U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

Prof. Karu Sankaralingam

Arithmetic Part B (3.6-3.9)

www.cs.wisc.edu/~karu/cs552/spring2008/

Slides combined and enhanced by Mark D. Hill from work
by Falsafi, Marculescu, Nagle, Patterson, Roth, Rutenbar,
Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Revisiting Computer Arithmetic

- Earlier in semester
  - Integer Representation
  - Addition/Subtraction
  - Logical ops
    - Barrel shifter

- Next:
  - Integer Multiplication
  - Integer Division
  - Floating point numbers
    - Representation
    - Addition/multiplication

Grade School Multiplication

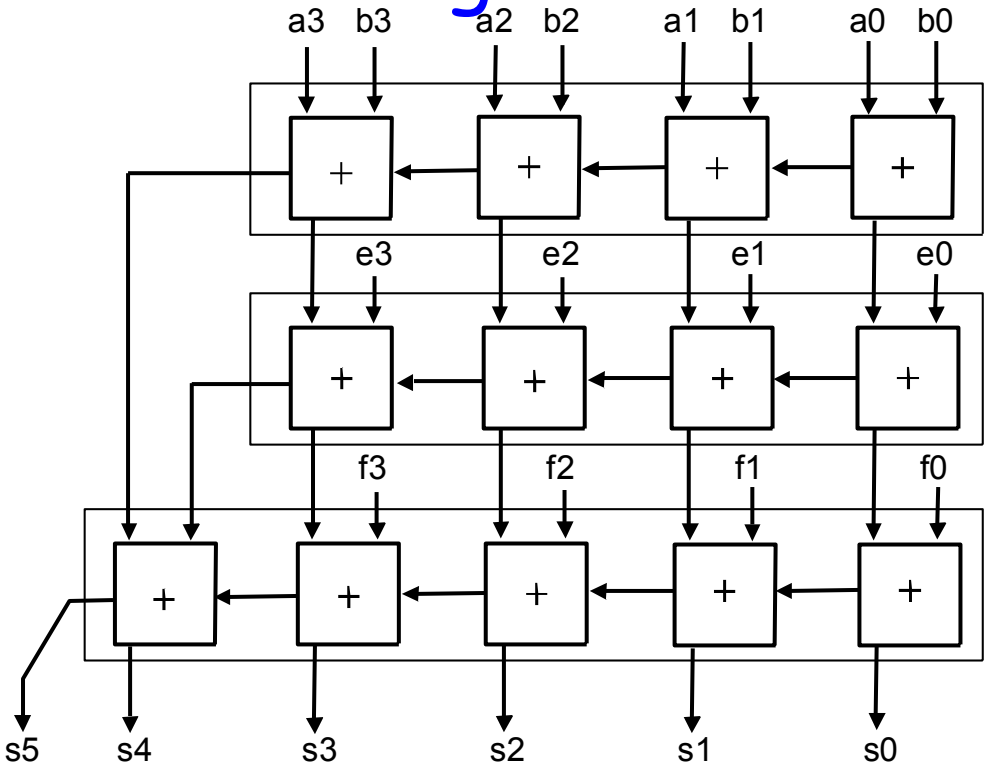
- Multiplicand (1000_{10}) and Multiplier (1001_{10})

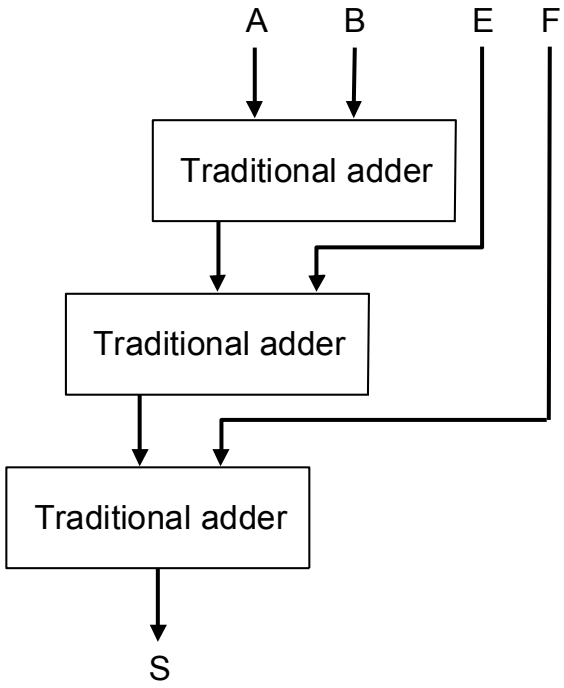
```
    1000
   0000
  0000
 1000
-------
1001000
```

1001000_{10}

- Two approaches
  - Purely combinational
  - Sequential

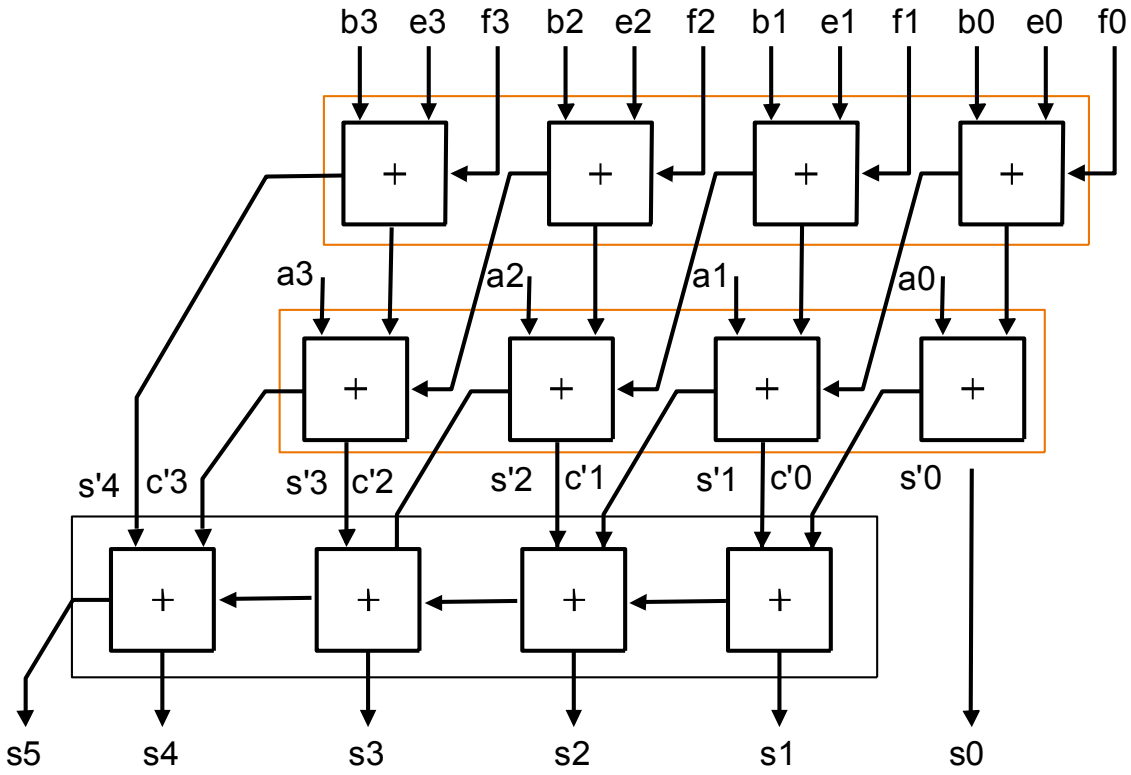
Adding more than two operands

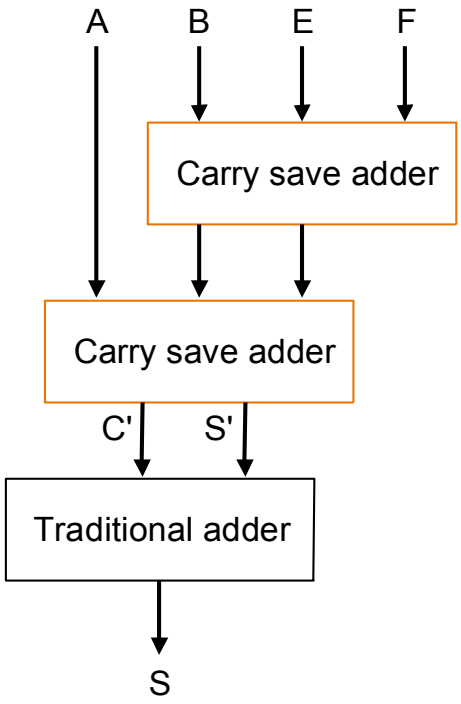




- Add four numbers (A,B,E,F)
- "Traditional adder" : ripple carry
- Basic block: Full adder
- N-1 adders for N numbers

Carry-save adders





- Same basic block
- Adds three numbers per stage (using Cins) and output 2 numbers (Cout and S)
  - Number of stages reduced to ~2/3N for N-numbers

Advantage of CSA approach

- 3 numbers => 1 CSA stage
- 4-9 numbers => 3 CSA stages
- Etc.

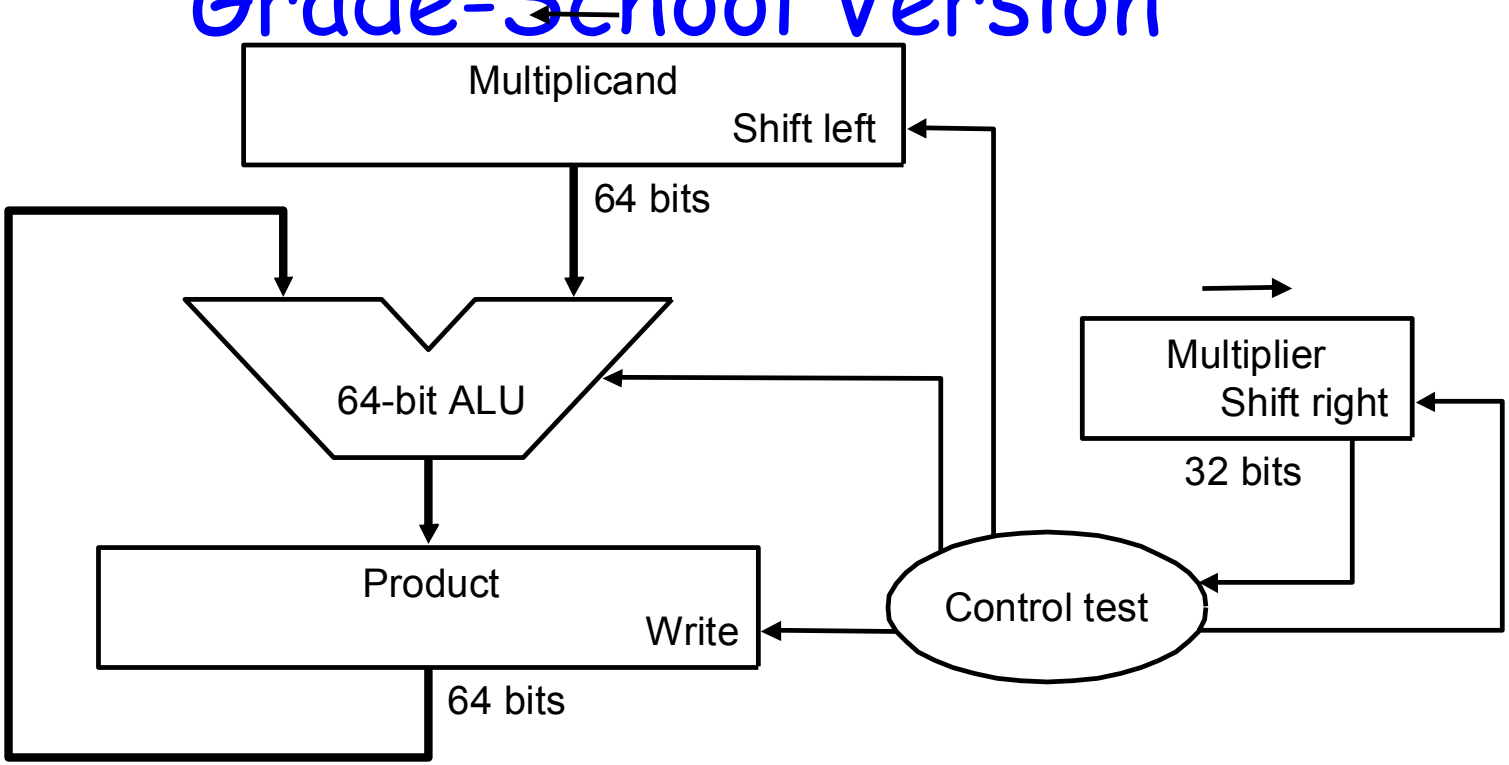
Purely combinational

- Uses carry-save adders (Wallace Tree)
- Partial products
  - AND multiplicand with multiplier bits

Grade School Multiplication

- Sequential approach
- Reuse hardware
  - Partial products generated, accumulated into product each cycle

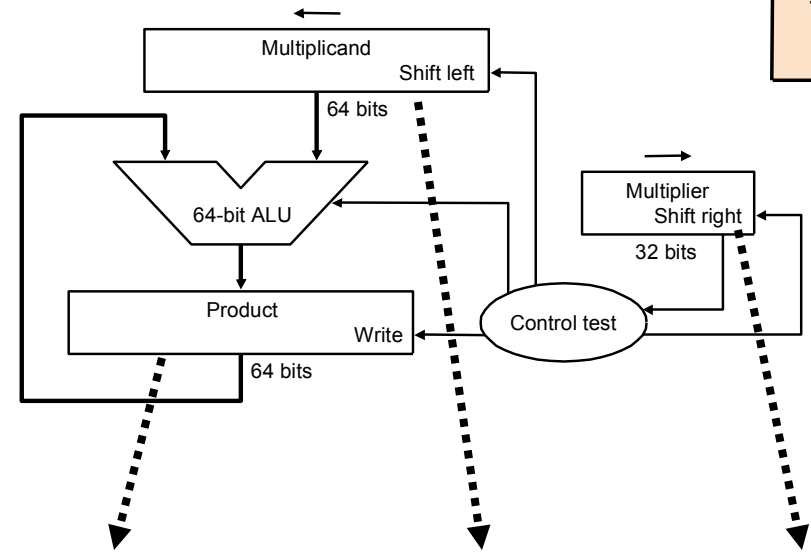
Grade-School Version

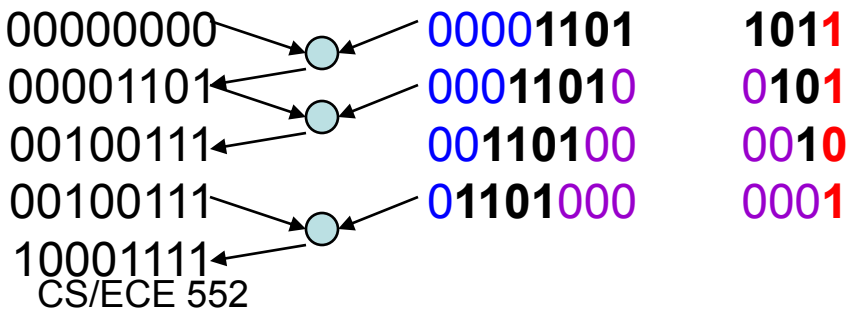


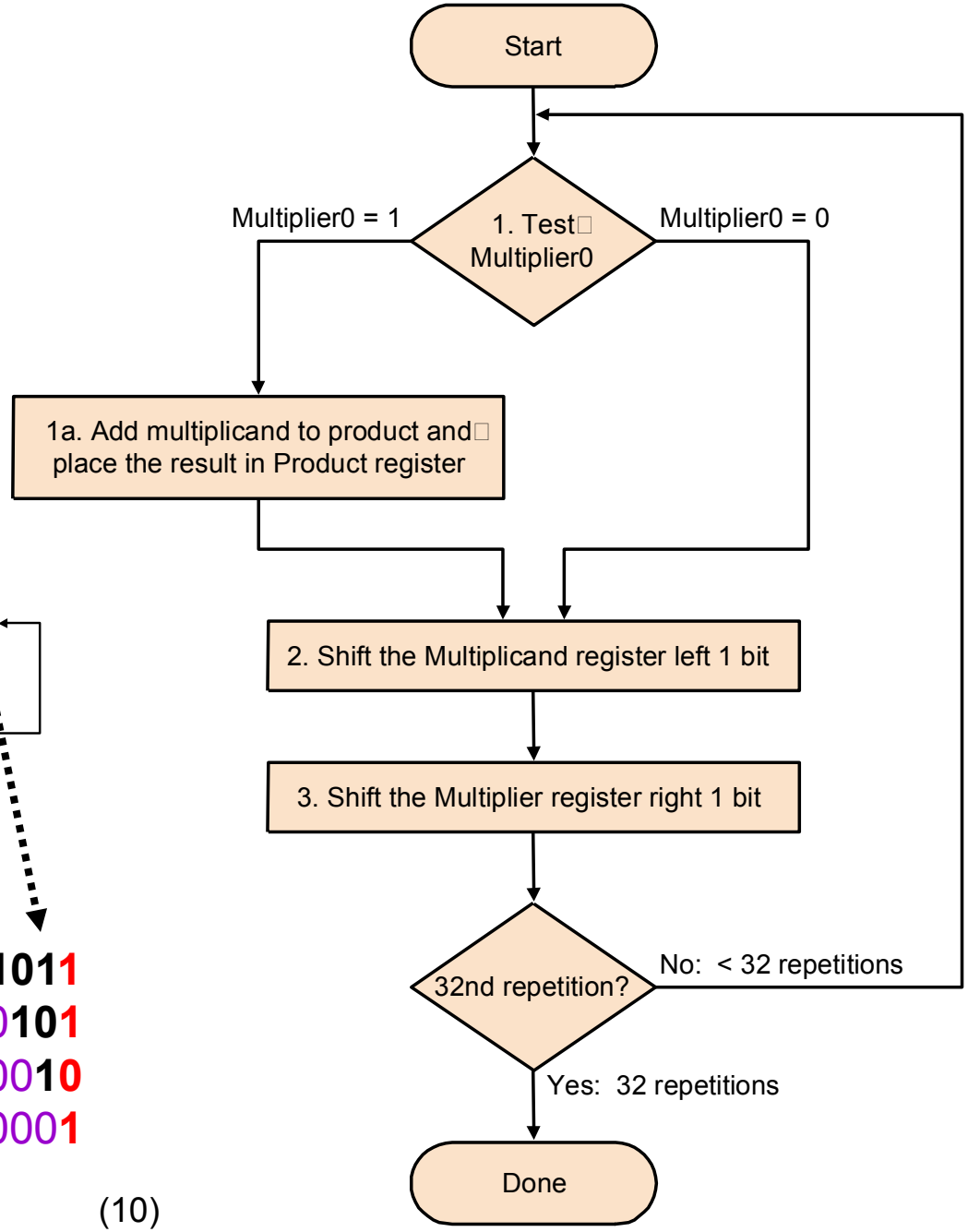
- Naïve implementation
  - 2x 64-bit registers, 1x 32 bit register
  - 1x 64-bit ALU

Flow chart

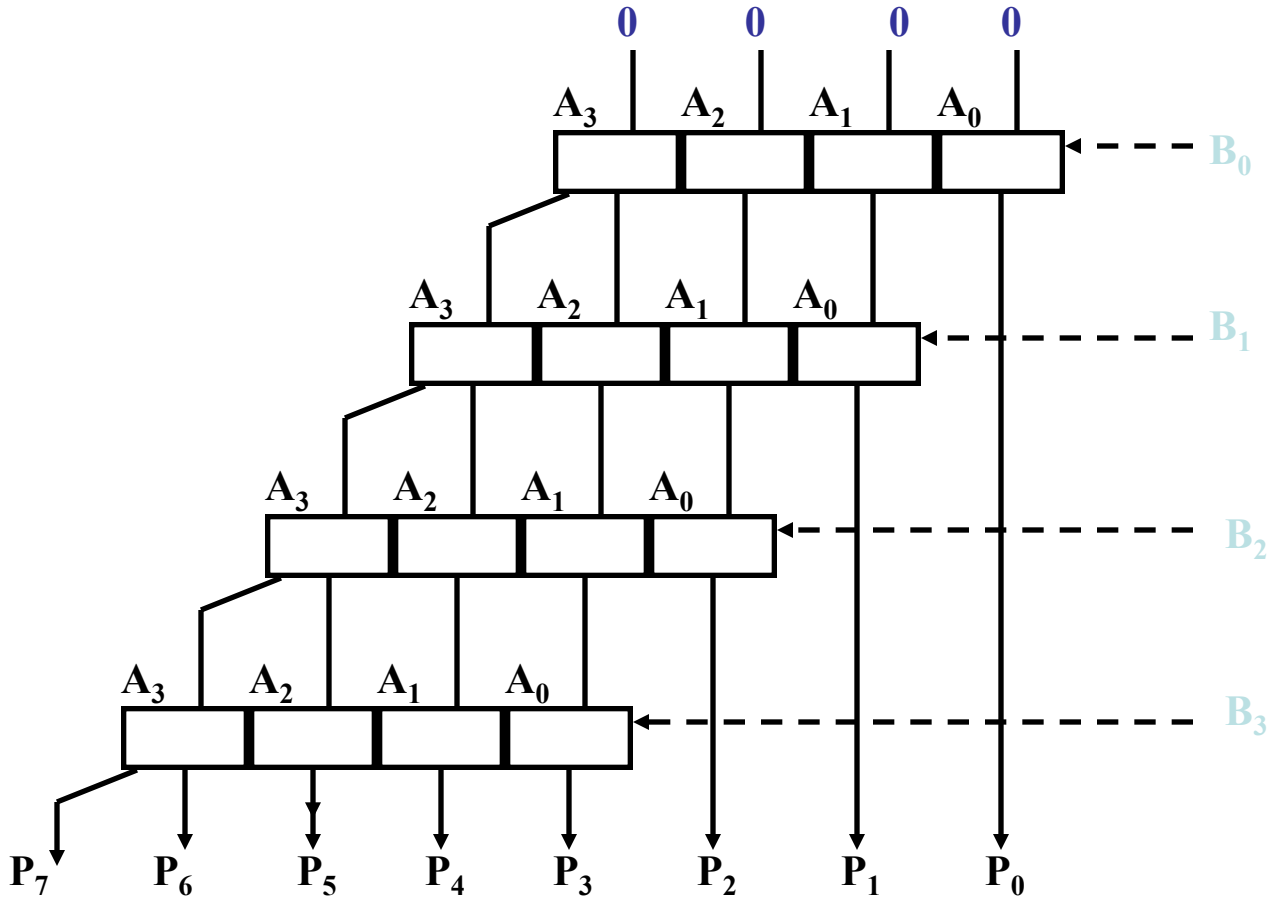
- Repeated shifts and adds
  - E.g. 13*11 = 143





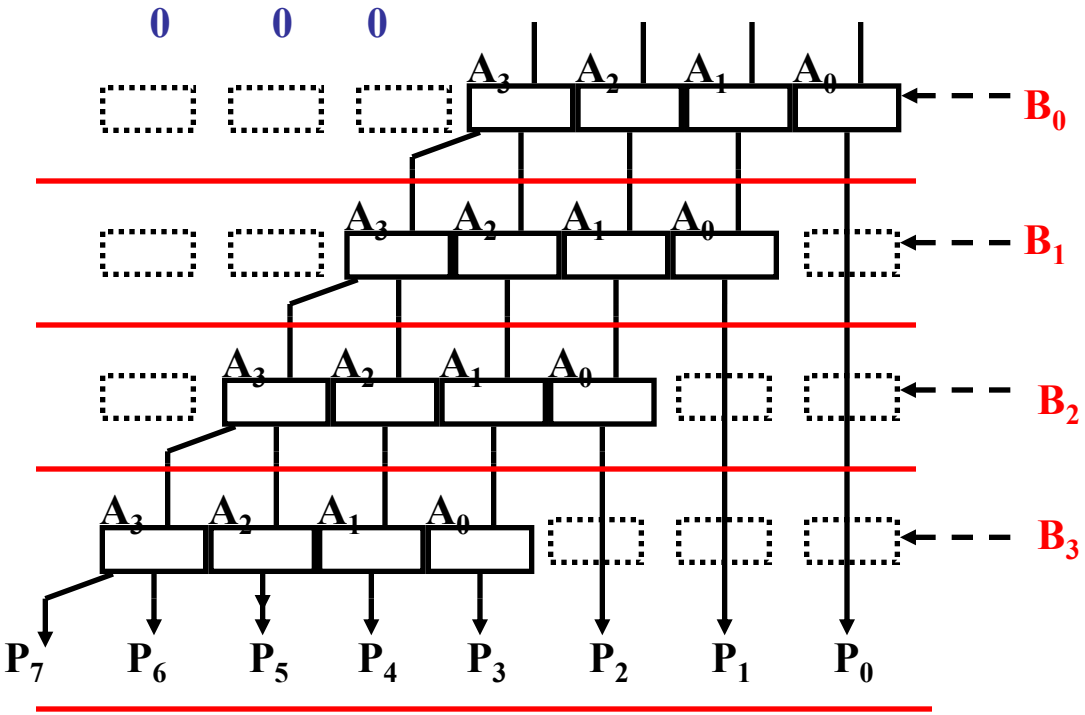


How does it work?



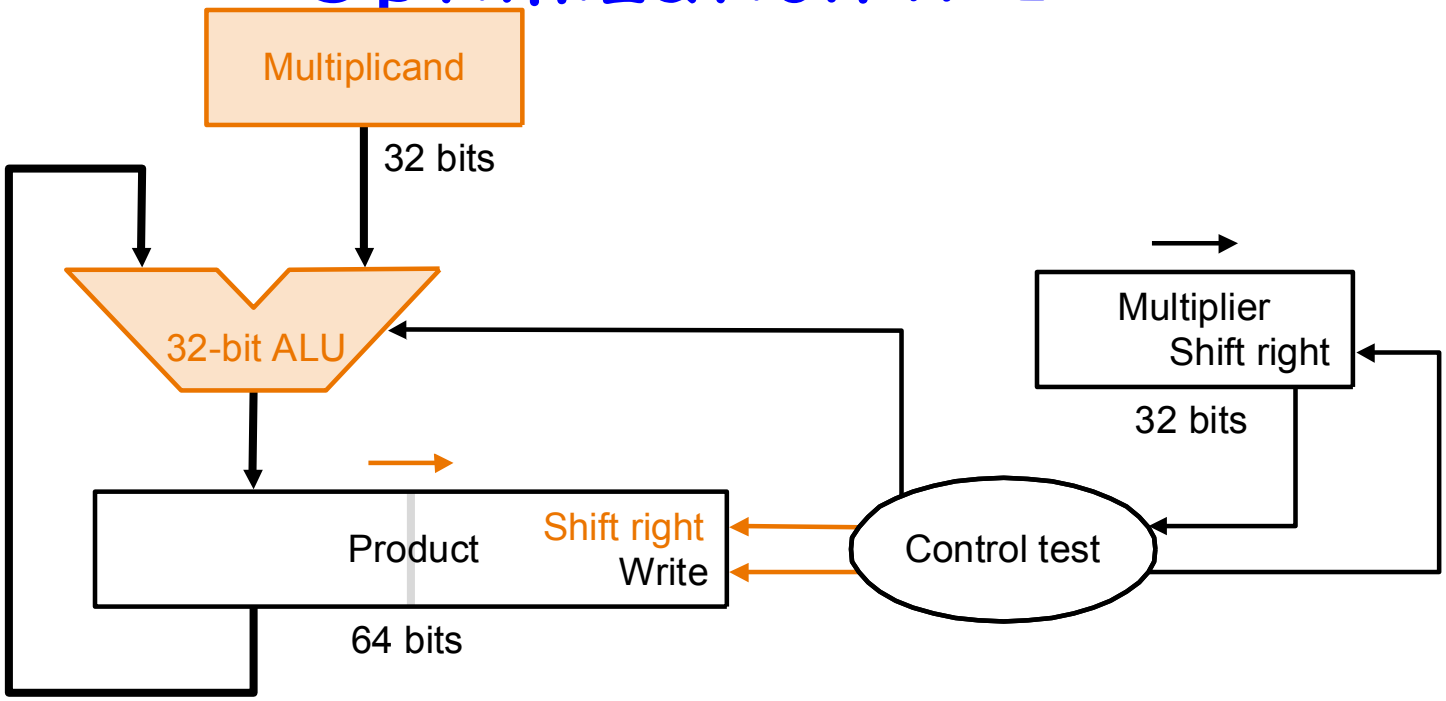
- Faithful reproduction of grade-school algorithm

Version #1



- Additional zeroes (padding)

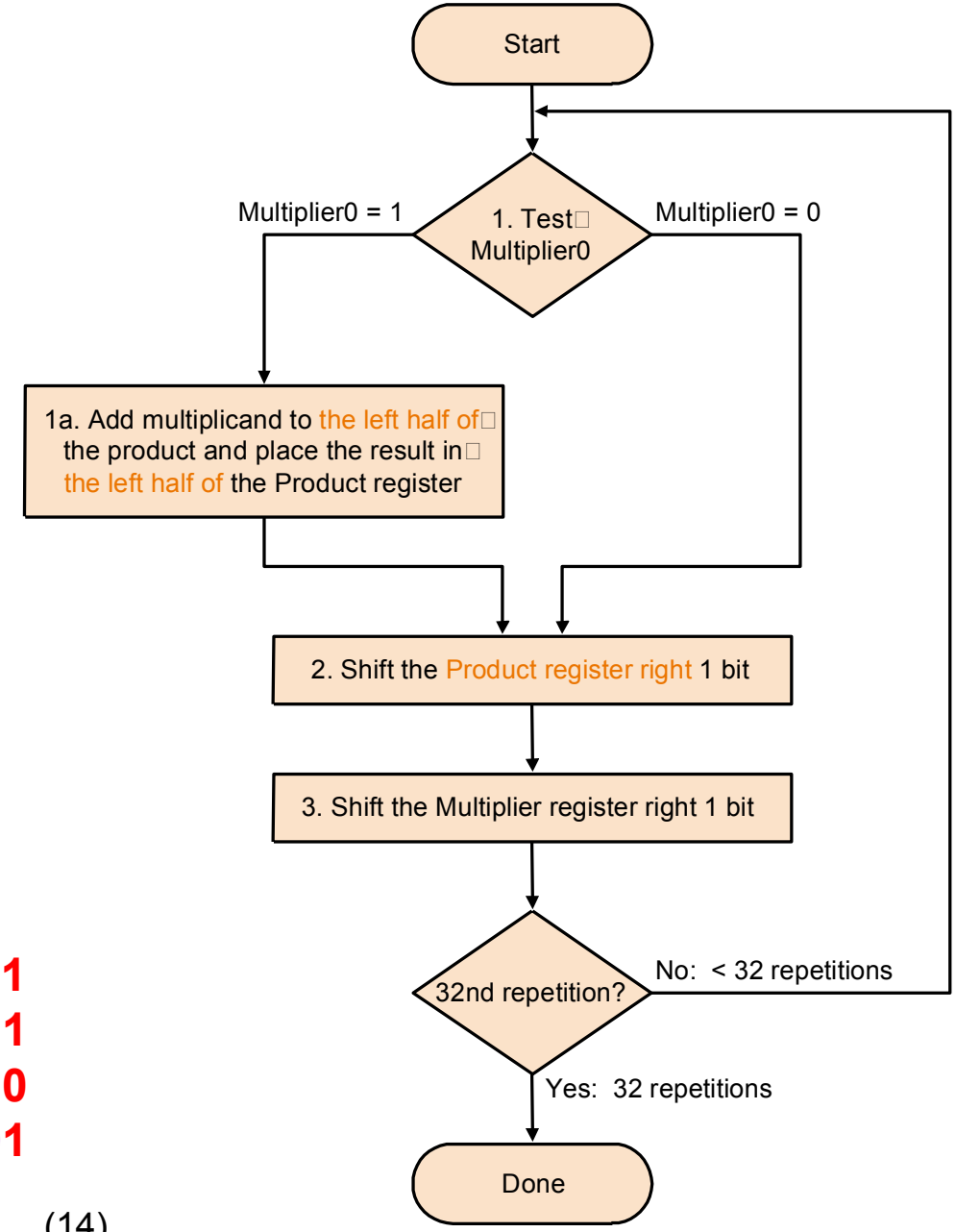
Optimization # 1

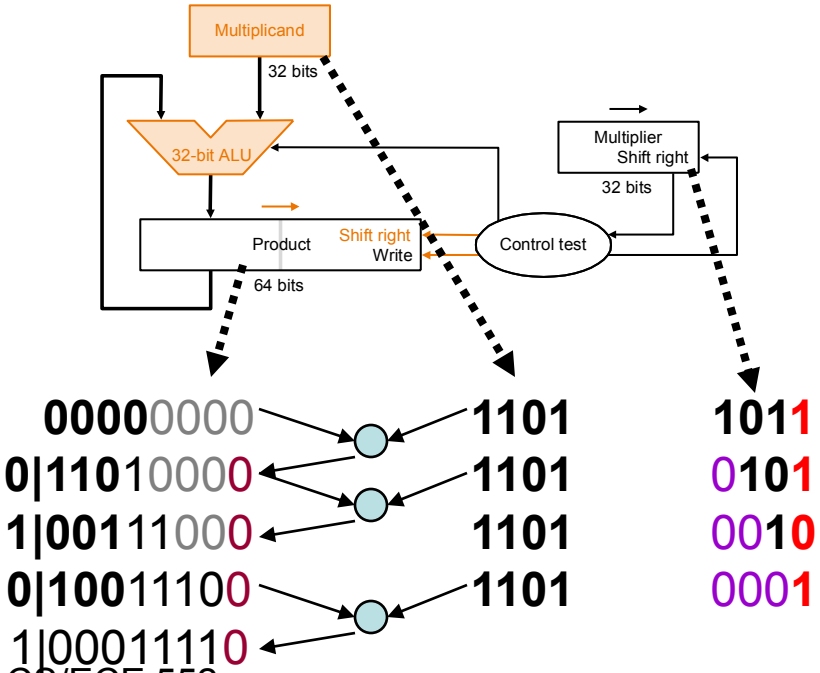


- Insight: LSB bits are frozen after each iteration
- Reduced 64-bit ALU to 32-bit ALU

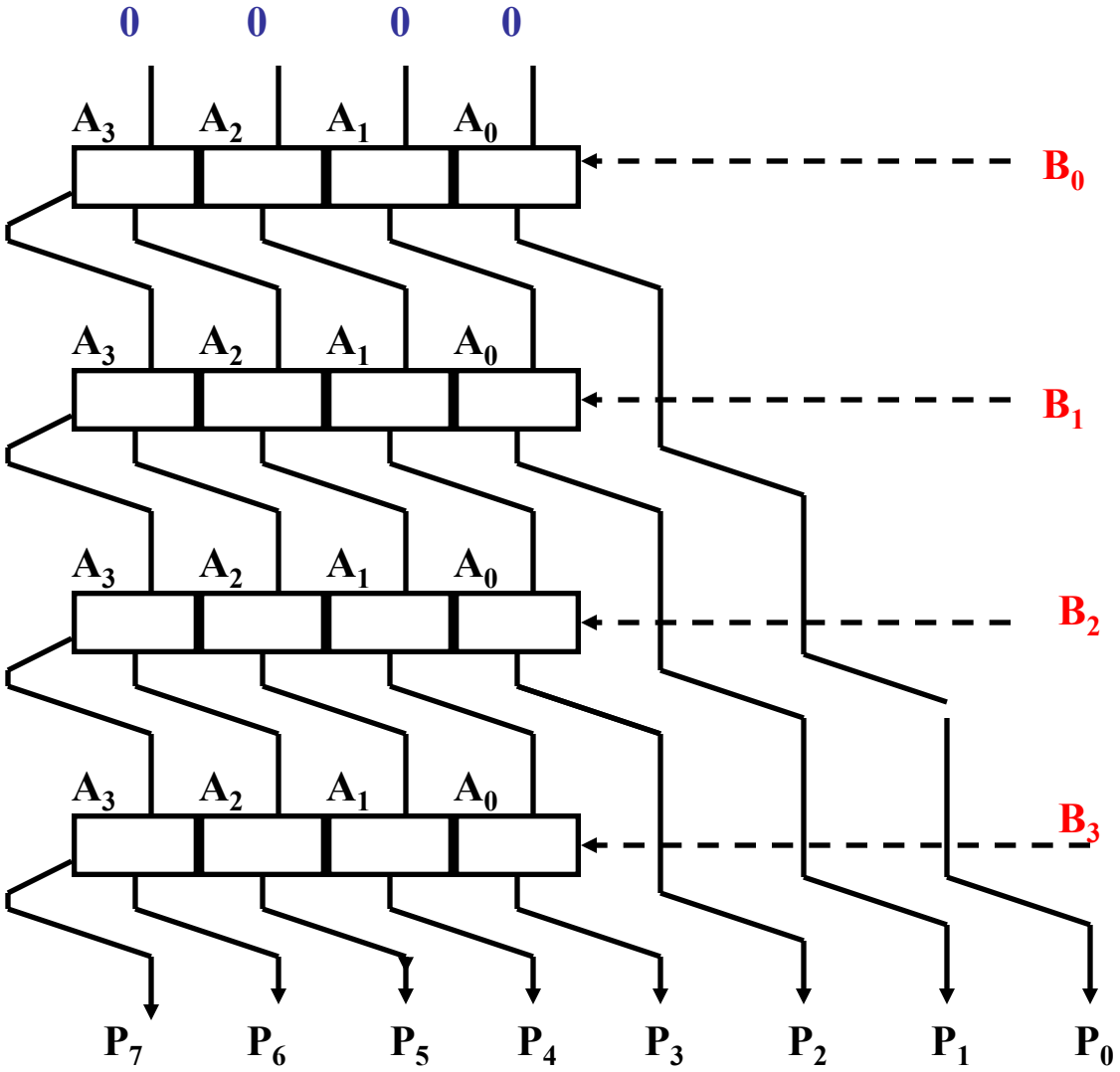
Flow chart

- Product register shifted right after LSB frozen
- 64-bit product computed with 32-bit adder



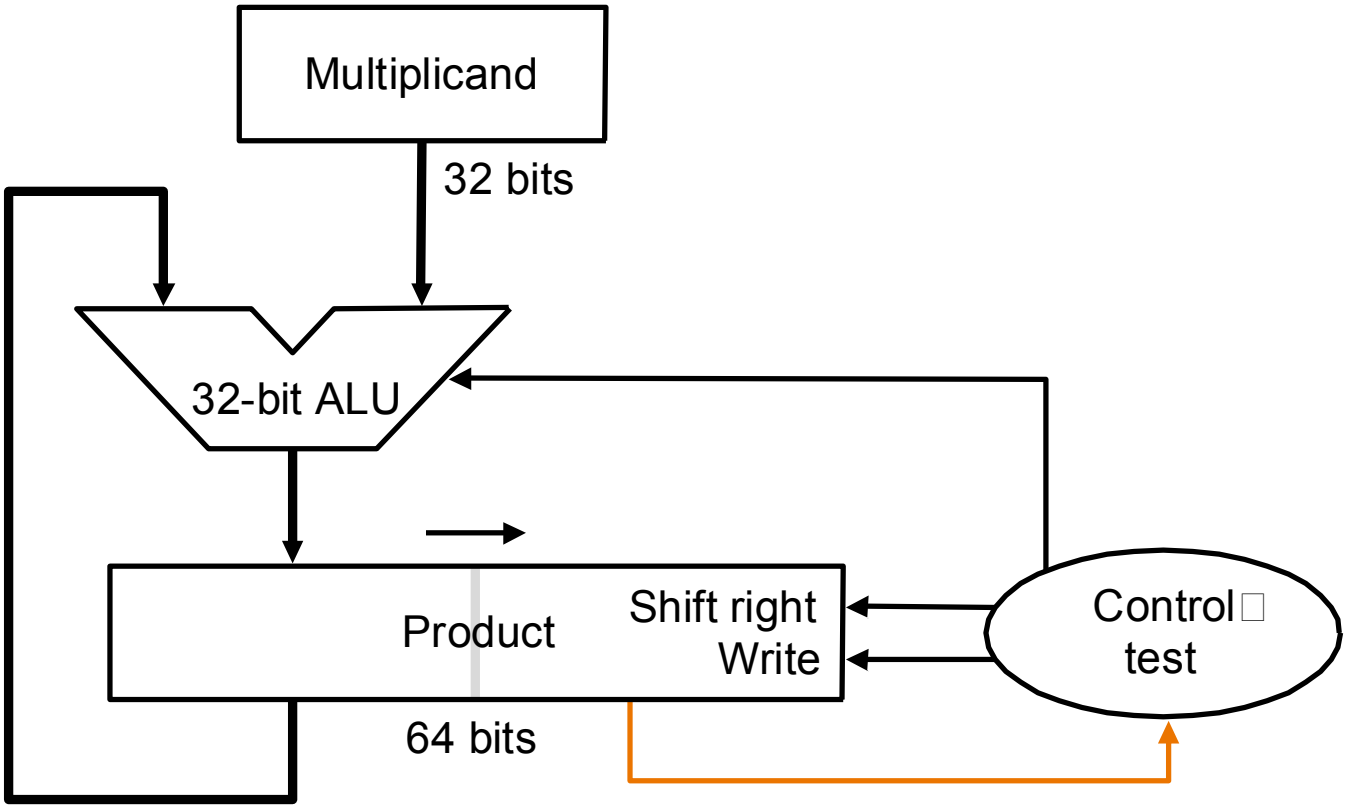


Flow of computation



- Shift product right

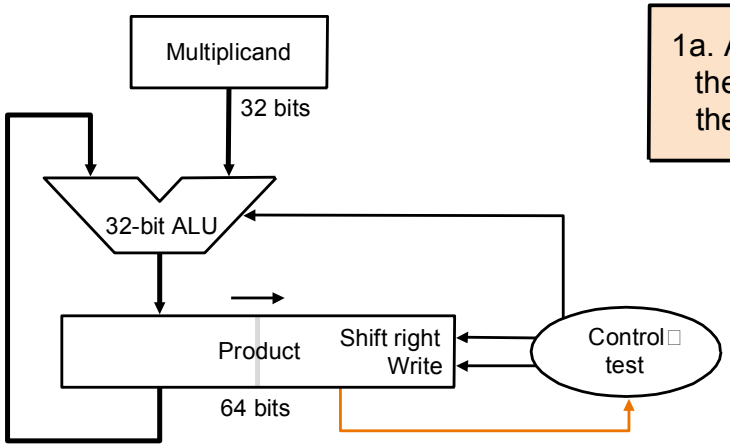
Optimization # 2

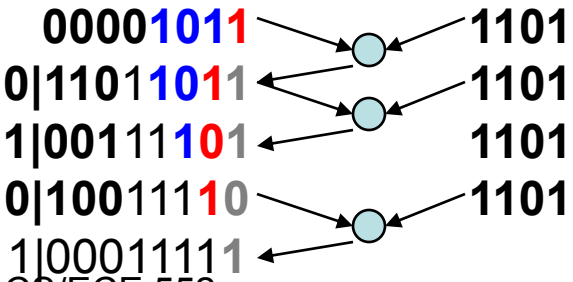


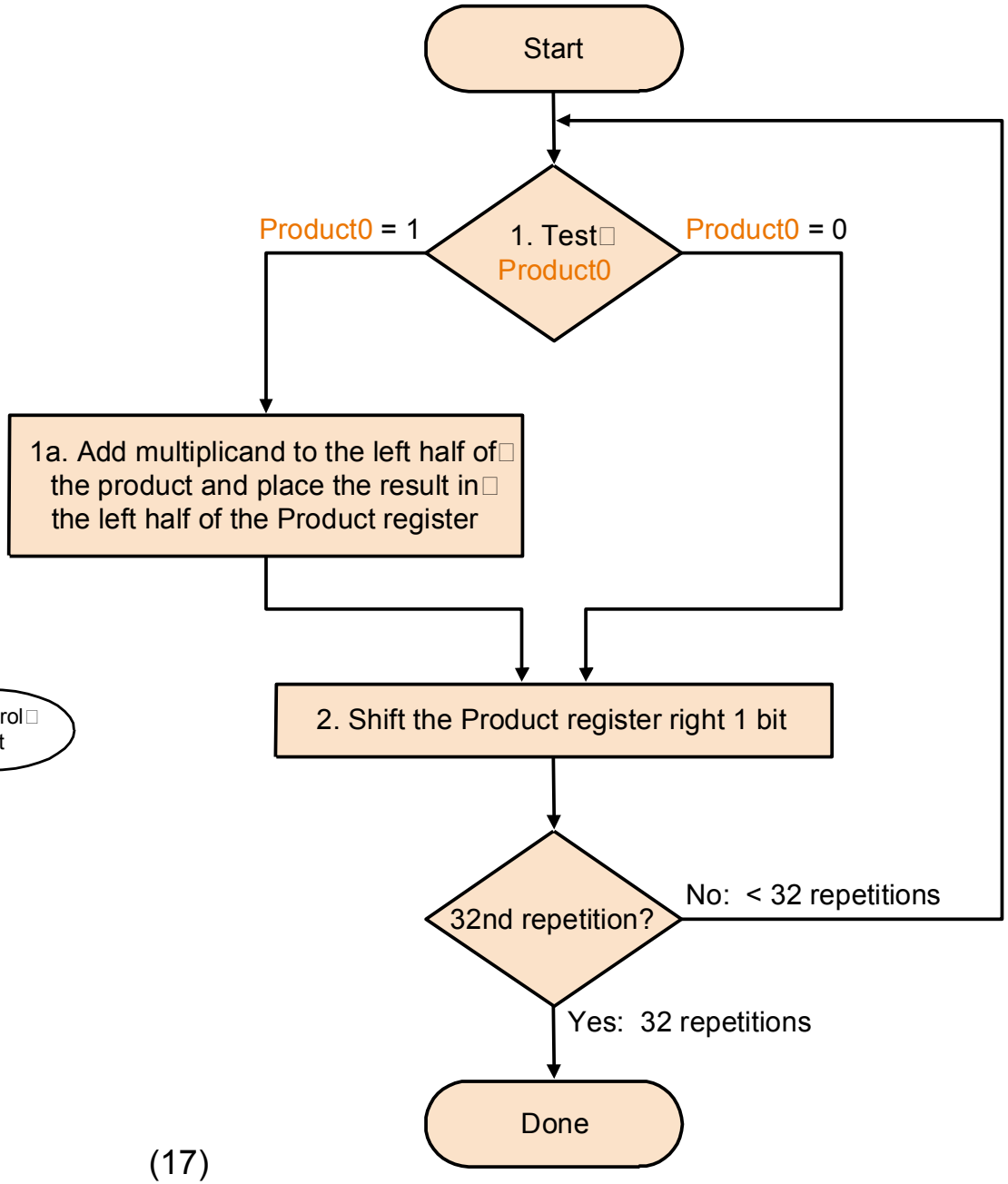
- Insight: bits of multiplier are consumed
  - 1x 32-bit register
  - 1x 64-bit register
  - 32-bit ALU

Flow chart

- Product register manipulates both multiplier and product







Signed Multiplication

- { <+,+>, <-,-> } : +
- { <+,->, <-,+> } : -

- If multiplier negative,
  - Multiplier = - Multiplier
  - Negative ++
- If multiplicand negative,
  - Multiplicand = -Multiplicand
  - Negative ++
- Product = Multiplicand * Multiplier
- If (Negative) sign = '-' , else sign = '+'

Booth's Algorithm

- Signed/Unsigned integer multiplication
  - Multiple bits at a time
  - Like CLA for addition
    - sophisticated
- Intuition : exploit run-lengths

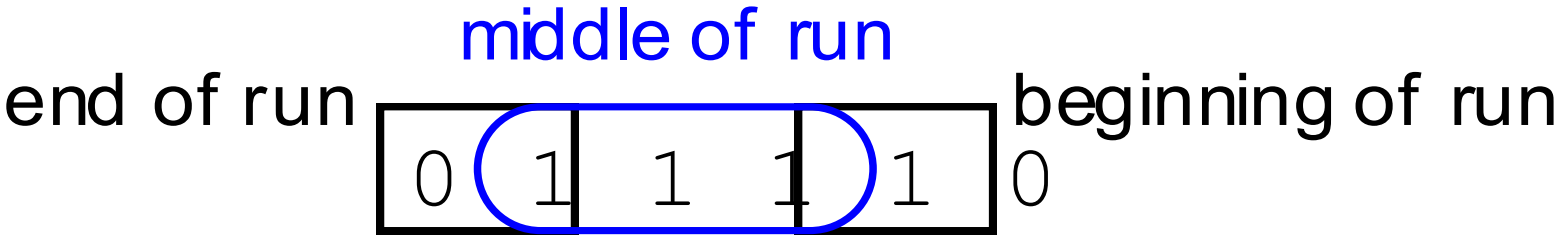
Booth's Algorithm Intuition

- How would you compute the decimal product:
  - 5345 * 999 ?
  - 5345 * (1000 – 1)
  - 5345000 – 5345 = 5339655
- What about
  - 5345 * 9990
  - 5345 * (10000 – 10)
  - 53450000 – 53450 = 53396550
- Now, what about
  - 5345 * 9900990
  - 5345 * (1000000 – 100000 + 1000 – 10)
  - 52915500000 + 5291550 = 52920791550
- Works only when multiplier has only '9's in the decimal system

Application to Binary

- 9 = 10-1 (10 is base of decimal system)
- 1 = 2-1 (2 is the base of binary system)
- For example
  - 0010 * 0011
  - 0010 * (0100 - 0001)

Run of '1's identification



Current bit	Bit to right	Explanation	Example	Action
1	0	Start of run of '1's	11001110	sub
1	1	Middle of run of '1's	11001110	nop
0	1	End of run of '1's	11001110	add
0	0	Middle of run of '0's	11001110	nop

Works for Signed numbers

- Consider 2's complement number 10110
  - Normal 2's complement view
    - 10110 = -16 + 0 + 4 + 2 + 0 = -10
  - Booth encoding view
    - 10110 = -16 + 8 - 0 - 2 + 0 = -10
    - 10110 :: $1\bar{1}0\bar{1}0$ in Booth encoding

      - 1011**0**(**0**)
      - 101**10**(0)
      - 10**11**0(0)
      - 1**01**10(0)
      - **10**110(0)

Booth's Algorithm

- Example: -4*6
  - 4 bit 2's complement 1100 * 0110
  - Extend multiplicand width
    - -4 = 1111 1100
  - Booth encoding of Multiplier
    - 6 = 10$\bar{1}$0 (corresponds to +8 -2)

1111 1100	0	0000 0000
1111 1000	$\bar{1}$	0000 1000
1111 0000	0	0000 0000
1110 0000	1	1110 0000
		1110 1000

The Mathematical Basis

- Recall the table
  - Multiplier = $a_{j-1} - a_j$
- 3-bit numbers
  - A (a_2,a_1,a_0)
  - B (b_2,b_1,b_0)
- Compute B x A
  - $B * \{ (a_1 - a_2).2^2 + (a_0 - a_1).2^1 + (0 - a_0).2^0 \}$
  - $B * \{ -a_2.(2^2) + a_1.(2^2-2^1) + a_0.(2^1-2^0) \}$
  - $B * \{ -a_2.(2^2) + a_1.(2^1) + a_0.(2^0) \}$
  - B * A !!

Current bit	Bit to right	Multiplier
1	0	-1
1	1	0
0	1	+1
0	0	0

Summary

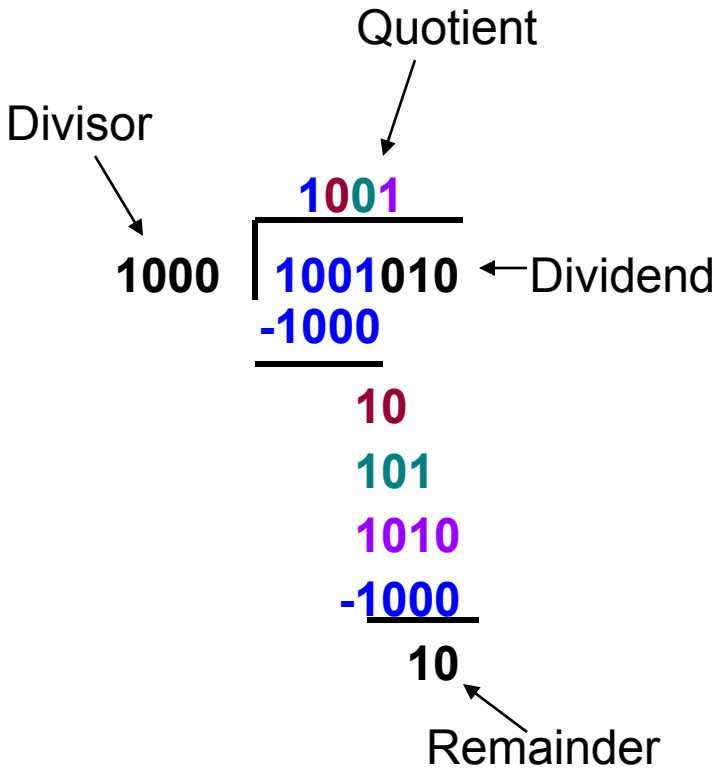
- Integer multiplication:
  - Grade school version with minor optimizations
- Sophisticated optimizations
  - Booth's algorithms
    - More involved
    - Scope for handling multiple bits at a time
  - Remember CLA vs. ripple-carry (grade-school version)

Integer Division

- Remember progressive optimization of multiplication hardware
  - Similar Story
- Start with naïve hardware
  - Faithful implementation of grade-school method (long division)
  - Optimize

Grade School Division

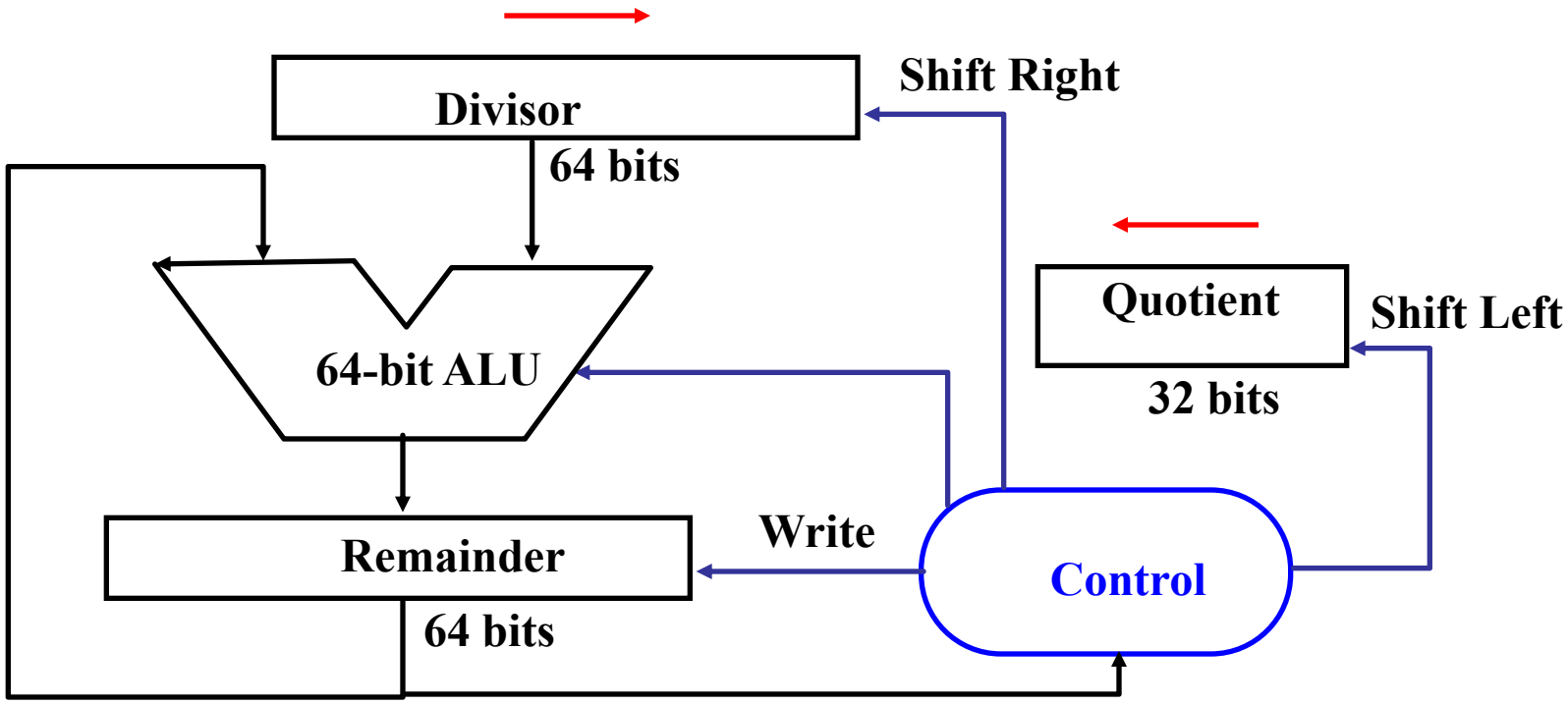
- Consider decimal division
  - Subtract largest possible multiple
  - Shift down one digit
  - "Rinse and repeat"
  - Binary
    - Exactly two choices: 0 or 1



Dividend = Quotient * Divisor + Remainder

|Dividend| = |Quotient| + |Divisor|

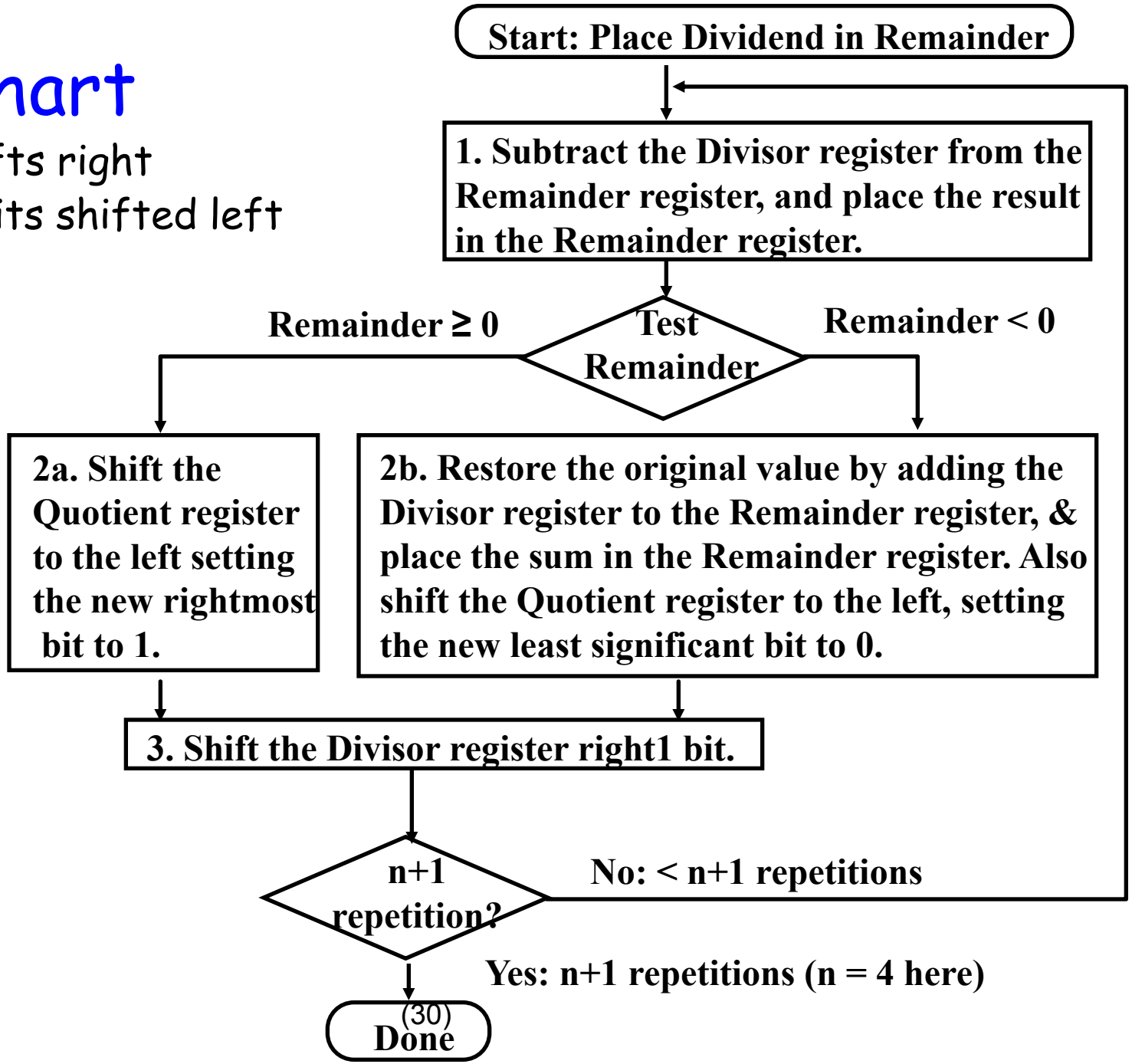
Version #1



- Dividend initially in Remainder reg
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg

Flow Chart

- Divisor shifts right
- Quotient bits shifted left



Example

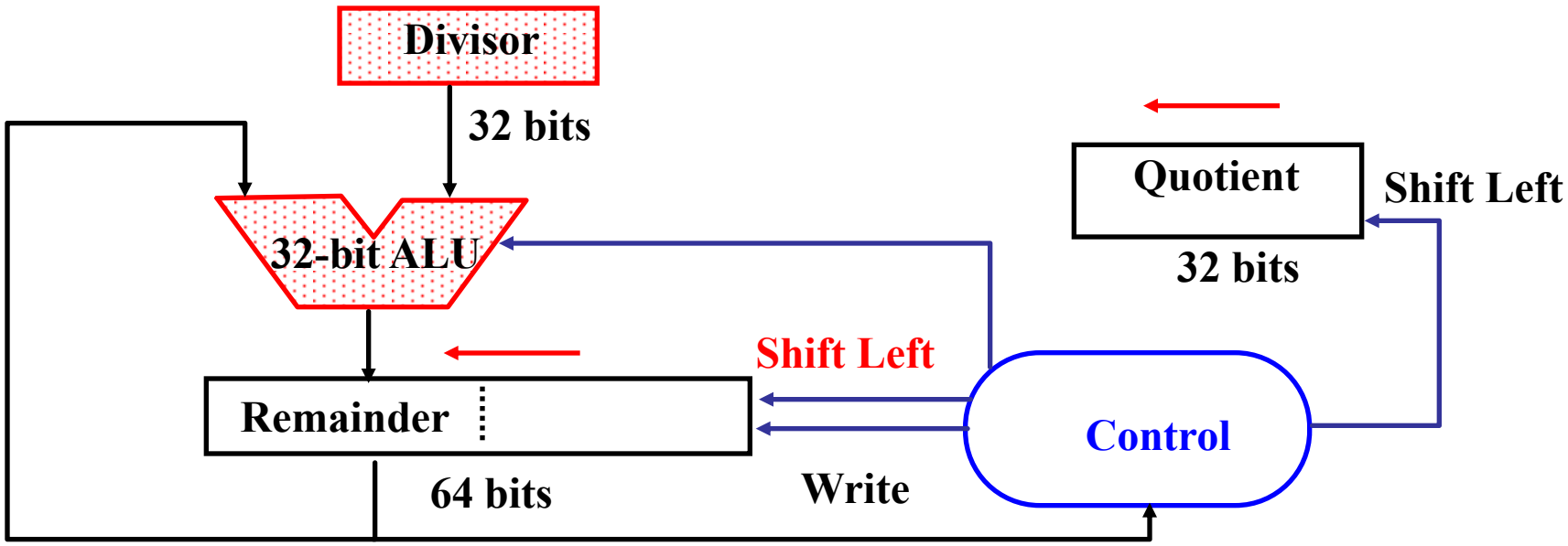
- n+1 steps for n-bit quotient
- 7 / 2 = (3,1)
- Use 8(4) bit 2's complement representation

Remainder	Divisor	Quotient
0000 0111	0010 0000	0000
1110 0111		
0000 0111	0001 0000	000**0**
1111 0111		
0000 0111	0000 1000	00**00**
1111 1111		
0000 0111	0000 0100	0**000**
0000 0011	0000 0010	**0001**
0000 0001	0000 0001	**0011**

Observations on Version #1

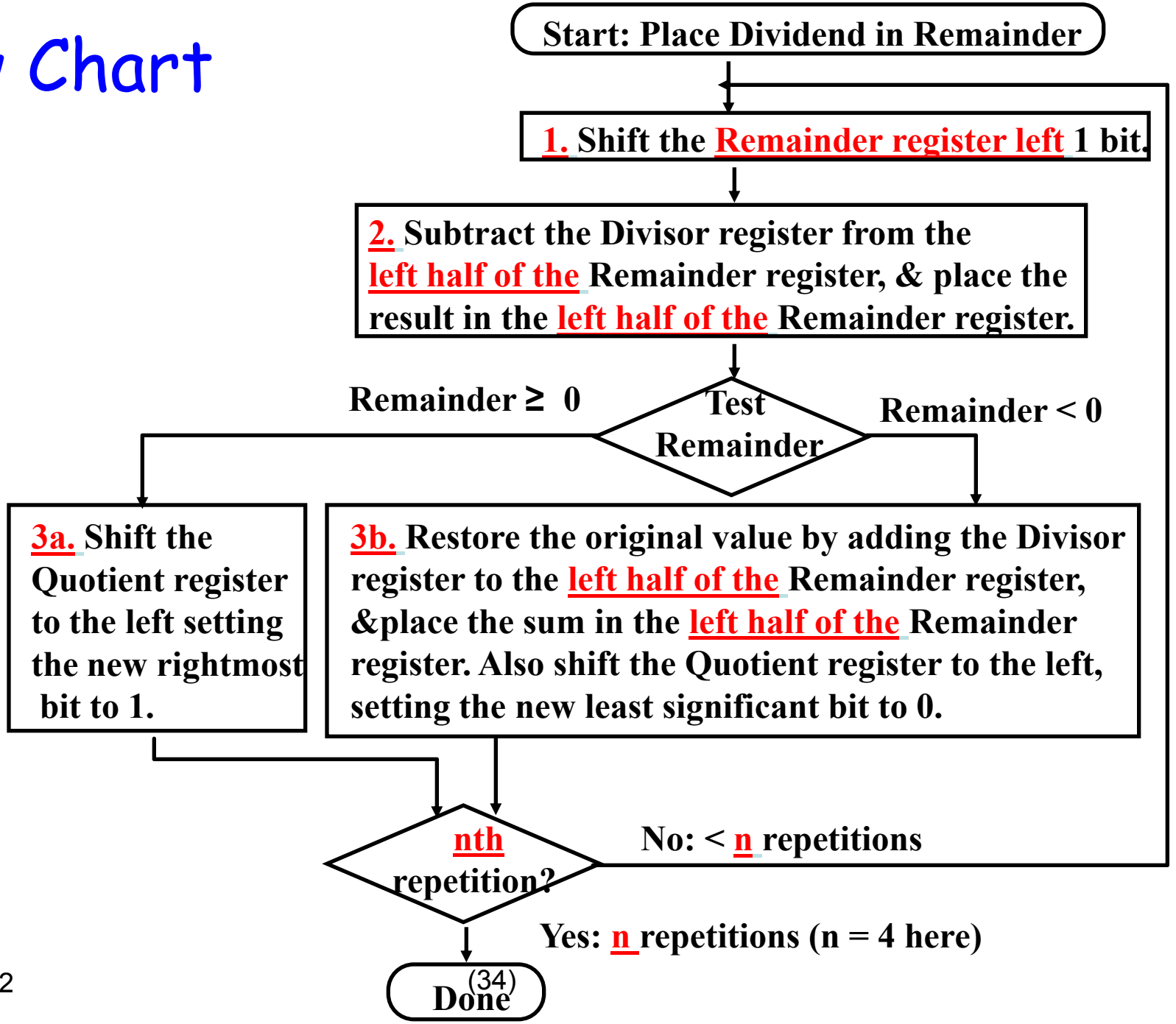
- 1/2 bits in divisor always 0
  - 1/2 of 64-bit adder is wasted
  - 1/2 of divisor is wasted
- Instead of shifting divisor to right, shift remainder to left?
- 1st step cannot produce a 1 in quotient bit (otherwise too big)
  - switch order to shift first and then subtract, can save 1 iteration

Version #2



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

Flow Chart


Start: Place Dividend in Remainder
1. Shift the Remainder register left 1 bit.
2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.
Test Remainder
Remainder ≥ 0
Remainder < 0
3a. Shift the Quotient register to the left setting the new rightmost bit to 1.
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, &place the sum in the left half of the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.
nth repetition?
No: < n repetitions
Yes: n repetitions (n = 4 here)
Done

Example

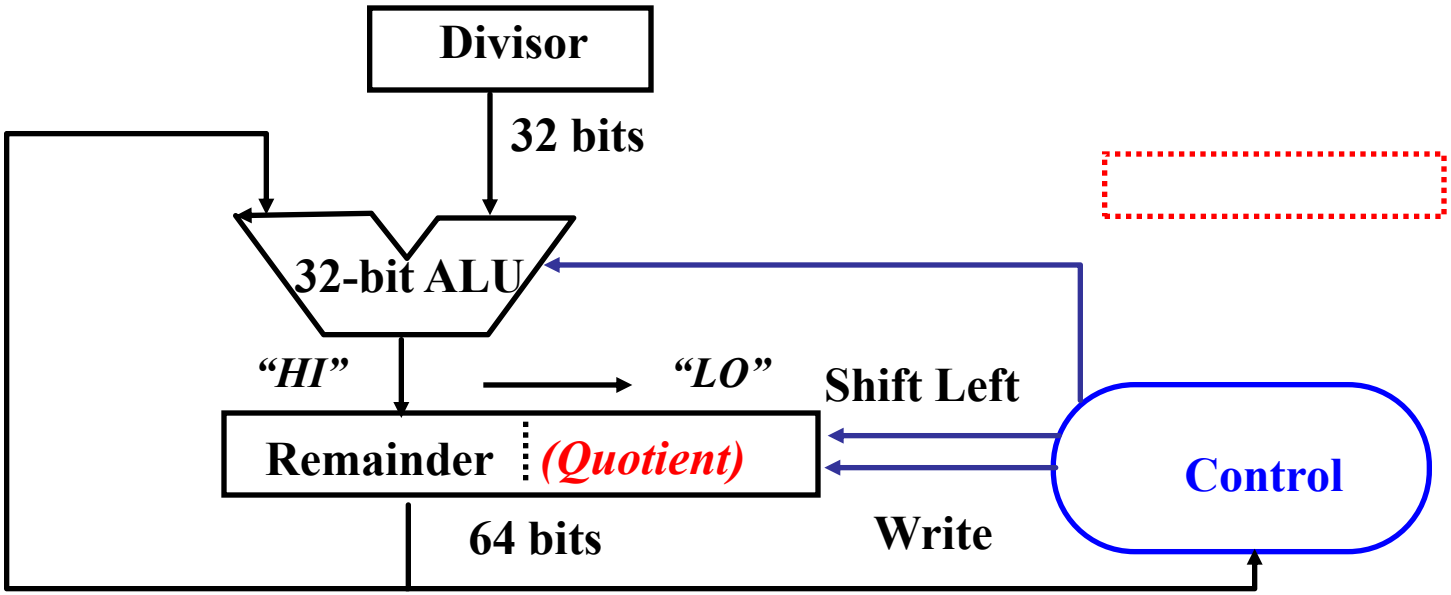
- n-steps only
- Because shift is done first

Remainder	Divisor	Quotient
0000 1110	1110	0000
1110 1110		
0000 1110	0010	000**0**
0001 1100	1110	
1111 1100		
0001 1100	0010	00**00**
0011 1000	1110	
0001 1000		0**001**
0011 0000	1110	
0001 0000		**0011**

Observations on Version #2

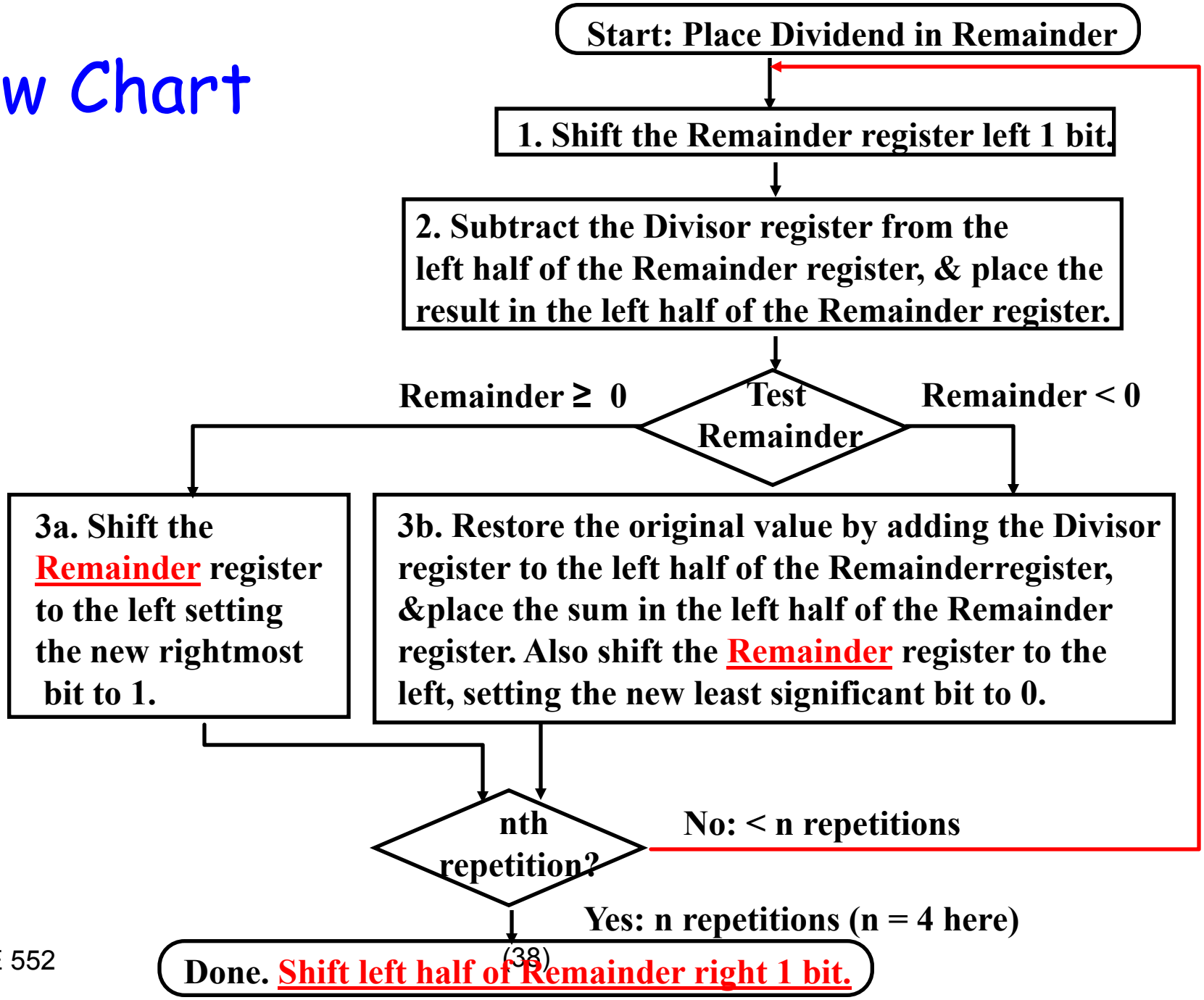
- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register

Version #3



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

Flow Chart


Start: Place Dividend in Remainder
1. Shift the Remainder register left 1 bit.
2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.
Test Remainder
Remainder ≥ 0
Remainder < 0
3a. Shift the Remainder register to the left setting the new rightmost bit to 1.
3b. Restore the original value by adding the Divisor register to the left half of the Remainderregister, &place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.
nth repetition?
No: < n repetitions
Yes: n repetitions (n = 4 here)
Done. Shift left half of Remainder right 1 bit.

Example

- Remember to shift left half of remainder register

Remainder	Divisor
0000 1110	1110
1110 1110	
0000 1110	0010
0001 1100	1110
1111 1100	
0001 1100	0010
0011 1000	1110
0001 1000	
0011 0001	1110
0001 0001	
0010 0011	
0001 0011	

Non-restoring division

- Restoring Division
  - Subtract largest possible multiple
    - Try '1'
    - If too large (i.e. remainder negative), restore
- Alternative
  - Non-restoring division
  - Introduce -1

Non-Restoring Division

- With +ve divisor
  - Subtract if positive remainder
  - Add if negative remainder
- Vice versa with –ve divisor

	Remainder	Divisor	Quotient
-	0000 0111	0010 0000	00000
+	1110 0111	0001 0000	00001
+	1111 0111	0000 1000	00011
+	1111 1111	0000 0100	00111
-	0000 0011	0000 0010	01111
	0000 0001	0000 0001	11111

Quotient = $\overline{0011}$ => $1 \times 2^1 + 1 \times 2^0$

Quotient = $\overline{11111}$ => $1 \times 2^4 - 1x2^3 - 1x 2^2 - 1x 2^1 + 1x2^0$

Floating Point

- Integer arithmetic till now
- How to represent real numbers:
  - Uncountably infinite
  - Realistically, can operate with limited number of significant digits (precision)
  - Remember exponent separately
- Remember scientific norms
  - Planck's constant 6.626068×10^{-34} m^2 kg/s
  - Avogadro's constant 6.022×10^{23}
  - Normalization:
    - Not 0.6022×10^{24}, Not 60.22×10^{22}
    - MSD in [1,9] range except for 0.0

FP in hardware

- Binary instead of decimal
- MSB = 1 (equivalent to [1,9] in decimal)
  - $(-1)^s \times f \times 2^e$
  - **s: Sign** stored separately
  - **f:** is of the form 1.F
  - **e:** is the (signed integer) exponent
- Optimizations:
  - Base not stored (implicit 2)
  - MSB not stored (always 1)

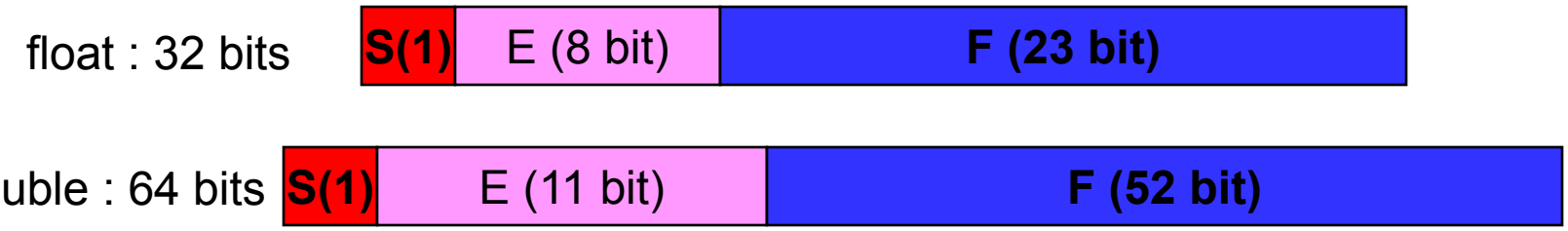
Binary Fractions Recap

- How do you represent $3.125_{(10)}$ in binary?
  - $11.001_{(2)}$
- For integer conversion to binary
  - Repeated division by 2 till quotient goes to zero
- For fraction conversion to binary
  - Repeated multiplication by 2 till fraction goes to zero
- Recurring binary fraction
  - $0.6_{(10)} = 0.1001\ 1001\ 1001\ \ldots._{(2)}$

0.125 * 2 = 0 . 25
0.25 * 2 = 0 . 5
0.5 * 2 = 1 . 0

0.6 * 2 = 1 . 2
0.2 * 2 = 0 . 4
0.4 * 2 = 0 . 8
0.8 * 2 = 1 . 6
0.6 * 2 = 1 . 2

IEEE 754



- Floating point representation standard
- Floating points stored as a packed triple
  - <S, E, F>
  - S : 1 bit
    - 0 → positive
    - **1** → negative
  - E : Biased representation for signed numbers
    - 8 bit (single precision—`float`) (Bias = +127)
    - 11 bit (double precision—`double`) (Bias = +1023)
  - F : (remember the 1 in 1.F is not stored)
    - 23 bit (single precision—`float`)
    - 52 bit (double precision—`double`)

Biased Exponent

- Why?
  - 2's complement arithmetic was elegant
  - Larger numbers appear to have larger magnitude with biased representation
    - Negative numbers appear large in 2's complement
  - Comparing two FP numbers is like comparing two integers with biased exponent

Examples

- 3.125 in double precision
  - 11.001
  - $(-1)^0 \times 1.1001 \times 2^1$
  - Exponent : 1 with bias of 1023 = 1024
  - 0100 0000 0000 1001 0000 0000 …
  - 0x4009 0000 0000 0000
- 0.6 in single precision
  - 0.100110011001…
  - $(-1)^0 \times 1.00110011001\ldots \times 2^{-1}$
  - Exponent : -1 with bias of 127 = 126
  - 0011 1111 0001 1001 1001…
  - 0x3F19 9999

Exceptions

- Specified in great detail in the document IEEE 754-1985
  - 20 pages

S	E	F	Number
0	0	0	0
0	Max	0	+inf
1	Max	0	- inf
X	Max	!= 0	NaN
X	0	!= 0	Denorm

Floating Point Addition

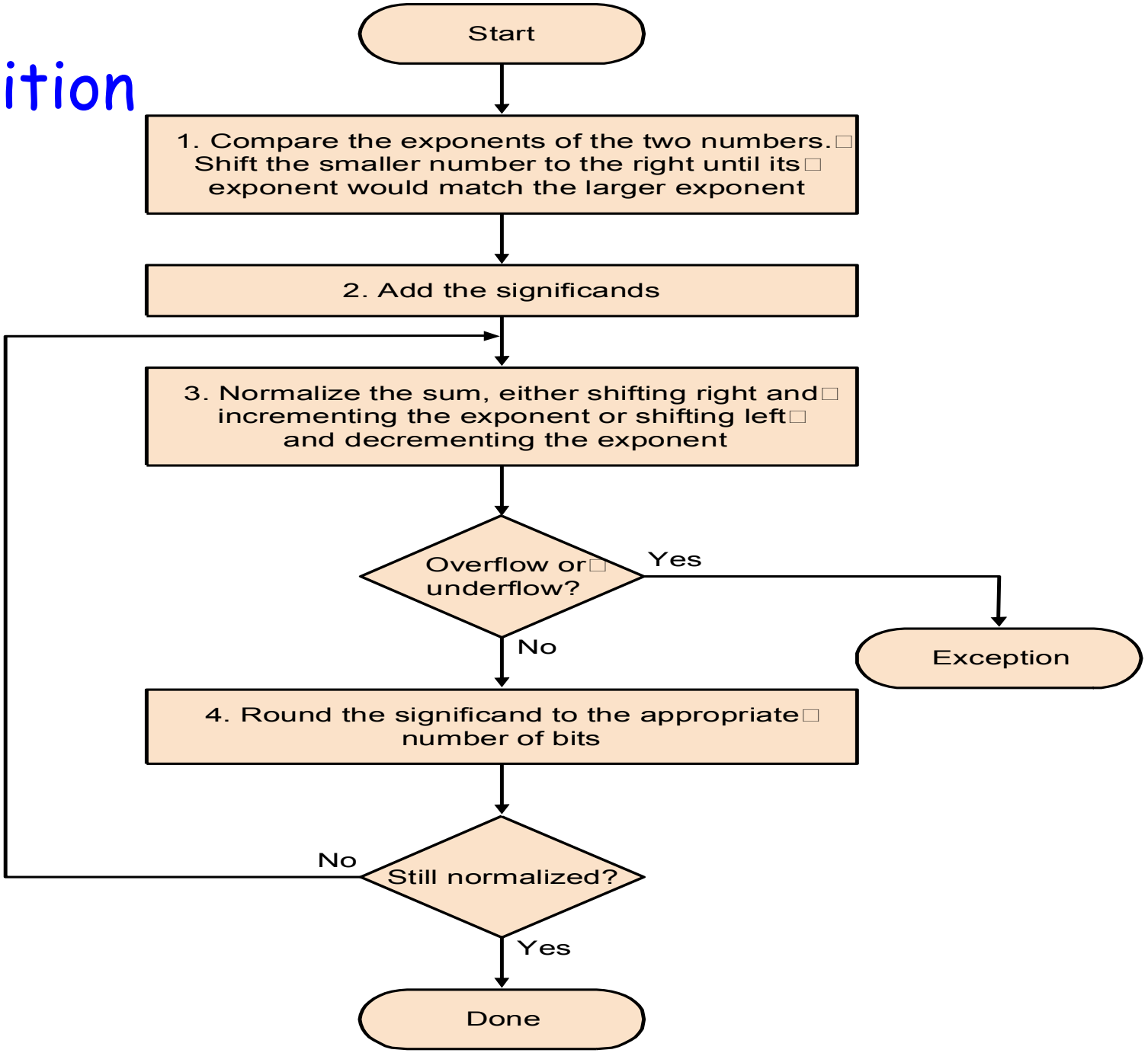
- FP addition
  - Align decimal point
  - Add
  - Normalize
- May involve rounding
  - The LSB may be shifted right beyond the limited precision

$$\begin{array}{r} 9.997 \times 10^{2} \\ +\ 4.6371 \times 10^{-1} \\ \hline \end{array}$$

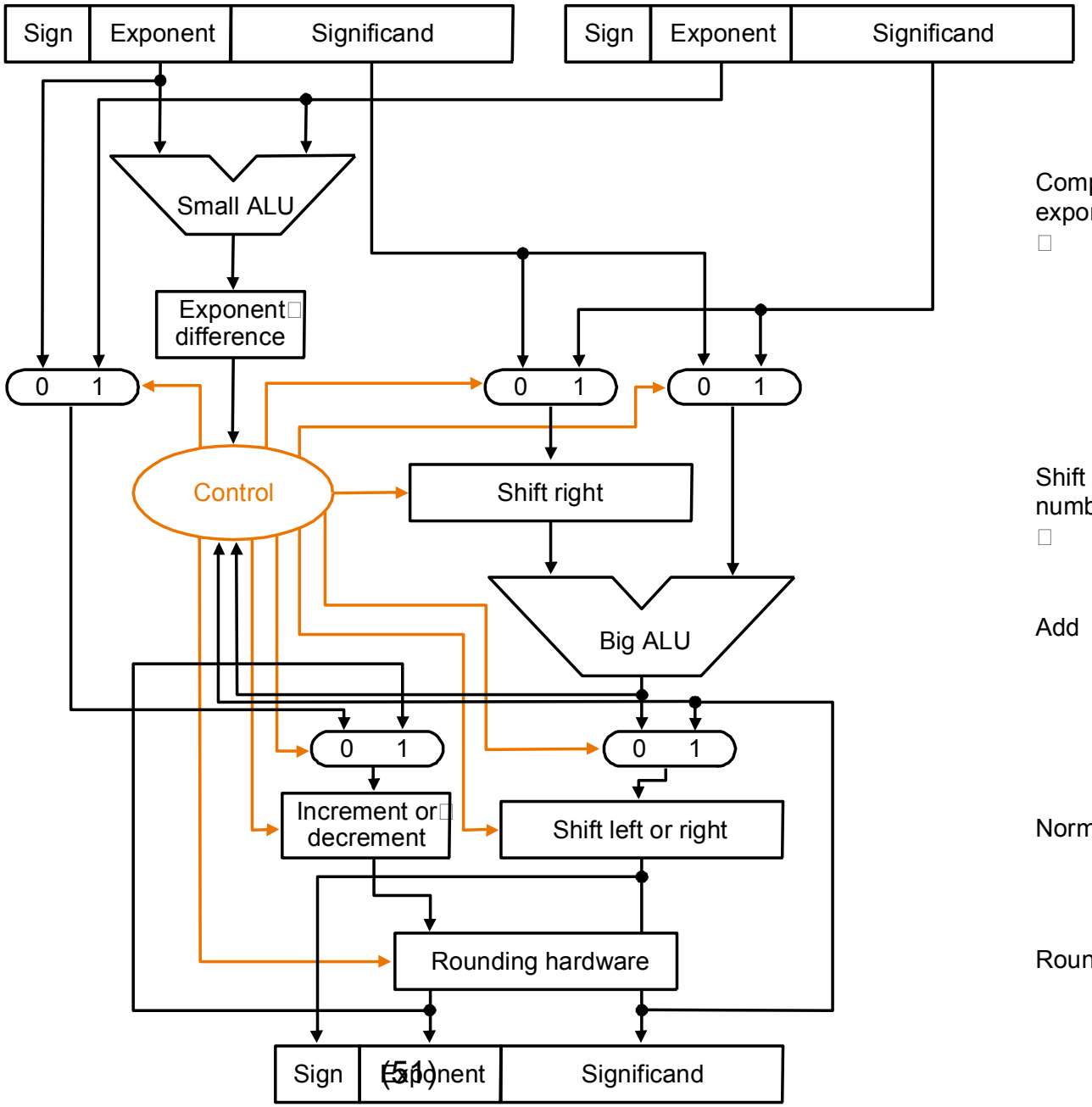
$$\begin{array}{r} 9.997 \times 10^{2} \\ +\ 0.0046371 \times 10^{2} \\ \hline 10.0016371 \times 10^{2} \end{array}$$

$$1.00016371 \times 10^{3}$$

FP Addition


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

- FP Addition Hardware



FP Subtraction

- At most one shift for normalization after add
  - Applies when two operands are of same sign
  - Also the case when subtracting two operands of different signs
- Subtraction
  - Subtract two numbers of similar sign
  - Add two numbers of dissimilar sign

Example

- $1.00010 \times 10^{13} - 1.00001 \times 10^{13}$
- 0.00001×10^{13}
  - Not normalized
- 1.0×10^{8}
  - Normalize involves multiple digit shifts

FP multiplication

- Example
  - $A = 3.0 \times 10^{1}$
  - $B = 5.0 \times 10^{2}$
  - $A \times B = (3.0 \times 5.0) \times 10^{(1+2)} = 15.0 \times 10^{3}$
  - $A \times B = 1.5 \times 10^{4}$
- Multiply mantissas
- Add exponents (no overflow/underflow)
- Normalize (and round)
- Set sign (easy, exor sign bits)

Adding exponents

- Biased representation
- Adding biased numbers.
- Raw number

$e_1 = 12, e_2 = 13 ; e_* = (e_1+e_2) = 25$

$E_1 = 12 + 127 = 139, E_2 = 13 + 127 = 140$

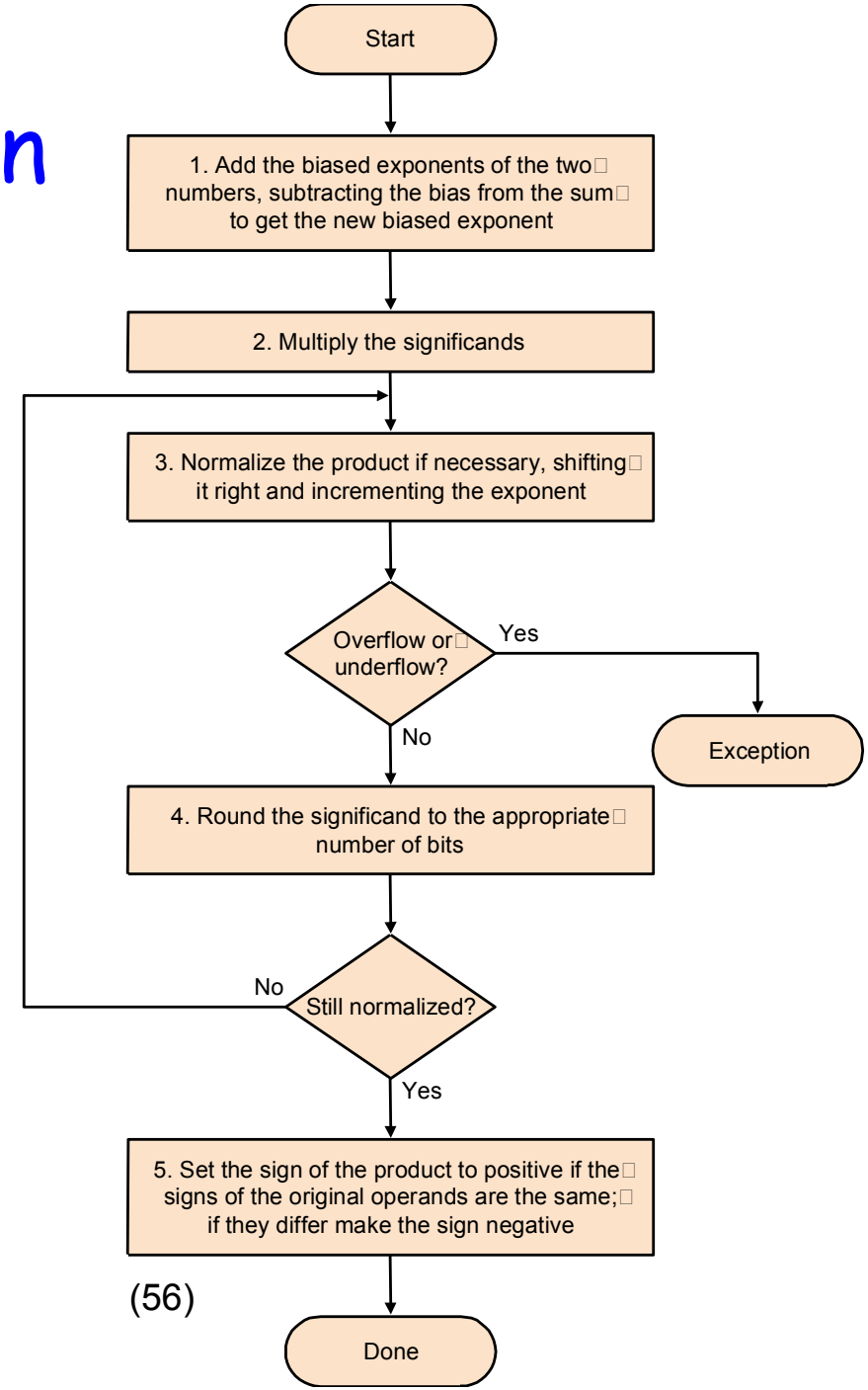
$E_* = e_*+127 = E_1 - 127 + E_2 - 127 + 127$

$E_* = E_1 + E_2 - 127$

$-127 = -(01111111) = (1000\ 0000) + 1$

```
1000 1011
1000 1100
1000 0001
---------
1001 1000
```

FP Multiplication



FP Multiplication

- Mantissa multiplication
  - Two non-negative integers
  - Remember combinational approach
    - Carry save adders in tree (Wallace tree)

FP division

- Exponent computation
  - $E_/ = E_1 - E_2 + 127$

- Raw number

  $e_1 = 12, e_2 = 13$; $e/ = (e_1 - e_2) = -1$

  $E_1 = 12 + 127 = 139, E_2 = 13 + 127 = 140$

  $E_/ = e_/ + 127 = (E_1 - 127) - (E_2 - 127) + 127$

  $E_/ = E_1 - (E_2 - 127)$

  $-E_2 = 1's\ C\ (E_2) + 1$

  $127 = (01111111)$

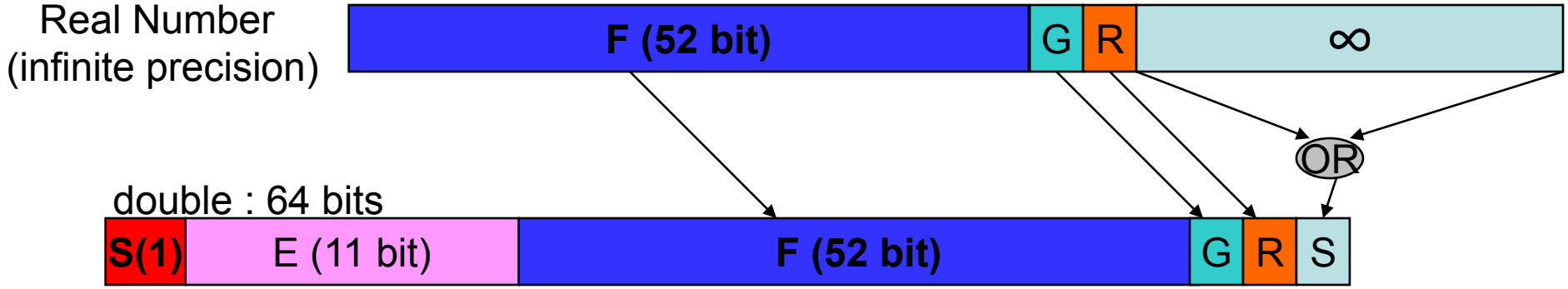
  $E_/ = E_1 + 1'sC\ (E_2) + 1000\ 0000$

```
  1000 1011
  1000 1100
  1000 0000
 ----------

  1000 1011
  0111 0011
  1000 0000
 ----------

  0111 1110
```

Rounding

- Decimal conventions
  - [5+,9] -> up
  - 5 -> round to even (4.5 ~ 4, 5.5 ~ 6)
  - [1,5-] -> down
  - 0 -> unchanged
- Binary rounding
  - xxx.1…1… -> up
  - xxx.10000 -> round to even
  - xxx.0xx1x -> down
  - xxx.00000 -> unchanged

Guard, Round and Sticky bits



- May shift to renormalize
  - Save one bit to right of LSB (Guard bit)
  - This is a significant bit in the normalized representation
  - Round-bit
    - one additional to the right of guard bit
    - Used only for rounding, never becomes significant bit
  - Sticky bit
    - Logical OR of all bits right of round bit

Round	Sticky	Action
1	1	Up
1	0	Even
0	1	Down
0	0	Unchanged

Rounding

- IEEE 754
  - Error bounds
  - No more than $\frac{1}{2}$ "units of the last place" (ULP)
  - Errors accumulate
    - Billions of FP ops in a single application
- Effects of limited precision
  - Commutativity, associativity etc. may no longer apply
  - $A = (3.1415 + 6.022 \times 10^{23}) - 6.022 \times 10^{23}$
  - $B = 3.1415 + (6.022 \times 10^{23} - 6.022 \times 10^{23})$
  - Is A==B?