18-447 Lecture 3: Computer Arithmetic: Multiplication and Division

James C. Hoe
Dept of ECE, CMU
January 26, 2009

Announcements: Handout00 survey due
Lab partner??
Read P&H Ch 3
Read IEEE 754-1985

Handouts: Handout02: Lab1 (download from Blackboard)
Handout03: HW1 (download from Blackboard)
IEEE 754-1985 (download from Blackboard)

Mult and Divide by Powers of 2

- left shift $b_{n-1}b_{n-2}...b_2b_1b_0$ by s positions yields

  $b_{n-1}b_{n-2}...b_2b_1b_0000...00$

  i.e., $$\sum_{i=0}^{n-1} 2^{i+s} b_i = 2^s \sum_{i=0}^{n-1} 2^i b_i$$

  Works for 2's-complement numbers (??)

- What about right shifts?
  - Does right-shift $b_{n-1}b_{n-2}...b_2b_1b_0$ by s positions yield

    $000...00\_b_{n-1}b_{n-2}...b_{s+1}b_s$

    or $111...11\ \_b_{n-1}b_{n-2}...b_{s+1}b_s$

    or $b_{n-1}...b_{n-1}\_b_{n-1}b_{n-2}...b_{s+1}b_s$

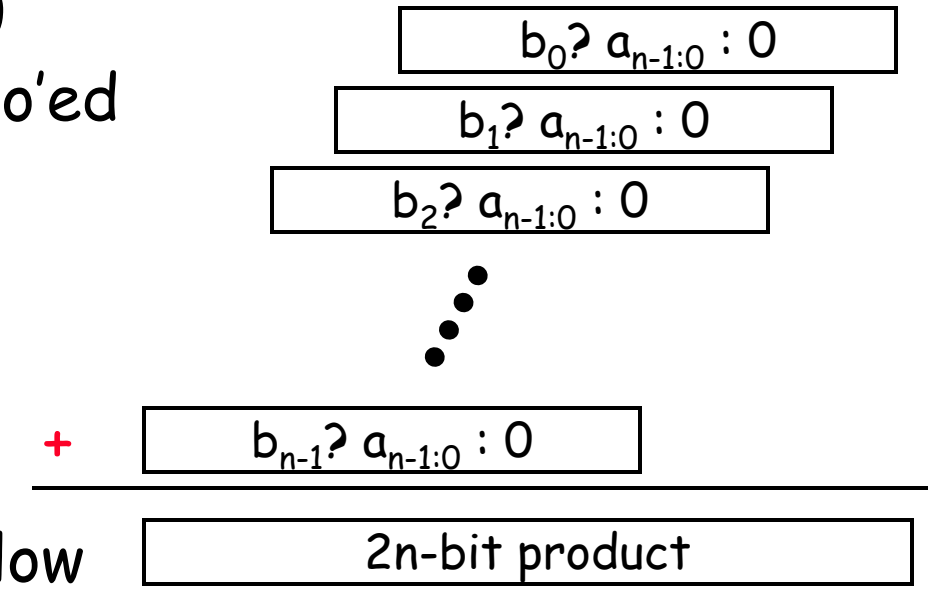
    logical right shift vs arithmetic r. shift

Multiply 2 n-bit numbers: a x b

- Given unsigned numbers $a_{n-1}a_{n-2}...a_2a_1a_0$ and $b_{n-1}b_{n-2}...b_2b_1b_0$

$$a \cdot b = \sum_{j=0}^{n-1} \left(2^j b_j \left(\sum_{i=0}^{n-1} 2^i a_i \right) \right)$$

- Construct a full adder array where the summand ($a_{n-1:0} \times 2^i$) can be conditionally zero'ed according to b_i of $b_{n-1:0}$



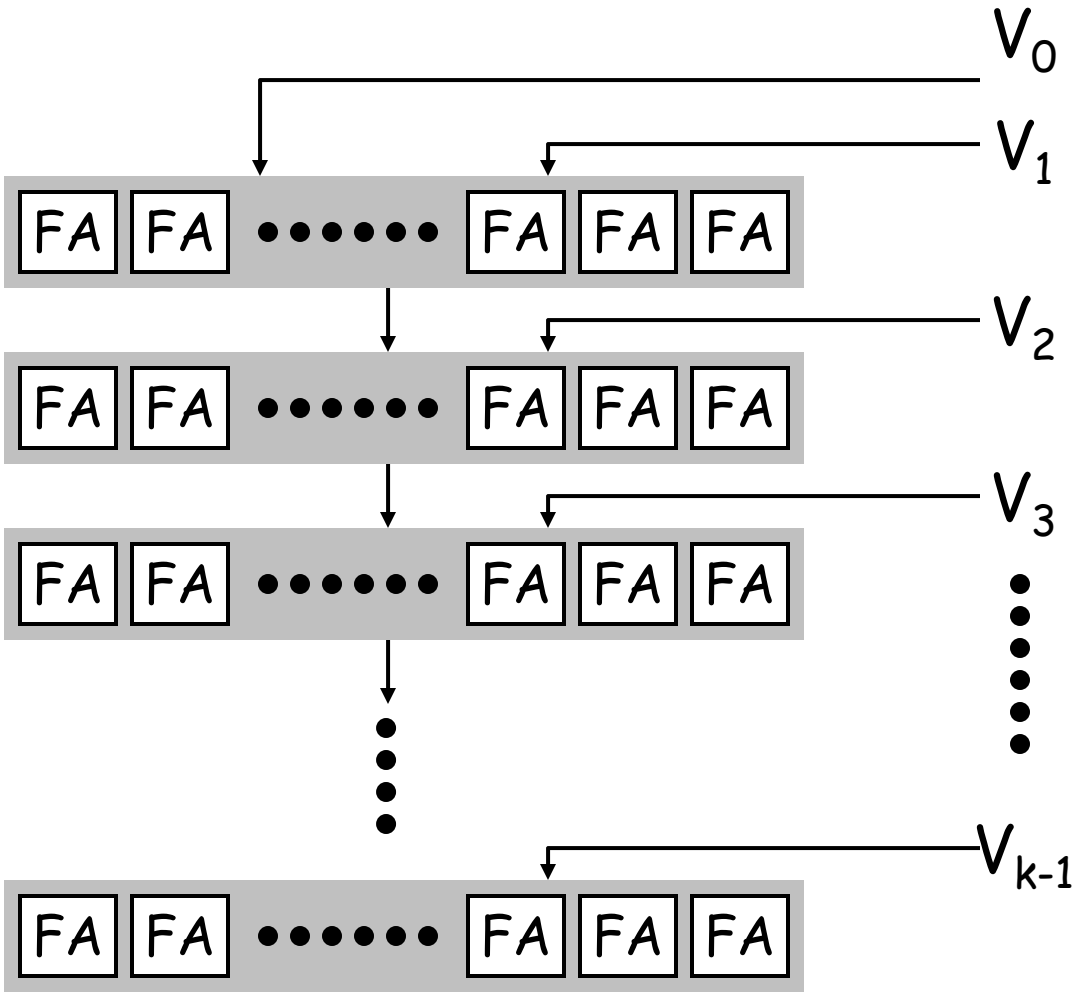
- 2n bits are required to represent all possible products without overflow

2's complement?

J. C. Hoe

Prelude to Multiply: adding many numbers quickly

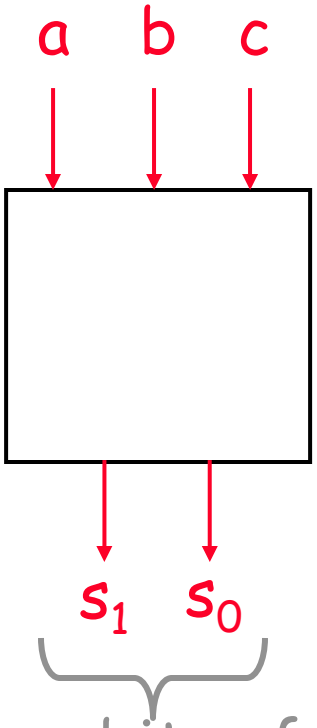
Adding k n-bit numbers



k-1 adders to sum k numbers

Critical Path: O(k + log n) or if n≈k then O(k)

Using "Pop. Count"



How many bits of a, b and c are set?

c	a	b	x	y
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

3-way parity

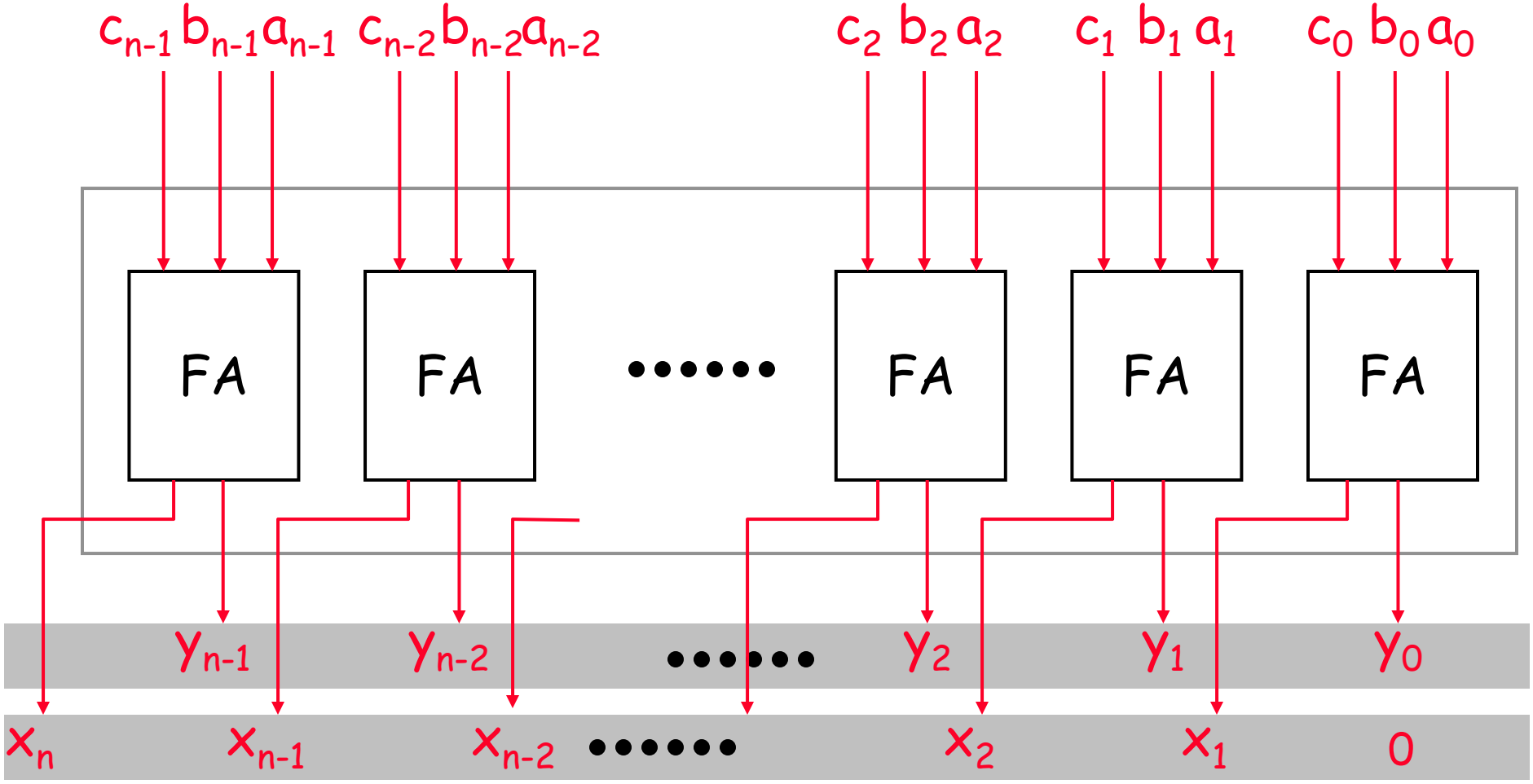
majority

$s_0 = a \oplus b \oplus c$

$s_1 = bc+ac+ab$

Where have you seen this before?

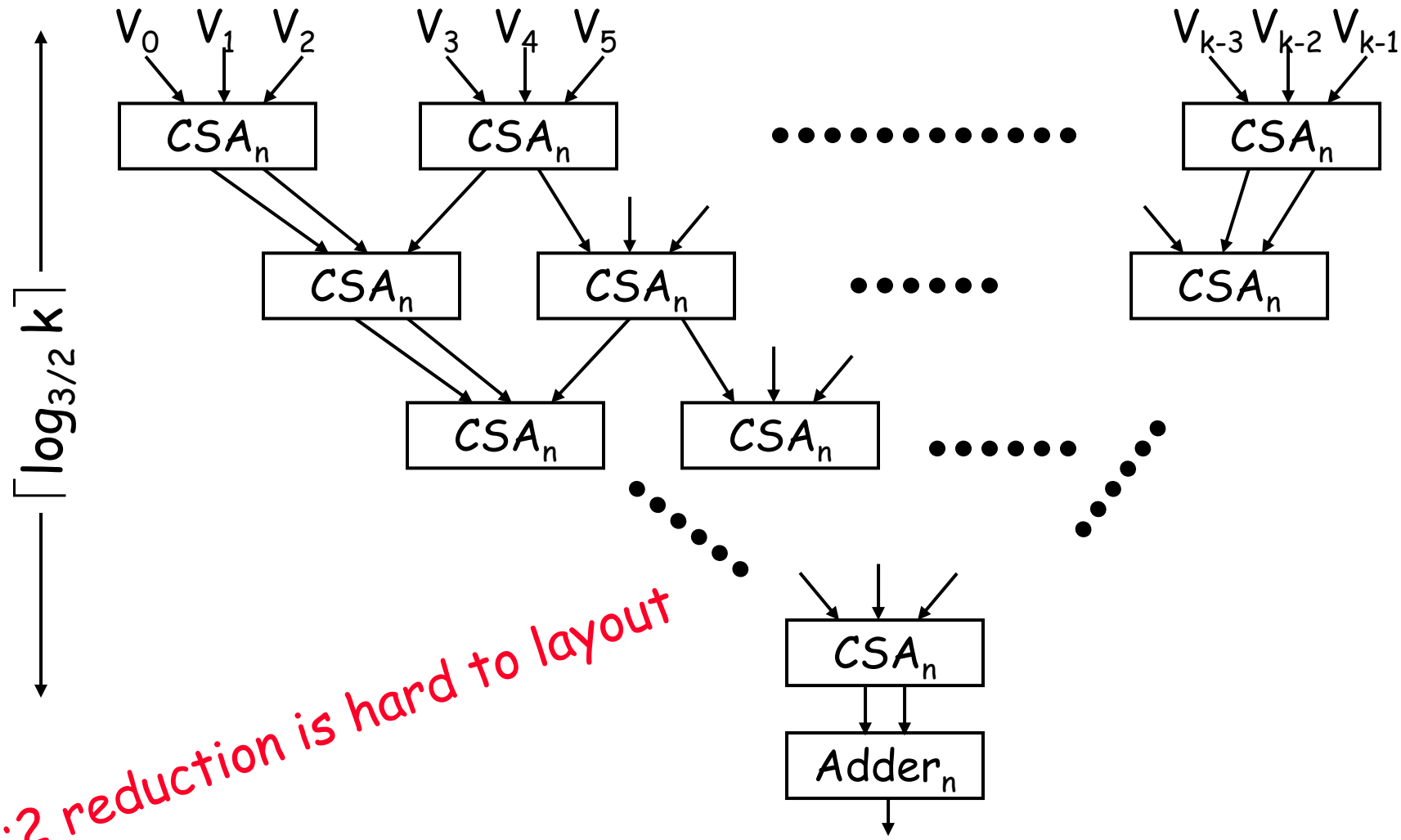
Carry-Save Adder (CSA)



$CSA_{delay}=FA_{delay}$!!

Takes A, B, and C and produce X and Y such that A+B+C=X+Y

3:2 CSA Reduction Tree



Critical Path: O(log k + log n) or if n≈k then O(log k)

SD representations: a detour

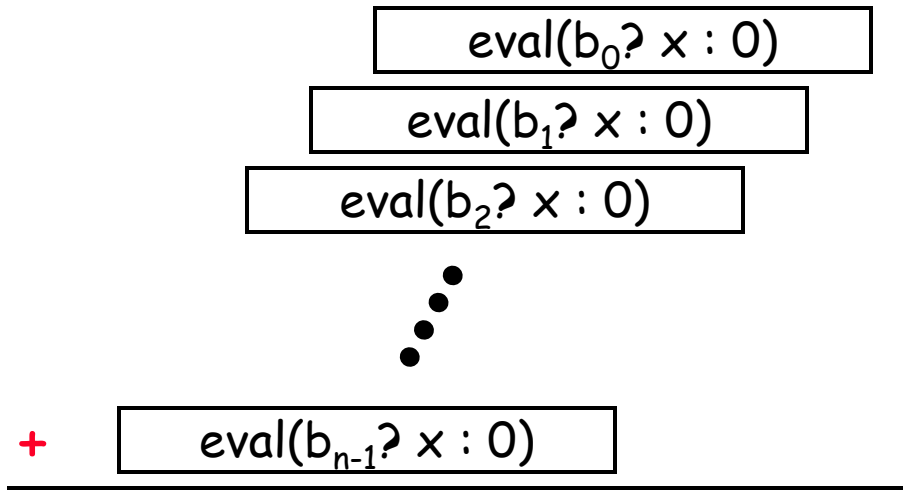
Multiplying by a Constant

- Let $b_{n-1}b_{n-2}\ldots b_2b_1b_0$ represent an unsigned constant c, the product

$$c \cdot x = \sum_{i=0}^{n-1} 2^i b_i x$$

- If c has k non-zero digits, this corresponds to summing k numbers, i.e., k – 1 additions

- Observation:
  1111·x is the same as 10000·x – 1·x, and 1 subtraction is the same cost as 1 addition



Signed-Digit Representation

- Let $d_{n-1}d_{n-2}...d_2d_1d_0$ be the signed-digit (SD) representation of a integer constant c
  - $d_i \in \{ 1, 0, -1 \}$ written as $1, 0, \overline{1}$
  - the product

$$c \cdot x = \sum_{i=0}^{n-1} 2^i d_i x$$

  - If c has k non-zero digits, the product requires k-1 additionsand subtractions
- Given an unsigned binary multiplicative constant, how to minimize the number of additions in the multiplication? For example 10111011

$$\Rightarrow 1100\bar{1}011 \Rightarrow 10\bar{1}00\bar{1}011 \Rightarrow 10\bar{1}00\bar{1}10\bar{1} \Rightarrow 10\bar{1}000\bar{1}0\bar{1}$$

Canonical Signed Digits (CSD)

- A canonical SD representation exists such that there are no 2 consecutive non-zero digits
- Reitwiesner Algorithm [1960]
  - assume $c_0 = 0$ (think a carry bit)
  - scan b_{i+1} and b_i from LSB to generate y_i and c_{i+1} in a single pass

e.g. 10111011, c_0=0

$\Rightarrow$ 1011101,$\bar{1}$ c_1=1

$\Rightarrow$ 101110,0$\bar{1}$ c_2=1

$\Rightarrow$ 10111,$\bar{1}$0$\bar{1}$ c_3=1

$\Rightarrow$ 1011,0$\bar{1}$0$\bar{1}$ c_4=1

$\Rightarrow$ 101,00$\bar{1}$0$\bar{1}$ c_5=1

$\Rightarrow$ 10,000$\bar{1}$0$\bar{1}$ c_6=1

$\Rightarrow$ 1,$\bar{1}$000$\bar{1}$0$\bar{1}$ c_7=1

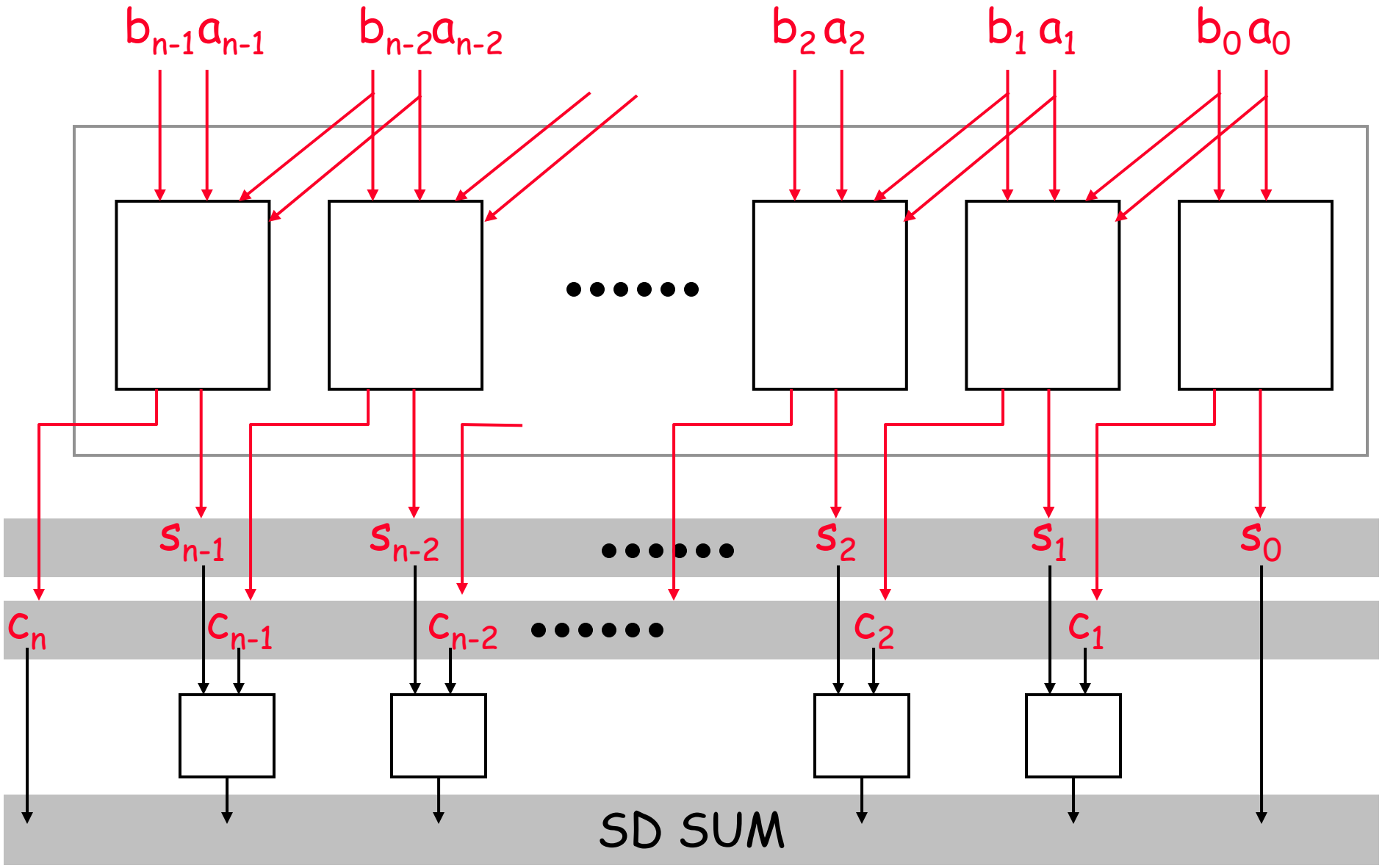
$\Rightarrow$,0$\bar{1}$000$\bar{1}$0$\bar{1}$ c_8=1

$\Rightarrow$,10$\bar{1}$000$\bar{1}$0$\bar{1}$ c_9=0

b_{i+1}	b_i	c_i	d_i	c_{i+1}	
0	0	0	0	0	string of 0s
0	0	1	1	0	end of 1s
0	1	0	1	0	single 1
0	1	1	0	1	string of 1s
1	0	0	0	0	string of 0s
1	0	1	$\bar{1}$	1	single 0
1	1	0	$\bar{1}$	1	beginning of 1s
1	1	1	0	1	string of 1s

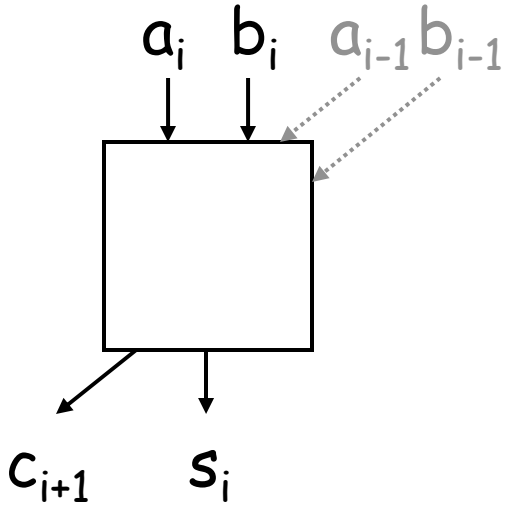
What else can you do with SD?

2:1 SD Adder



Binary Tree Multiplier using SD

- 2 SD numbers can be added digit-by-digit into their SD sum without ripple-carry (2:1 reduction!!)
- Enables a binary-tree-based multiplier array
- Requires a special SD bit-slice adder to make use of redundancy in the SD representation
  - a_{i-1} and b_{i-1} are examined to choose between <u>different</u> but <u>equivalent</u> combinations of c_{i+1} and s_i to output

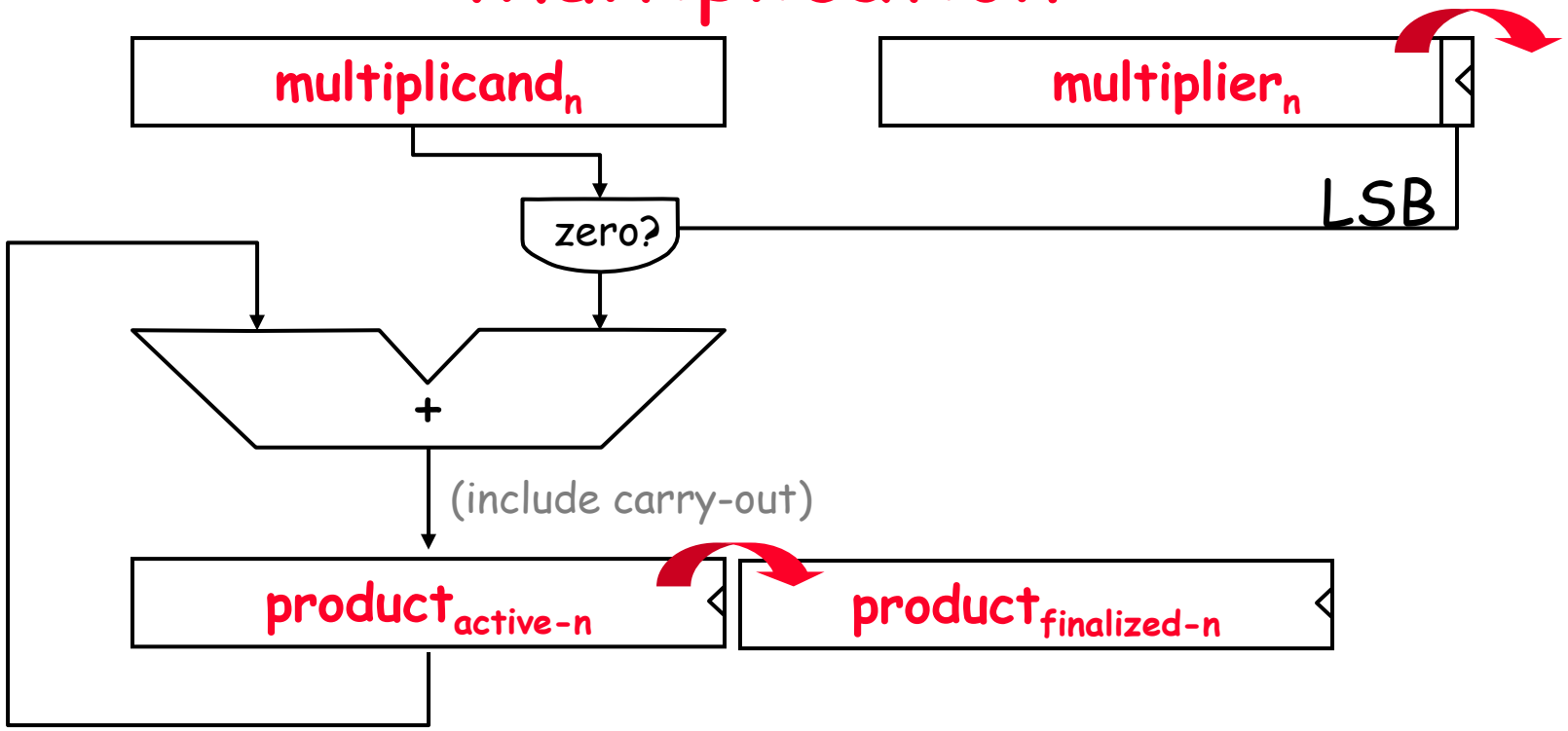


a_i	a_{i-1}	b_i	b_{i-1}	c_{i+1}	s_i
1	x	1	x	1	0
1	x	$\bar{1}$	x	0	0
$\bar{1}$	x	$\bar{1}$	x	$\bar{1}$	0
0	x	0	x	0	0
1	≥0	0	≥0	1	$\bar{1}$
1	else	0	else	0	1
$\bar{1}$	≥0	0	≥0	0	$\bar{1}$
$\bar{1}$	else	0	else	$\bar{1}$	1

J. C. Hoe

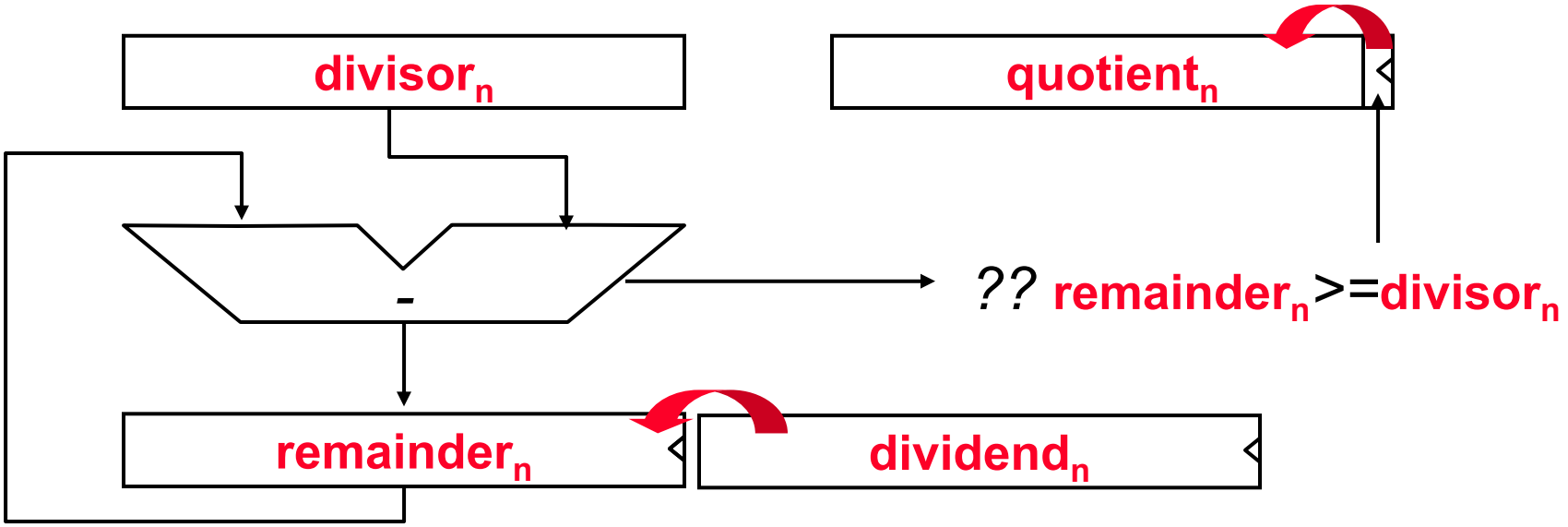
Iterative Multiplication and Division

Iterative Shift-and-Add Multiplication



- requires n iterations of "shift-and-add"
- multiplier and product$_{finalized}$ can share the same register since the in-used portions never overlap
- Could combine with earlier techniques to improve performance, e.g. CSA and SD

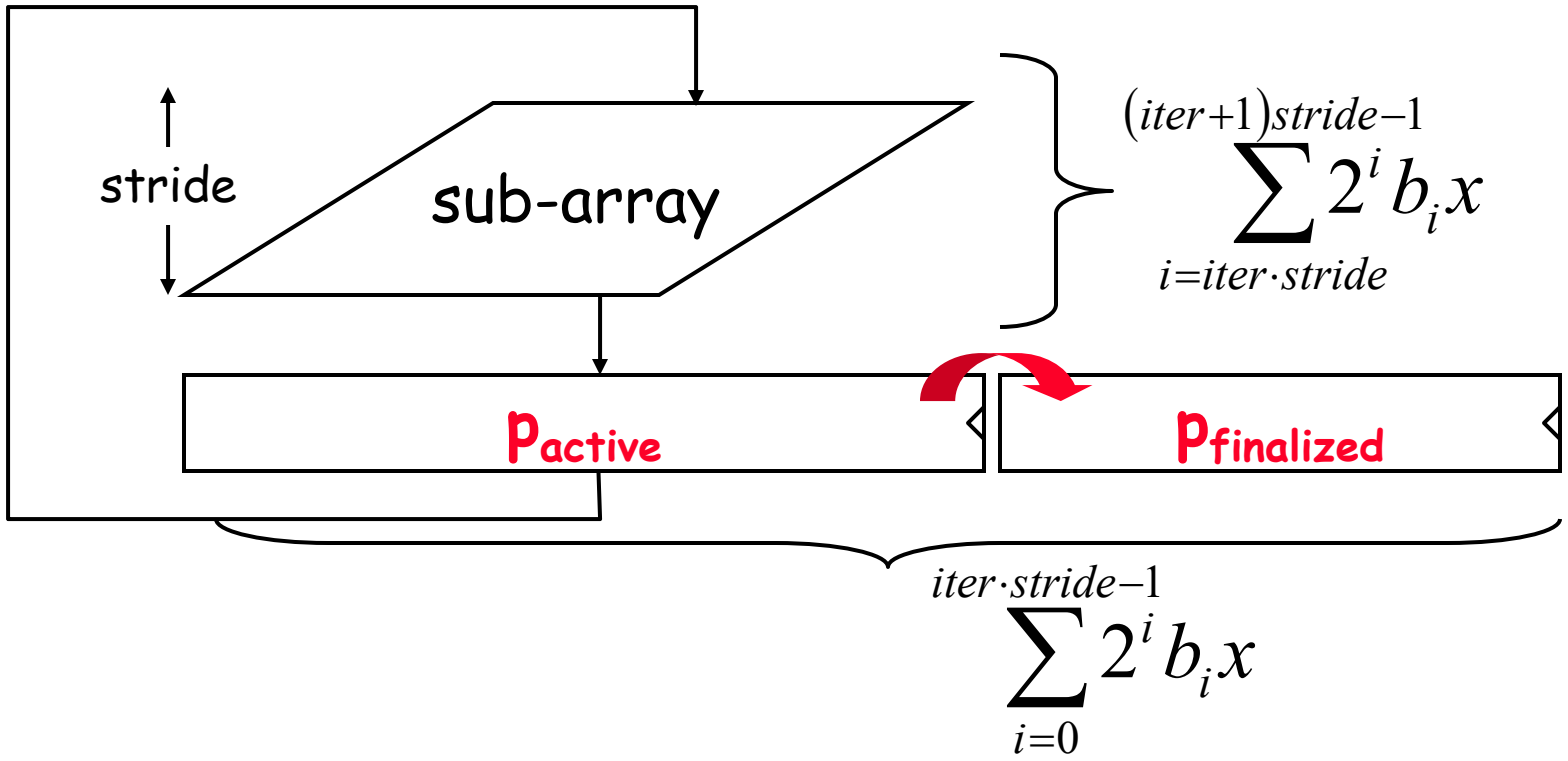
Iterative Division



- For A/B initialize: dividend$_n$=A; divisor$_n$=B; quotient$_n$=remainder$_n$=0;
- In each iteration,
  1. left-shift {remainder$_n$,dividend$_n$} and
  2. if remainder$_n$>=divisor$_n$ then
     - subtract divisor$_n$ from remainder$_n$
     - left-shift a 1-bit into quotient$_n$

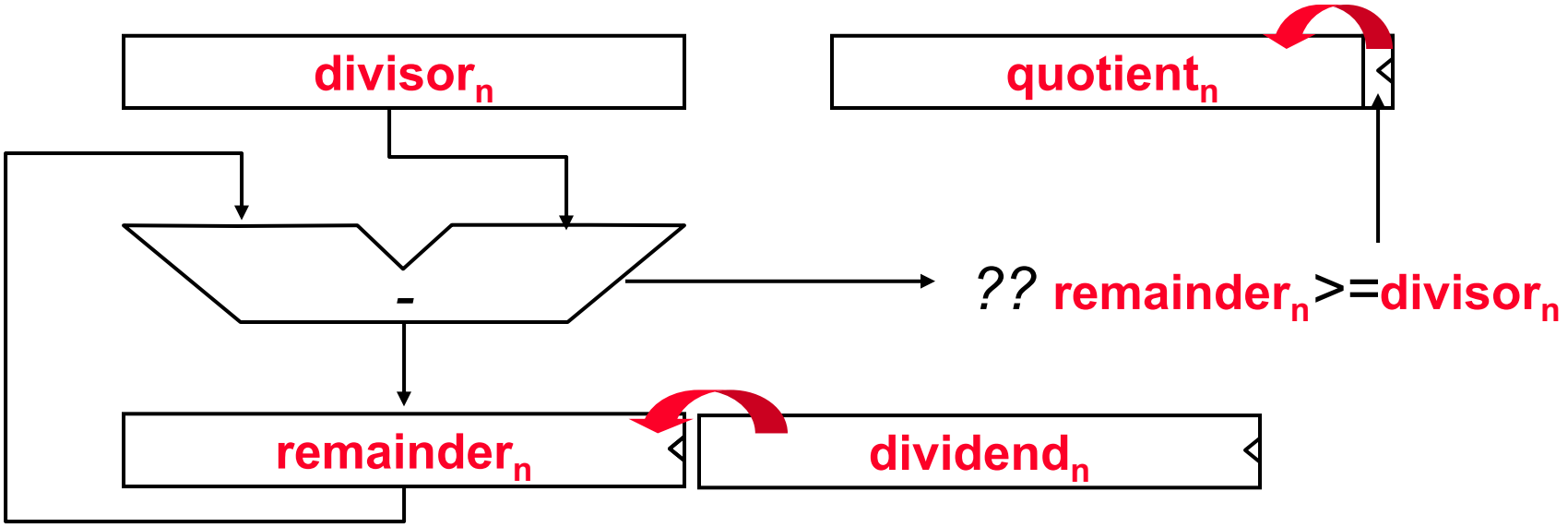
     else
     - left-shift a 0-bit into quotient$_n$

Higher Radix Multiplier Array



- requires n/stride iterations of "shift-and-add"
- right-shift the product registers (P_{active} and $P_{finalized}$) by stride-positions after each iteration
- compatible with CSA, binary, or regular adder arrays

Faster (Higher-Radix) Division



- Is it possible to unroll the iterative circuit to generate multiple quotient bits per cycle?
  - how to choose the new quotient bits? (Think about how you normally do long division in base 10)
  - it would have to be a non-propagating adder (i.e. CSA or SD) to have performance advantages

SRT Division (Sweeney, Robertson, and Tocher)

- Represent quotient in higher-radix SD
  - redundant ways to represent the same number
  - 1_{10} could be written as $01_{2\text{-SD}}$ or $1\bar{1}_{2\text{-SD}}$
- after **remainder$_n$** is in range of **divisor$_n$**, in each iteration, look at only the top-few digits of **divisor$_n$** and **remainder$_n$** to "guess" the new SD quotient bits
  - e.g. radix-4 SRT looks at high-order 4 bits of **divisor$_n$** and 6 bits of **remainder$_n$** to chose a quotient between -2 and 2
  - typically done with a lookup table

The intuition is to guess new quotient bits "in the right ballpark" to guarantee you can compensate for the worst-case errors in subsequent iterations

e.g. what are the possible values for 2-bit number, 1"?"

$\{2,3\}_{\text{radix-2}}$, $\{3,2,1\}_{\text{radix2-SD}}$

Dividing by Powers of 2 via ARS

- Not quite right for negative 2's-complement numbers

  - $4'b1000 \rightarrow_r 4'b1100 \rightarrow_r 4'b1110 \rightarrow_r 4'b1111 \rightarrow_r 4'b1111$
    $-8 \qquad -4 \qquad -2 \qquad -1 \qquad -1$

  - $4'b1011 \rightarrow_r 4'b1101 \rightarrow_r 4'b1110 \rightarrow_r 4'b1111 \rightarrow_r 4'b1111$
    $-5 \qquad -3\ (-2.5) \qquad -2\ (-1.25) \qquad -1(-0.625) \qquad -1$

"Rounding" to the more negative direction whenever a 1 is shifted off the right!!

Nevertheless, good enough approximation most of the time----can be off by at most 1.

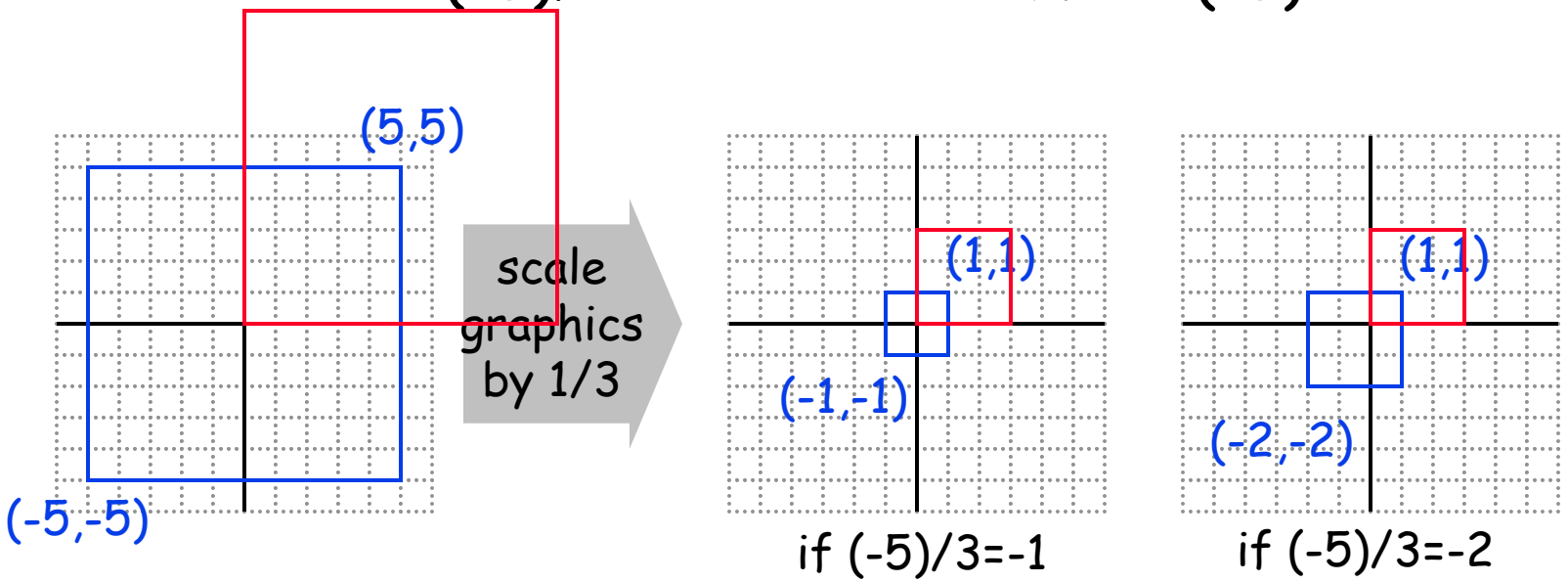
Quotient and Remainder

- Quotient(A,B) = $\lfloor A / B \rfloor$
- Remainder(A,B) = A - Quotient(A,B) · B
- In C

  5/3=1 and 5%3=2

- What is

  (-5)/3= and (-5)%3=



Further Readings

- If you are interested about computer arithmetic, a great place to start is Appendix H of *Computer Architecture: A quantitative approach* by Hennessy and Patterson

- Next Lecture: IEEE Floating Point