ECE 362

Microprocessor Systems and Interfacing

Computer Arithmetic

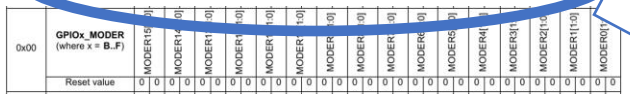
Spring 2025

Reading

- P&H textbook chapter 3 & Appendix A: Computer Arithmetic

Final Piece!

```
GPIOC->MODER |= 0x01555555;
```



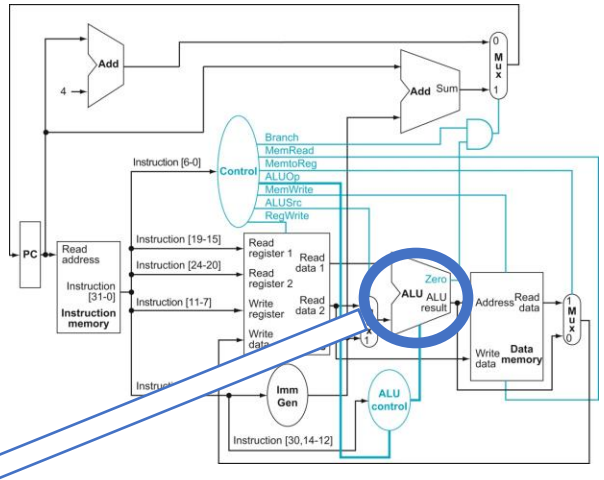
We started with embedded C.
No idea what instructions are.

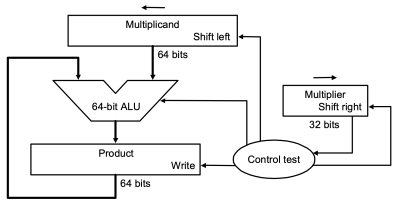
Built high-level systems in lab.

```
fact:
  addi sp, sp, -8   // allocate 2 items on the stack
  sw x1, 4(sp)      // save the return address
  sw x10, 0(sp)     // save the argument n
  addi x5, x10, -1  // x5 = n -1;
  bge x5, x0, L1    // if (n - 1) >= 0, goto L1
  addi x10, x0, 1   // return 1
  addi sp, sp, 8    // pop 2 items off the stack
  jalr x0, 0(x1)    // return to the caller
L1: addi x10, x10
  jal fact          // call fact with (n-1)
  addi x6, x10, 0   // return from jal: move results of fact (n-1) to x6
  lw x10, 0(sp)     // restore argument n
  lw x1, 4(sp)      // restore return address
  addi sp, sp, 8    // adjust stack pointer to pop 2 items
  mul x10, x10, x6  // return n * fact(n-1)
  jalr x0, 0(x1)    // return to the caller
```

Moved down to assembly instructions.

Understand how programs are constructed.





Final jump: What does the hardware that implements arithmetic really look like?

Moved down to how these get implemented in hardware

Introduction Computer Organization/Architecture

Outline

- Basic arithmetic (Ch 3.1-3.3)
  - Representing numbers
  - 2's Complement, unsigned
  - Addition and subtraction
  - Add/Sub ALU
    - full adder, ripple carry, subtraction, together
  - Logical operations
    - and, or, xor, nor, shifts - barrel shifter
  - Carry lookahead, overflow
- Integer Multiplication
- Integer Division
- Floating point numbers
  - Representation
  - Addition/multiplication

Unsigned Integers

Flashback

- Recall:
  - n bits give rise to 2^n combinations
  - let us call a string of 32 bits as "$b_{31}\ b_{30}\ldots b_3\ b_2\ b_1\ b_0$"
- $f(b_{31}\ldots b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
- Treat as normal binary number
  - e.g., 0 . . .011010101

  $= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

  $= 128 + 64 + 16 + 4 + 1 = 213$
- max f (111 . . . 11) = $2^{32} - 1 = 4,294,967,295$
- min f(000 . . . 00) = 0
- range [0, 2^{32}-1] => # values $(2^{32} - 1) - 0 + 1 = 2^{32}$

Numbers

Flashback

- Bits are just bits (no inherent meaning)
  — conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001…
  decimal: 0…2^n-1
- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers ***
  negative numbers
- How do we represent negative numbers?
  i.e., which bit patterns will represent which numbers?

Number Representation

Flashback

- Sign Magnitude:

	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Balance, number of zeros, ease of arithmetic

Signed Integers

Flashback

- 2's complement
- $f(b_{31}\ b_{30}\ \ldots\ b_1\ b_0) = -b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
  - max $f(0111\ \ldots\ 11) = 2^{31} - 1 = 2147483647$
  - min $f(100\ \ldots\ 00) = -2^{31} = -2147483648$ (asymmetric)
- range $[-2^{31}, 2^{31}-1]$ => #values $(2^{31}-1 - -2^{31} + 1) = 2^{32}$
- E.g., -6
- 000 . . . 0110 --> 111 . . 1001 + 1 --> 111 . . .1010

Two's Complement Operations

Flashback

- Negating a two's complement number: invert all bits and add 1
  - remember: “negate” and “invert” are quite different!

- Converting n bit numbers into numbers with more than n bits:
  - Converting immediate values into 32/64 bits for arithmetic
  - copy the most significant (the sign) bit into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - "sign extension"

Negation in 2's complement

- Negation: Invert all bits, add 1
  - Why?
- If the (k+1) bit 2's complement representation of a number N is $<b_k\ b_{k-1}\ .\ .\ .\ b_1\ b_0>$

  $N_k = -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
- Show that the negation procedure is correct

Negation in 2's complement

- Key trick:
  - Complement of bit b can be written as (1-b)
- $N_k = -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
- Inversion gives us

  $-(1-b_k) \times 2^k + (1-b_{(k-1)}) \times 2^{(k-1)} + \ldots + 2^1 \times (1-b_1) + 2^0 \times (1-b_0)$
- Separating the red and blue terms and adding 1

  $-2^k + 2^{(k-1)} + \ldots + 2^1 + 2^0 + 1 - N_k$
- Blue terms plus 1 goes to zero. Q.E.D.

Sign extension

- Consider representation of -2:

3bit (decimal)	2-bit (decimal)
011 (+3)	
010 (+2)	
001 (+1)	01 (+1)
000 (0)	00 (0)
111 (-1)	11 (-1)
110 (-2)	10 (-2)
101 (-3)	
100 (-4)	

Mathematical basis

- Inductive proof
  - if the (k+1)-bit 2's complement representation of a number N is $<b_k\ b_{k-1}\ \ldots\ b_1\ b_0>$
    - $N_k = -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
  - Then the (k+2)-bit 2's complement representation $<b_k\ b_k\ b_{k-1}\ \ldots\ b_1\ b_0>$ also represents N
    - $N = -b_k x 2^{k+1} + b_k x\ 2^k + b_{(k-1)} x 2^{(k-1)} + \ldots + 2^1 x b_1 + 2^0 x b_0$
    - $= (-2 x b_k + b_k) \times 2^k + b_{(k-1)} x 2^{(k-1)} + \ldots + 2^1 x b_1 + 2^0 x b_0$
    - $= -b_k \times 2^k + b_{(k-1)} x 2^{(k-1)} + \ldots + 2^1 x b_1 + 2^0 x b_0$

Addition and Subtraction

- Similar to decimal (carry/borrow twos instead of tens)
- Identical operation for signed and unsigned
  - E.g. Unsigned vs signed

0011	3	3
1010	10	-6
1101	13	-3

Interesting cases

- Show computation in 4-bit 2's complement representation

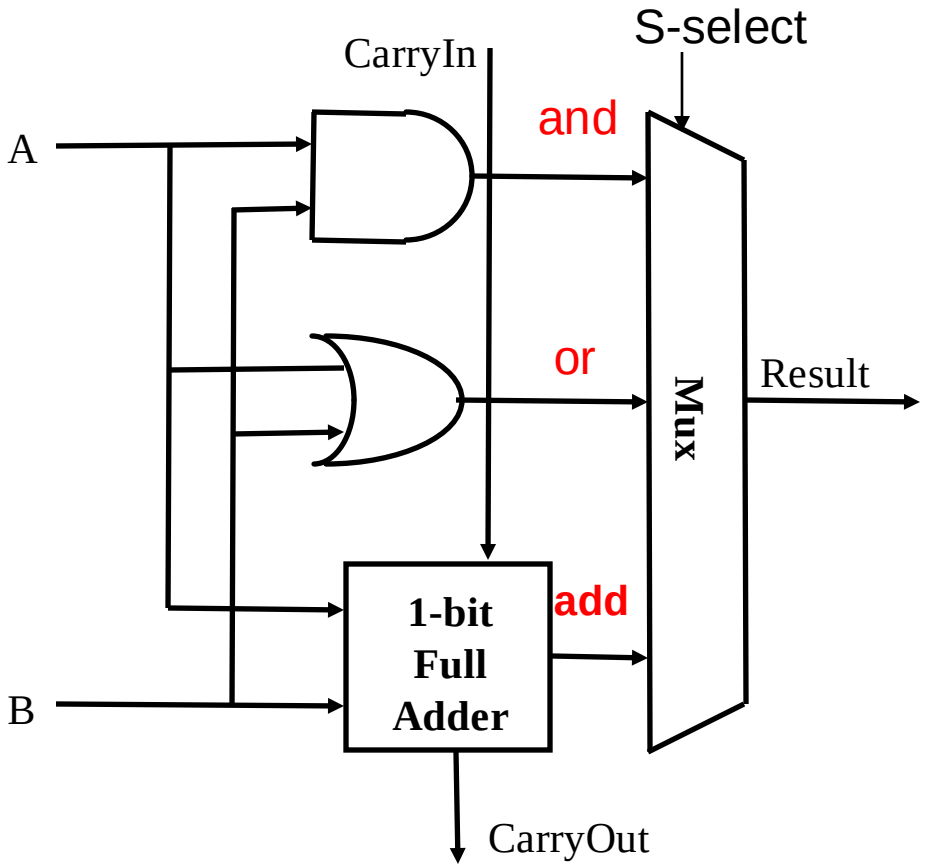
  4+4

  (-4) + (-4)

- Overflow: later

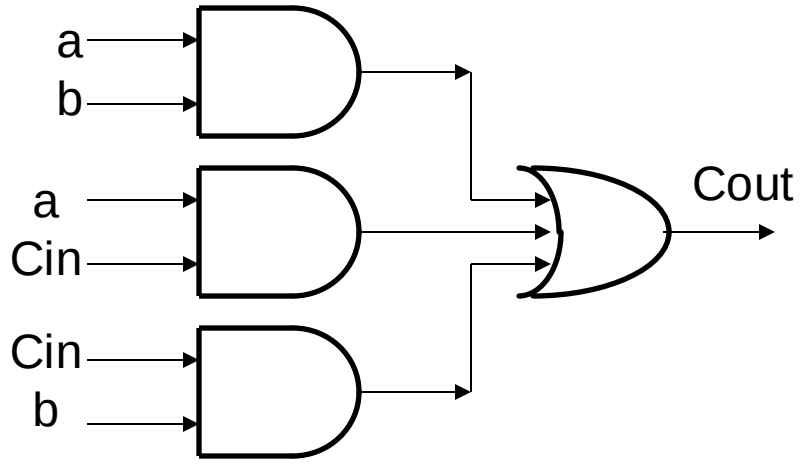
ALU bit-slice

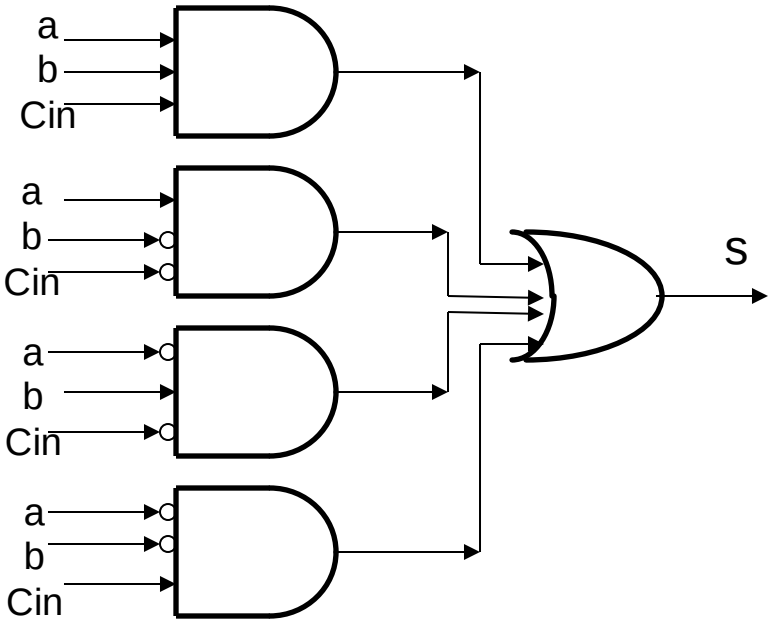
- Bit-wise operation
  - and, or, add
  - Full adder?
  - Sub?



Full adder

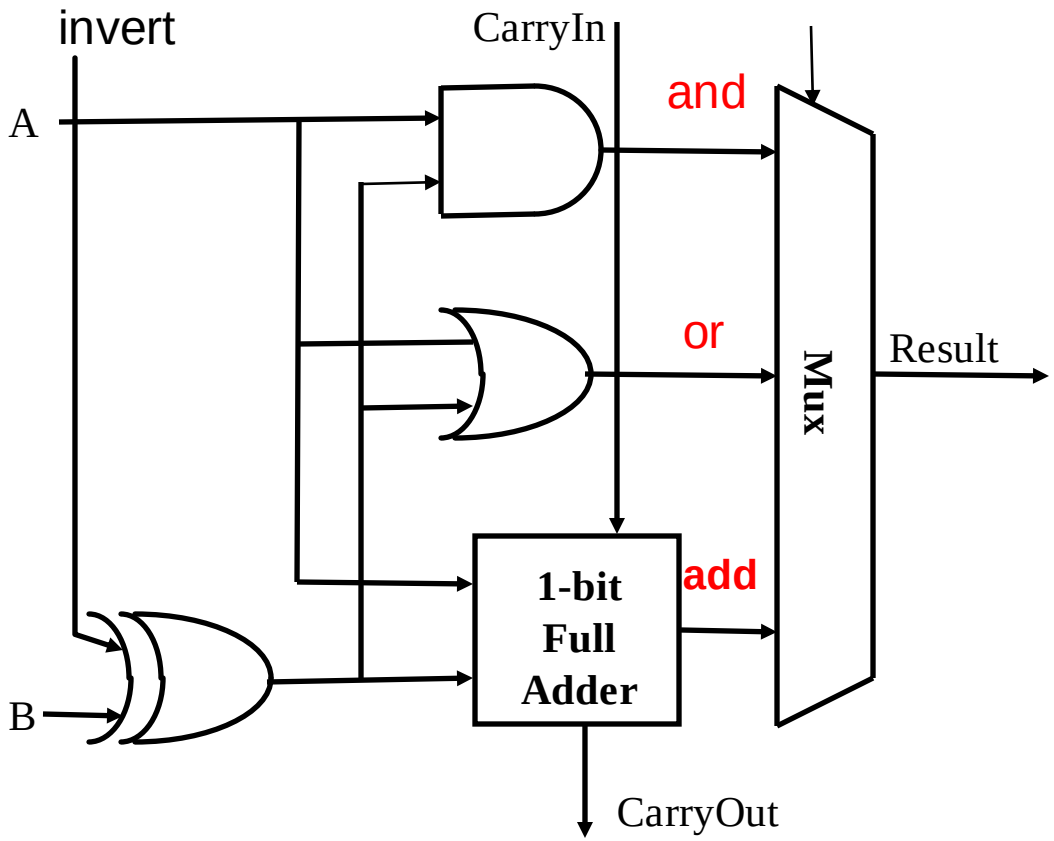
- Three inputs and two outputs
- Cout, s = F(a,b,Cin)
  - Cout : only if at least two inputs are set
  - S : only if exactly one input or all three inputs are set
- Logic?





Subtract

- A - B = A + (– B)
  - form two's complement by invert and add one



Self-Exercises

- How do I convert 237 to binary?

- What is the 2's complement representation (3-bit) of:

  -3:

  +5:

- If a number X is represented as $b_3\ b_2\ b_1\ b_0$ in 4 bit 2's complement arithmetic, what is the 8-bit 2's complement representation of X?

- Show the computation in 4-bit 2's complement arithmetic:

  5 – (-3)

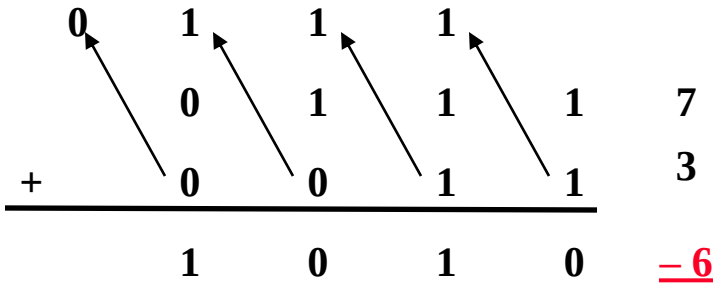
Overflow

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

Decimal	2's Complement
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

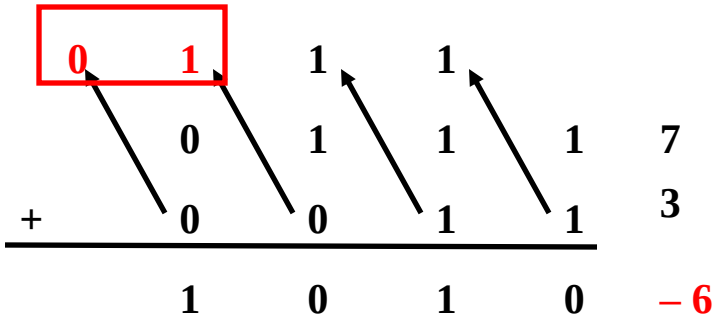
- **Examples: 7 + 3 = 10 but …**

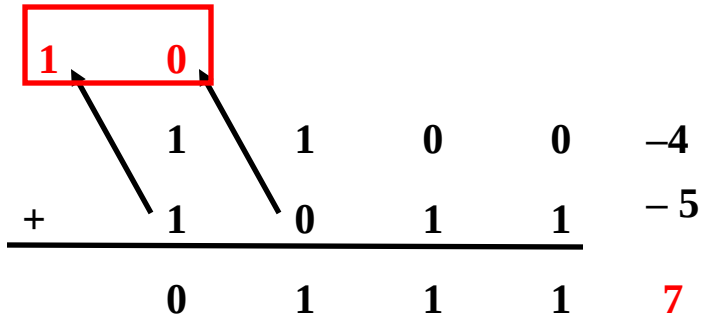
- 4 - 5 = - 9 but …



Overflow

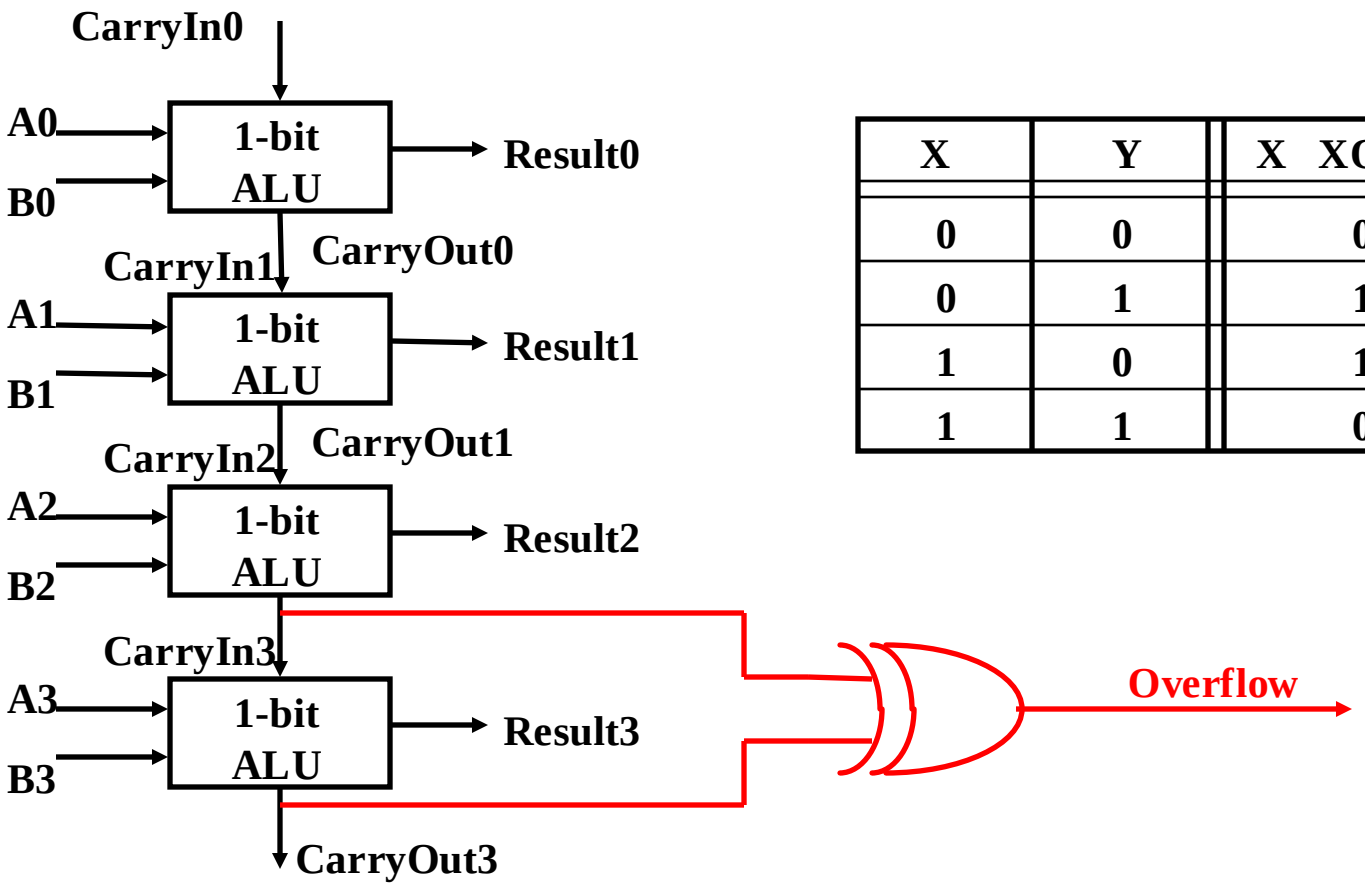
- Overflow: the result is too large (or too small) to represent properly
  - Example: - 8 < = 4-bit binary number <= 7
- When adding operands with different signs, overflow cannot occur!
- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive
- On your own: Prove you can detect overflow by:
  - Carry into MSB XOR Carry out of MSB





Overflow detection

- Carry into MSB XOR Carry out of MSB
  - For N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]

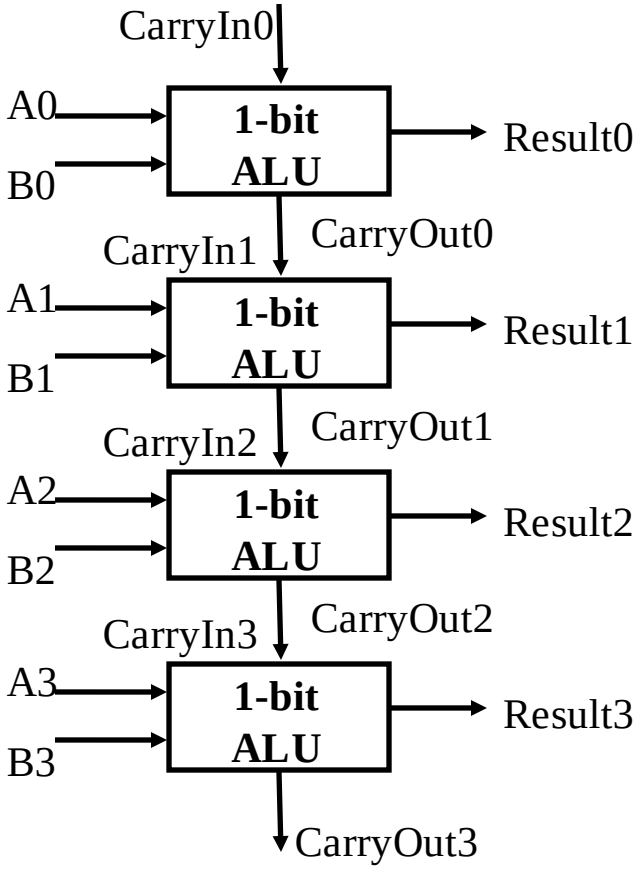


X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Negative, Zero

- Required for conditional branches
- Zero
  - How?
  - NOR all 32 bits
  - Avoid 33rd bit (carry out)
- Negative may be required on overflow
  - If (a<b) jump : jump taken if a-b is negative
- Tempting to consider MSB
  - E.g. if (-4 < 3) branch
  - Branch should be taken, but (-4-3) computation results in overflow for 3 bits… so MSB is 0
  - E.g. if (3 < -3) branch
  - Branch should not be taken but (3- (-3)) results in overflow for 3 bits … so MSB is 1.
  - Negative = ??
  - If not overflow -> look at MSB, if overflow -> look at carry out bit

Ripple-carry adder



Problem : Slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
  - Delay = 32 x Critical-path(Fast adder) + XOR

- Is there more than one way to do addition?
  - Two extremes: ripple carry and sum-of-products
  - Flatten expressions to two levels

Can you see the ripple? How could you get rid of it?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $\quad c_2 = \langle 7 \text{ min-terms} \rangle$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $\quad c_3 = \langle 15 \text{ min-terms} \rangle$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $\quad c_4 = \langle 31 \text{ min-terms} \rangle$

Not feasible! Why? Exponential fanin

Blocked Ripple Carry

- Flatten Logic in blocks of “k” say 4
  - But not the naïve version
  - Block of 4 requires 31-input OR gate
- Ripple carry from one block to the next
- Delay through N-bit addition
  - (N/4) * 2 + 1 (XOR)

- Reduction by a constant factor
  - Still linear in number of inputs
  - Can do better: Logarithmic delay

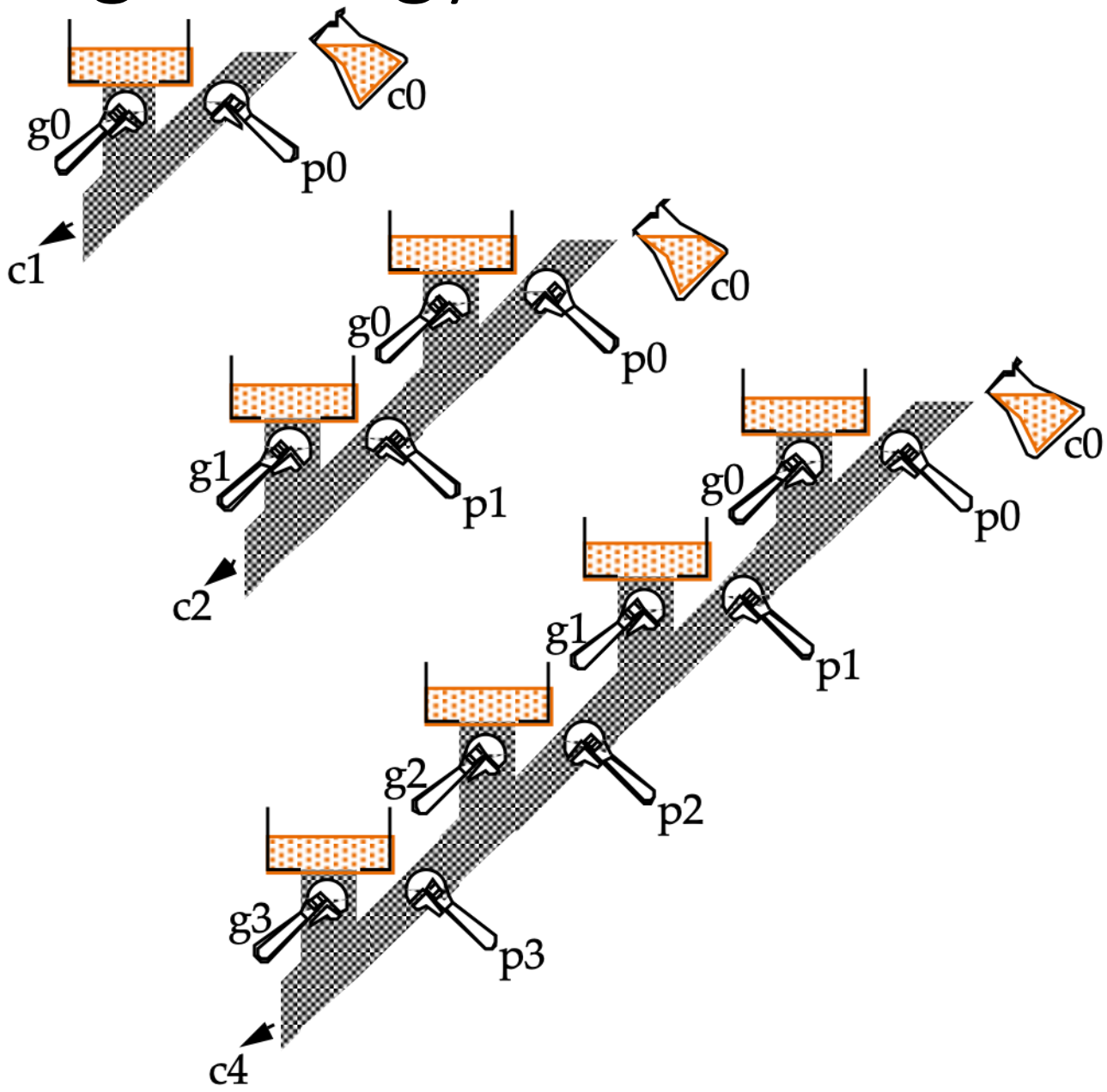
Carry Lookahead Adder

- Reformulate addition
  - Facilitates block computation
  - Facilitates hierarchical, parallel computation
  - Key concepts: Generate and Propagate
- An approach in-between our two extremes
  - Ripple carry
  - Flattened 2-level

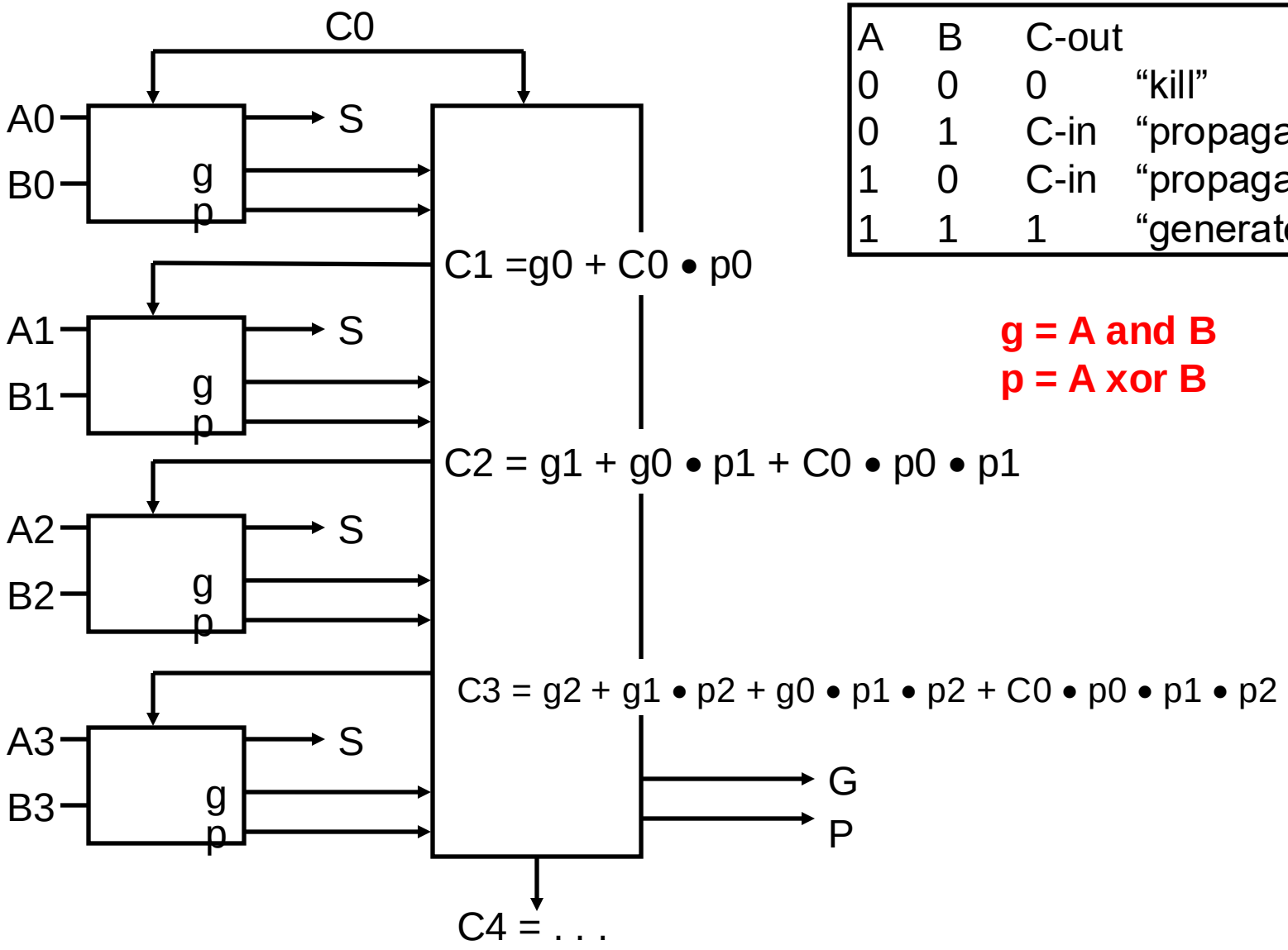
Carry look-ahead

- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry?
    - $g_i = a_i . b_i$
  - When would we propagate the carry?
    - $p_i = a_i + b_i$ (slightly corrected later)
- Did we get rid of the ripple?

CLA: Plumbing Analogy



Carry-lookahead adder



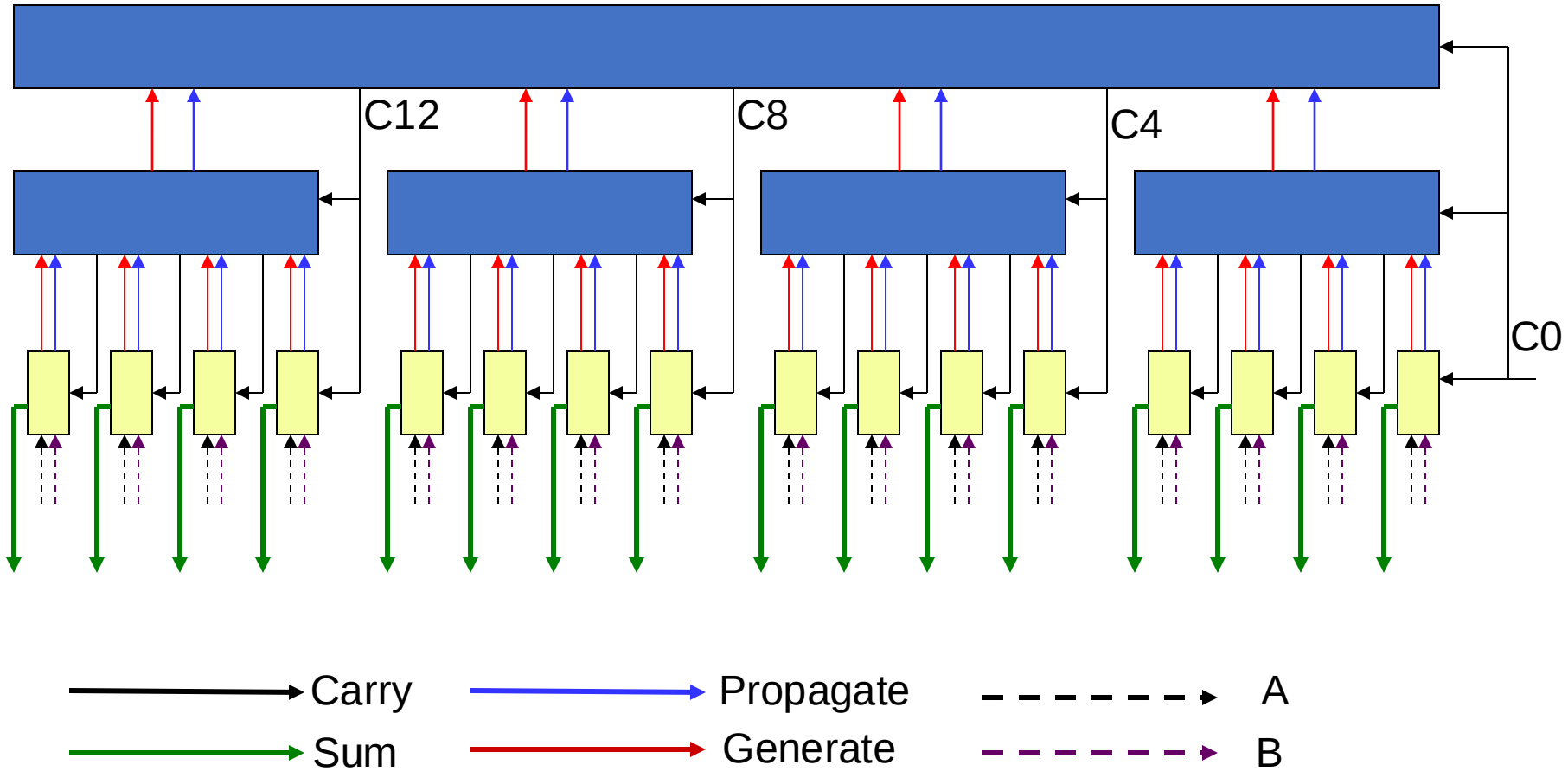
A	B	C-out	
0	0	0	“kill”
0	1	C-in	“propagate”
1	0	C-in	“propagate”
1	1	1	“generate”

g = A and B
p = A xor B

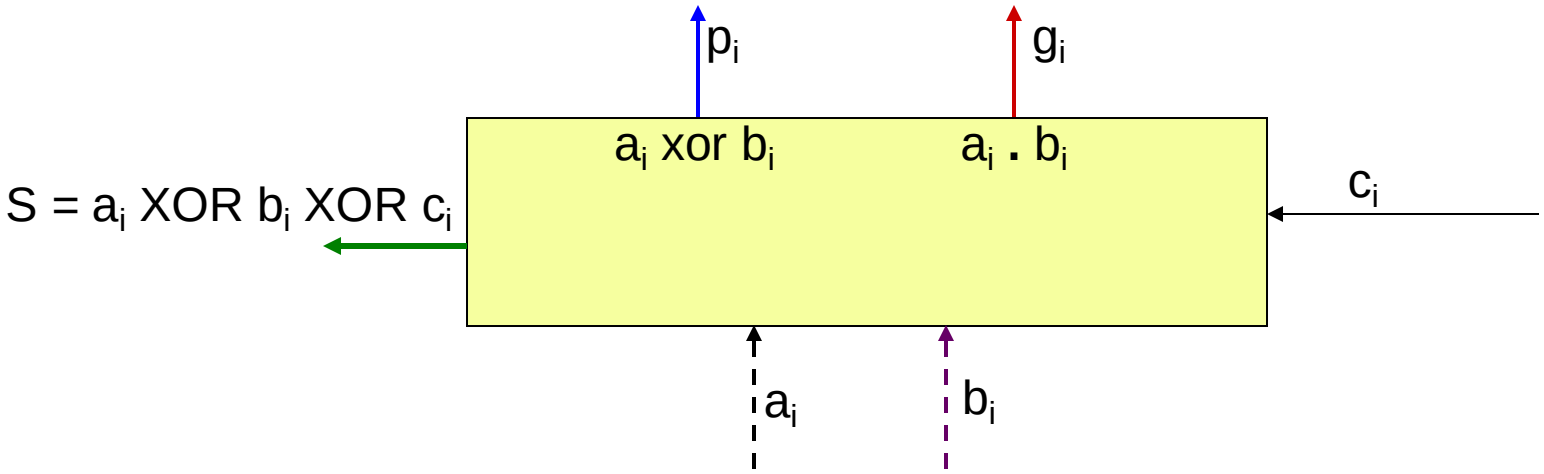
Carry-Lookahead Adder

- Waitaminute!
  - Nothing has changed
  - Fanin problems if you flatten!
    - Not really, Linear fanin, not exponential
  - Ripple problem if you don't!
- Enables divide-and-conquer
- Figure out Generate and Propagate for k-bits together
- Compute hierarchically
- Instead of linearly rippling thru blocks of k-bits (our previous idea) go up a tree of k-bit blocks (logarithmic)

2-level 16-bit CLA



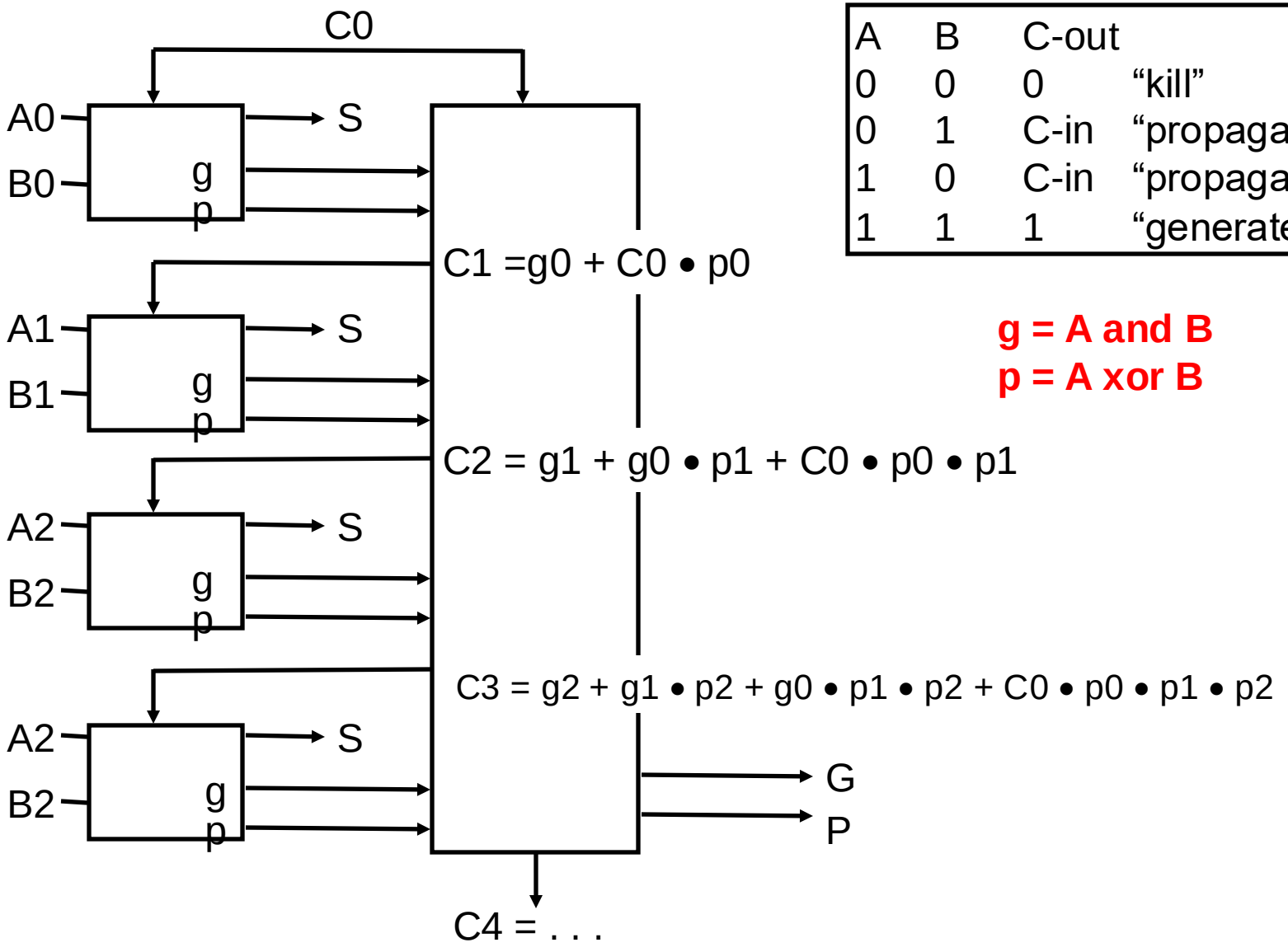
Leaf Node



- Zero level gi and pi generation
  - One gate delay
- Part of Full adder (only sum bit)
  - Two gate delays (ignoring inverters)

Intermediate nodes

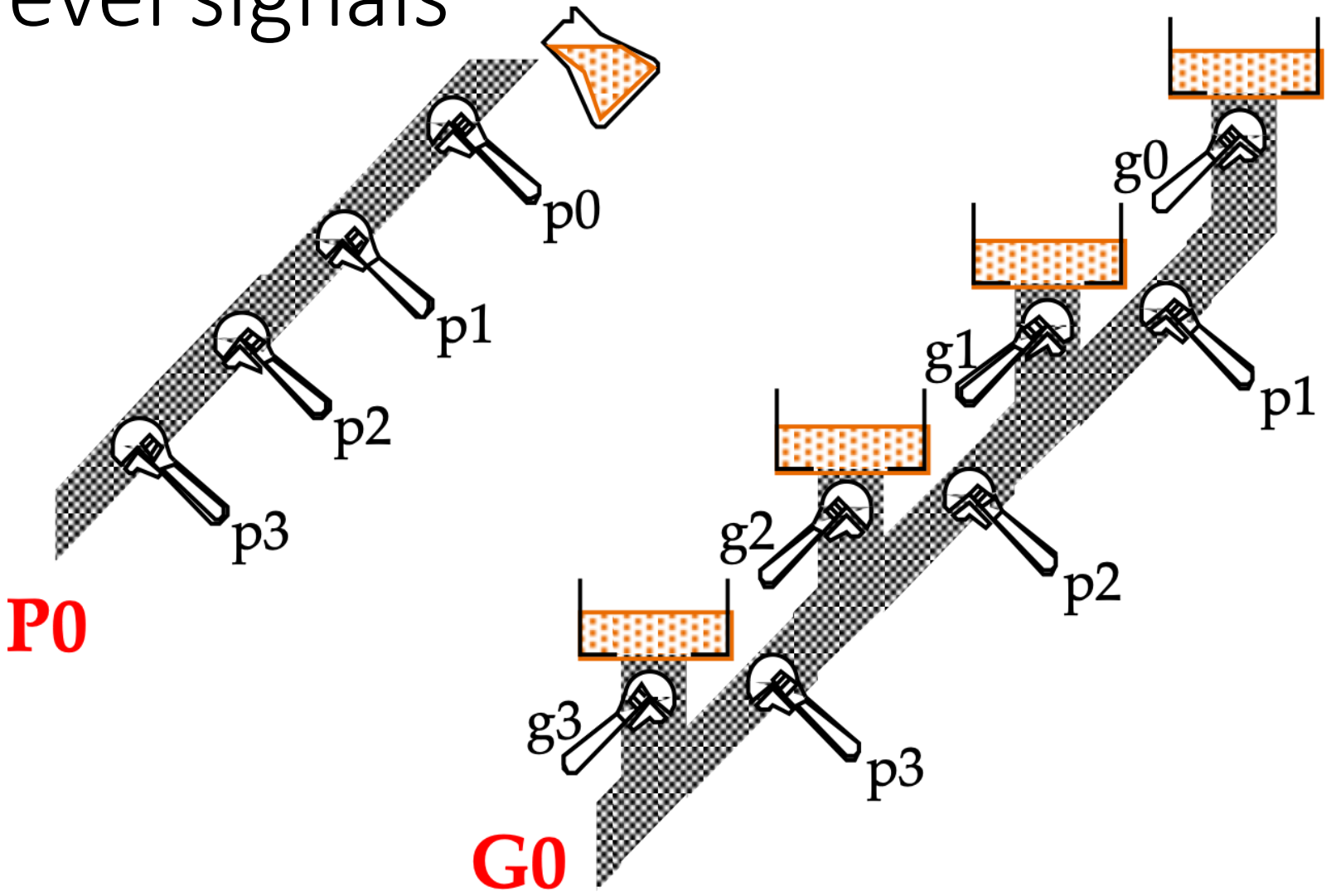
- We know how carry's are generated



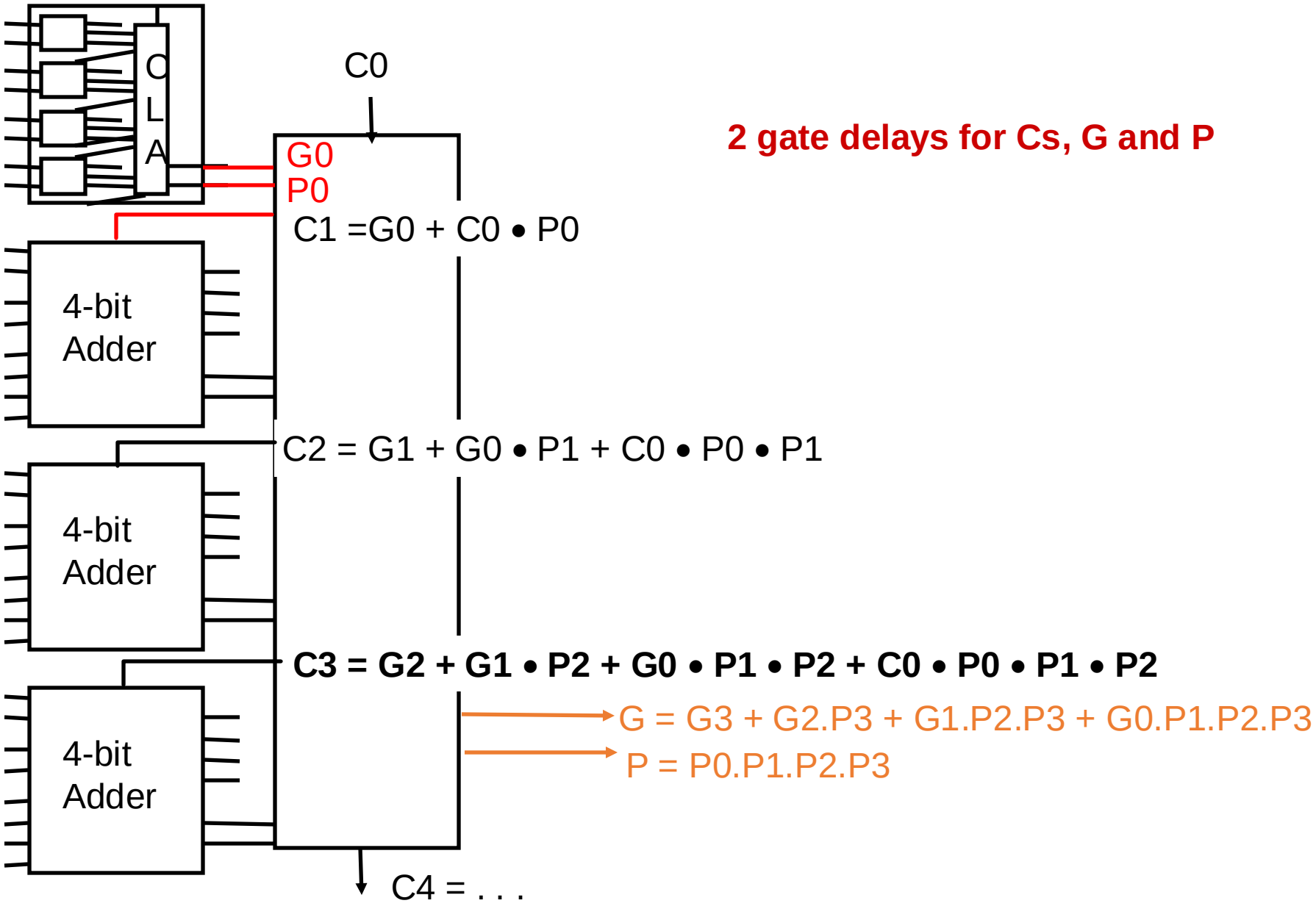
A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

g = A and B
p = A xor B

Block level signals

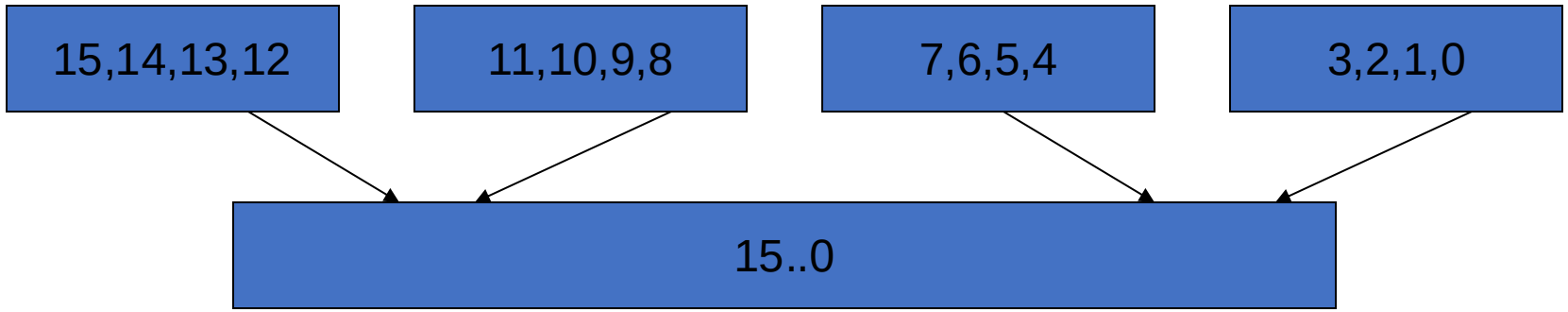


CLA Logic



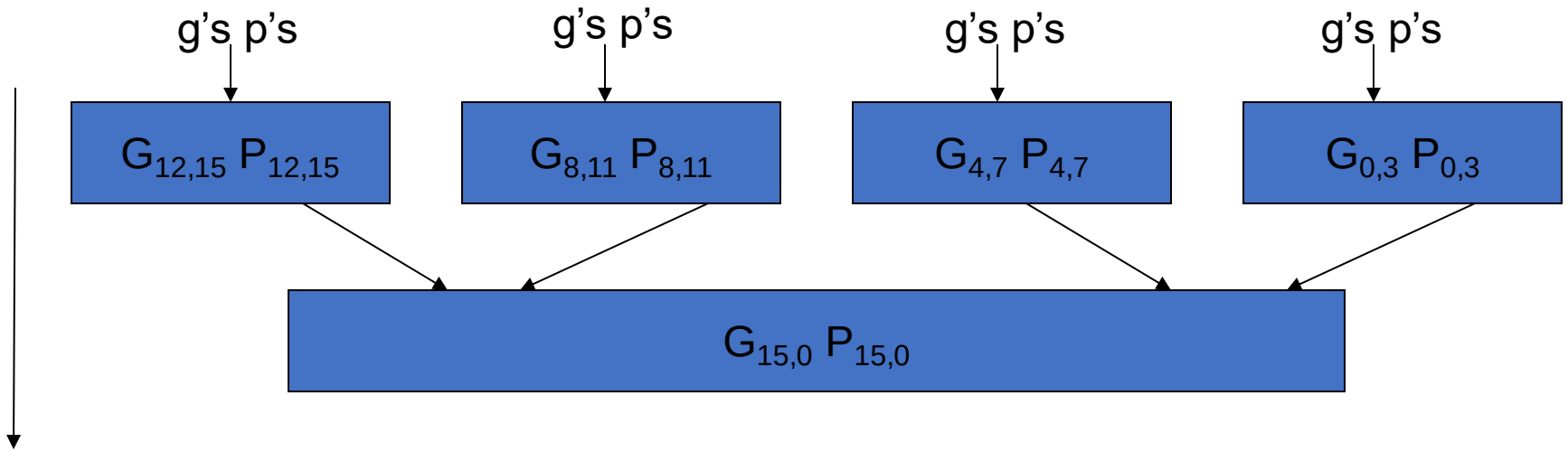
2 gate delays for Cs, G and P

CLA: Delay Analysis



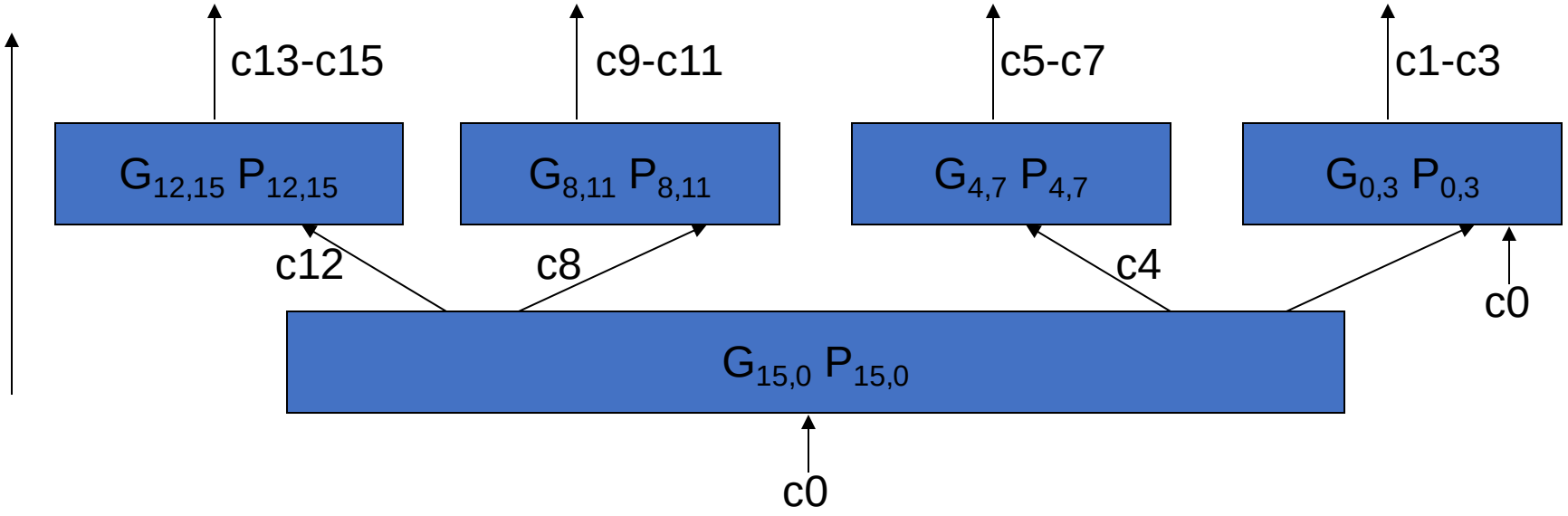
- Height of tree = $O(\log_4(n))$
- 16-bit addition : $k*\log_4(16) = k*2$ (slightly inaccurate)

Computing G's and Ps



- UP the tree
- Parallel algorithm in hardware
  - g's and p's : 1
  - First level Gs and Ps : 1 + 2 = 3
  - Second level Gs and Ps : 1 + 2 + 2 = 5
    - (not needed for 16 bit adder example)

Computing C's



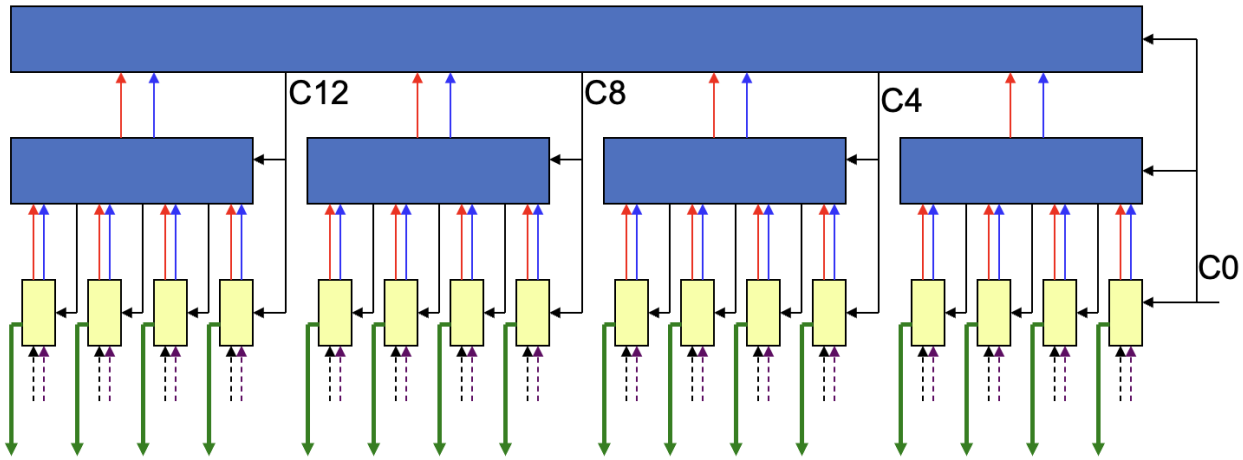
- DOWN the tree
- Worth spending some time to think about the intricacies

Delays

- c1—c3 : 1 + 2
- c4 : 1 + 2 + 2 **
- c5-c7 : 1 + 2 +2 + 2 = 7
- c8 : 1 + 2 + 2
- c9-c11 : 1 + 2 + 2 + 2 = 7
- c12: 1 + 2 + 2
- c13-c15 : 1 + 2 + 2 + 2 = 7

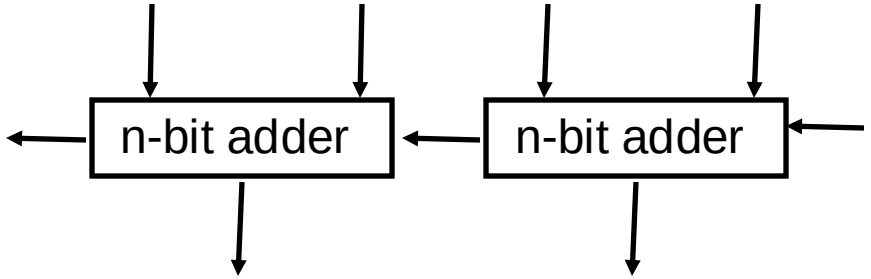
- What about sum-bits?

**** Can be 1 + 2 if computed in first level CLA block. No overall improvement by doing this**

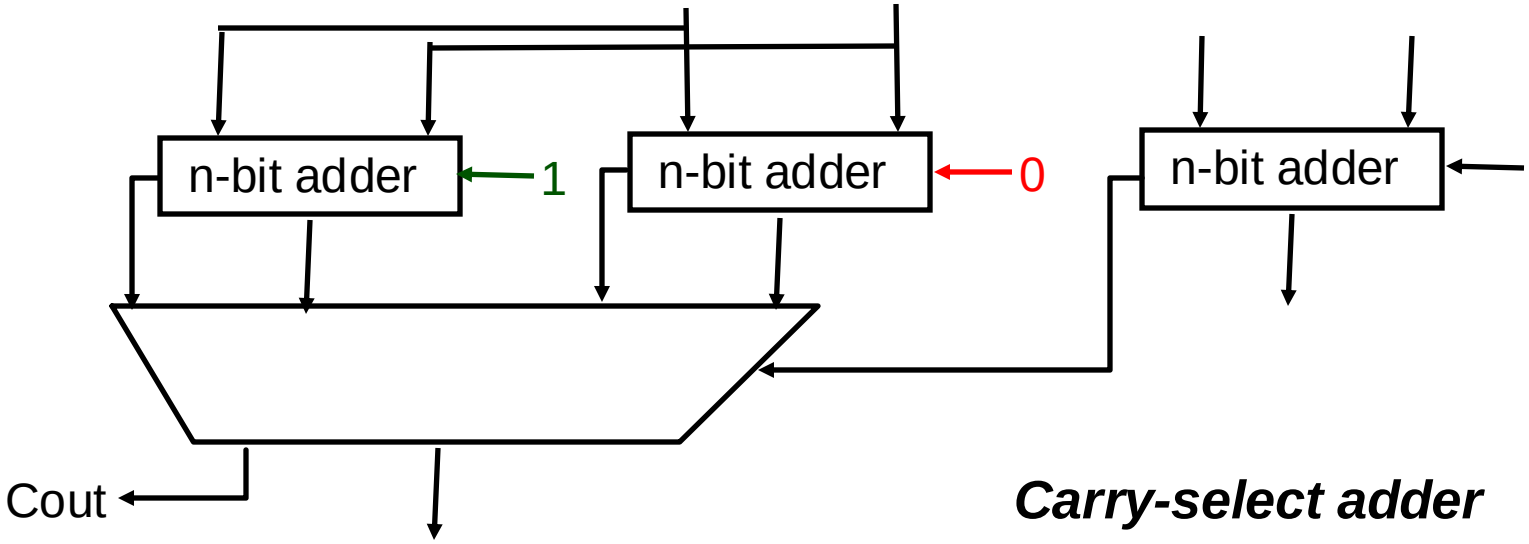


Carry-selection: Guess

CP(2n) = 2*CP(n)



CP(2n) = CP(n) + CP(mux)

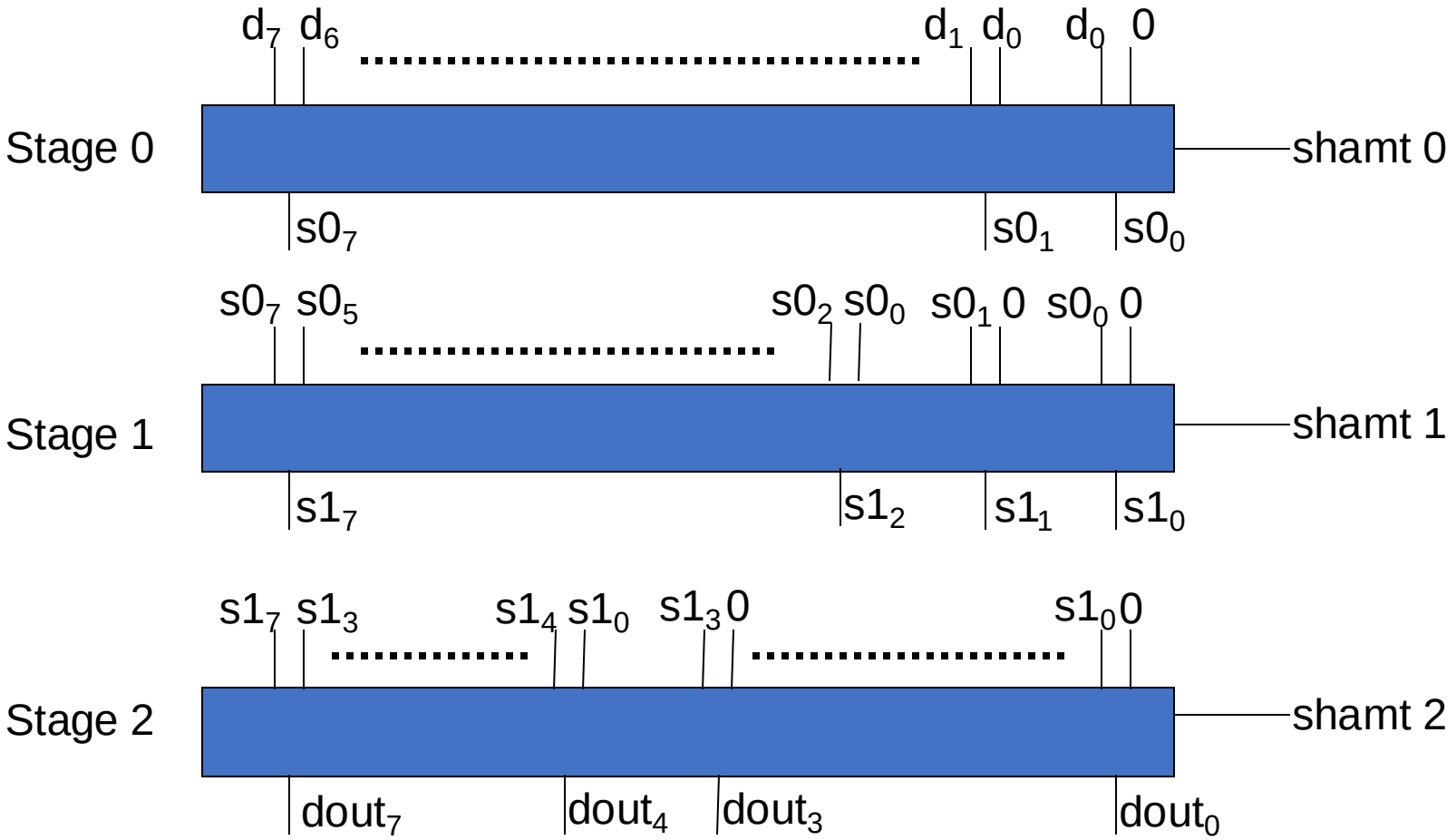


Carry-select adder

Shift

- E.g., Shift left logical for d<7:0> and shamt<2:0>
  - Using 2-1 muxes called Mux(select, in0, in1)
  - stage0<7:0> = Mux(shamt<0>,d<7:0>, d<6:0> || 0)
  - stage1<7:0> = Mux(shamt<1>, stage0<7:0>, stage0<5:0> || 00)
  - dout<7:0> = Mux(shamt<2>, stage1<7:0>, stage1<3:0> || 0000)
- Other operations
  - Right shift
  - Arithmetic shifts
  - Rotate

Barrel Shifter



Extensions

- Design a barrel shifter unit that can do
  - Right shift
  - Left shift
- Design a shifter/rotator combo that can do
  - Right rotate
  - Left rotate
  - In addition to shifts

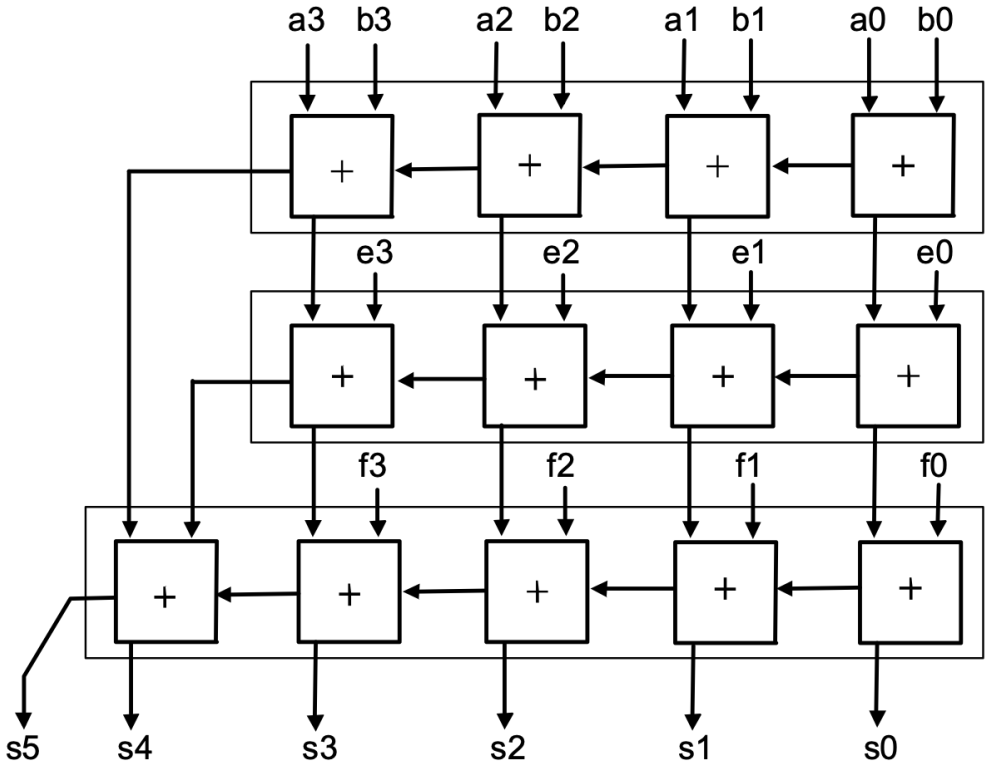
Grade School Multiplication

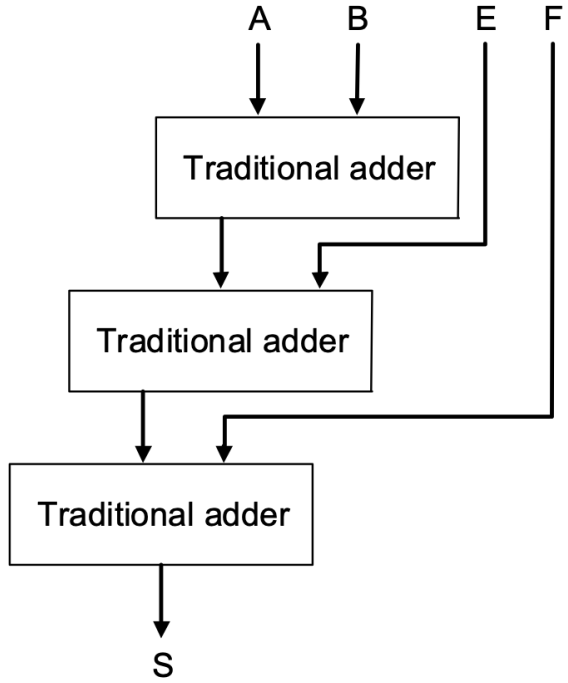
- Multiplicand (1000_{10}) and Multiplier (1001_{10})

```
    1000
   0000
  0000
 1000
-------
1001000₁₀
```

- Two approaches
  - Purely combinational
  - Sequential

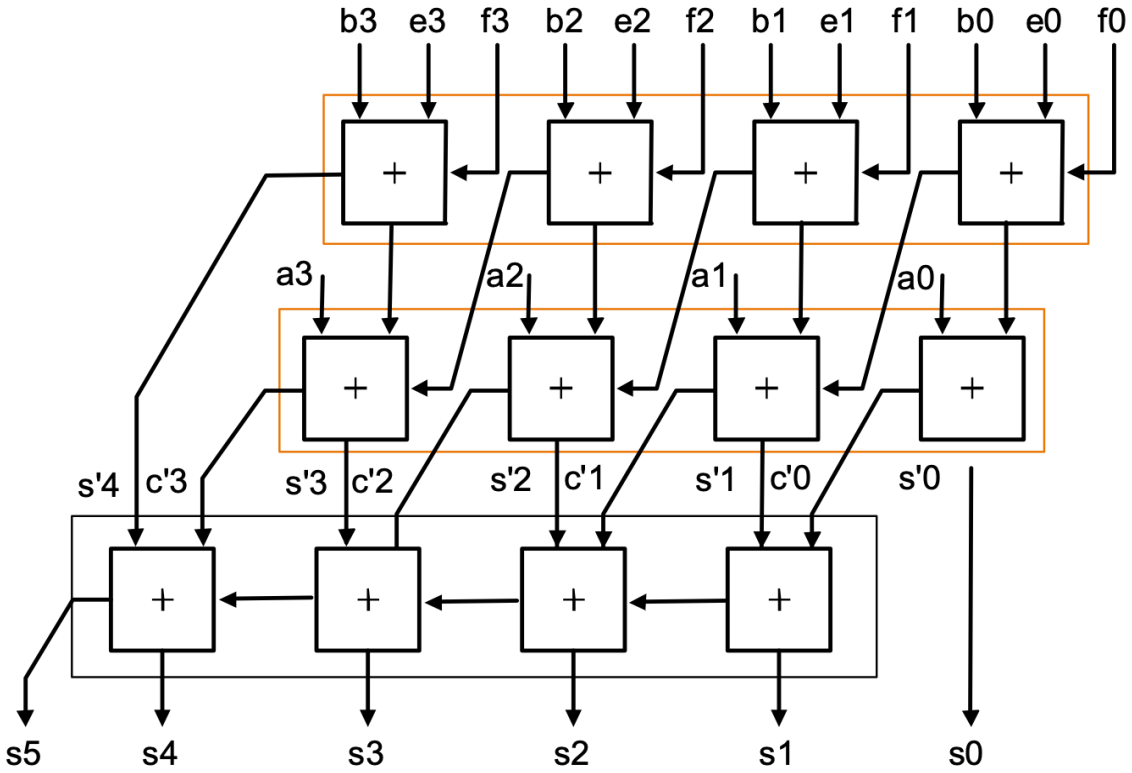
Adding more than two operands

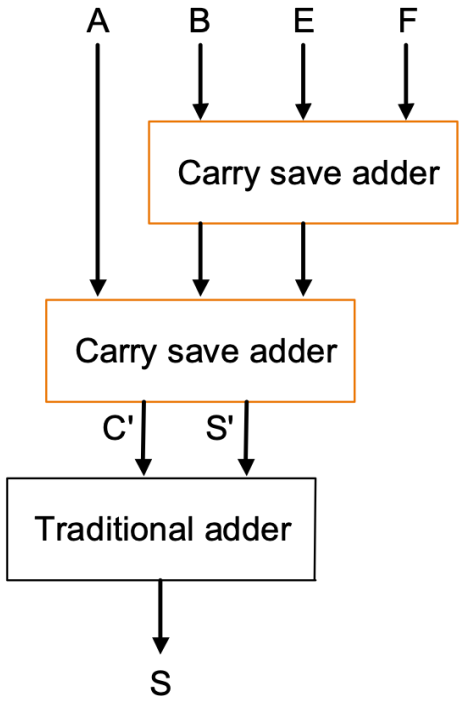




- Add four numbers (A,B,E,F)
- “Traditional adder” : ripple carry
- Basic block: Full adder
- N-1 adders for N numbers

Carry-save adders





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

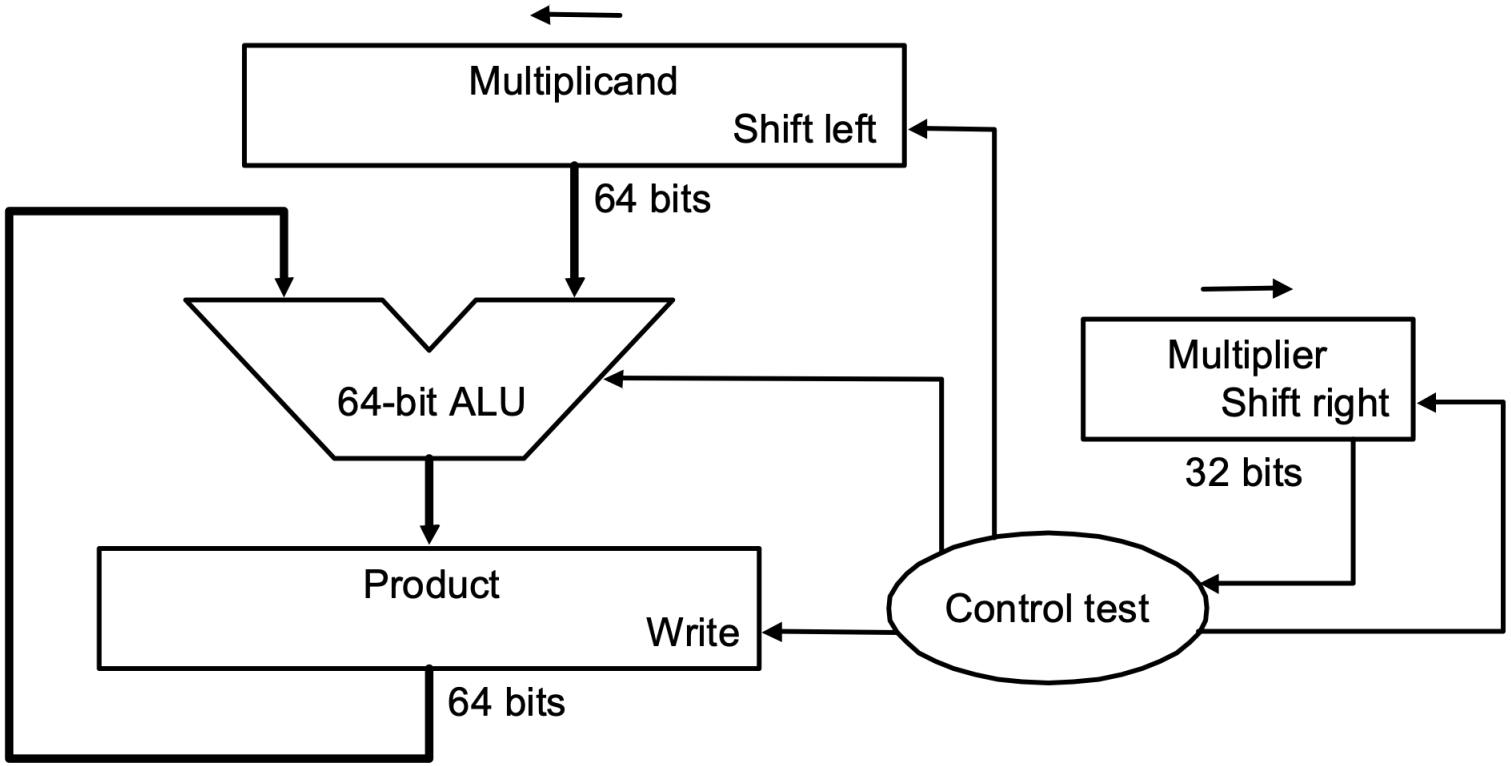
Purely combinational

- Uses carry-save adders (in a tree organization)
  - Called Wallace tree for multiplication
  - Fast but lots of hardware (quite big)
- Partial products
  - AND multiplicand with multiplier bits

Grade School Multiplication

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

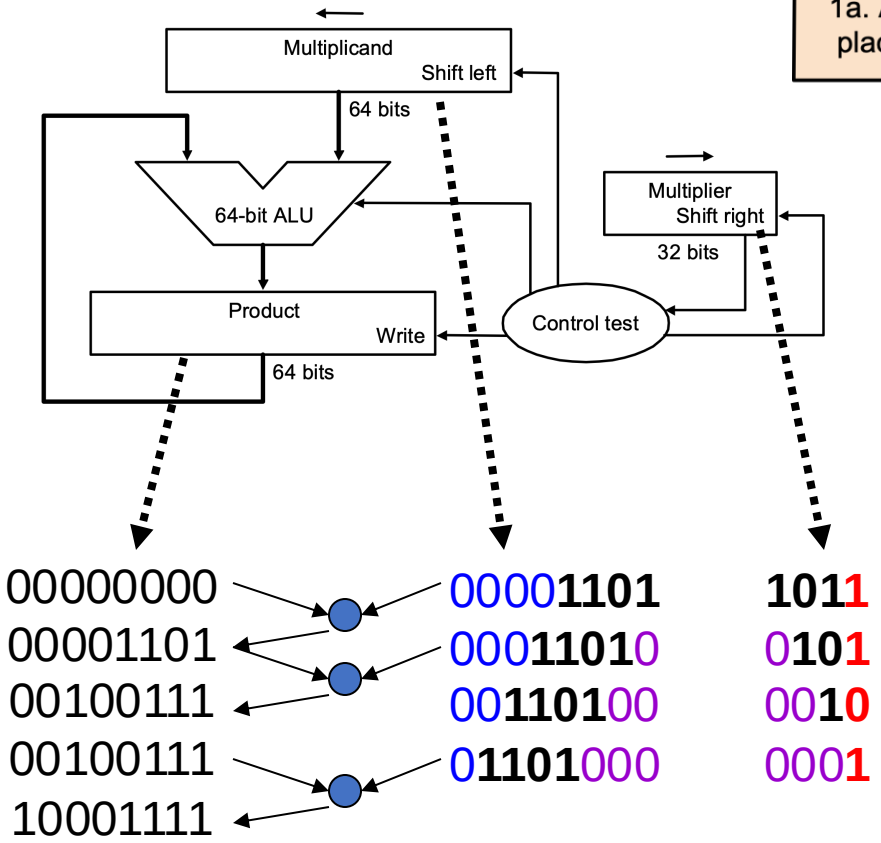
Grade-School Version

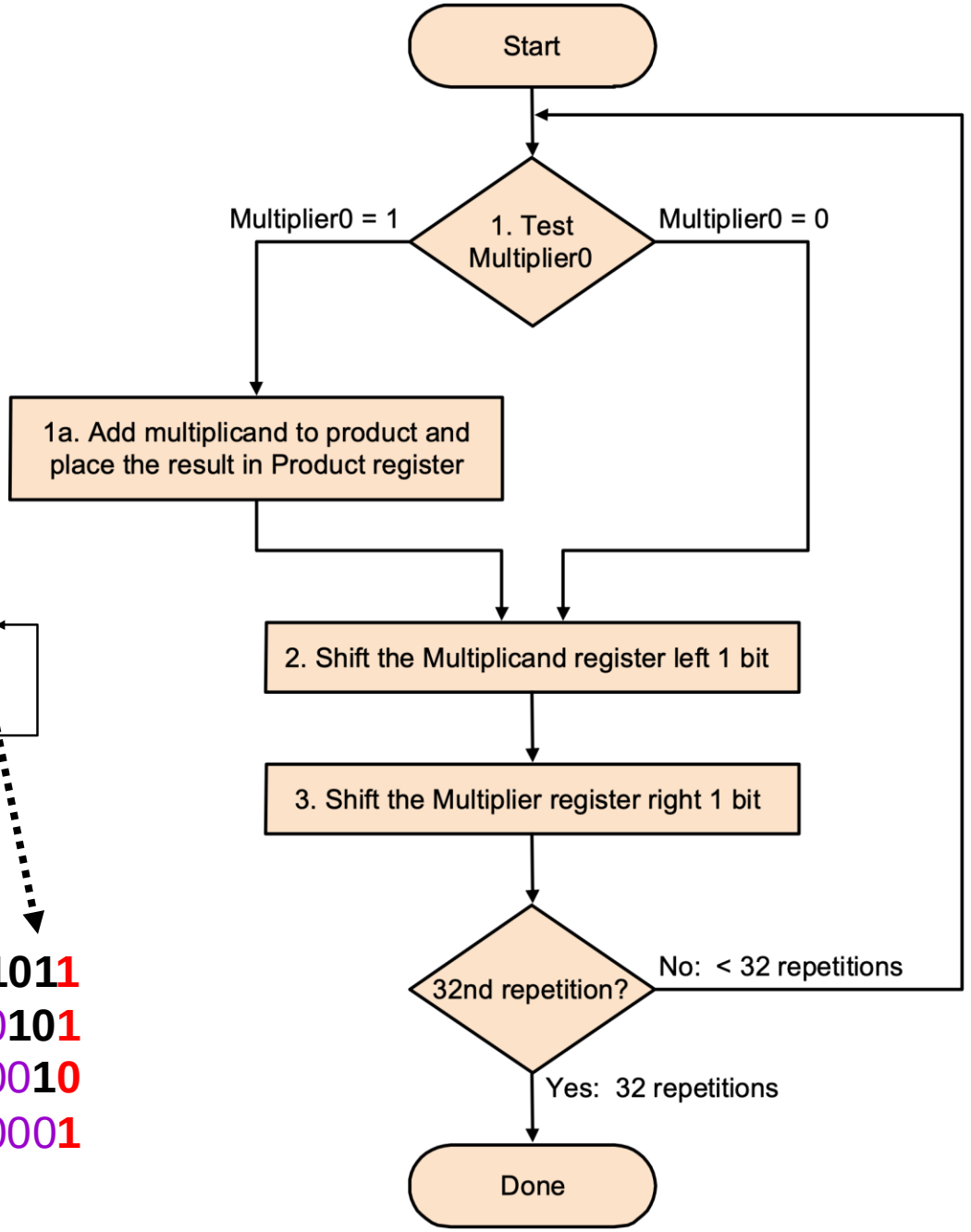


- Naïve implementation
  - 2x 64-bit registers, 1x 32 bit register
  - 1x 64-bit ALU

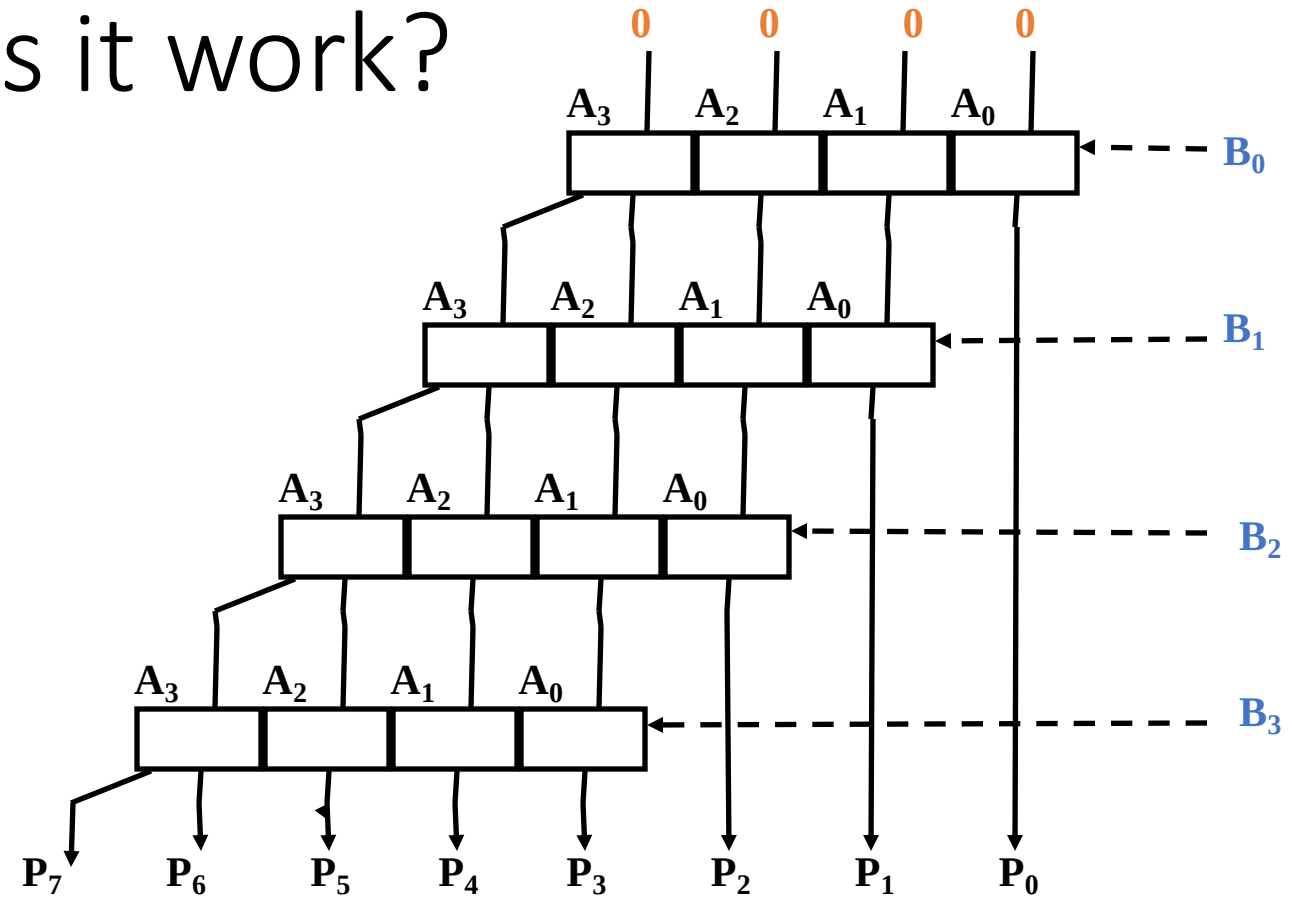
Flow chart

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



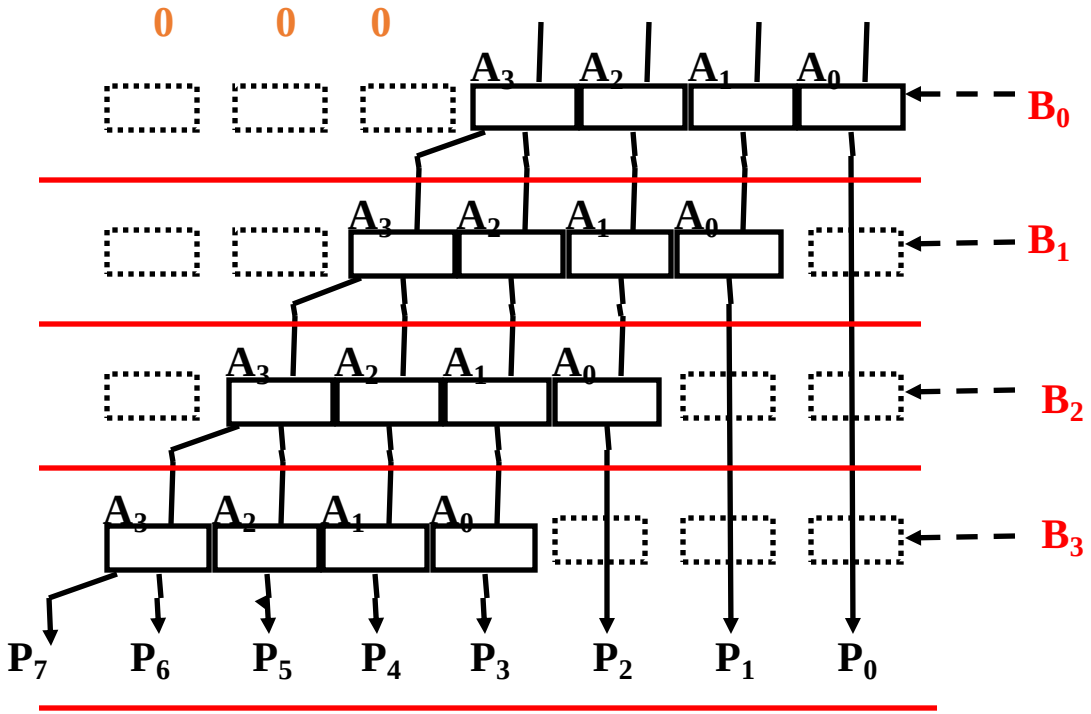


How does it work?



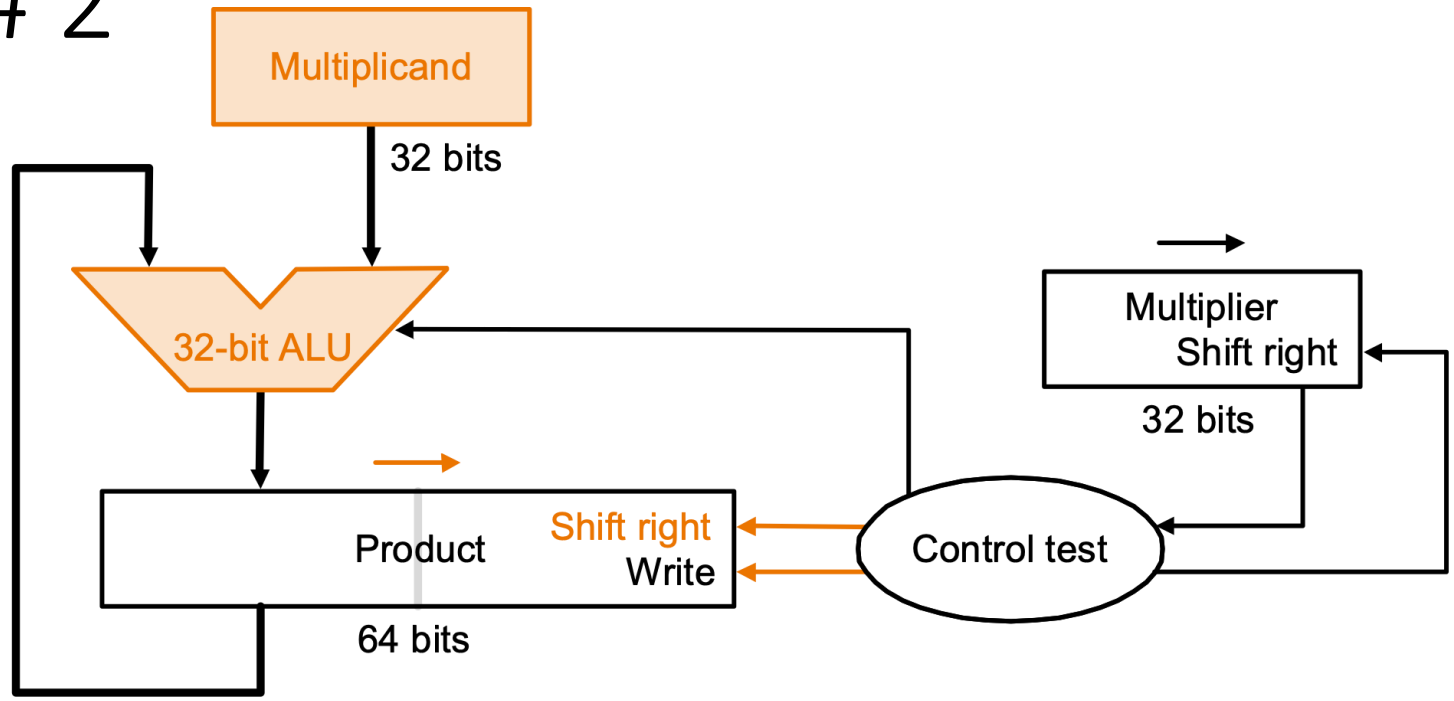
- Faithful reproduction of grade-school algorithm

Version #1



- Additional zeroes (padding)
- Only n valid bits – other bits are zeros

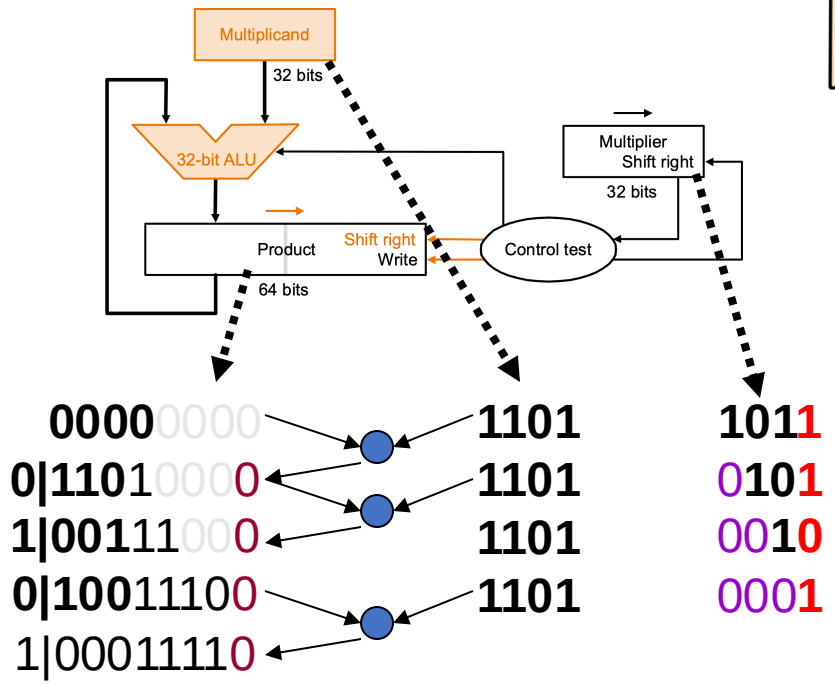
Version # 2



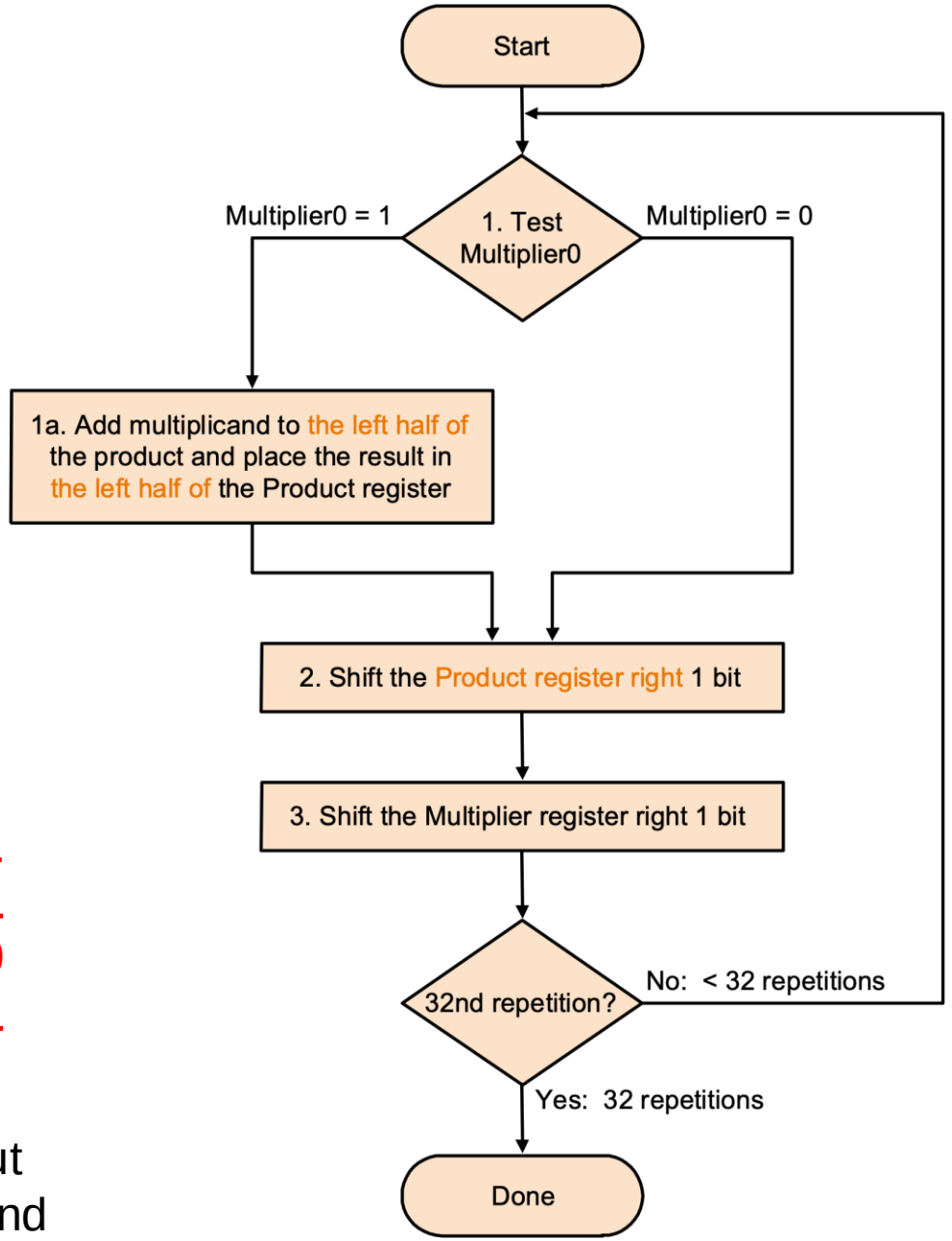
- Insight: Product LSB bits are frozen after each iteration
- Reduce 64-bit ALU to 32-bit ALU
- Very simple optimization – no big deal

Flow chart

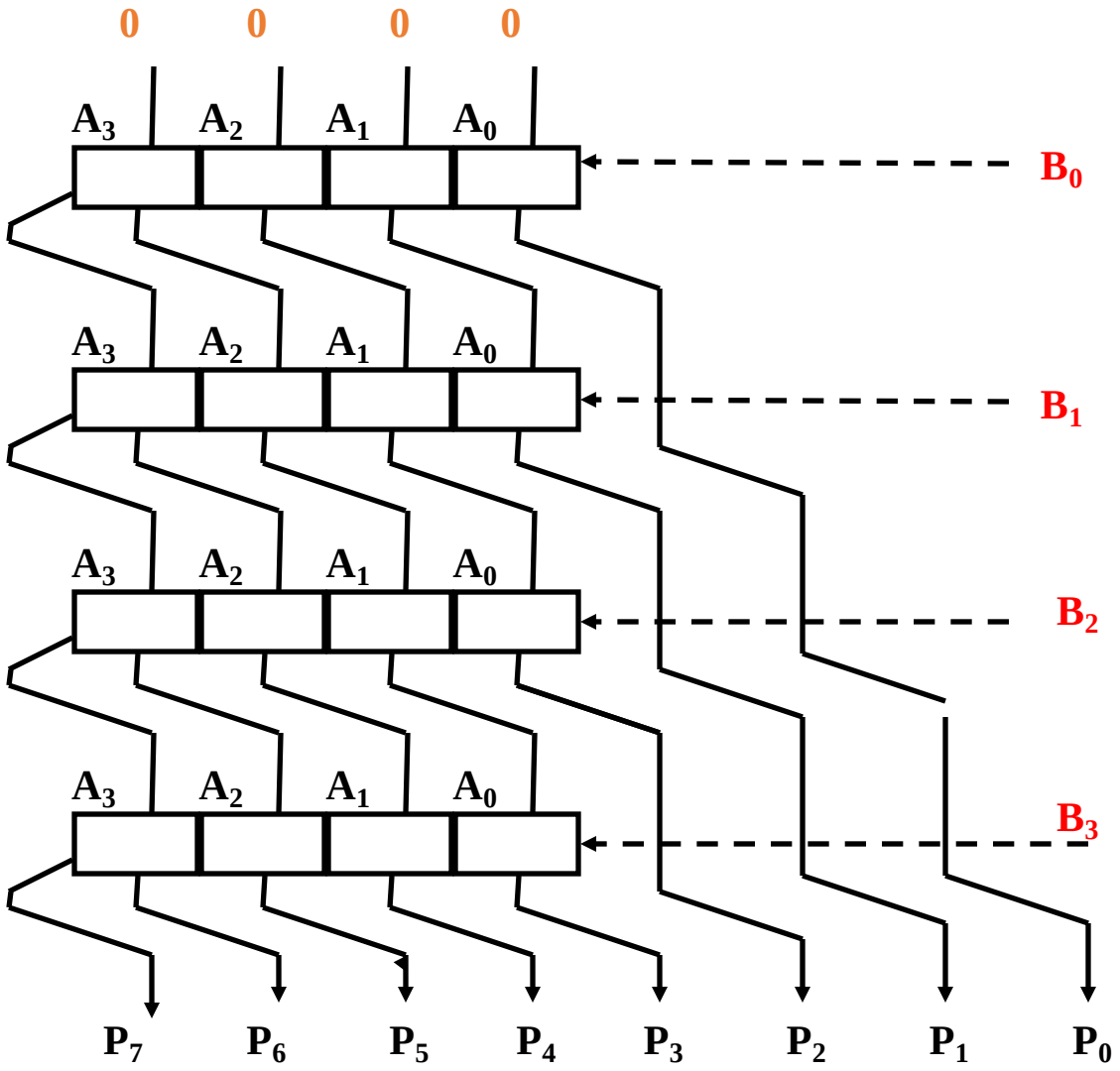
- Product register shifted right after LSB frozen
- 64-bit product computed with 32-bit adder



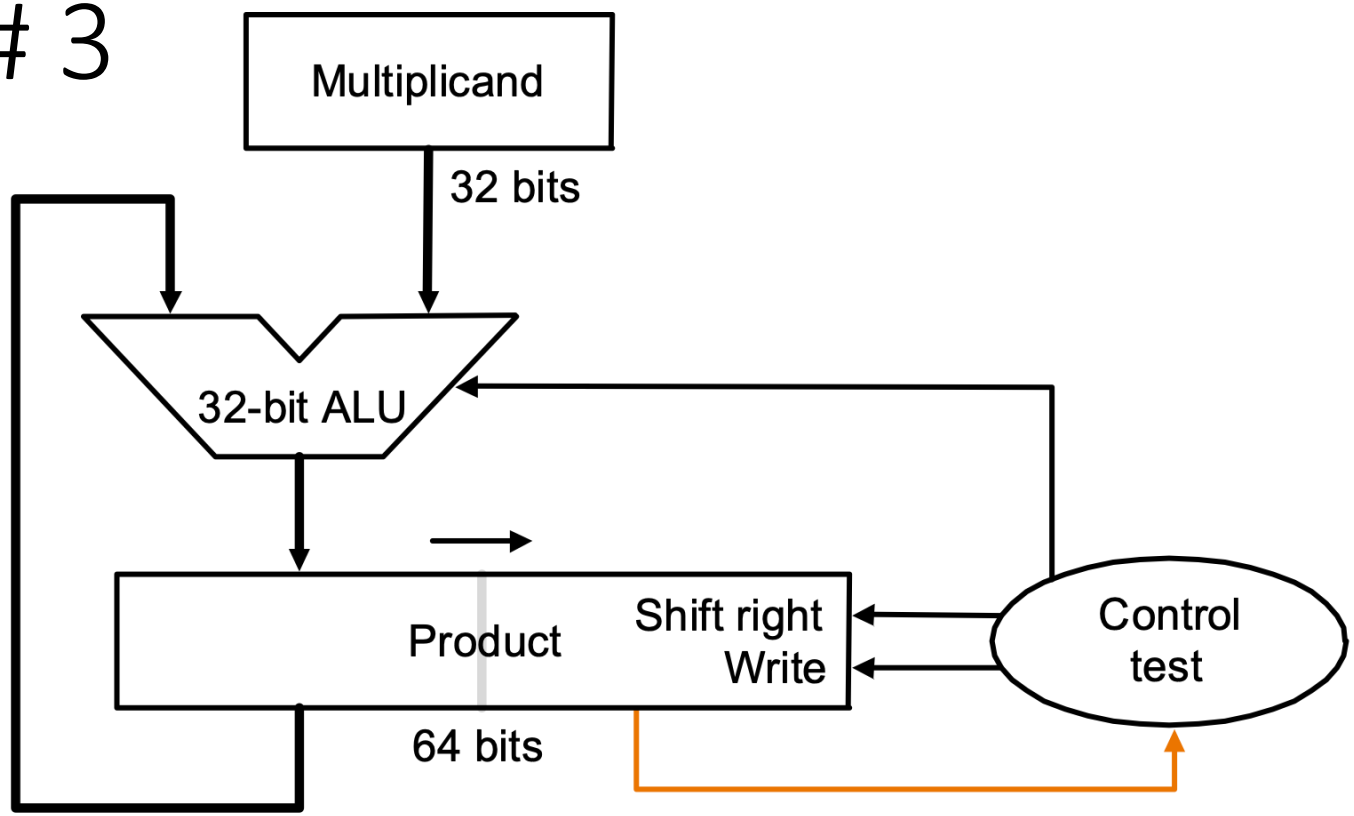
Each step shows product after add but before shift → needs a shift at the end



Flow of computation



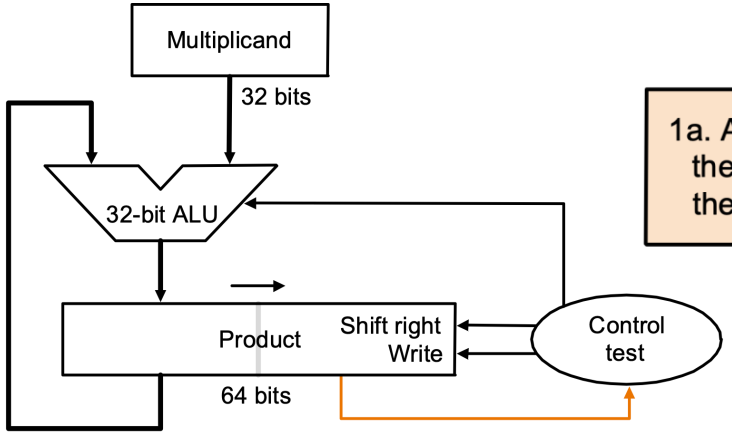
Version # 3

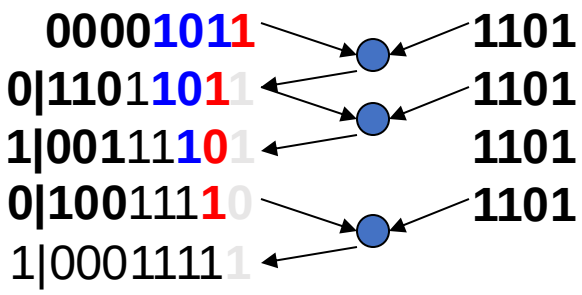


- Insight: bits of multiplier are consumed
  - Put multiplier in not-yet-used part of product
  - 1x 32-bit register, 1x 64-bit register, 32-bit ALU
  - Simple optimization – no big deal

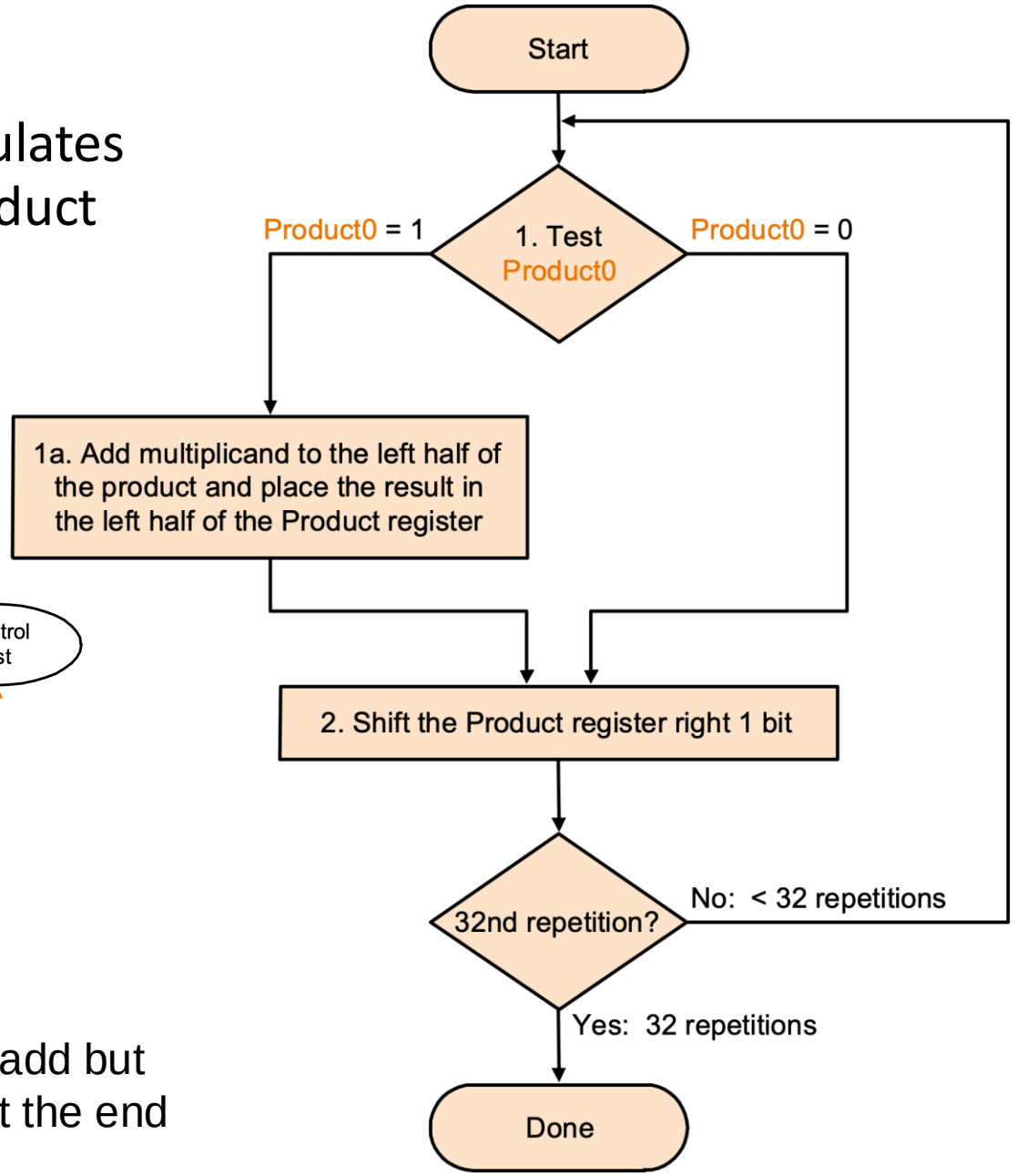
Flow chart

- Product register manipulates both multiplier and product





Each step shows product after add but before shift → needs a shift at the end



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
- Previous algo has n steps – can we do better?
- Intuition : exploit run-lengths

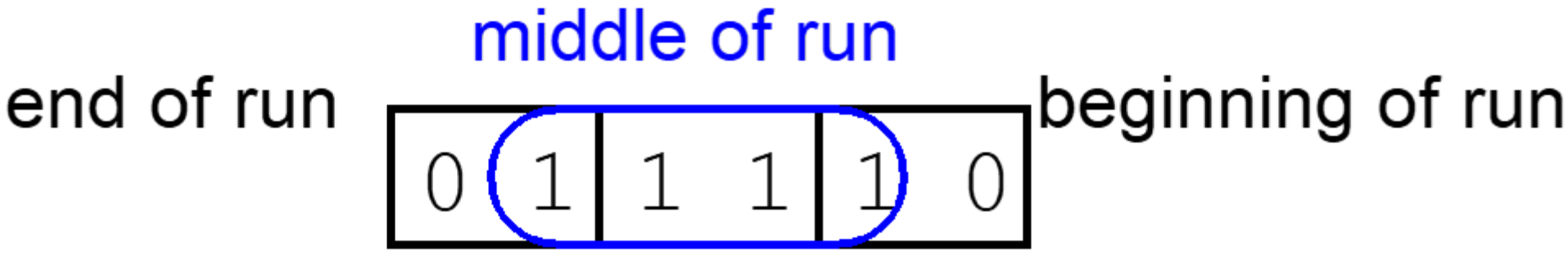
Booth's Algorithm Intuition

- 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

- “1 is the new 9”
- 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

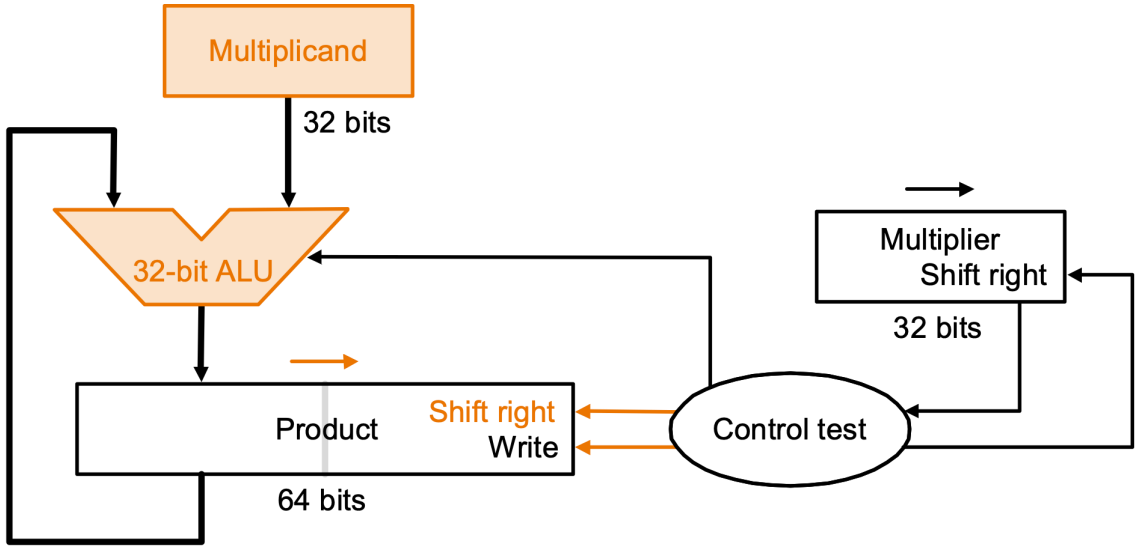
The Mathematical Basis

- 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

Booth's in our multiplier

- Change the algorithm slightly.
- Remember the last bit you shifted out (to detect run start/end)
  - if start: have ALU subtract
  - if end: have ALU add
  - else: nothing



Summary

- Integer multiplication:
  - Grade school version with minor optimizations
- Sophisticated optimizations
  - Booth's algorithms
    - Can do multiple bits at a time
    - Like CLA for addition
    - We won't go into this topic
  - Remember CLA vs. ripple-carry (grade-school version)

Recap

- Booth's Multiply Algorithm
  - Identify and exploit runs of '1's
  - Remember the analogy of multiplication by 999
    - 0110 normal encoding (4+2 = 6)
    - $10\bar{1}0$ Booth encoding (8-2 = 6)

Outline

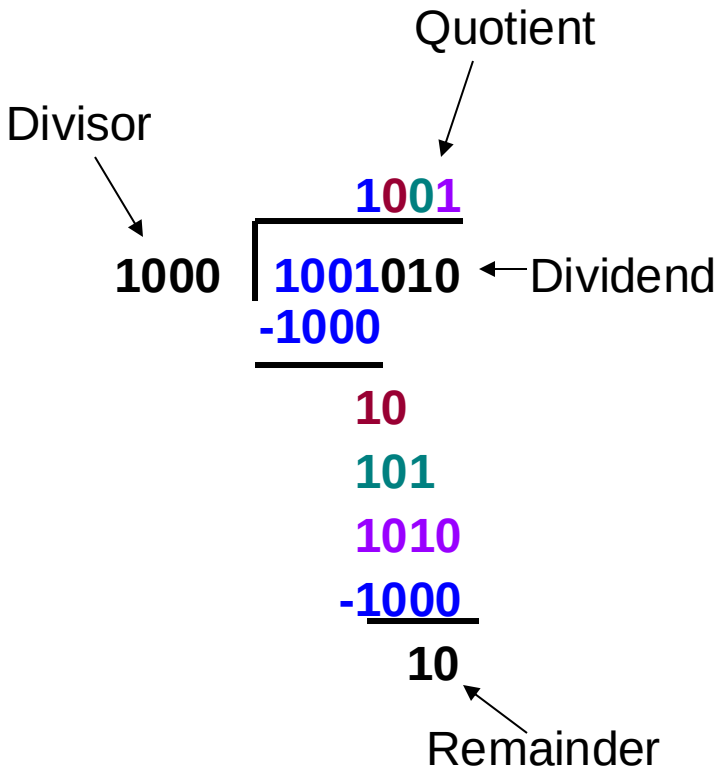
- Integer Division
  - Restoring/Non-restoring
- Floating Point Arithmetic

Integer Division

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

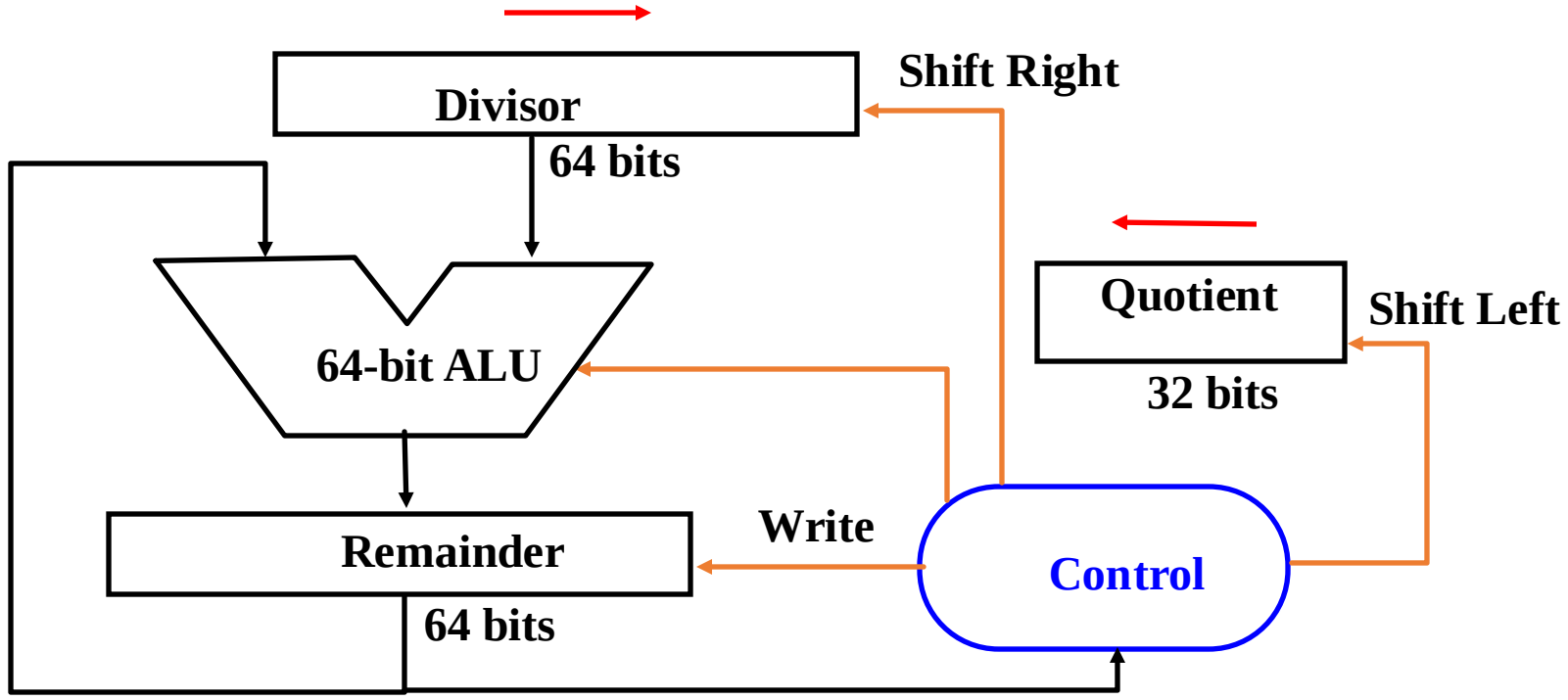
Grade School Division

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



Dividend = Quotient * Divisor + Remainder

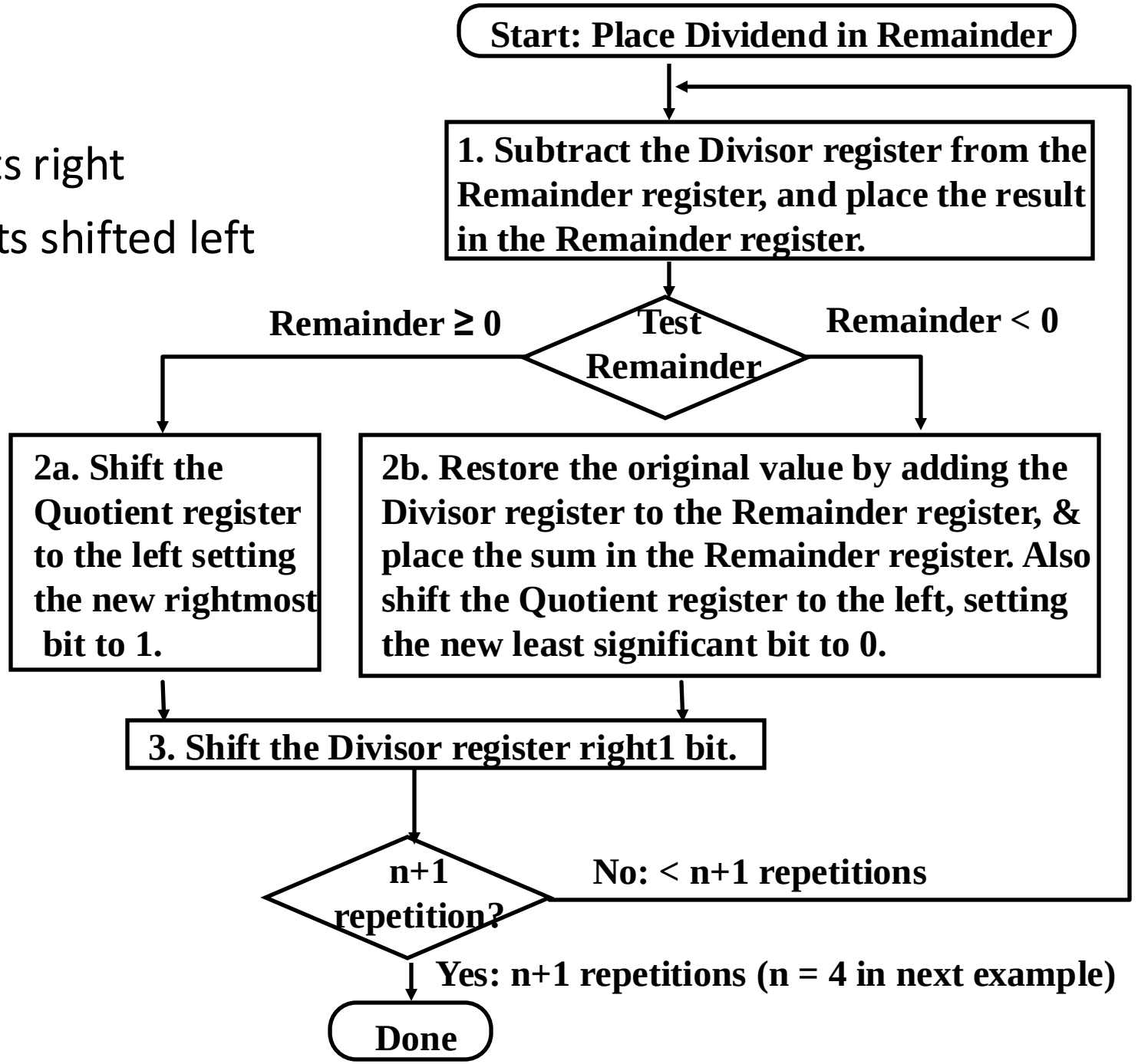
Version #1



- 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

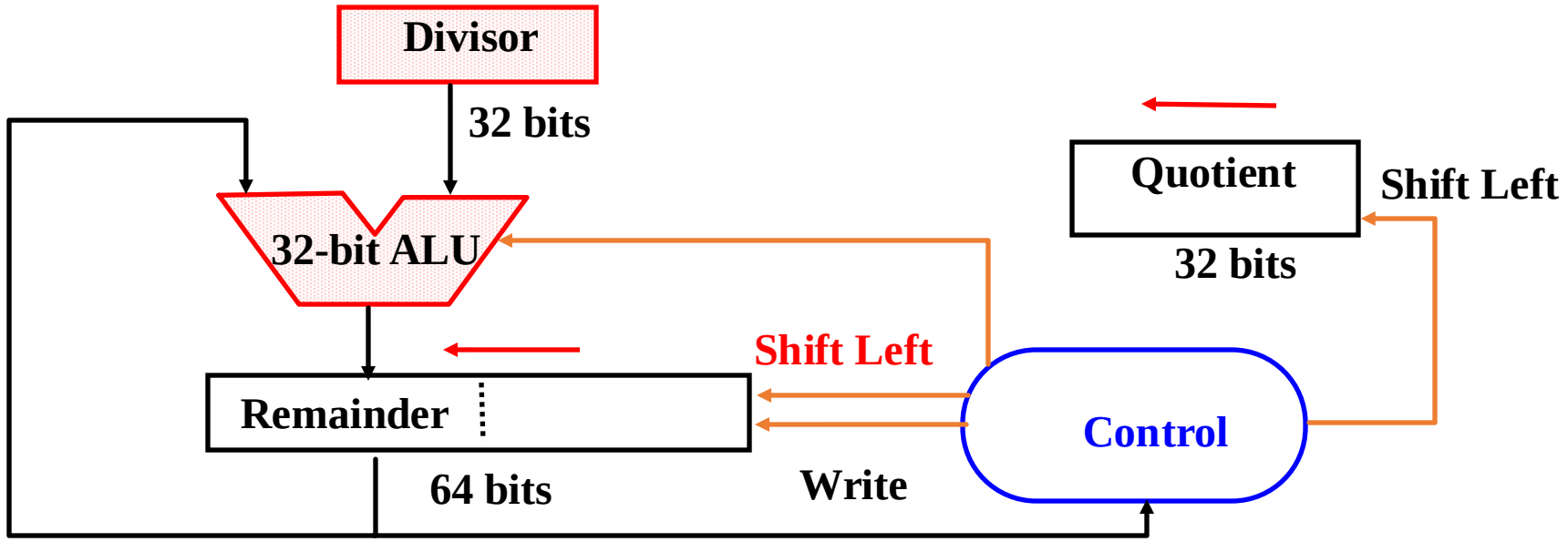
- n+1 steps for n-bit quotient
- 7 / 2 = (3,1)
- Use 8(4) bit 2's complement representation
- Register size/ALU size optimizations possible

	Remainder	Divisor	Quotient
	0000 0111	0010 0000	0000
	1110 0111	(after subtraction)	
(restore)	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

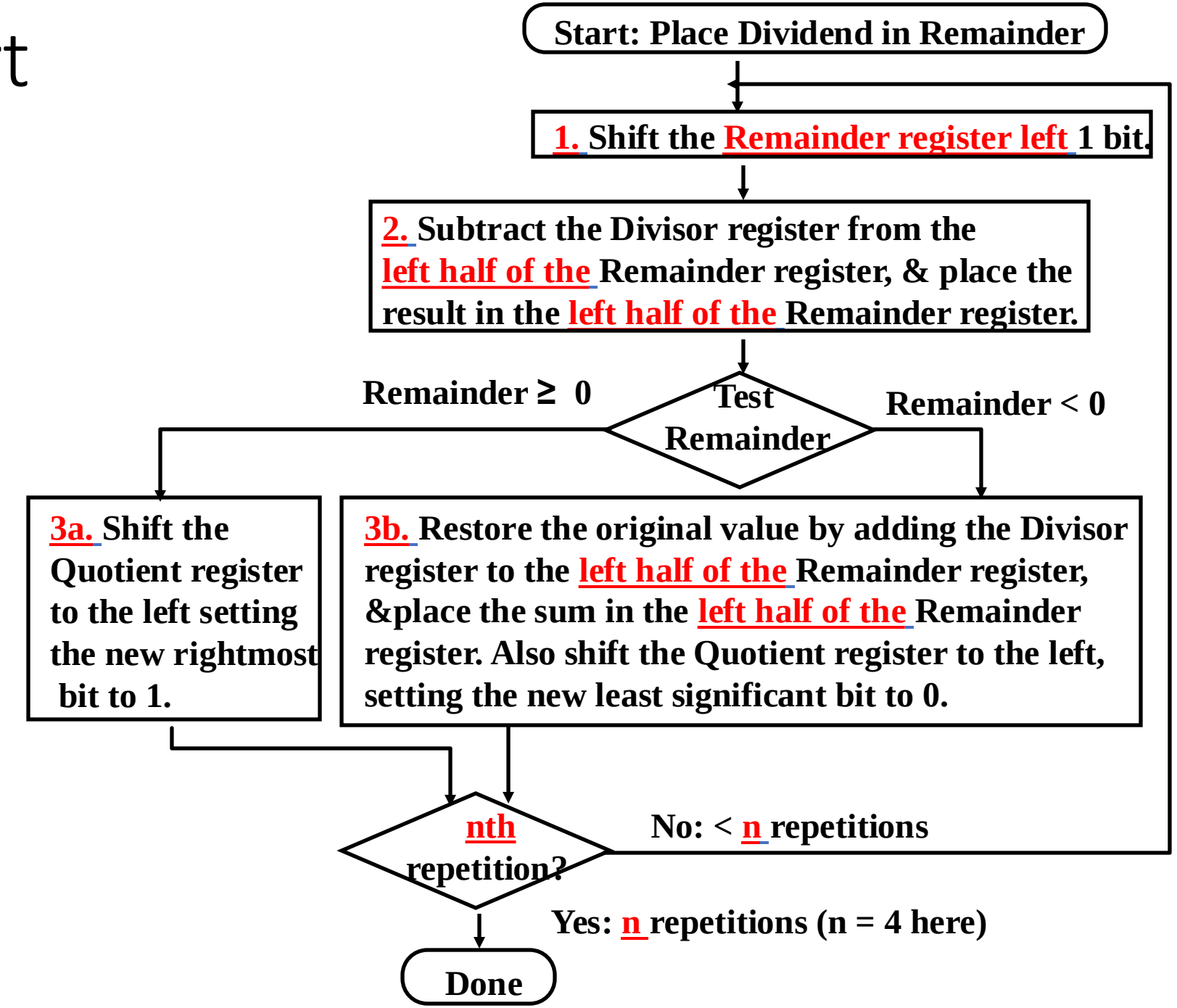
- 1/2 bits in divisor always 0
  - 1/2 of 64-bit adder is wasted
  - 1/2 of divisor is wasted
- Instead of shifting divisor to right,
  shift remainder to left?
- Has n+1 steps where 1st step cannot produce a 1 in quotient bit
  - switch order to shift first and then subtract,
    can save 1 iteration

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

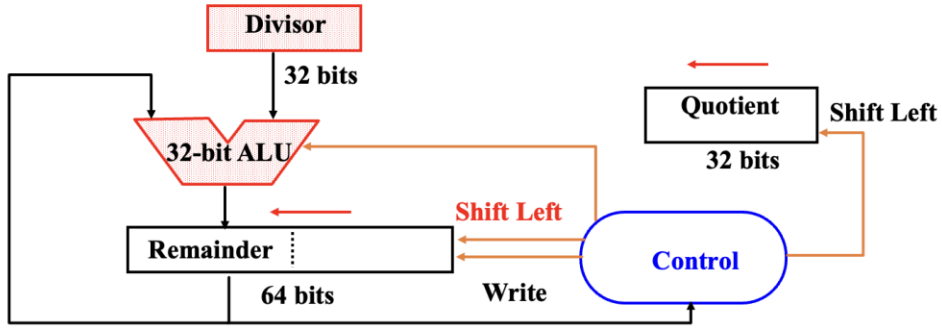
- n-steps only
- Because shift is done first

Example: 7/2

Writing the divisor as -2 to make the math easier

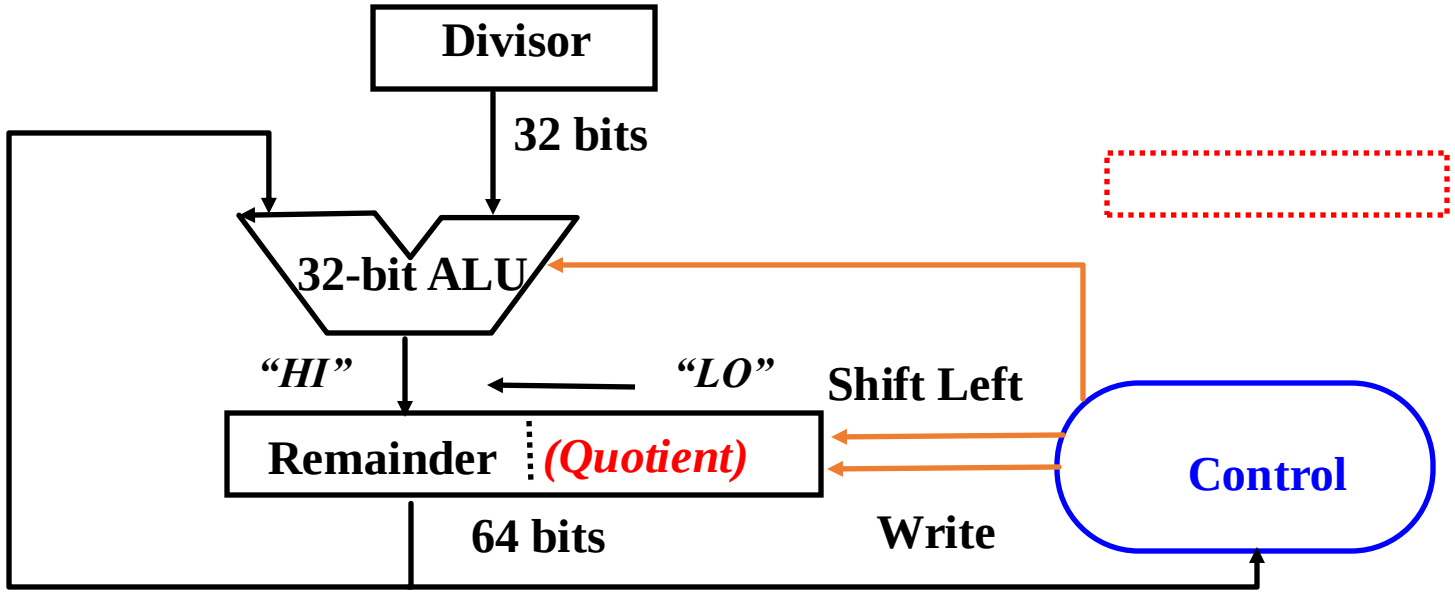
Remainder	Divisor	Quotient
0000 1110	1110	0000
1110 1110		
0000 1110	0010	0000
0001 1100	1110	
1111 1100		
0001 1100	0010	0000
0011 1000	1110	
0001 1000		0001
0011 0000	1110	
0001 0000		0011

Observations on Version #2

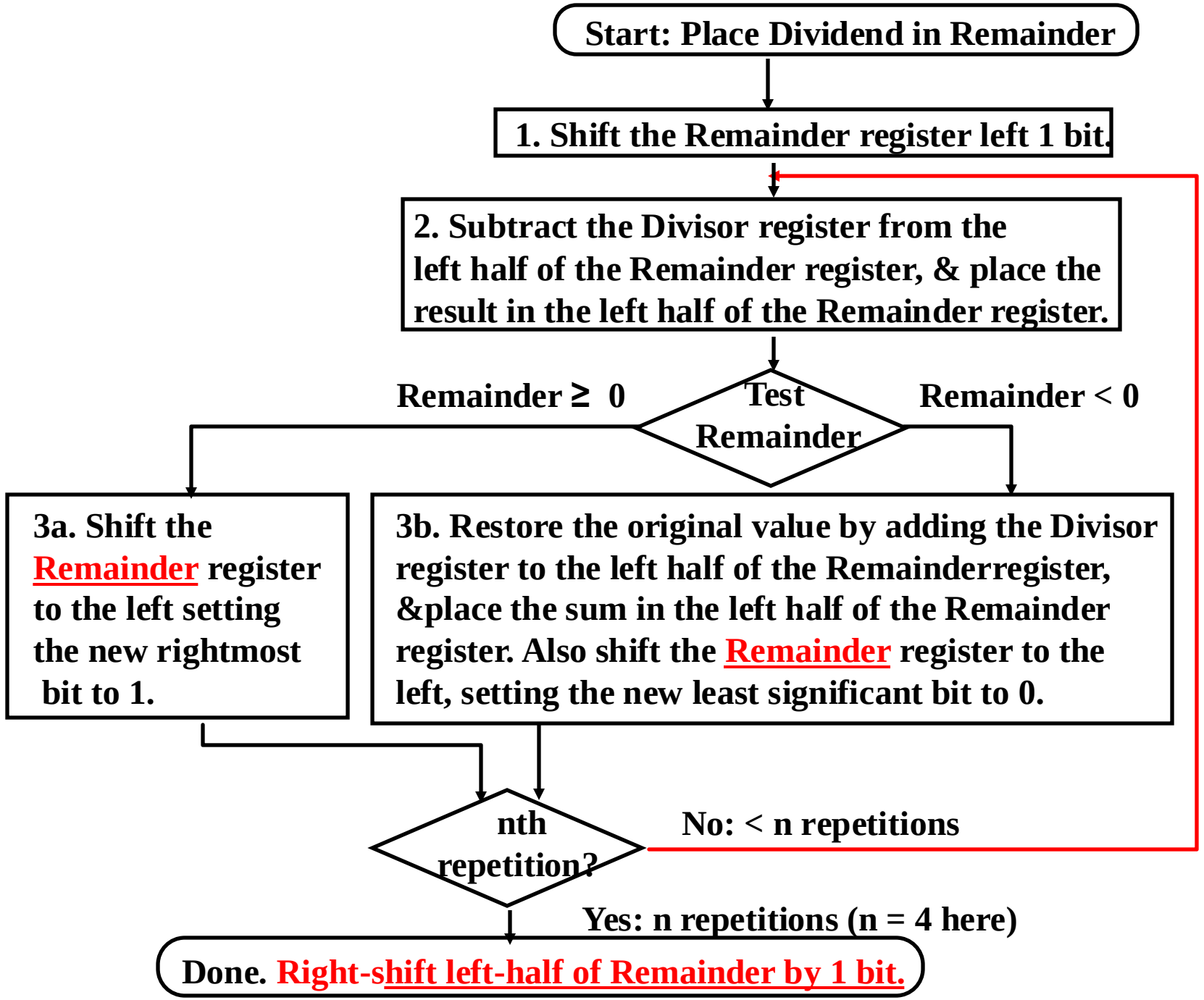


- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The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

Version #3



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


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. Right-shift left-half of Remainder by 1 bit.

Example

- Remember to right-shift the 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 $\overline{1}$ (Like Booth’s)

Non-Restoring Division (this example is based on version #1)

- 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	1111 1

If starting remainder positive then subtract and shift a quotient of 1 irrespective of the resulting remainder being positive or negative
If starting remainder negative then add and shift a quotient of 1bar irrespective of ……

Quotient = 0011 => $1 \times 2^1 + 1 \times 2^0$
Quotient = $1\overline{1}\overline{1}1$ => $1 \times 2^4 - 1x2^3 - 1x\ 2^2 - 1x\ 2^1 + 1x2^0$

Convert quotient to 2's C

- Non-restoring division uses two unique symbols for quotient
  - Can use "0" for $\bar{1}$ (but means the value is -1 and not 0)
  - To convert quotient to 2's C, use 0 for $\bar{1}$ and shift left once while shifting in a 1
  - Previous slide eg quotient = $1\bar{1}\bar{1}1$ 1 is represented as 10001
    - Final step: shift left once while shifting in a 1 to get 00011

A correction step

- If the remainder at the end is negative, then add the divisor one more time
  - can happen if the actual quotient is even or if you divide a number by a larger number (eg 3/7)
  - First convert the existing quotient to 2's complement and then adjust the quotient by subtracting a 1
- You can apply non-restoring diving to any version of the divider we have studied.
  - When you do non-restoring version#3 use arithmetic right shift for the left half to get the remainder

Issues in non-restoring

- Issues
  - Imperfect division before all bits are computed
  - Converting quotient to 2’s complement

Another restoring division

- The key problem in restoring division is the restore add
- Why not do the usual subtract but NOT store the result if negative (and set quotient to 0). Then no need to restore and no drama of 1, 1bar, conversion etc.
- Called non-performing restoring division
- But not as good as non-restoring which can handle multiple bits at a time

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 x 10^{23}
  - Normalization:
    - Not 0.6022 x 10^{24}, Not 60.22 x 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 to save bits
  - 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 ….**$_{(2)}$

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`)

Examples

- 3.125 in double precision
  - 11.001
  - $(-1)^{0} \times 1.1001 \times 2^{1}$
  - Exponent : 1 with bias of 1023 = 1024 (add bias to actual exponent to get IEEE exponent)
  - **0**100 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
  - **0**011 1111 0001 1001 1001…
  - 0x3F19 9999

FP Representation Worksheet

- Consider two floating point numbers (represented in single precision, IEEE 754 format). A = 0xBFC00000 and B = 0x41600000. If C = A x B, what is the hexadecimal representation of C in IEEE 754 format.

Solution

float : 32 bits

- A = 0xBFC00000
  = 1011 1111 1100 0000 0000 0000 0000 0000
  = -1 * 1.5 * 2^0
- B = 0x41600000
  = 0100 0001 0110 0000 0000 0000 0000 0000
  = +1 * 1.75 * 2^3
- C = -21 = -1 * 1.0101 (binary)* 2^4
  =**1***100 0001 1*010 1000 0000 0000 0000 0000
  = 0xC1A80000

Biased Exponent

- Why?
  - 2’s complement arithmetic is elegant
  - Larger numbers appear to have larger magnitude with biased representation
    - Negative numbers appear large in 2’s complement
  - Biased so that comparing two FP numbers is like comparing two unsigned integers

Exceptions

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

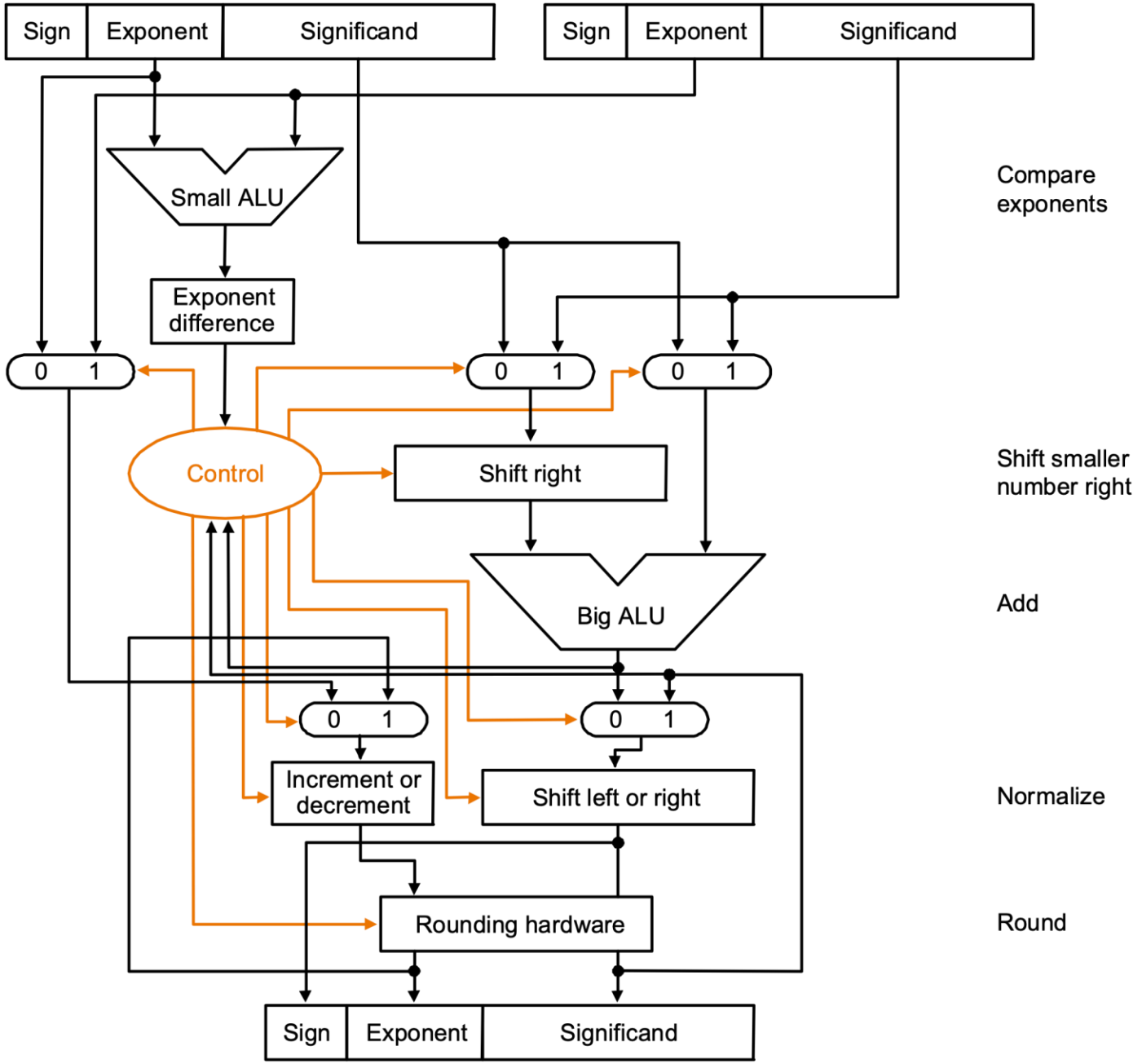
S	E	F	Number
0	0	0	0
0	Max	0	+inf
1	Max	0	- inf
X	Max	!= 0	NaN

Floating Point Addition

$$9.997 \times 10^{2}$$
$$+ 4.6371 \times 10^{-1}$$

- FP addition
  - Align decimal point → 9.997×10^{2} / $+ 0.0046371 \times 10^{2}$
  - Add → 10.0016371×10^{2}
  - Normalize → 1.00016371×10^{3}
- May involve rounding
  - The LSB may be shifted right beyond the limited precision

- FP Addition Hardware



Normalization issues

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

Adding exponents

- Biased representation
- Adding biased numbers
- Raw number

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

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

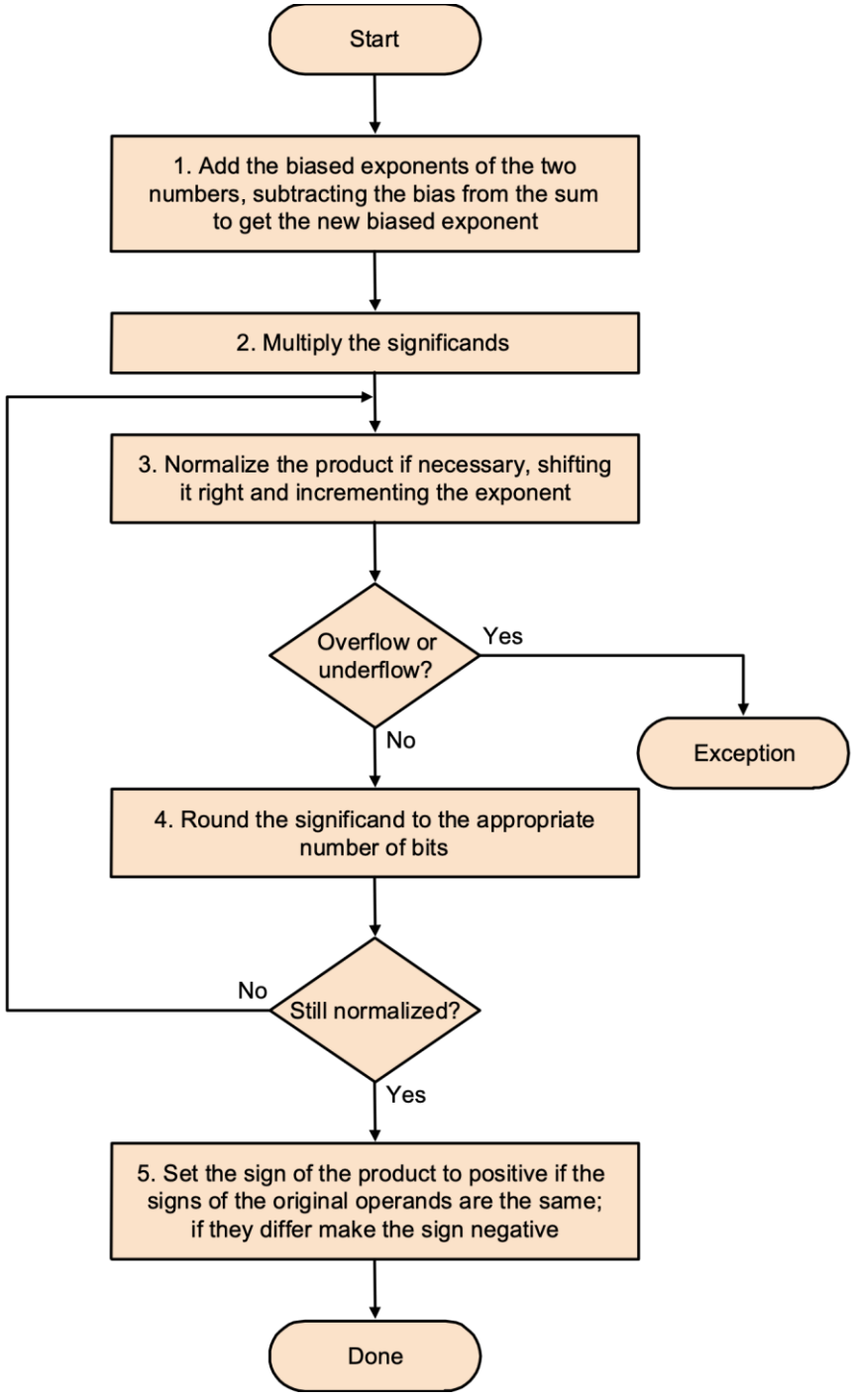
  $E_* = e_*+127 = E_1 - 127 + E_2 - 127 + 127$

  $E_* = E_1 + E_2 - 127$

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

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

FP Multiplication



FP Multiplication

- Mantissa multiplication
  - Two non-negative integers
  - Recall
    - Carry save adders in tree (Wallace tree)
    - Booth’s multiplication

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 $(E_2) + 1000\ 0000$

139_{10}: 1000 1011
140_{10}: 1000 1100

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
- E.g., round to nearest hundredth

  1.2349999 1.23 (Less than half way)

  1.2350001 1.24 (Greater than half way)

  1.2350000 1.24 (Half way—round up)

  1.2450000 1.24 (Half way—round down)

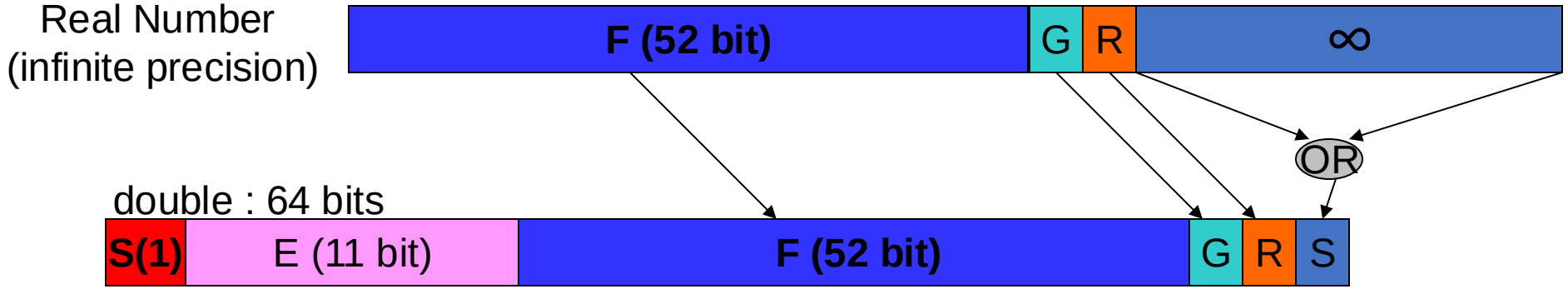
Binary Rounding

- Binary rounding
  - x.xxx1…1… -> up
  - x.xxx10000 -> round to even (LSB=0)
  - x.xxx0xx1x -> down
  - x.xxx00000 -> unchanged

- Eg.: Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

Guard, Round and Sticky bits



- After compute, may shift to renormalize
  - Save one bit to right of LSB (Guard bit)
  - 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 ½ “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?
- Ch3 done!