EE204: Computer Architecture

Today’s Lecture

- Signed and Unsigned Numbers
- Integers:
  - Multiply
  - Divide
- Floating Points:
  - Addition

Multiply

- Grade school shift-add method:

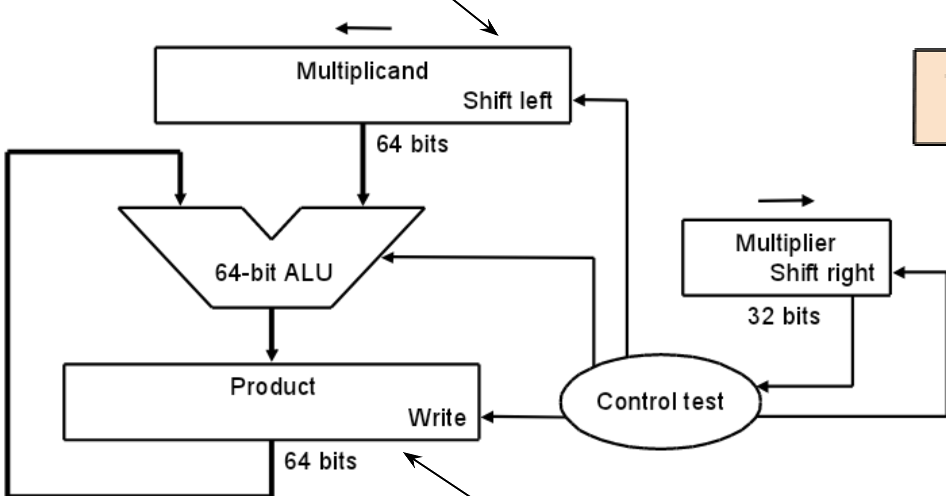
```
Multiplicand          1000
Multiplier       x       1001
                    ---------
                         1000
                        0000
                       0000
                      1000
                   ----------
Product            01001000
```

- m bits x n bits = m+n bit product
- Binary makes it easy:
  - multiplier bit 1 => copy multiplicand (1 x multiplicand)
  - multiplier bit 0 => place 0 (0 x multiplicand)

Shift-add Multiplier Version 1

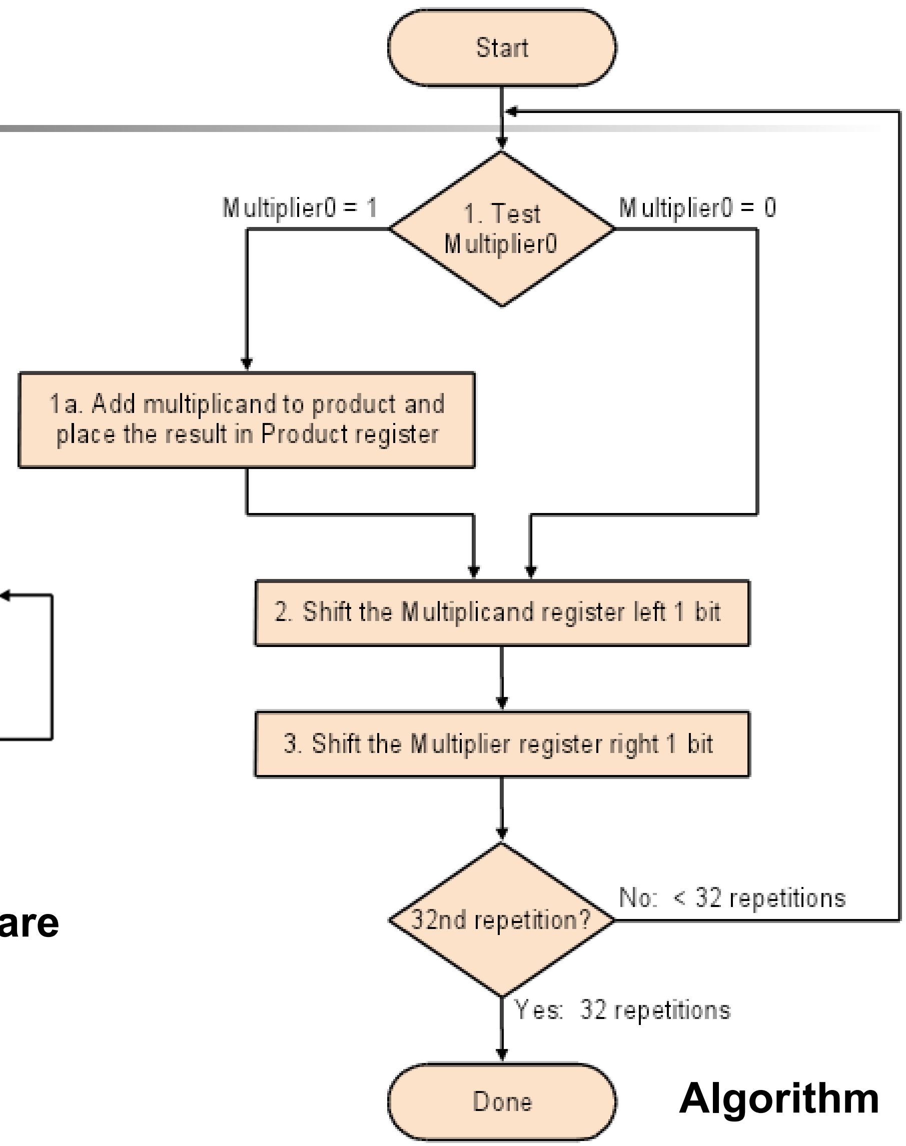
Multiplicand: 1000
Multiplier: 1001

32-bit multiplicand starts at right half of multiplicand register



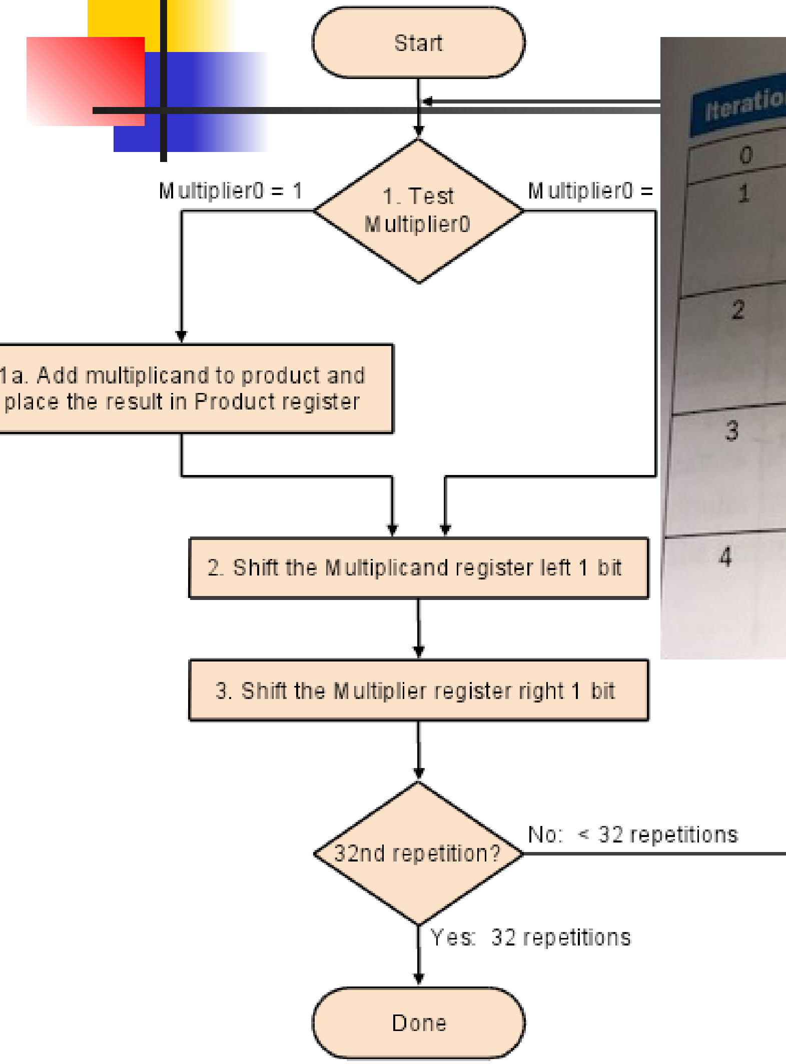
Product register is initialized at 0

Multiplicand register, product register, ALU are 64-bit wide; multiplier register is 32-bit wide



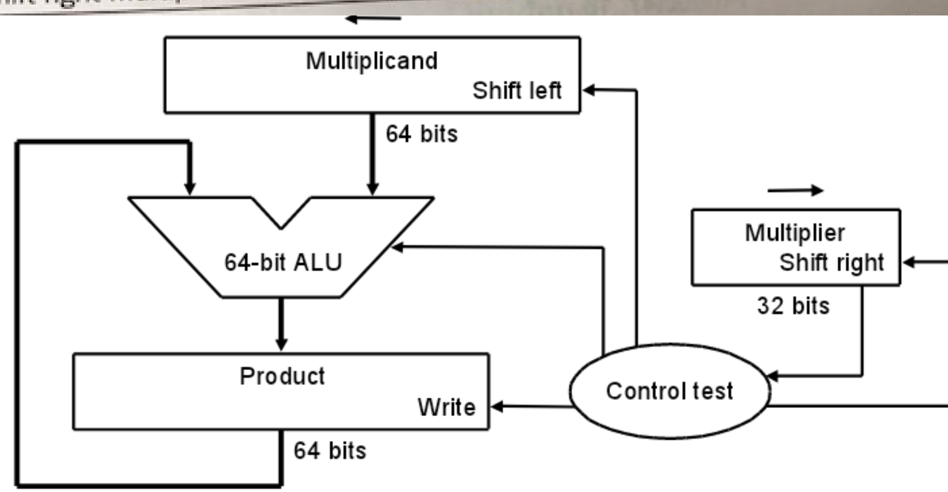
Algorithm

Shift-add Multiplier Version1



Algorithm

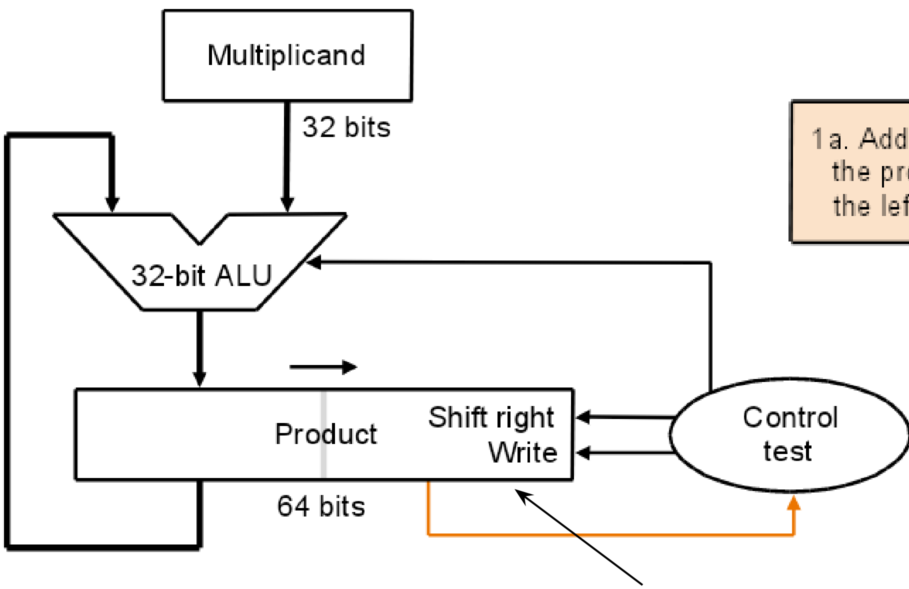
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110



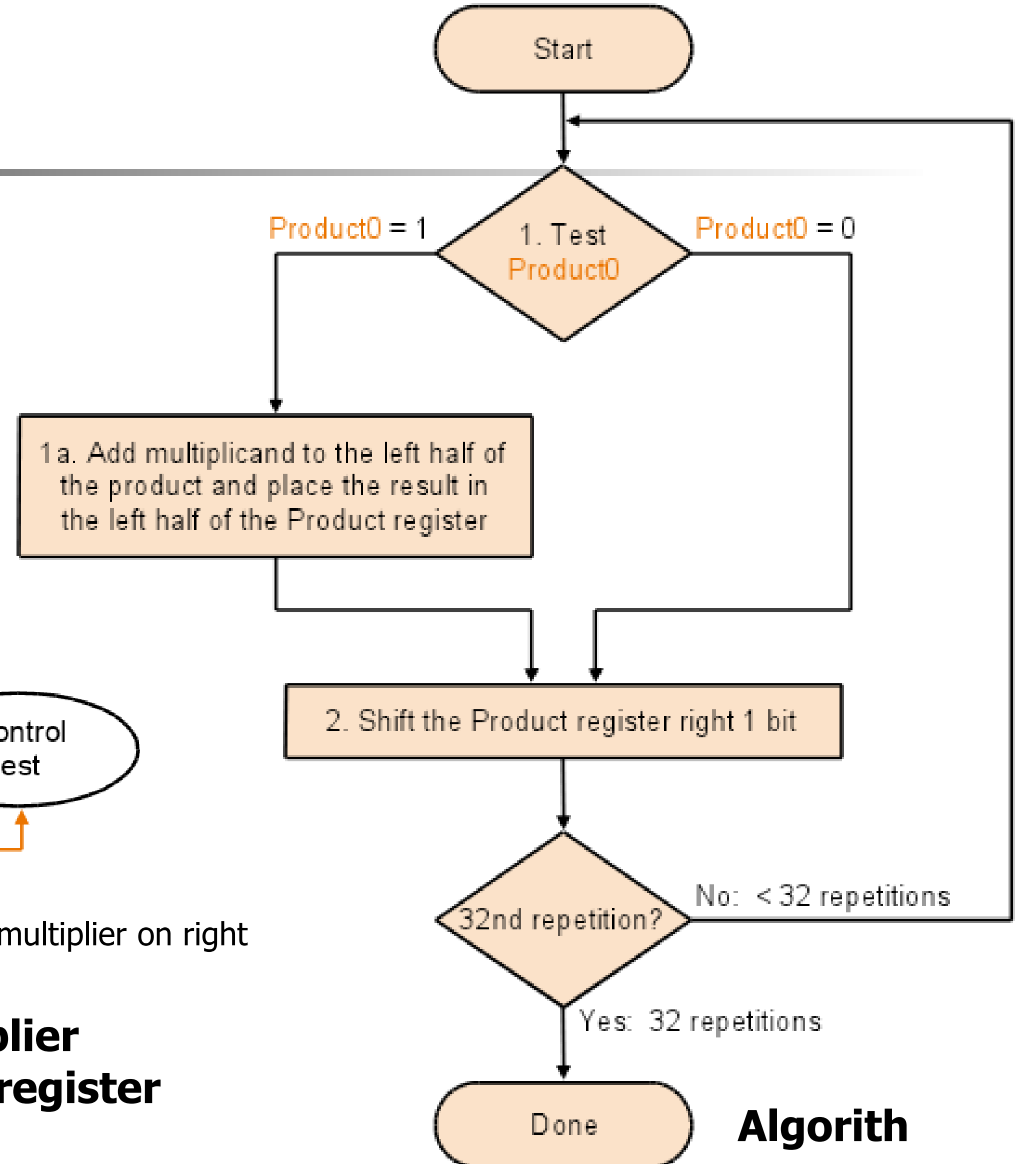
Observations on Multiply Version 1

- 1 step per clock cycle $\Rightarrow$ nearly 100 clock cycles to multiply two 32-bit numbers
- Half the bits in the multiplicand register always 0
  $\Rightarrow$ 64-bit adder is wasted
- 0's inserted to right as multiplicand is shifted left
  $\Rightarrow$ least significant bits of product never change once formed
- Improved version ….

Shift-add Multiplier Version 2



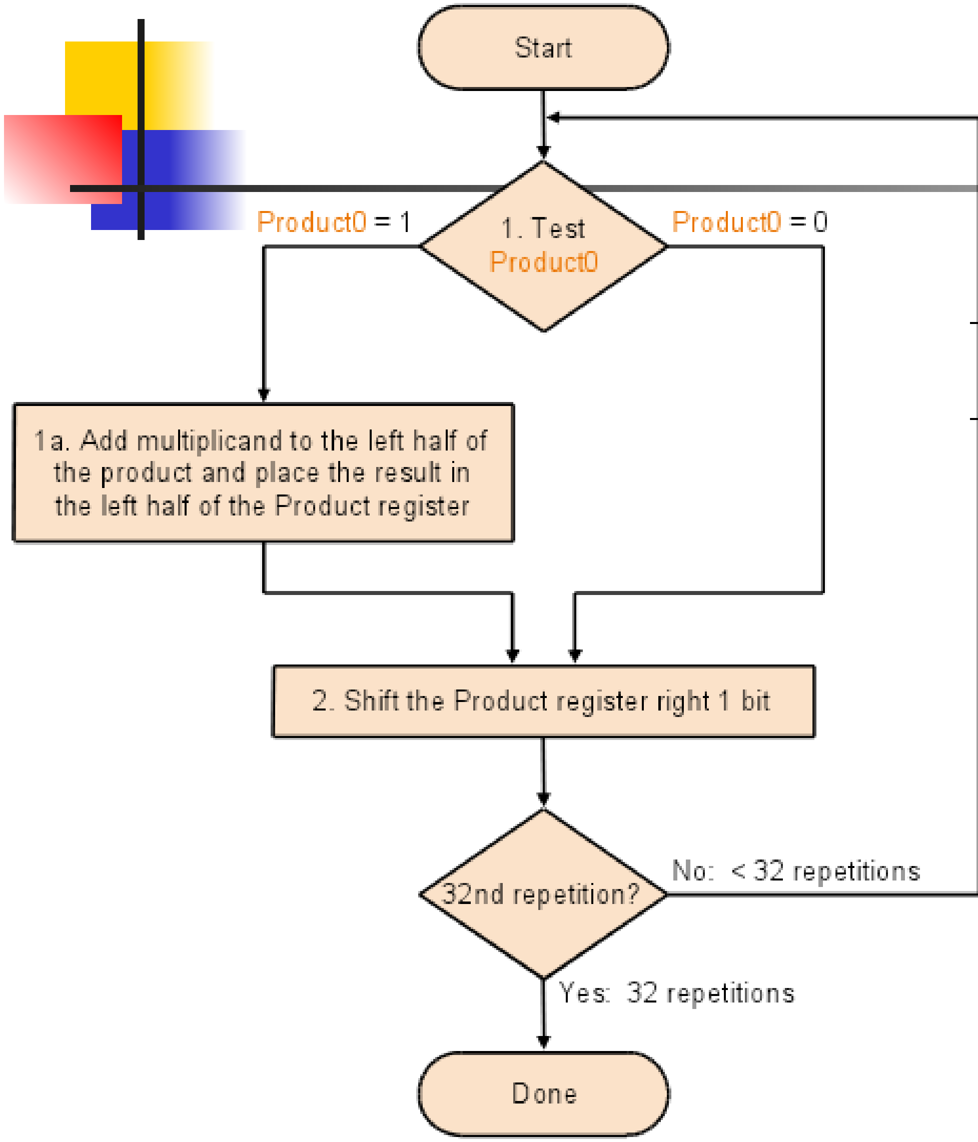
Product register is initialized with multiplier on right



No separate multiplier register; multiplier placed on right side of 64-bit product register

Algorithm

Shift-add Multiplier Version 2



Algorithm

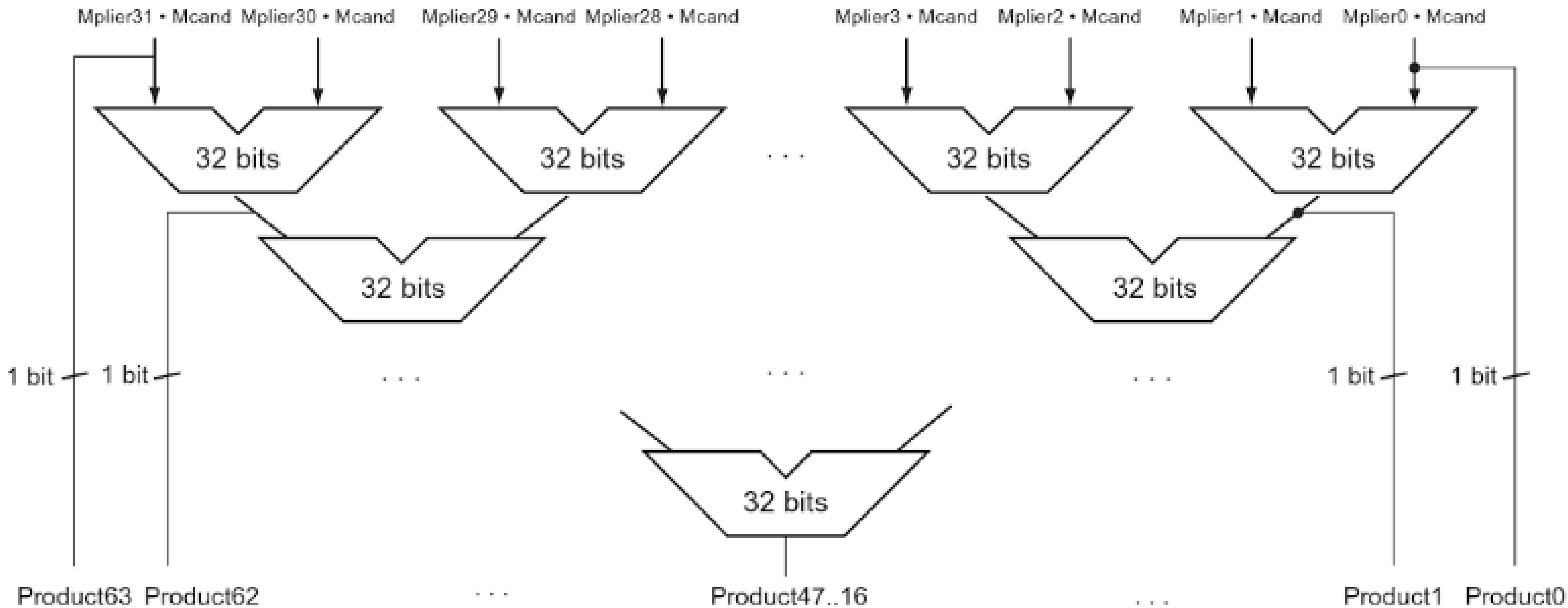
Example: 0010 * 0011:

Itera-tion	Step	Multiplicand	Product
0	init values	0010	0000 0011
1	1a	0010	0010 0011
	2	0010	0001 0001
2	…		

Observations on Multiply Version 2

- 2 steps per bit because multiplier & product combined
- What about *signed* multiplication?
  - An easy solution is to make both positive and remember whether to negate product when done, i.e., leave out the sign bit, run for 31 steps, then negate if multiplier and multiplicand have opposite signs

Fast Multiplication Hardware



Divide

```
                     1001   Quotient
Divisor 1000    | 1001010   Dividend
            -1000
                 10
                 101
                 1010
                -1000
                   10   Remainder
```

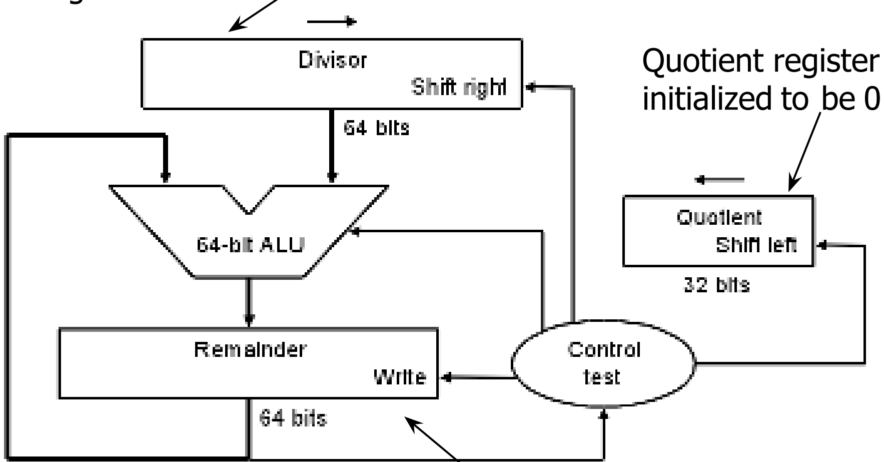
- Junior school method: see how big a multiple of the divisor can be subtracted, creating quotient digit at each step
- Binary makes it easy $\Rightarrow$ *first* , try 1 * divisor; *if too big* , 0 * divisor
- Dividend = (Quotient * Divisor) + Remainder

Example: 0111 / 0010:

Divide Version 1

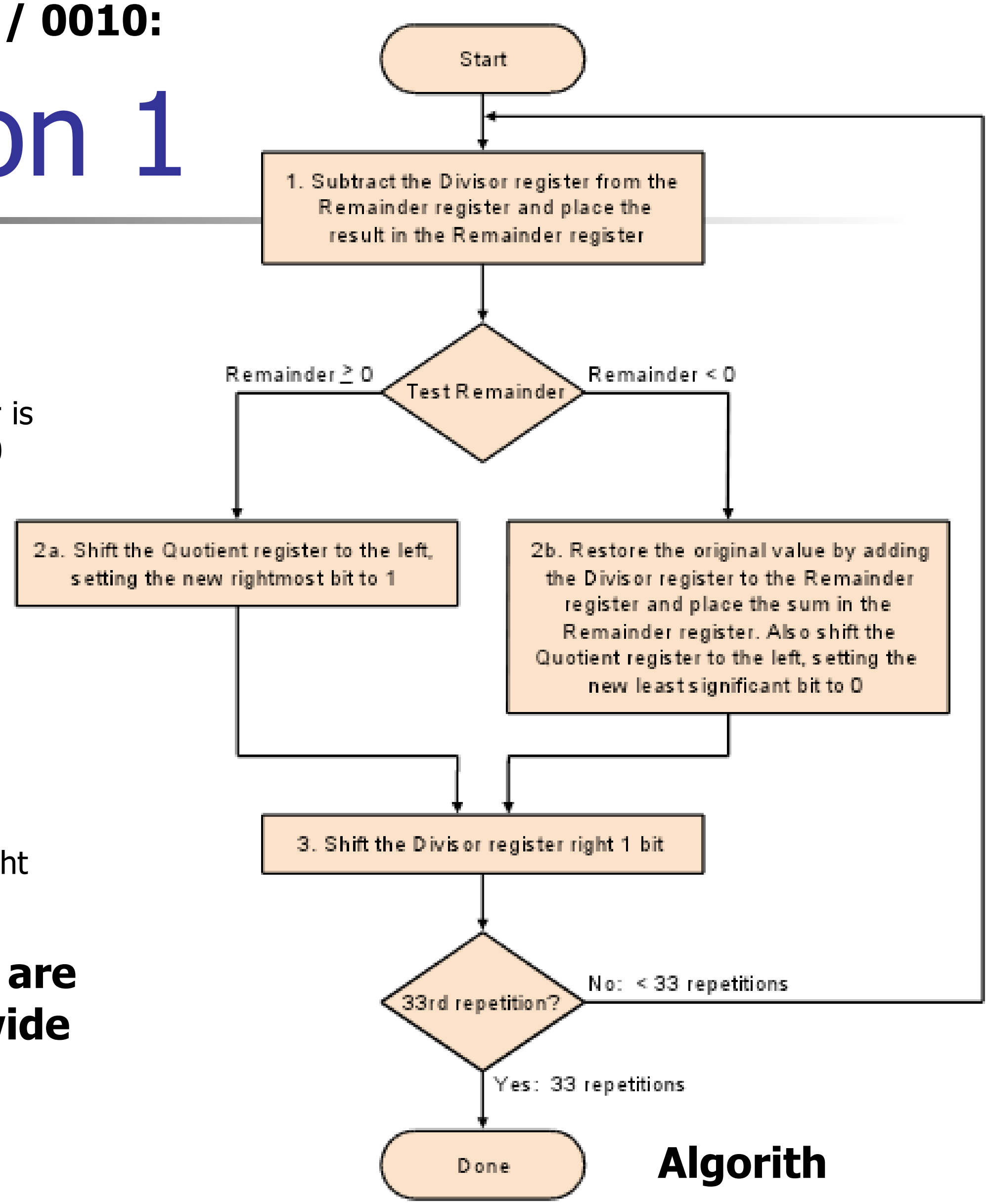
32-bit divisor starts at left half of divisor register

Quotient register is initialized to be 0

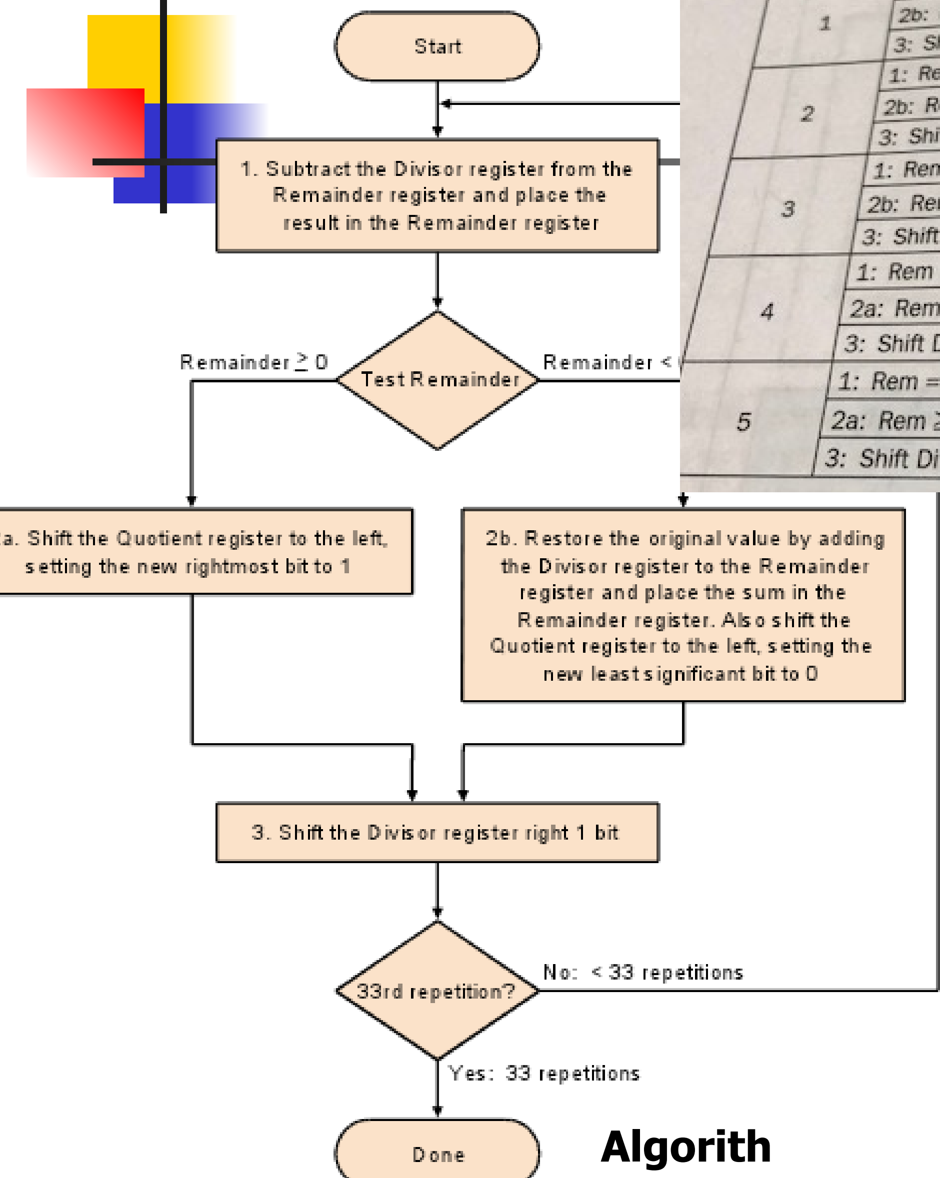


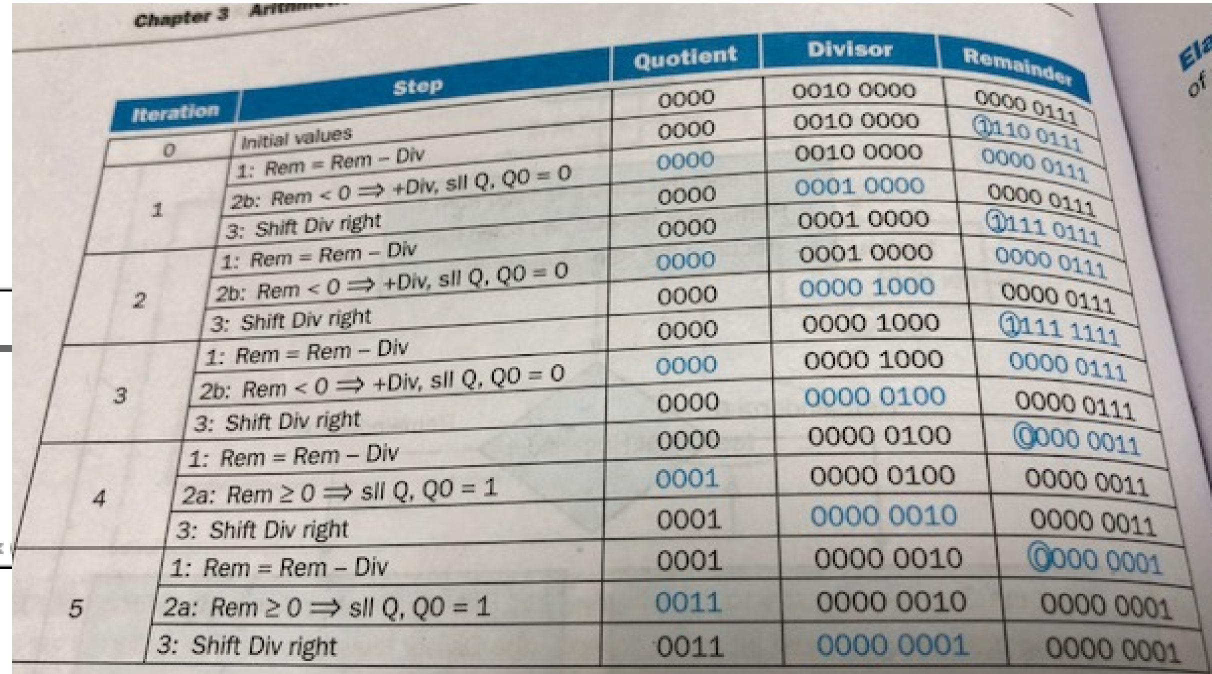
Remainder register is initialized with the dividend at right

Divisor register, remainder register, ALU are 64-bit wide; quotient register is 32-bit wide



Algorithm

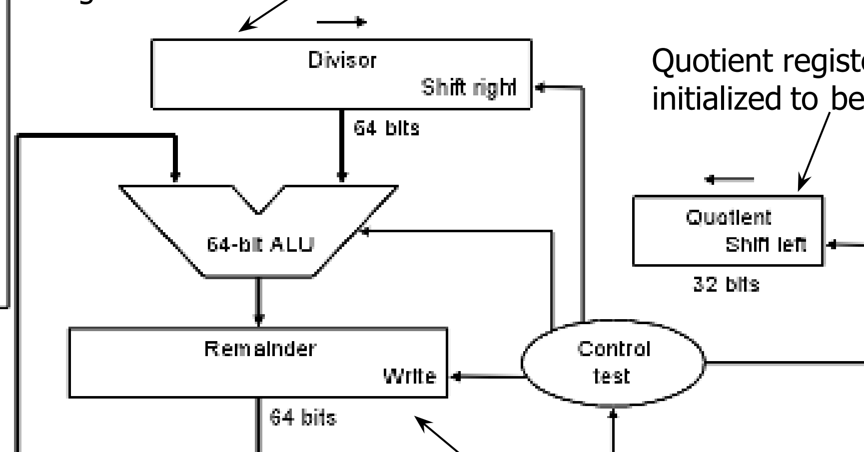




Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

32-bit divisor starts at left half of divisor register

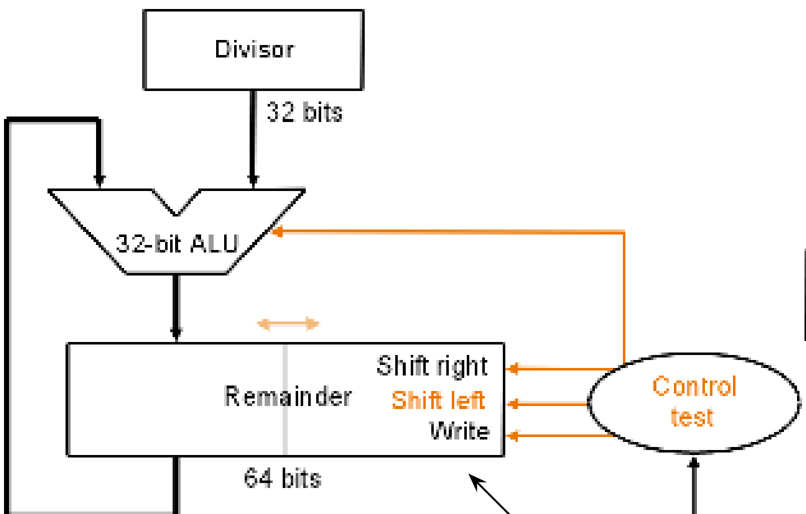
Quotient register is initialized to be 0



Remainder register is initialized with the dividend at right

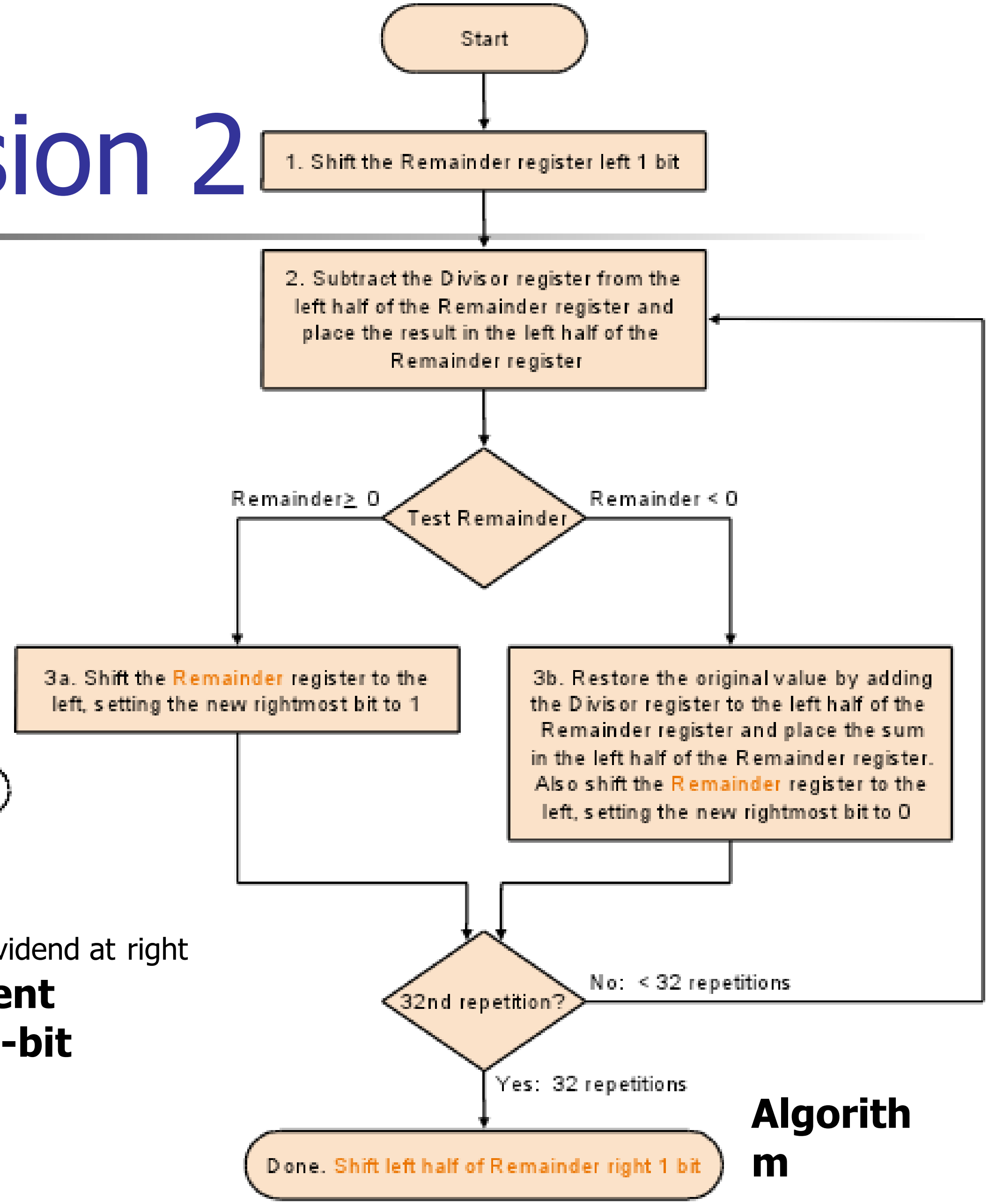
Algorithm

Divide Version 2



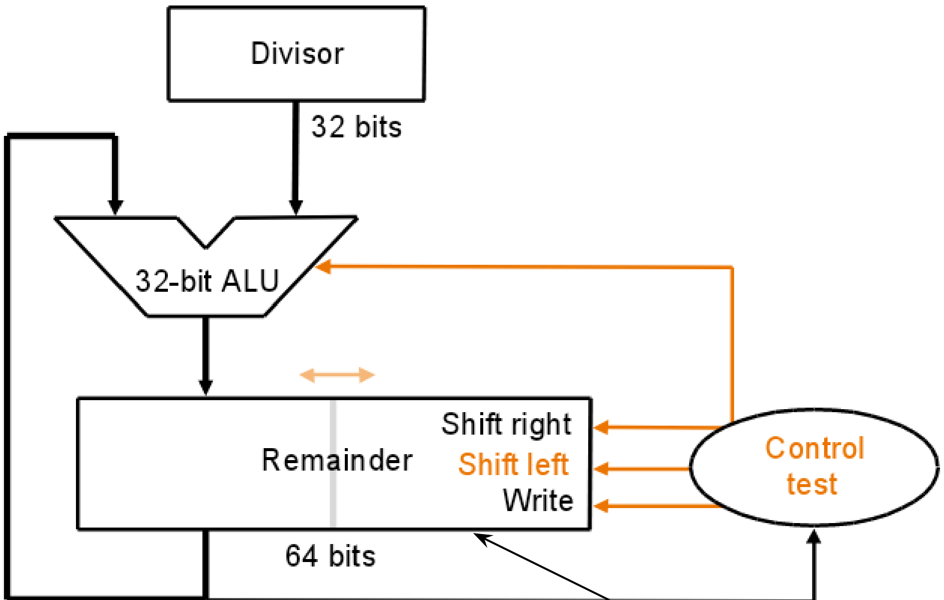
Remainder register is initialized with the dividend at right

No separate quotient register; quotient is entered on the right side of the 64-bit remainder register



Algorithm

Divide Version 2



Remainder register is initialized with the dividend at right

No separate quotient register; quotient is entered on the right side of the 64-bit remainder register

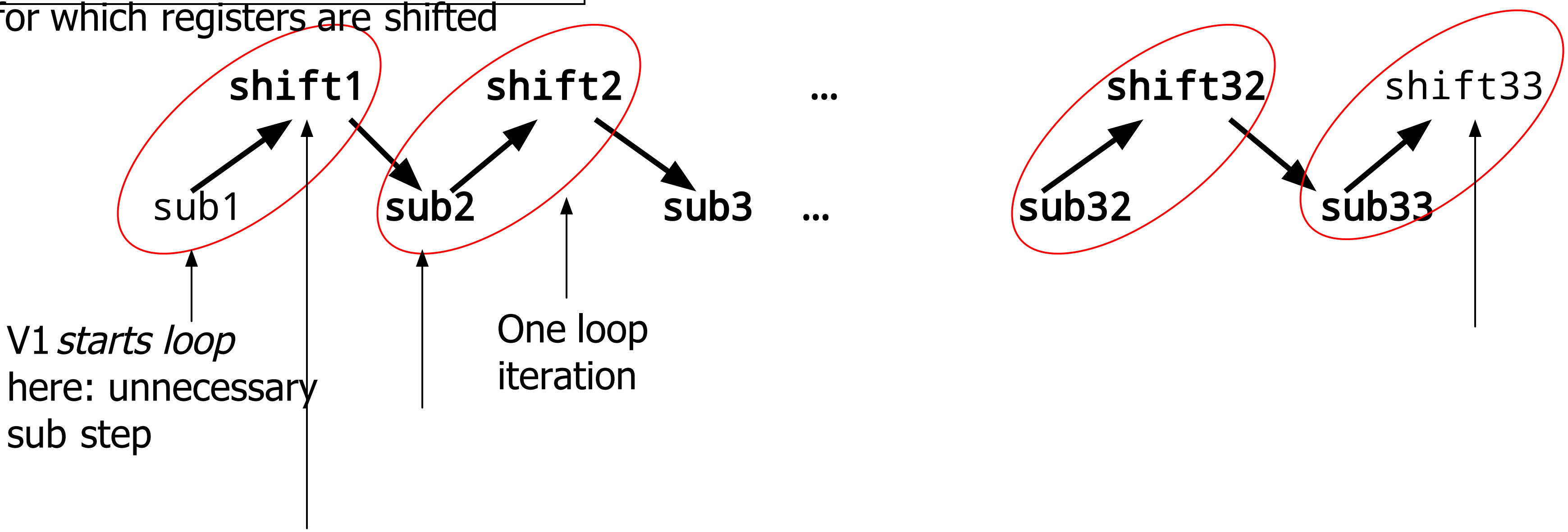
Number of Iterations

Why the extra iteration in Version 1?

Ovals represent loop iterations

Shift: see the version descriptions for which registers are shifted

Main insight – sub(i+1) must actually follow shifti of the divisor (or remainder, depending on version) and the resulting bit in the quotient appears on shift(i+1)



Floating Point

- We need a way to represent
  - numbers with fractions, e.g., `3.1416`
  - very small numbers (in absolute value), e.g., `.00000000023`
  - very large numbers (in absolute value) , e.g., $-3.15576 * 10^{46}$
- Representation:
  - *scientific* : sign, exponent, fraction form:
    binary point
    - $(-1)^{sign}$ * fraction* $2^{exponent}$. E.g., $-101.001101 * 2^{111001}$
  - more bits for *significand* gives more accuracy
  - more bits for *exponent* increases range
  - if $1 \leq$ significand • 10_{two} $(=2_{ten})$ then number is *normalized* , **except for** number 0 which is normalized to significand 0
    - E.g., $-101.001101 * 2^{111001} = -1.01001101 * 2^{111011}$ (normalized)

IEEE 754 Floating-point Standard

- IEEE 754 floating point standard:
  - single precision: one word

31	bits 30 to 23	bits 22 to 0
sign	8-bit exponent	23-bit significand

  - double precision: two words

31	bits 30 to 20	bits 19 to 0
sign	11-bit exponent	upper 20 bits of 52-bit significand

bits 31 to 0
lower 32 bits of 52-bit significand

IEEE 754 Floating-point Standard

- Sign bit is 0 for positive numbers, 1 for negative numbers
- Number is assumed normalized and leading 1 bit of fraction left of binary point (for non-zero numbers) is *assumed* and not shown
  - e.g., fraction 1.1001… is represented as 1001…,
  - **exception** is number 0 which is represented as all 0s (see next slide)
  - for other numbers:
    value = $(-1)^{\text{sign}} * (1 + \text{fraction}) * 2^{\text{exponent value}}$
- 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
  - therefore, for non-0 numbers:
    value = $(-1)^{\text{sign}} * (1 + \text{fraction}) * 2^{\overbrace{(\text{stored\_exponent} - \text{bias})}^{\text{equals real exponent value}}}$

IEEE 754 Floating-point Standard

- Special treatment of 0:
  - if exponent is all 0 and significand is all 0, then the value is 0 (sign bit may be 0 or 1)
  - if exponent is all 0 and significand is *not* all 0, then the value is $(-1)^{\text{sign}} * (1 + \text{significand}) * 2^{-127}$
    - therefore, all 0s is taken to be 0 and not 2^{-127} (as would be for a non-zero normalized number); similarly, 1 followed by all 0's is taken to be 0 and not -2^{-127}

- *Example* : Represent -0.75_{ten} in IEEE 754 single precision
  - decimal: $-0.75 = -3/4 = -3/2^2$
  - binary: $-11/100 = -.11 = -1.1 \times 2^{-1}$
  - IEEE single precision floating point exponent = bias + exponent value
    $= 127 + (-1) = 126_{\text{ten}} = 01111110_{\text{two}}$
  - IEEE single precision: 10111111010000000000000000000000

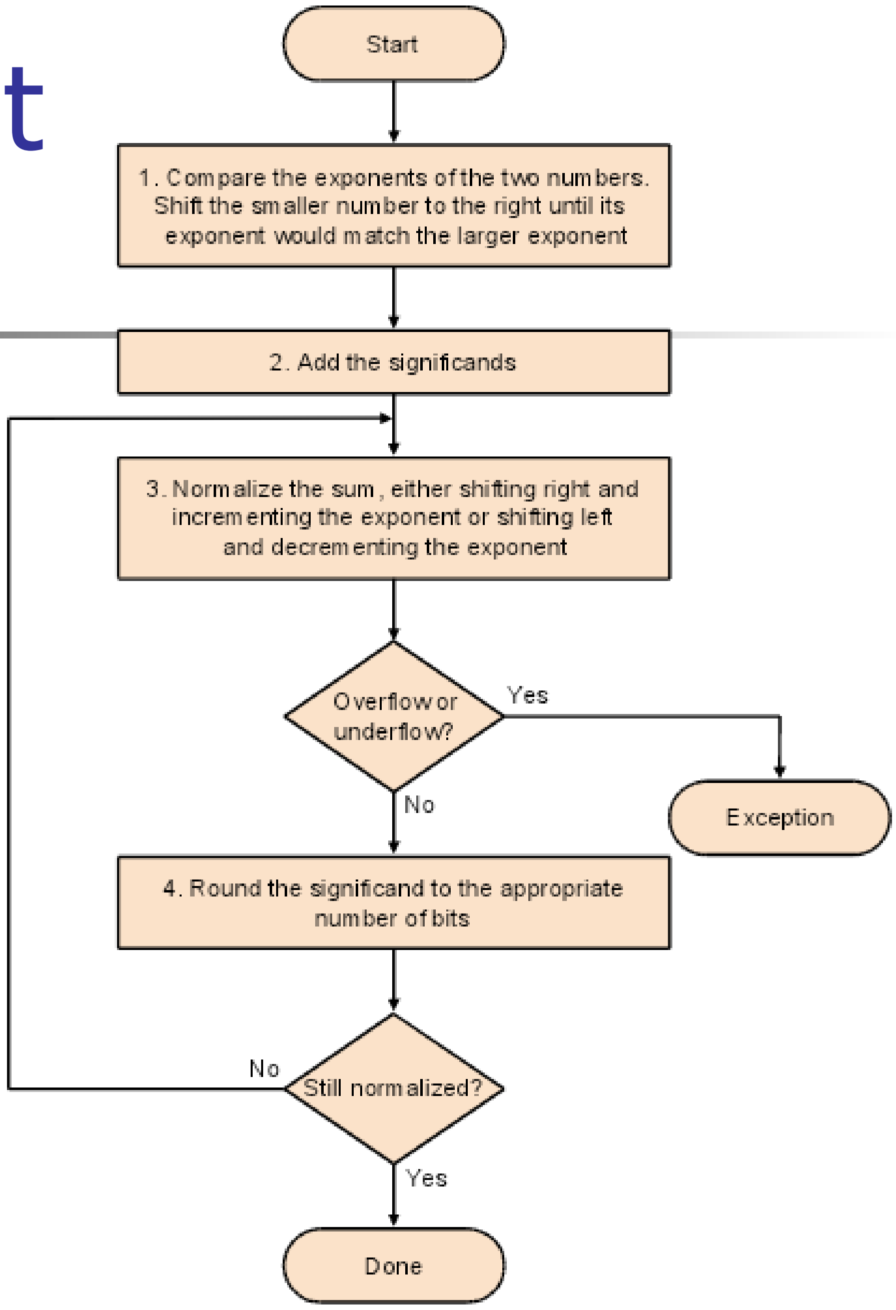
sig
n

exponen
t

fractio
n

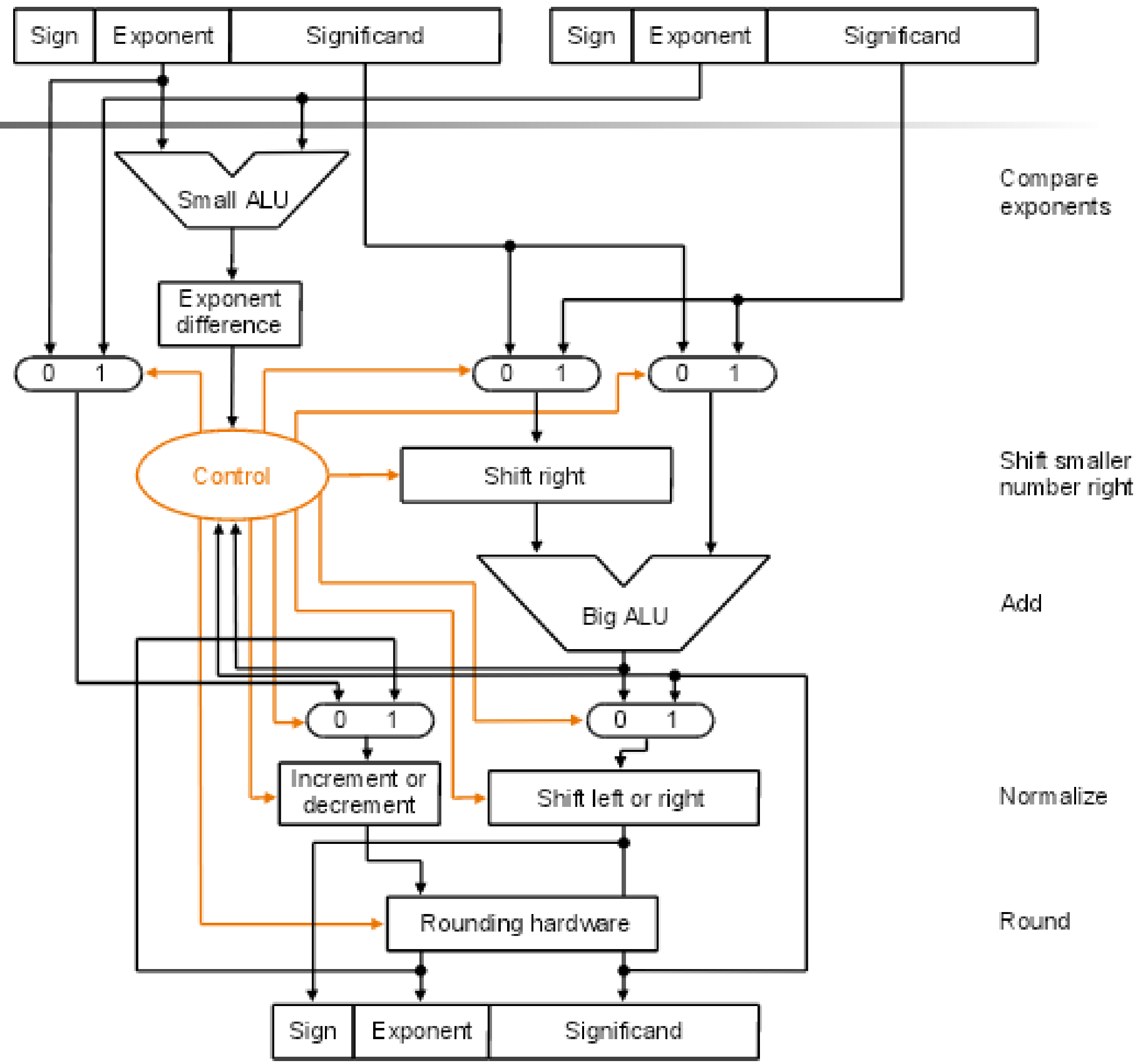
Floating Point Addition

- Algorithm:



Floating Point Addition

- Hardware:



Floating Point Complexities

- In addition to *overflow* we can have *underflow* (number too small)
- *Accuracy* is the problem with both overflow and underflow because we have only a finite number of bits to represent numbers that may actually require arbitrarily many bits
  - limited precision $\Rightarrow$ rounding $\Rightarrow$ rounding error
  - 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 implementing the standard can be even worse
  - see text for discussion of Pentium bug!

Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist:
  - two's complement
  - IEEE 754 floating point
- Computer instructions determine *meaning* of the bit patterns.
- Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation)

- Covered the following sections of the Patterson Book:
  - 2.4 Signed and Unsigned Numbers
  - 3.3: Multiplication
  - 3.4: Division
  - 3.5 Floating Point