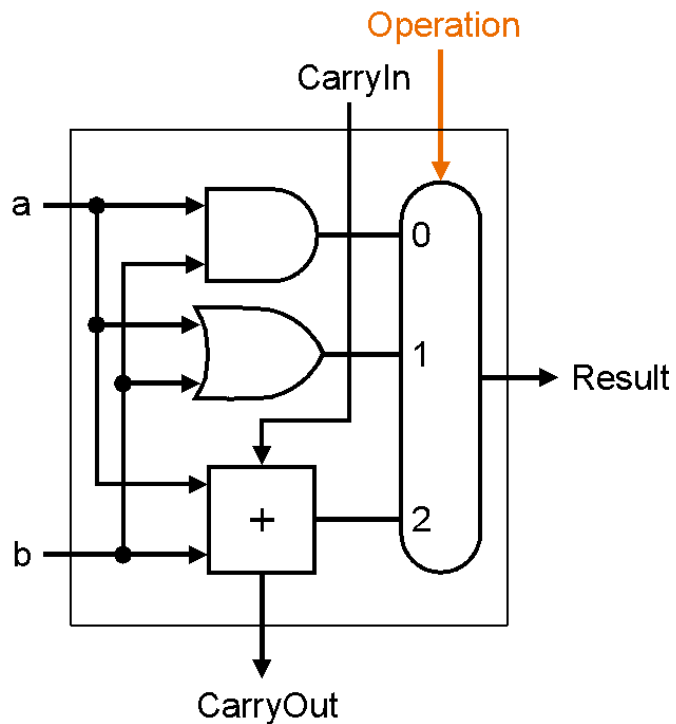
A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD

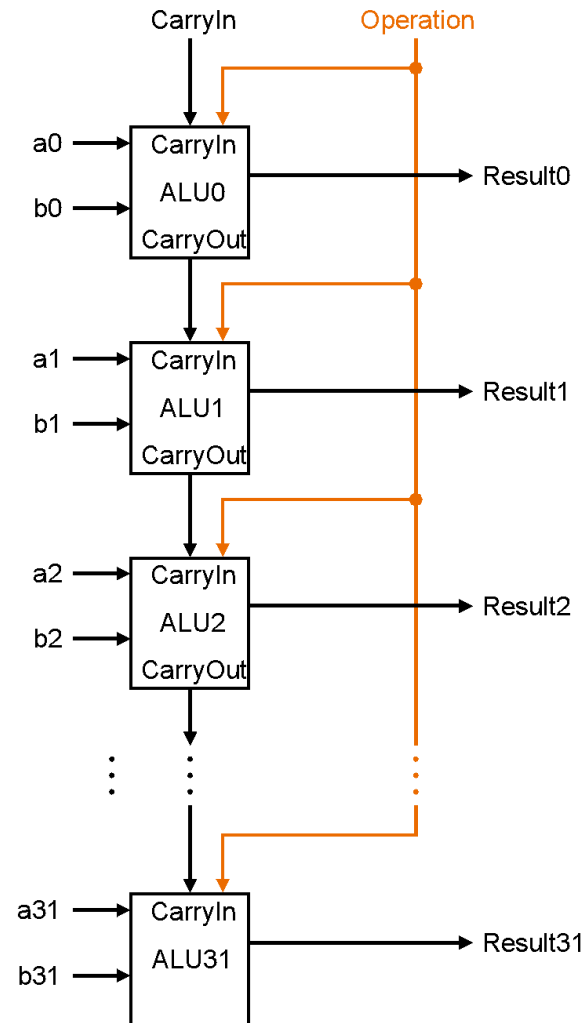


A 32-bit ALU

1-bit ALU

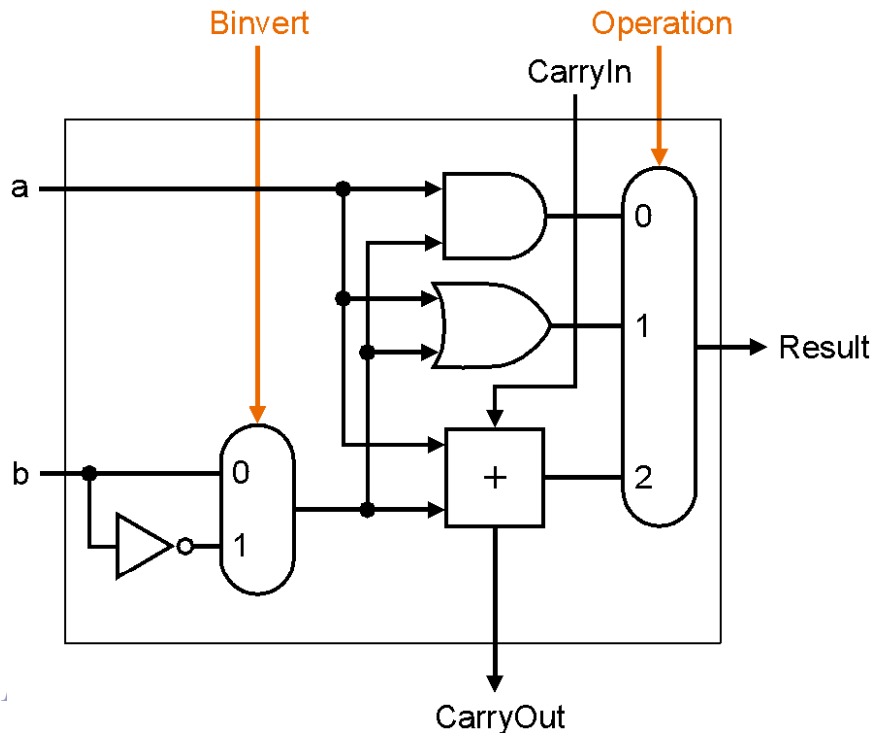


32-bit ALU



How About Subtraction?

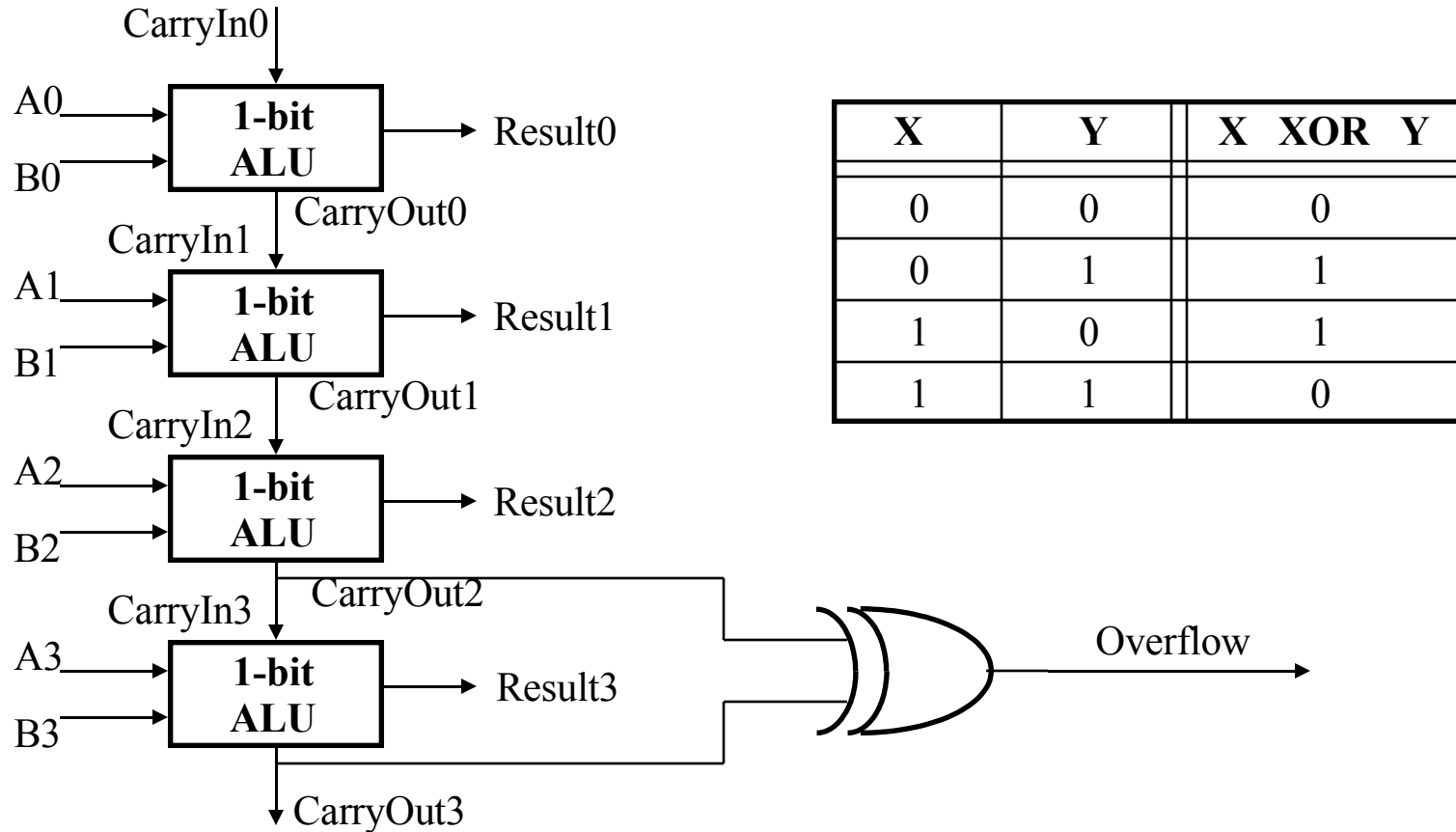
- Keep in mind the following:
 - $(A - B)$ is the same as: $A + (-B)$
 - 2's Complement negate: Take the inverse of every bit and add 1
- Bit-wise inverse of B is $\neg B$:
 - $A - B = A + (-B) = A + (\neg B + 1) = A + \neg B + 1$



$$\begin{array}{r} 1 \\ + 1 1 0 \\ \hline 0 1 1 1 \end{array}$$

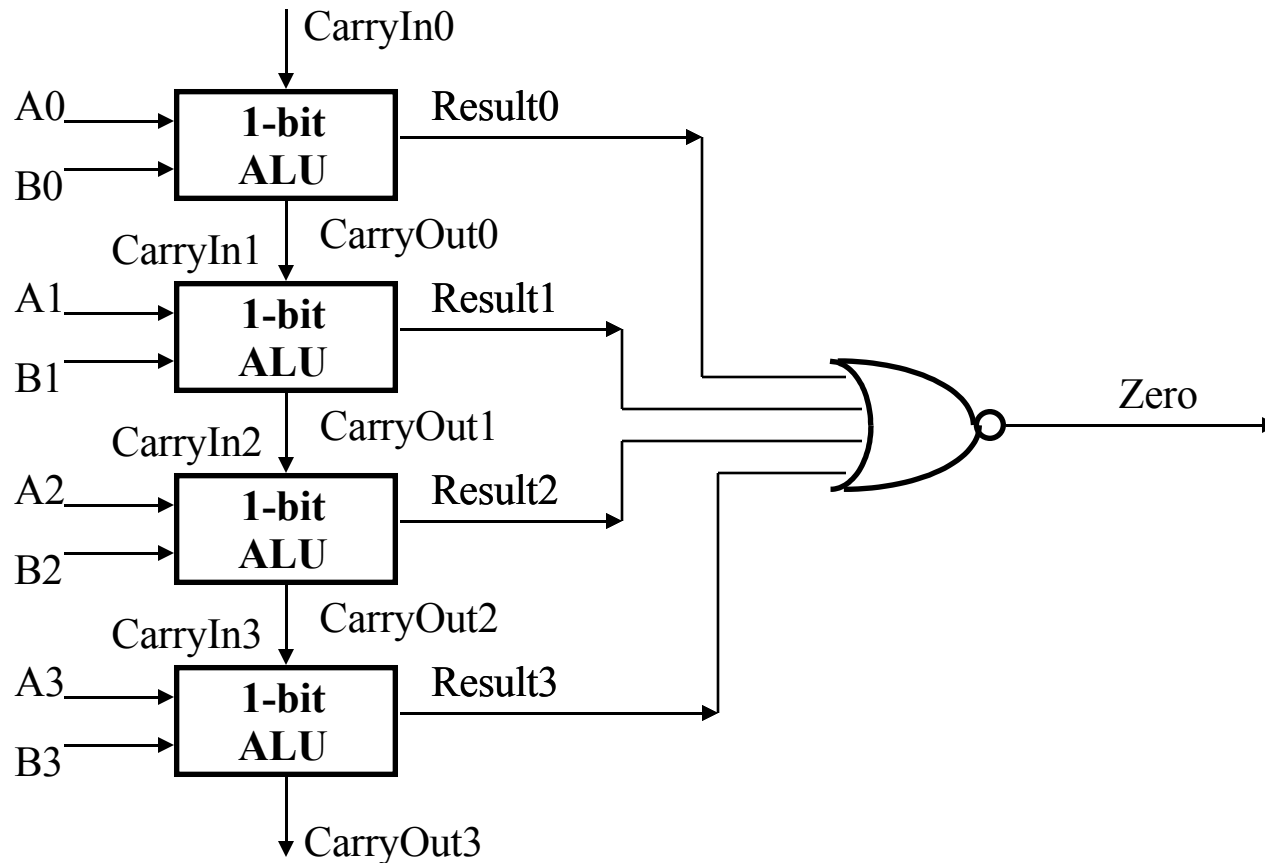
Overflow Detection Logic

- Carry into MSB $\neq$ Carry out of MSB
 - For a N-bit ALU: $\text{Overflow} = \text{CarryIn}[N - 1] \text{ XOR } \text{CarryOut}[N - 1]$



Zero Detection Logic

- Zero Detection Logic is just one BIG NOR gate
 - Any non-zero input to the NOR gate will cause its output to be zero



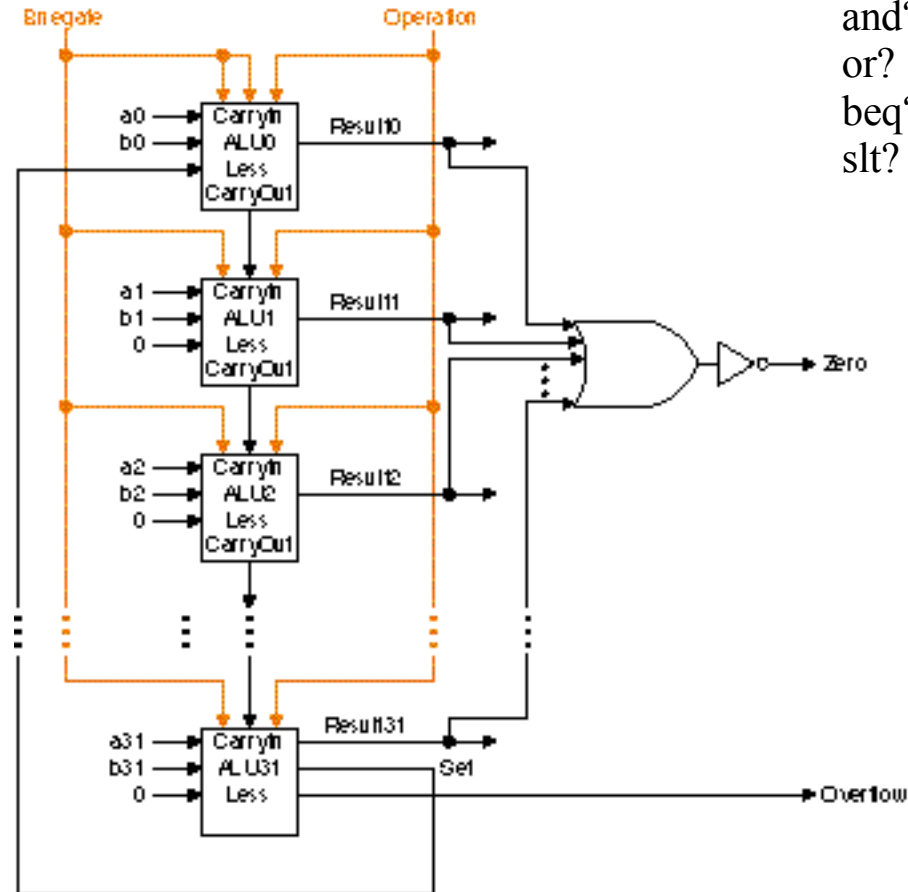
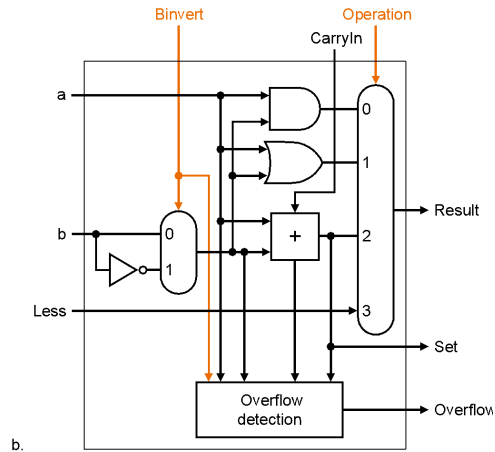
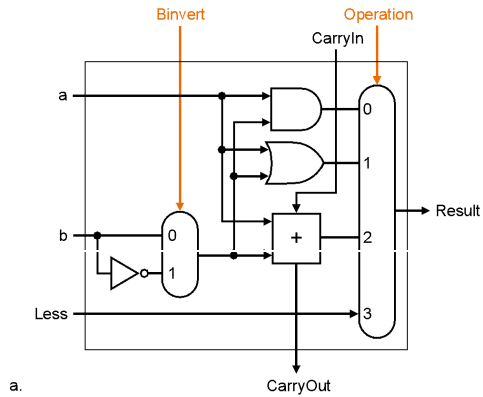
Set-on-less-than

- Do a subtract
- use sign bit
 - route to bit 0 of result
 - all other bits zero

Full ALU

give signals for:
neg oper

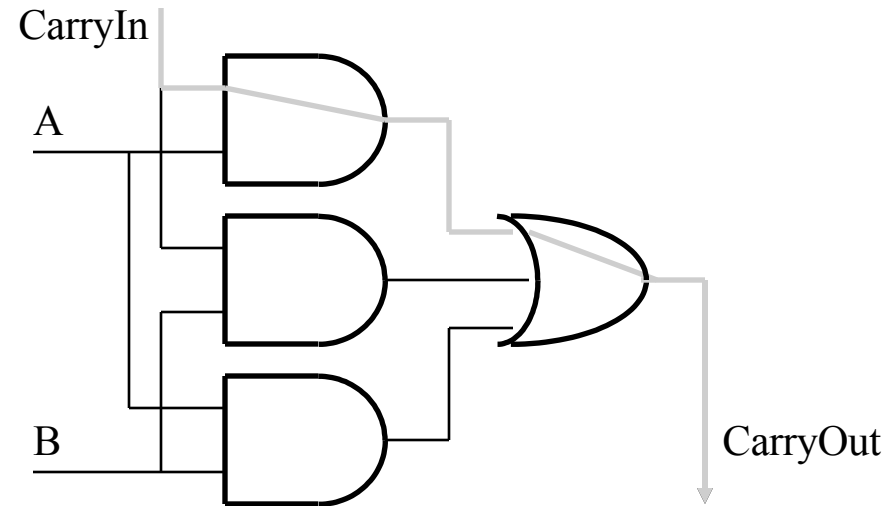
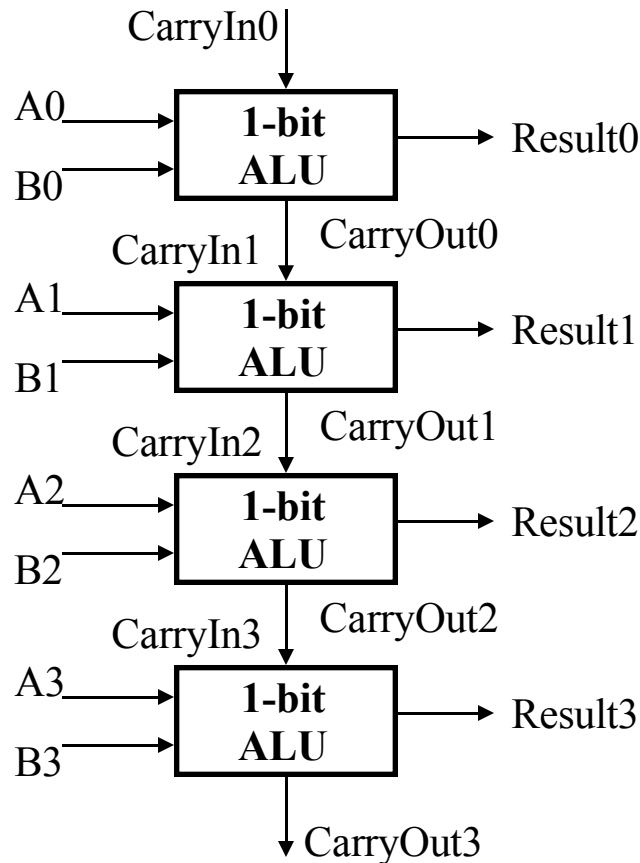
add?
 sub?
 and?
 or?
 beq?
 slt?



sign bit (adder output from bit 31)

The Disadvantage of Ripple Carry

- The adder we just built is called a “Ripple Carry Adder”
 - The carry bit may have to propagate from LSB to MSB
 - Worst case delay for an N-bit RC adder: $2N$ -gate delay



Ripple carry adders are slow. Faster addition schemes are possible that *accelerate* the movement of the carry from one end to the other.

MULTIPLY

- Paper and pencil example:

Multiplicand		1000
Multiplier	x	<u>1011</u>

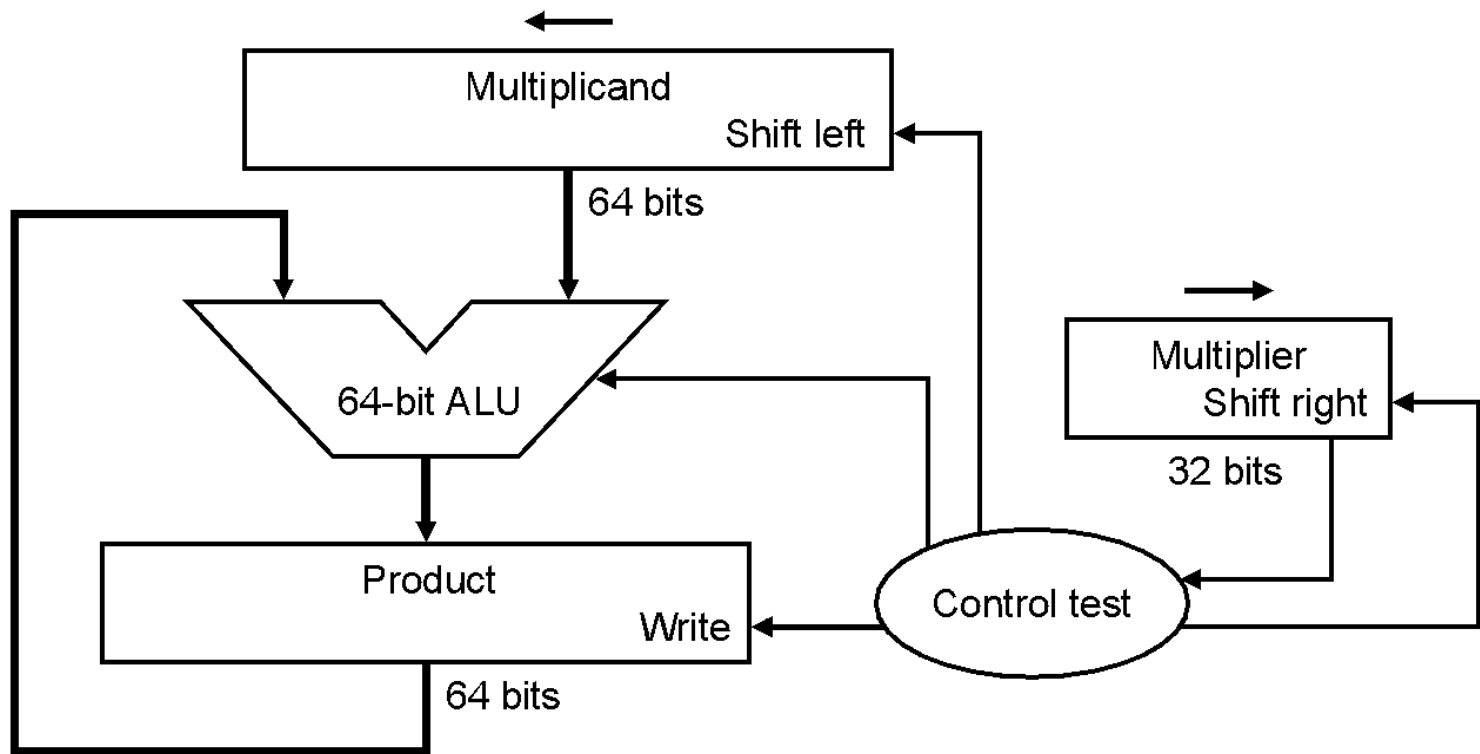
Product = ?

- m bits x n bits = m+n bit product
- Binary makes it easy:
 - 0 => place 0 (0 x multiplicand)
 - 1 => place multiplicand (1 x multiplicand)
- we'll look at a couple of versions of multiplication hardware

MULTIPLY HARDWARE

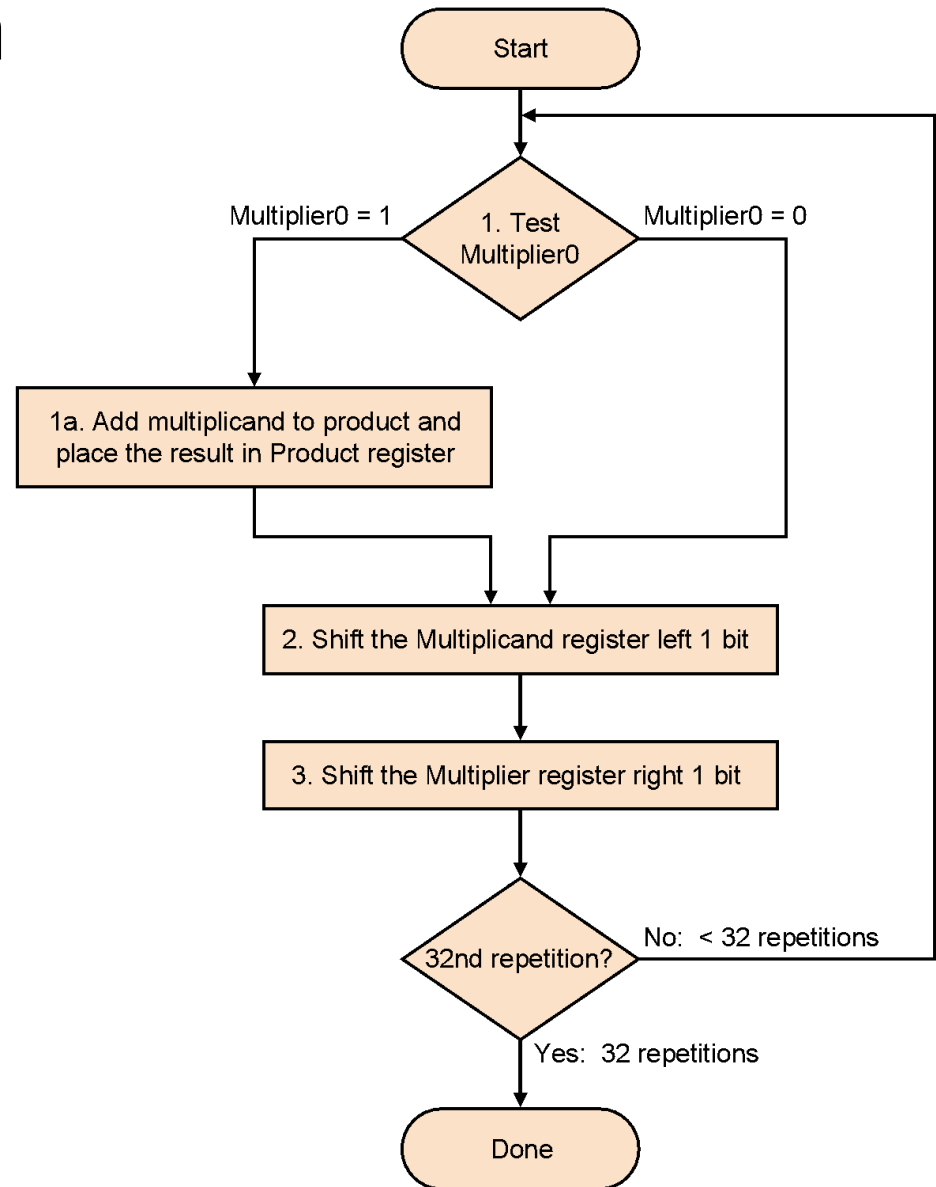
Version 1

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiply Algorithm Version 1

Multiplier	Multiplicand	Product
0101	0000 0110	0000 0000



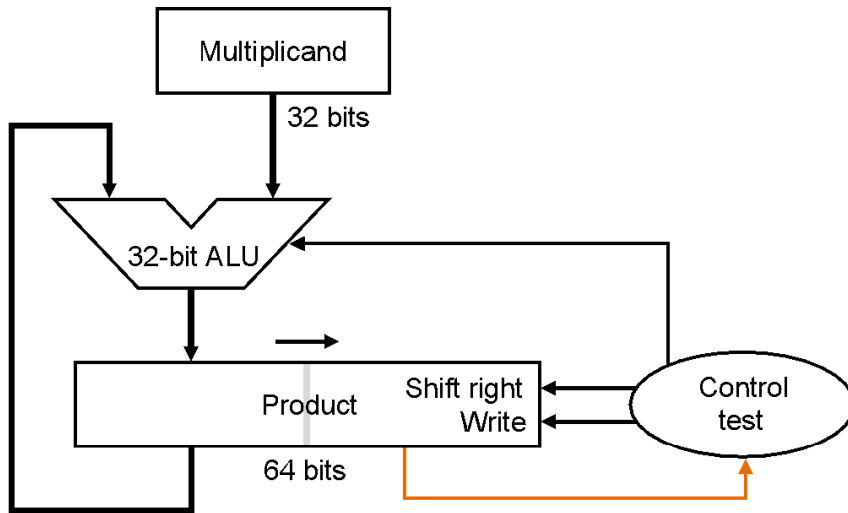
Observations on Multiply Version 1

- 1 clock per cycle $\Rightarrow$ 100 clocks per multiply
 - Ratio of multiply to add 100:1
- 1/2 bits in multiplicand always 0
 $\Rightarrow$ 64-bit adder is wasted
- 0's inserted in left of multiplicand as shifted
 $\Rightarrow$ least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?
- Wasted space (zeros) in product register exactly matches meaningful bits of multiplier at all times. Combine?

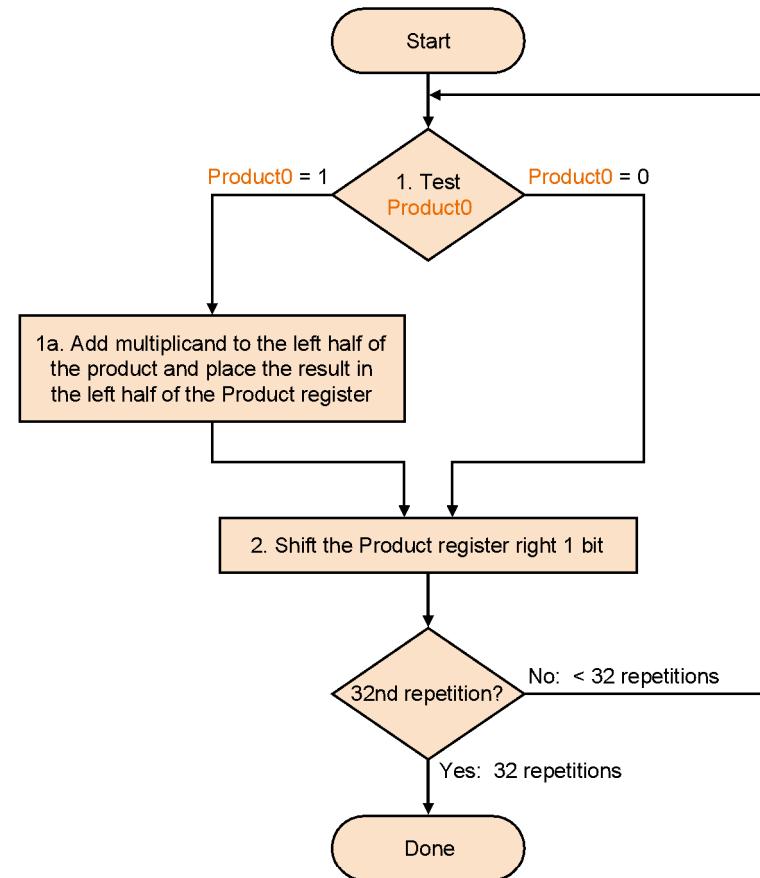
MULTIPLY HARDWARE

Version 2

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (Q-bit Multiplier reg)



Multiplicand	Product
0110	0000 0101
	0011 0010
	0001 1001
	0011 1100
	0001 1110



Observations on Multiply Version 2

- 2 steps per bit because Multiplier & Product combined
- 32-bit adder
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- What about **signed** multiplication?
 - easiest solution is to make both positive & remember whether to complement product when done.

Divide: Paper & Pencil

			Quotient
Divisor	1000	$\overline{)1101010}$	Dividend

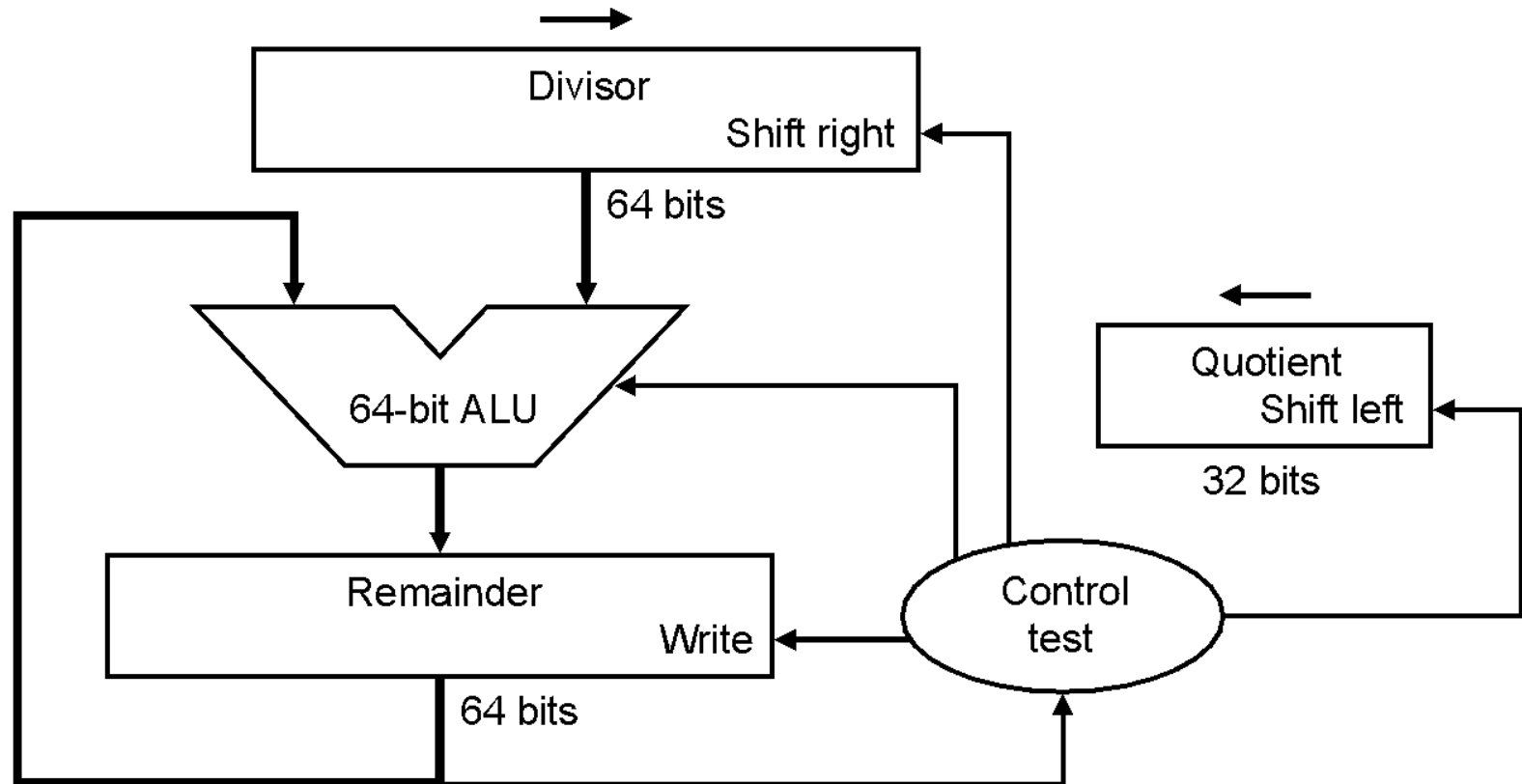
Remainder

- See how big a number can be subtracted, creating quotient bit on each step
 - Binary $\Rightarrow$ 1 * divisor or 0 * divisor
- Dividend = Quotient * Divisor + Remainder

DIVIDE HARDWARE

Version 1

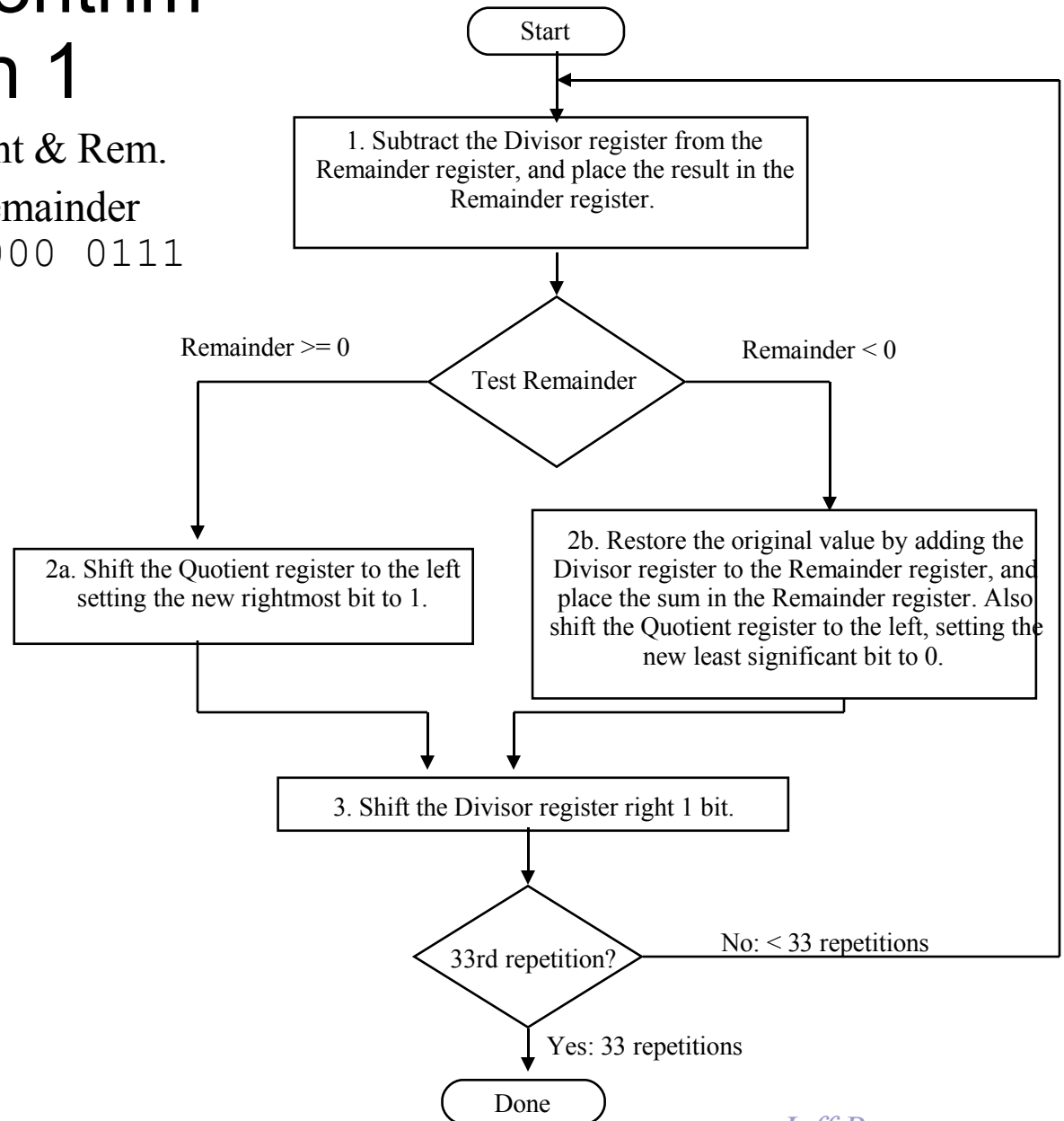
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 1

- Takes $n+1$ steps for n -bit Quotient & Rem.

Quotient	Divisor	Remainder
0000	0011 0000	0000 0111



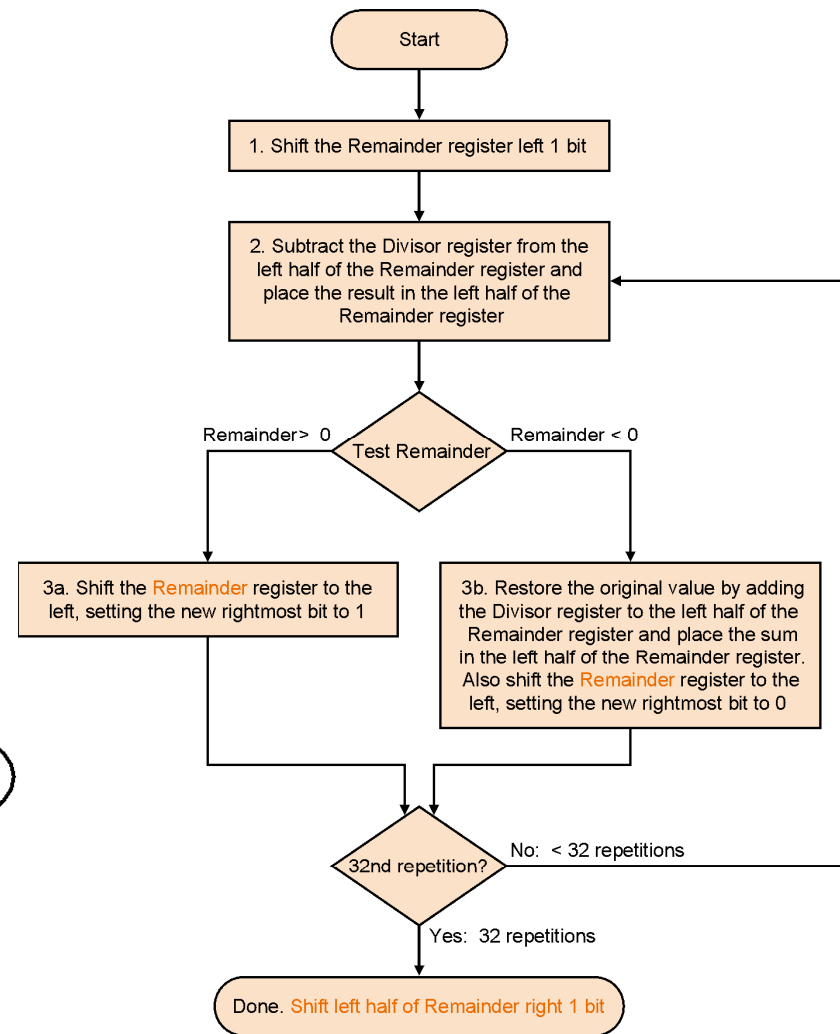
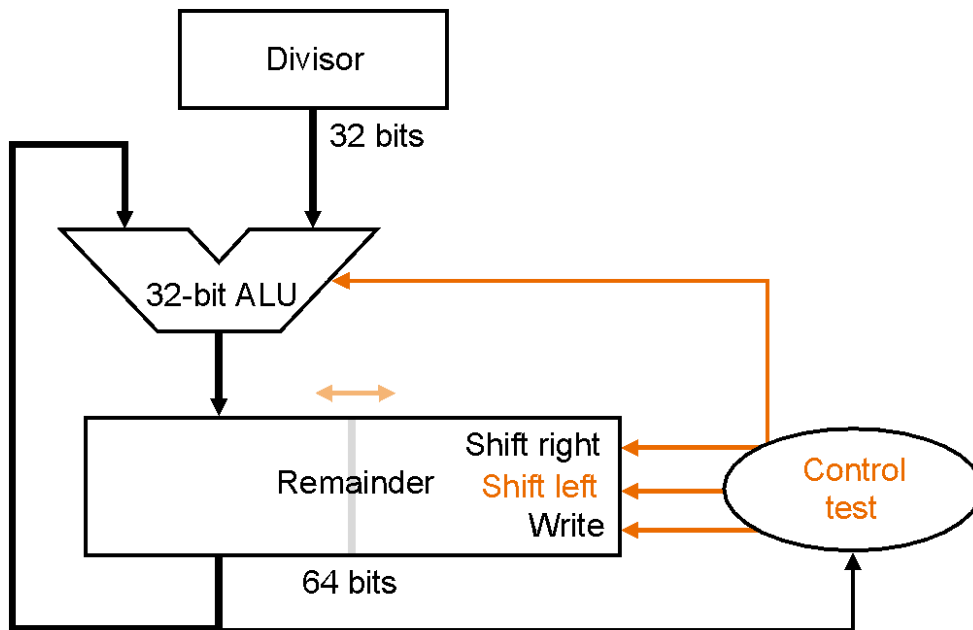
Divide Hardware Version 1

- Again, 64-bit adder is unnecessary.
- Quotient grows as remainder shrinks

DIVIDE HARDWARE

Version 2

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)



Observations on Divide Version 2

- Same Hardware as Multiply: just need ALU to add or subtract, and 63-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
 - Note: Dividend and Remainder must have same sign
 - Note: Quotient negated if Divisor sign & Dividend sign disagree

Key Points

- Instruction Set drives the ALU design
- ALU performance, CPU clock speed driven by adder delay
- Multiplication and division take much longer than addition, requiring multiple addition steps.

So Far

- Can do logical, add, subtract, multiply, divide, ...
- But.....
 - what about fractions?
 - what about really large numbers?

Binary Fractions

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

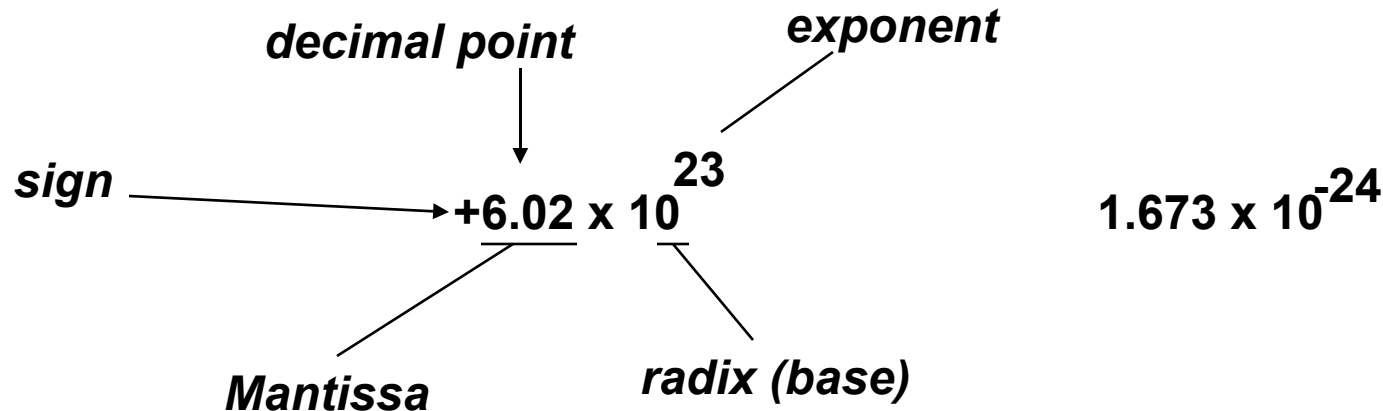
so...

$$101.011_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

e.g.,

$$.75 = 3/4 = 3/2^2 = 1/2 + 1/4 = .11$$

Recall Scientific Notation

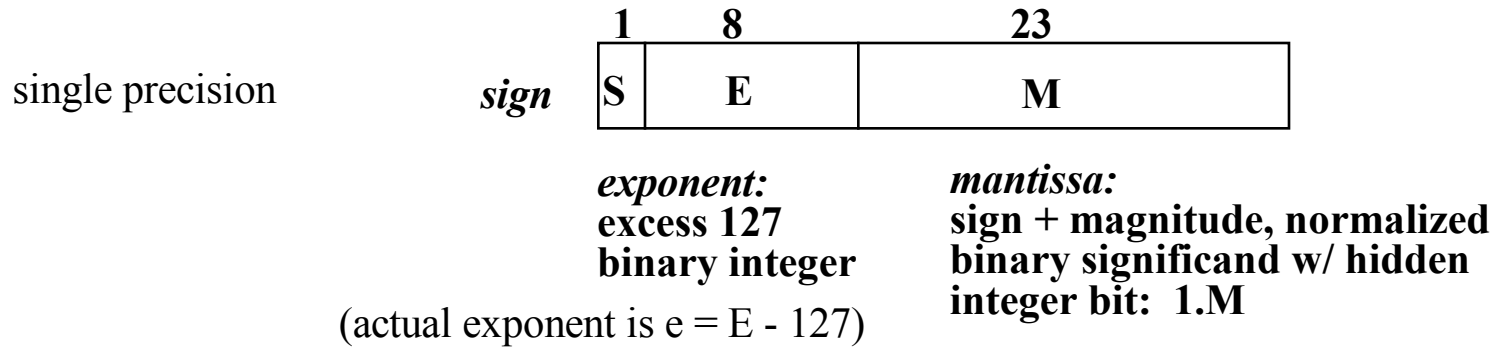


Issues:

- Arithmetic (+, -, *, /)
- Representation, Normal form
- Range and Precision
- Rounding
- Exceptions (e.g., divide by zero, overflow, underflow)
- Errors
- Properties (negation, inversion, if $A = B$ then $A - B = 0$)

Floating-Point Numbers

Representation of floating point numbers in IEEE 754 standard:



$$N = (-1)^S 2^{E-127} (1.M)$$

$0 < E < 255$

$0 = 0\ 00000000\ 0\ \dots\ 0$ $-1.5 = 1\ 01111111\ 10\ \dots\ 0$
 $325 = 101000101\ X\ 2^0 = 1.01000101\ X\ 2^8$
 $\quad = 0\ 10000111\ 010001010000000000000000$
 $.02 = .0011001101100\dots\ X\ 2^0 = 1.1001101100\dots\ X\ 2^{-3}$
 $\quad = 0\ 01111100\ 1001101100\dots$

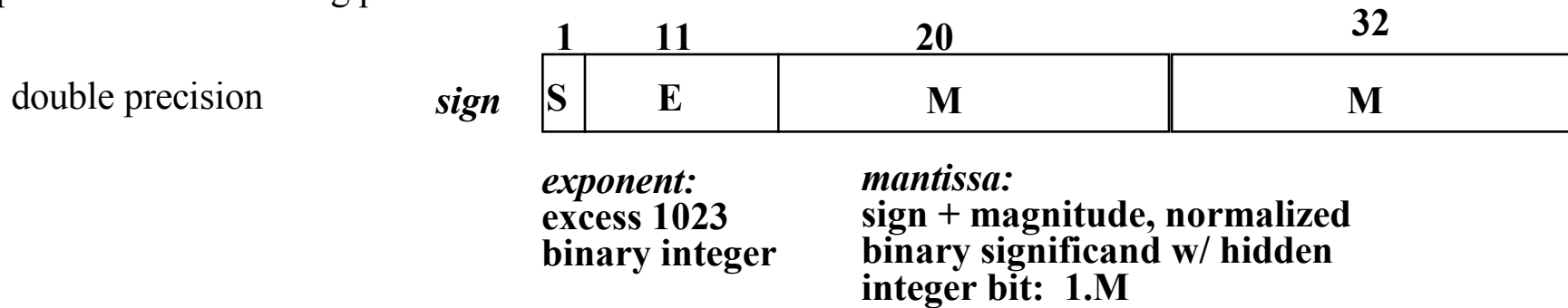
- range of about 2×10^{-38} to 2×10^{38}
- always normalized (so always leading 1, thus never shown)
- special representation of 0 ($E = 00000000$) (why?)
- can do integer compare for greater-than, sign

What do you notice?

- 0 0 00000000 0000000000...
 - $1.5 * 2^{-100}$ 0 00011011 1000000000...
 - $1.75 * 2^{-100}$ 0 00011011 1100000000...
 - $1.5 * 2^{100}$ 0 11100011 1000000000...
 - $1.75 * 2^{100}$ 0 11100011 1100000000...
-
- Does this work with negative numbers, as well?

Double Precision Floating Point

Representation of floating point numbers in IEEE 754 standard:



actual exponent is $e = E - 1023$

$$N = (-1)^S 2^{E-1023} (1.M) \quad \text{where } 0 < E < 2048$$

- 52 (+1) bit mantissa
- range of about 2×10^{-308} to 2×10^{308}

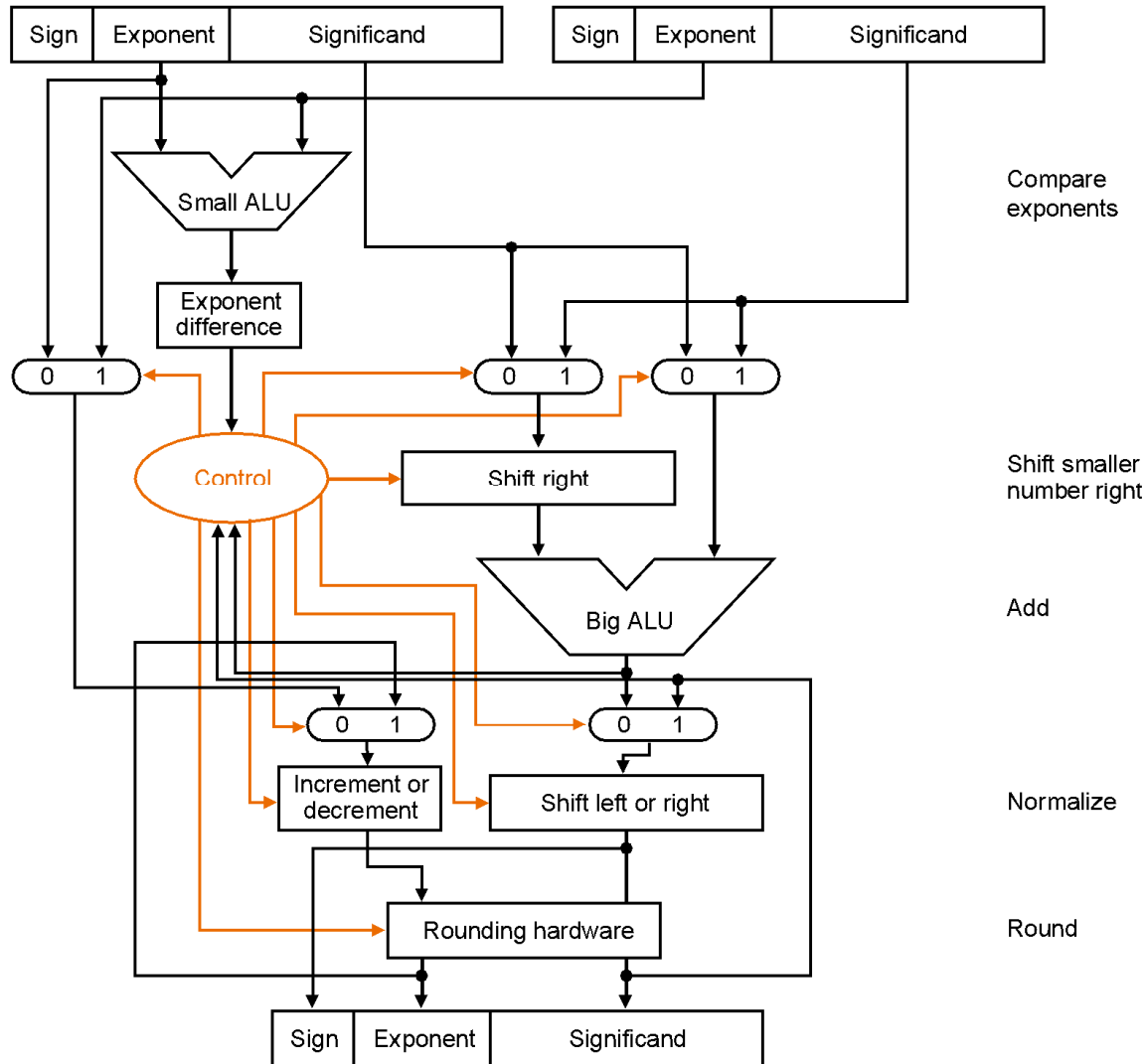
Floating Point Addition

- How do you add in scientific notation?

$$9.962 \times 10^4 + 5.231 \times 10^2$$

- Basic Algorithm
 1. Align
 2. Add
 3. Normalize
 4. Round

FP Addition Hardware



Floating Point Multiplication

- How do you multiply in scientific notation?

$$(9.9 \times 10^4)(5.2 \times 10^2) = 5.148 \times 10^7$$

- Basic Algorithm
 1. Add exponents
 2. Multiply
 3. Normalize
 4. Round
 5. Set Sign

FP Accuracy

- Extremely important in scientific calculations
- Very tiny errors can accumulate over time
- IEEE 754 FP standard has four rounding modes
 - always round up (toward $+\infty$)
 - always round down (toward $-\infty$)
 - truncate
 - round to nearest
 - => in case of tie, round to nearest even
- Requires extra bits in intermediate representations

Extra Bits for FP Accuracy

- *Guard bits* -- bits to the right of the least significant bit of the significand computed for use in normalization (could become significant at that point) and rounding.
- IEEE 754 has two extra bits and calls them *guard* and *round*.

When Keepin' it Real Goes Wrong

- Machine FP only *approximates* the real numbers
- Just calculating extra bits of precision is often enough
 - but not always!
 - knowing when, how to tell, and what to do about it can be subtle
 - analysis involves algorithm, program implementation, and hardware
- <http://www.cs.berkeley.edu/~wkahan/>
 - "How JAVA's Floating-Point Hurts Everyone Everywhere"
 - "Roundoff Degrades an Idealized Cantilever"
 - (and lots more)

Key Points

- Floating Point extends the range of numbers that can be represented, at the expense of precision (accuracy).
- FP operations are very similar to integer, but with pre- and post-processing.
- Rounding implementation is critical to accuracy over time.