National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 2: Hardware

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

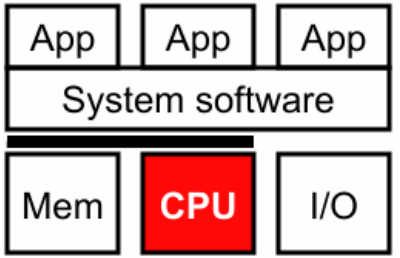
- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - EEC 170 at University of UC Davis
    - https://www.ece.ucdavis.edu/~soheil/private/EEC170/

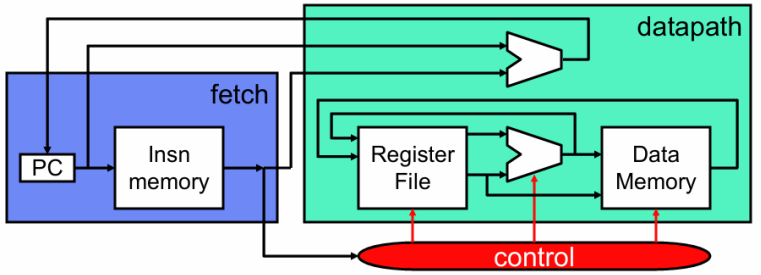
Outline

- Binary Arithmetic
- Adder
- Subtraction
- Multiplier
- Division

Central Processing Unit (CPU)

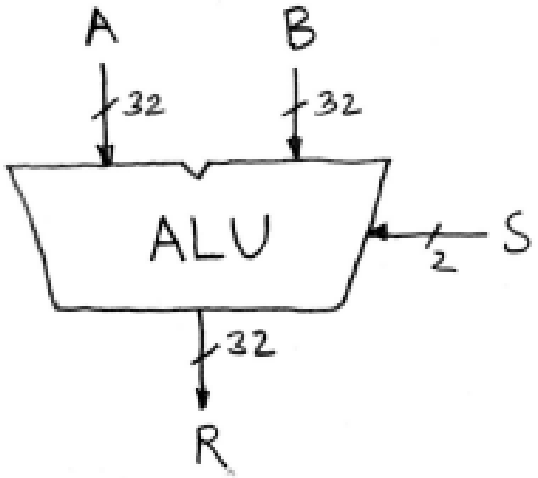
- Inside a CPU
  - Datapath: performs computation
    - Includes registers, ALUs …
  - Control: determines which computation is performed
    - Routes data through datapath
  - Fetch: get insn, translate opcode into control
  - Fetch->Decode->Execute “cycle”





Arithmetic Logic Unit (ALU)

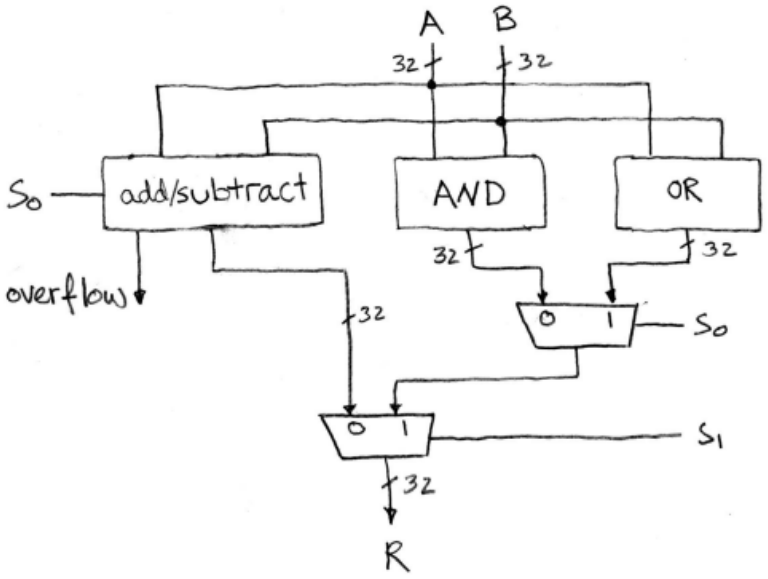
- The ALU is used to compute the result of R-type instructions (ADD, SUB, ADDI, AND, OR)



When S = 00, R = A + B
When S = 01, R = A – B
When S = 10, R = A & B
When S = 11, R = A | B

Arithmetic Logic Unit (ALU)

- The ALU is used to compute the result of R-type instructions (ADD, SUB, ADDI, AND, OR)
  - A 32-bit bitwise AND unit
  - A 32-bit bitwise OR unit
  - A 32-bit ADD/SUBTRACT unit with a control line
  - The logic to output carry
  - Overflow
  - Zero
  - Negative



Binary Arithmetic

- Computers represent integers in binary (base2)
  - $3 = 11_2$, $4 = 100_2$, $5 = 101_2$, $30 = 11110_2$
- Addition take place as usual (carry the 1, etc.)

```
17 =    10001
+5 =      101
-------------
22 =    10110
```

Carry out from the second column / Carry into the third column

Carry in to the second column

```
  111
  101
+ 111
-----
 100
```

Binary Arithmetic

- In hardware, integers have fixed width
  - N bits: 16, 32, or 64
  - LSB is 2^0, MSB is 2^{N-1}
  - Range: 0 to 2^N-1

Binary Arithmetic

- What about negative integers in binary numbers
  - Unsigned plus one bit for sign
    - 10 = 000001010, -10 = 100001010
    - Range: $-(2^{N-1}-1)$ to $2^{N-1}-1$
  - Option II: two's complement (2C)
    - Leading 0s mean positive number, leading 1s negative
    - 10 = 00001010, -10 = 11110110
    - + One representation for 0 (all zeros)
    - + Easy addition
    - Range: $-(2^{N-1})$ to $2^{N-1}-1$

Binary Arithmetic

- Sign Magnitude Representation
  - The Most Significant bit of the number if a sign bit
  - The remaining bit represents the magnitude of the number in a binary form

MSB **Magnitude**

0 0 1 0 0 0 1 0

- Example: 8-bit sign-magnitude form

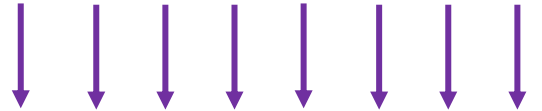
+34 = 0 0 1 0 0 0 1 0

-34 = 1 0 1 0 0 0 1 0

Binary Arithmetic

- 1's Complement Representation
  - The representation of the negative number is different from the positive number representation
  - Example: The represent -34 in 1's complement form

+34 = 0 0 1 0 0 0 1 0



-34 = 1 1 0 1 1 1 0 1

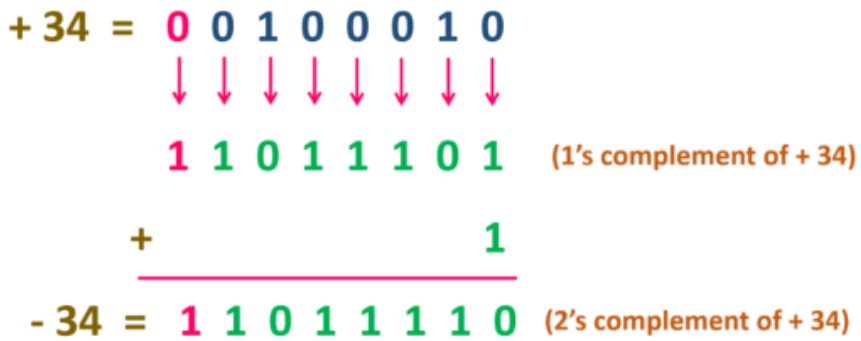
Invert all 1s in that number by 0s and 0s by 1s

Binary Arithmetic

- 2’s Complement Representation
  - The representation of the positive number as the 1’s complement form
- Translate negative number from 1’s complement to 2’s complement form
  - Write the number corresponding to +34
  - Find 1’s complement of +34
  - Add 1 to the 1’s complement number

Binary Arithmetic

- 2's Complement Representation
  - Translate negative number from 1's complement to 2's complement form
  - Write the number corresponding to +34
  - Find 1's complement of +34
  - Add 1 to the 1's complement number



Takeaway Questions

- What is the 2C representation in the following numbers?
  - -1
  - 1
  - 0

Takeaway Questions

- Still more on 2C
  - Trick to negating a number quickly: -B = B’ + 1
    - -(1) = (0001)’ + 1 = 1110 + 1 = 1111 = -1
    - -(-1) = (1111)’ + 1 = 0000 + 1 = 0001 = 1
    - -(0) = (0000)’ + 1 = 1111 + 1 = 0000 = 0

Understanding of overflow

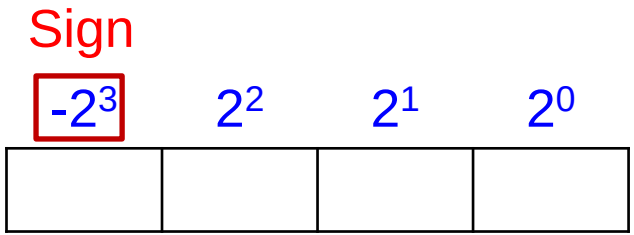
- Carry indicates overflow

	1	0	1	1	7(DEC)
+	0	1	1	1	11(DEC)
1	0	0	1	0	19(DEC)

Overflow -> 19 is out of the range of the 4-bit value representation (0-15)

A	B	C	D	Unsigned
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Overflow in Signed numbers (2C)

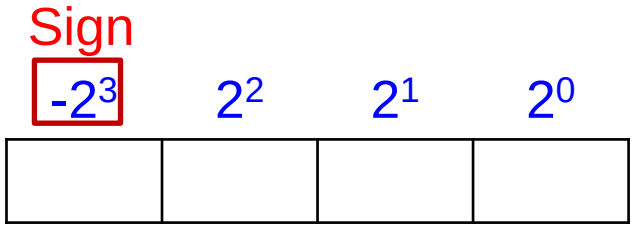


The range of 4-bits signed number
-2^{n-1} <-> $(2^{n-1} - 1)$ ==> -8 <-> 7

	1	0	0	1	-7(DEC)
+	1	1	0	1	-3(DEC)
1	0	1	1	0	-10(DEC)

Overflow !

Overflow in Signed numbers (2C)

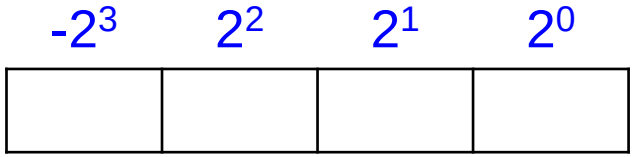


The range of 4-bits signed number
-2^{n-1} <-> $(2^{n-1} - 1)$ ==> -8 <-> 7

	0	1	1	1	7(DEC)
+	0	0	0	1	1(DEC)
	1	0	0	0	-8(DEC)

Overflow !

Overflow in Signed numbers (2C)

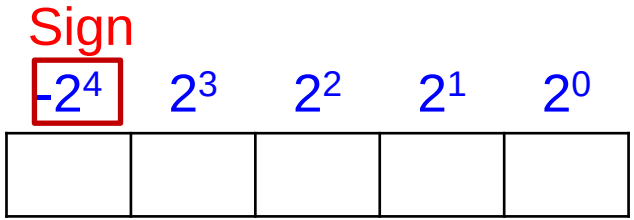


The range of 4-bits signed number
-2^{n-1} <-> $(2^{n-1} - 1)$ ==> -8 <-> 7

	0	1	1	1	7(DEC)
+	0	0	0	1	1(DEC)
	1	0	0	0	-8(DEC)

Overflow ! How to fix this problem?

Overflow in Signed numbers (2C)



The range of 5-bits signed number
-2^{n-1} <-> $(2^{n-1} - 1)$ ==> -16 <-> 15

	0	0	1	1	1	7(DEC)
+	0	0	0	0	1	1(DEC)
	0	1	0	0	0	8(DEC)

Extend 4-bit value to 5 bits to hold the correct result

What is sign extension?

- Sign-extension
  - Copying the sign bit of the un-extended value to all bits on the left side of the larger-size value
  - **SEXT** instruction widens the data while maintaining its sign and value.
  - e.g. widen the data while maintaining its sign and value
  - Unsigned number, converts positive values, provided the sign bit is zero

01001000 <- 8-bit value of 72
00000000 01001000 <- extended to 16-bit value
00000000 00000000 00000000 01001000 <- extended 32-bit value

What is sign extension?

- 8-bit encoding of decimal signed number -56 can be sign-extended as follows:

Sign
00111000 <- 8-bit value of 56
11000111 <- 8-bit value of -56 (1's complement)
11001000 <- 8-bit value of -56 (2's complement)
11111111 11001000 <- extended to 16-bit value
11111111 11111111 11111111 11001000 <- extended 32-bit value

Binary Addition

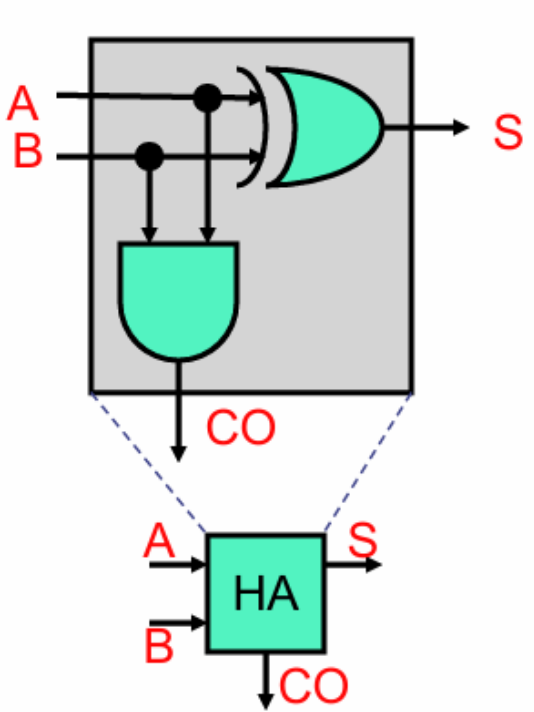
- Repeat N times
  - Add least significant bits and any overflow from previous add
  - Carry the overflow to next addition
  - Shift two addends and sum one bit to the right
- Sum of two N-bit numbers can yield an N+1 bit number
  - - More steps (smaller base)
  - + Each one is simple (adding just 1 and 0)
    - So simple, then we can do it in hardware

```
  1      111111
 43 = 00101011
+29 = 00011101
 72 = 01001000
```

The Half Adder

- How to add two binary integers in hardware?
- Start with adding two bits
  - S = A^B
  - CO (Carry out) = AB
  - This is called a half adder

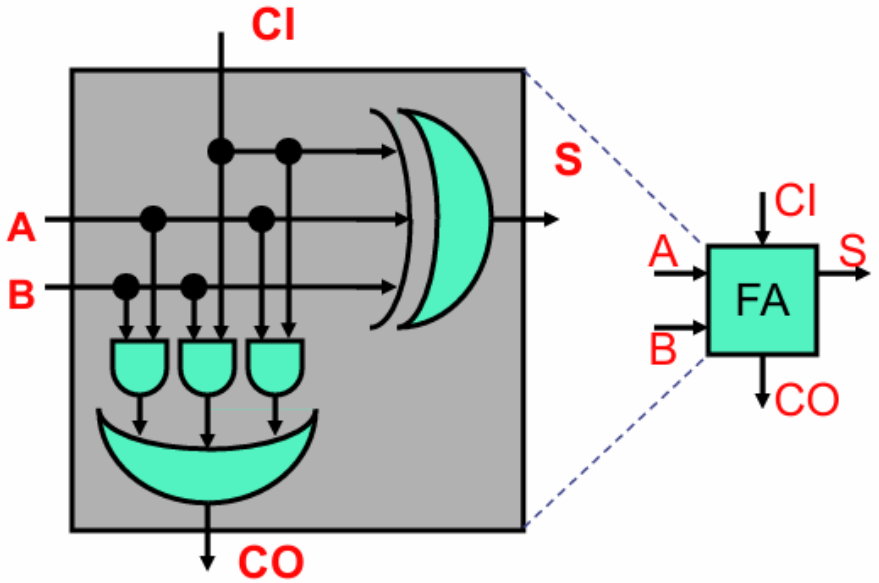
A	B	=	CO	S
0	0	=	0	0
0	1	=	0	1
1	0	=	0	1
1	1	=	1	0



The Full Adder

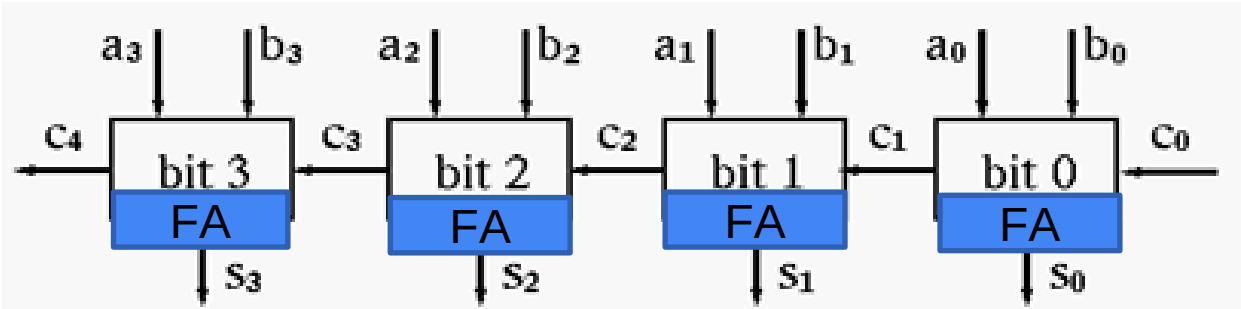
- The full adder
  - S = C'A'B+C'AB' + CA'B' + CAB = C^A^B
  - CO = C'AB+CA'B+CAB'+CAB = CA + CB + AB

C	A	B	=	CO	S
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



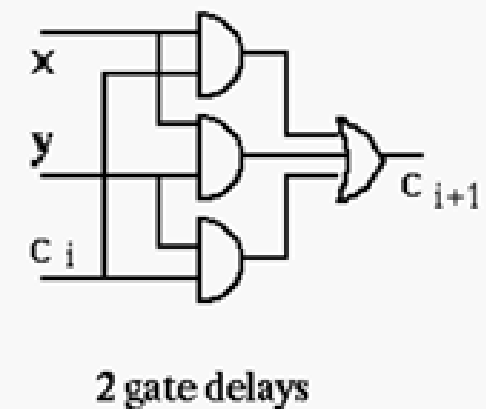
Ripple-Carry Adder (RCA)

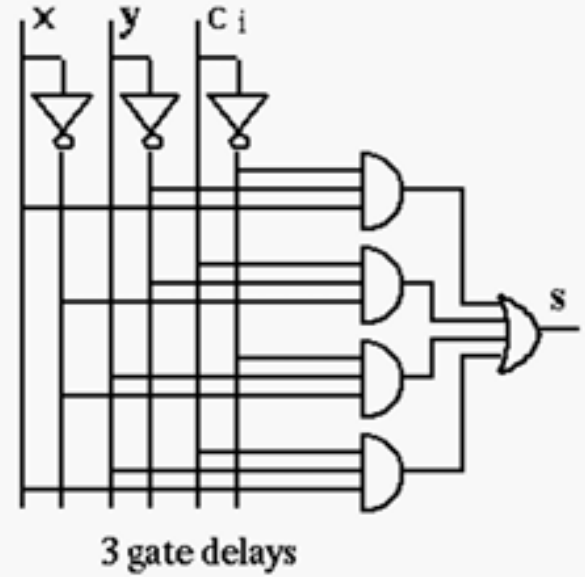
- N-bit ripple-carry adder (RCA)
  - N 1-bit full adders (FA) “chained” together
  - As the carry c_i needs to be passed on through all lower bits to compute the sums for the higher bits



Ripple-Carry Adder (RCA)

- Gate delay of FA

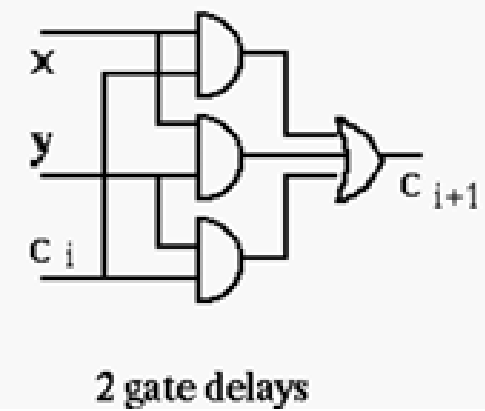


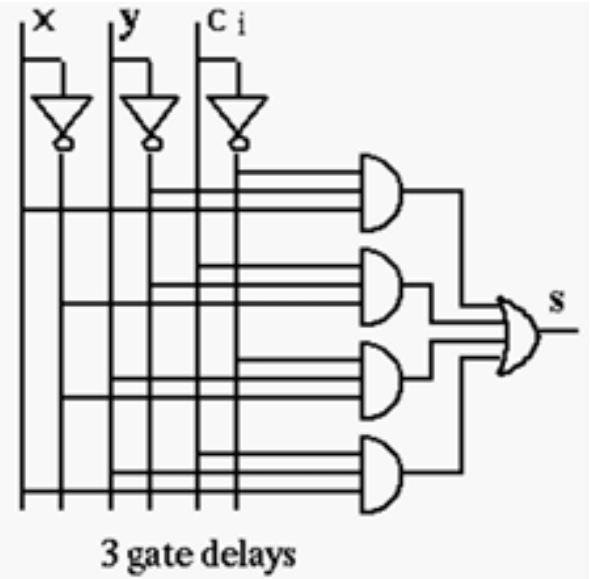


- $C_i+1 = x_iy_i + x_ic_i+y_ic_i$ (2 gate delays)
- $S_i = x'_iy'_ic_i + x'_iy_ic'_i+x_iy'_ic'_i+x_iy_ic_i$ (3 gate delays)

Ripple-Carry Adder (RCA)

- Gate delay of RCA

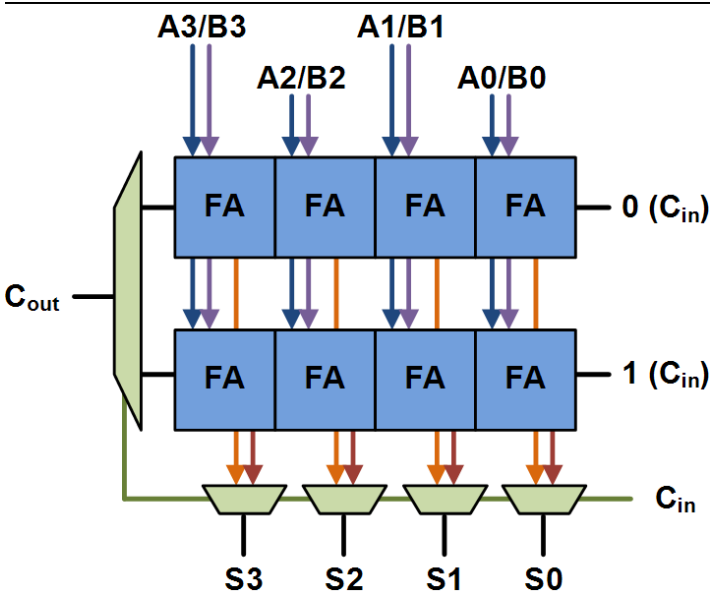




  - The total gate delays of n-bit sum of RCA is 2(n-1)+3
    - When n = 64, there will be 129 gate delays
    - When n = 16, there will be 33 gate delays

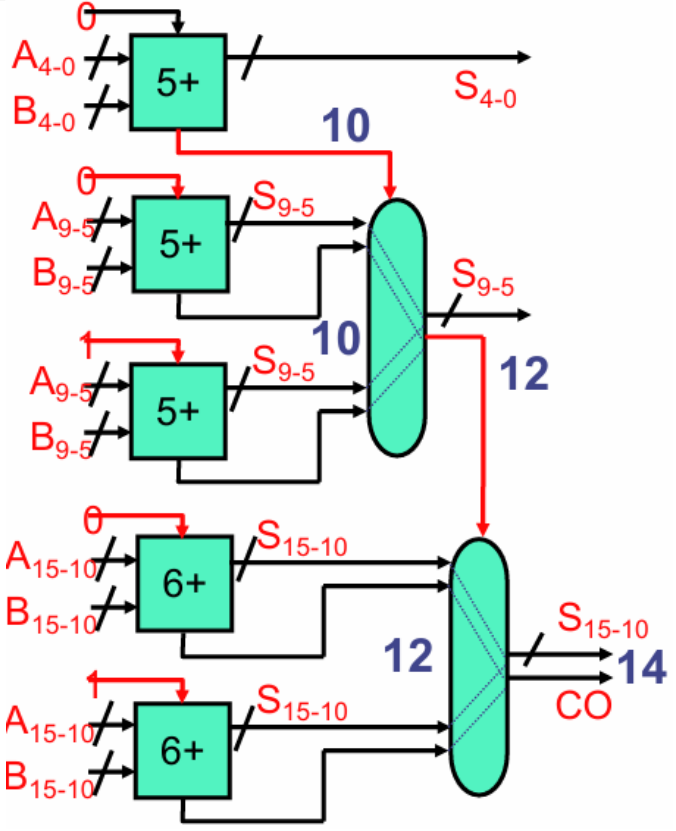
Carry-Select Adder (CSA)

- Consists of RCAs and a multiplexer
  - Compute the (n + 1)-bit sum of two n-bit numbers
  - Simple but fast, having a gate level depth of O(sqrt(n))



Carry-Select Adder (CSA)

- Multi-Segment CSA
  - Example: 5, 5, 6 bit = 16 bit
  - Hardware cost
    - Compute each segment with 0 and 1 carry-in
    - Serial mux chain
  - Delay
    - 5-bit adder (10) + Two muxes (4) = 14



Carry-Select Adder (CSA)

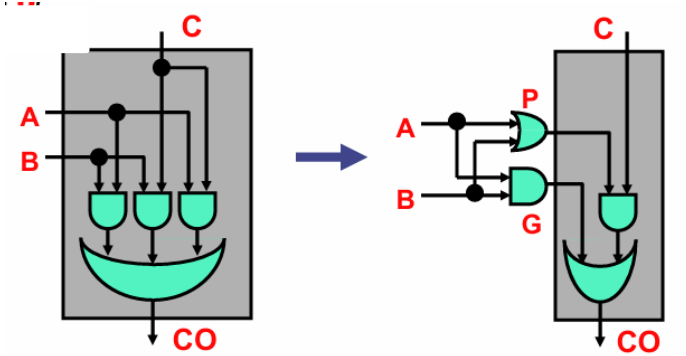
- What is CSA delay (two segment)?
  - $d(CO_{15}) = MAX(d(CO_{15-8}), d(CO_{7-0})) + 2$
  - $d(CO_{15}) = MAX(2*8, 2*8) + 2 = 18$
  - In general: $2*(N/2) + 2 = N + 2$ (vs about 2N for RCA)
- What if we cut adder into 4 equal pieces?
  - $d(CO_{15}) = MAX(d(CO_{15-12}), d(CO_{11-0})) + 2$
  - $d(CO_{15}) = MAX(2*4, MAX(d(CO_{11-8}),d(CO_{7-0})) + 2) + 2$
  - $d(CO_{15}) = MAX(2*4, MAX(2*4, MAX(d(CO_{7-4}), d(CO_{3-0}))+2)+2)+2$
  - d(CO15) = MAX(2*4, MAX(2*4, MAX(2*4, 2*4)+2)+2)+2
  - d(CO15) = 2*4 + 3*2 = 14
- N-bit adder in M equal pieces: 2*(N/M) + (M-1)*2

Takeaway Questions

- What is the delay of a 16-bit CSA adder with 8 parts?
  - (A) 16
  - (B) 18
  - (C) 20
  - (D) 32

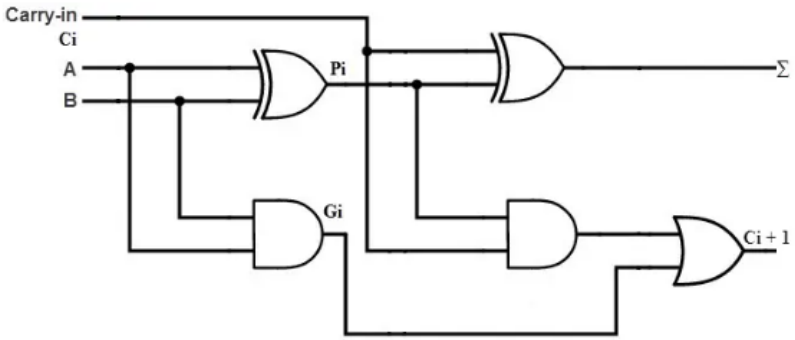
Carry Lookahead Adder (CLA)

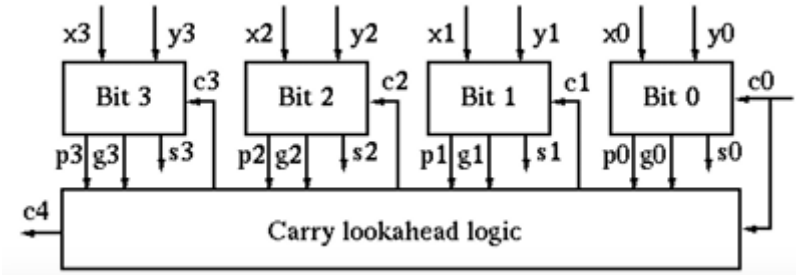
- Let's look at the single-bit carry-out function
  - CO = AB+AC+BC = (AB)+(A+B)C
  - (AB): Generates carry-out regardless of incoming C -> rename to G
    - Generate function: $g_i = x_i \bullet y_i$
    - If $g_i = 1$, the i^{th} bit generate a carry, ci = 1
  - (A+B): Propagates incoming C -> rename to P
    - Propagate function: $p_i = x_i + y_i$
    - p_i is true when A_i or B_i is 1 => $p_i = A_i$
  - $CO_{i+1} = G_i + P_i C_i$



Carry Lookahead Adder (CLA)

- The CLA requires
  - AND and OR gates with many inputs as n + 1 (for C_n)
  - Both gi and pi can be generated for all n-bits in constant time (1 gate delay)





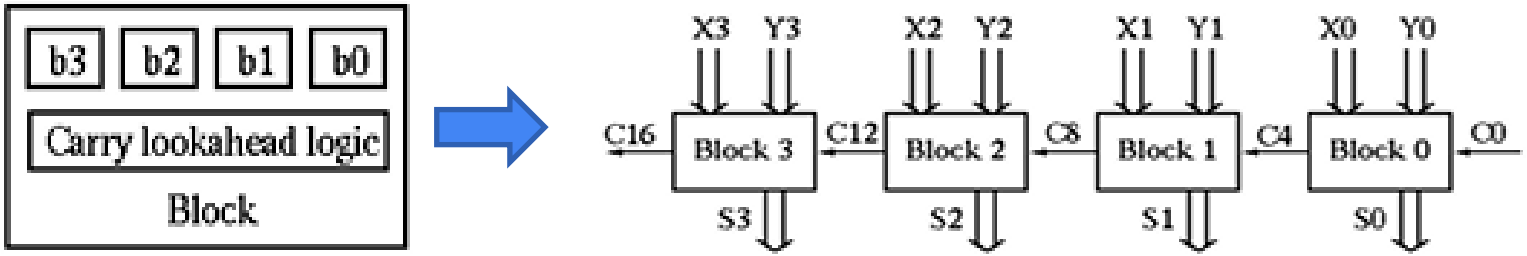
A	B	Ci	C i +1	Condition
0	0	0	0	No carry generate
0	0	1	0	
0	1	0	0	
0	1	1	1	No carry propagate
1	0	0	0	
1	0	1	1	
1	1	0	1	Carry generate
1	1	1	1	

Carry Lookahead Adder (CLA)

- Infinite Hardware CLA
  - Can expand C1…N in terms of G's, P's, and C0
  - Example C_{16}
    - $C_{16} = G_{15}+P_{15}C_{15}$
    - $C_{16} = G_{15}+P_{15}(G_{14}+P_{14}C_{14})$
    - $C_{16} = G_{15}+P_{15}G_{14}+\ldots+P_{15}P_{14}\ldots P_2P_1G_0 + P_{15}P_{14}\ldots P_2P_1P_0C_0$
  - A CLA
    - Generates c_i has logic in two gate delay after g_i and p_i are available
    - Generates S_i has logic in three gate delay after g_i and p_i are available
    - Example: the total gate delay of a 4-bit CLA is 6 (= 1 + 2 + 3)

Carry Lookahead Adder (CLA)

- Is there a compromise?
  - Reasonable number of small gates?
  - Sublinear (doesn't have to be constant) latency?
  - Multi-level CLA exploits hierarchy to achieve this
  - Example, we pack n = 4 bits as a block with carry lookahead
    - Still use ripple carry between the blocks (C_4, C_8, C_{12}, C_{16})



Carry Lookahead Adder (CLA)

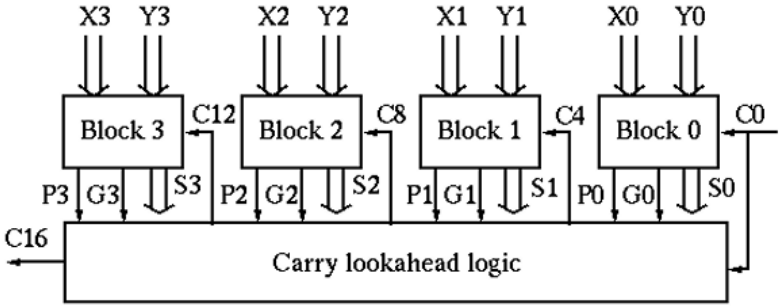
- Multi-level CLA
  - There are n/4 blocks in an n-bit adder, the total gate delays is

Operations	Number of Gate Delays
Generate g_i and p_i	1
Generate c_i (i=1,2,3,4) in block 1	2
Generate c_i (i=5,6,7,8) in block 2	2
.....	.....
Generate s_i	3
Total	$1+2(n/4)+3$

  - Example: When n = 64, the number of gate delays is 36

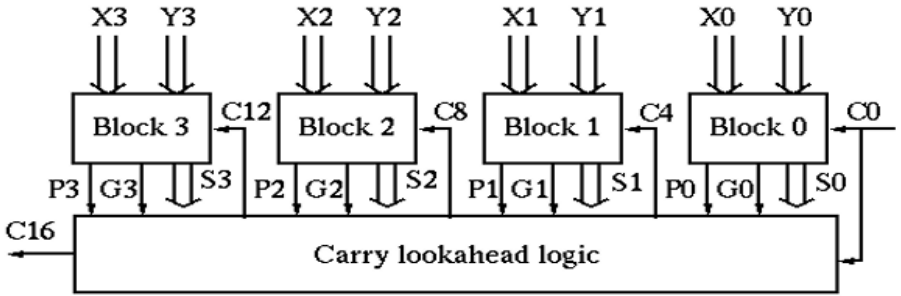
Carry Lookahead Adder (CLA)

- Two-Level CLA
  - The second-level generate and propagate functions
    - $P_i = P_{4i+3}P_{4i+2}P_{4i+1}P_{4i}$
      - If all four bits in a block propagate, the block propagates a carry
    - $G_i = g_{4i+3} + p_{4i+3}g_{4i+2} + p_{4i+3}p_{4i+2}g_{4i+1} + p_{4i+3}p_{4i+2}p_{4i+1}g_{4i}$
      - If at least one of the four bits generates carry and it can be propagated to the MSB, the block generates a carry



Carry Lookahead Adder (CLA)

- Two-Level CLA
  - The second-level generate and propagate functions
    - C4 can be generated in constant time (independent of n)
      - $C_4 = (g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0)+(p_3p_2p_1p_0)C_0 = G_0+P_0C_0$
      - $C_8 = G_1 + P_1C_4$, $C_{12} = G_2 + P_2C_8$, $C_{16} = P_3C_{12}$
    - Combine four blocks of 4-bit CLA as a super block
    - We get a 16-bit adder with two levels of carry-lookahead logic



Carry Lookahead Adder (CLA)

- Two-Level CLA
  - There are n/16 super blocks in a n-bit adder
  - The total gate delays can be found as

Operations	Number of Gate Delays
Generate all g_i and p_i	1
Generate all G_i and P_i $(i = 0, 1, 2, \cdots)$	2
Generate c_i $(i = 4, 8, 12, 16)$ in Block 0	2
Generate c_i $(i = 20, 24, 28, 32)$ in Block 1	2
.....	.....
Generate s_i	3
Total	$1 + 2 + 2(n/16) + 3$

  - When n = 64, the number of gate delays is 14

Carry Lookahead Adder (CLA)

- The third level of CLA
  - With the carries C_{16}, C_{32}, C_{48}, C_{64} generated simultaneously by the third-level carry-look ahead logic

Operations	Number of Gate Delays
Generate all g_i and p_i	1
Generate all G_i and P_i $(i = 0, 1, 2, \cdots)$	2
Generate all $\mathbf{G}_i$ and $\mathbf{P}_i$ $(i = 0, 1, 2, \cdots)$	2
Generate c_i $(i = 16, 32, 48, 64)$ in super block 0	2
.....	.....
Generate s_i	3
Total	$1 + 2 + 2 + 2(n/64) + 3$

  - When n = 64, the number of gate delays is 10

Subtraction

- Sign/magnitude subtraction is reverse addition
  - 2C subtraction is addition
- How to subtract using an adder?
  - sub A B = add A -B
  - Negate B before adding (fast negation trick: -B = B’ + 1)

Shift and Rotation

- Left/right shifts are useful
  - Fast multiplication/division by small constants
  - Bit manipulation: extracting and setting individual bits in words
- Right shifts
  - Can be logical (shift in 0s) or arithmetic (shift in copies of MSB)
    - srl 110011, 2 = 001100
    - sra 110011, 2 = 111100
    - Caveat: for negative numbers, sra is not equal to division by 2

```
-53 = 1111111111001011

          sra 2

     1111111111110010(11) = -14
      ^^               ^^
     sign           dropped
   extension
```

```
 53 sra 2 = floor( 53 / 2^2) = floor( 13.25) =  13
-53 sra 2 = floor(-53 / 2^2) = floor(-13.25) = -14
```

A Simple Shifter

- The simplest 16-bit shifter: can only shift left by 1
  - Implement using wires (no logics)
- Logical shift operators << >>
  - Performs zero-extension for >>
    - wire [15:0] a = b << c[3:0];
- Arithmetic shift operator >>>
  - Performs sign-extension
  - Require a signed wire input
    - wire signed [15:0] b;
    - wire [15:0] a = b >>> c[3:0];

Multiplication

- How humans multiply
  - We first generate all partial product terms

```
    1010     <- Multiplicand
x   1101     <- Multiplier
=========
    1010     <- Partial Product
```

```
    1010
x   1101
=========
    1010
   0000      <- Partial Product
```

Multiplication

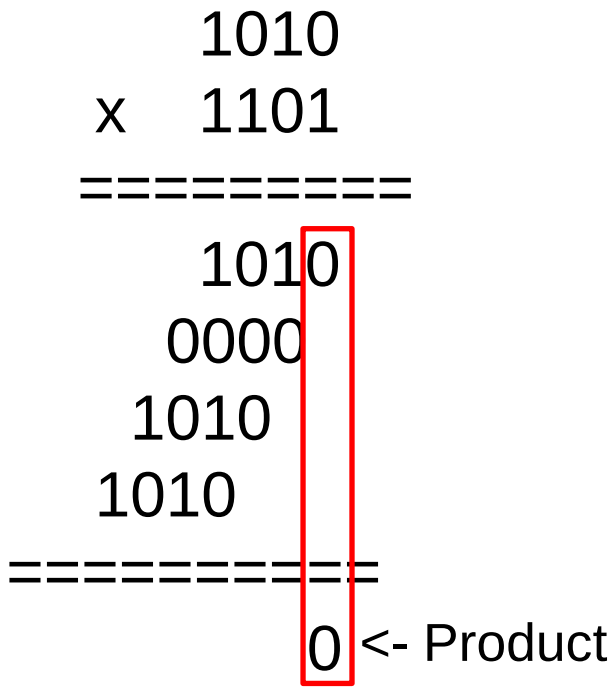
- How humans multiply
  - We first generate all partial product terms

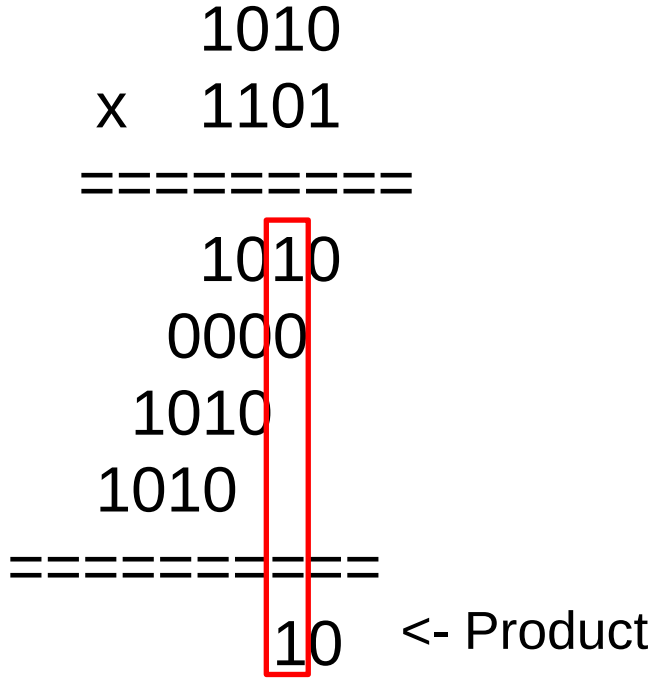
```
    1010     <- Multiplicand
x   1101     <- Multiplier
=========
    1010
   0000
  1010       <- Partial Product
```

```
    1010
x   1101
=========
    1010
   0000
  1010
 1010        <- Partial Product
```

Multiplication

- Then add column by column, right to left





Multiplication

- Sometimes with one or more carry digits

```
          1010
        x 1101
        ========
Carry-> 1
          1010
         0000
        1010
       1010
       ==========
       0010 <- Product
```

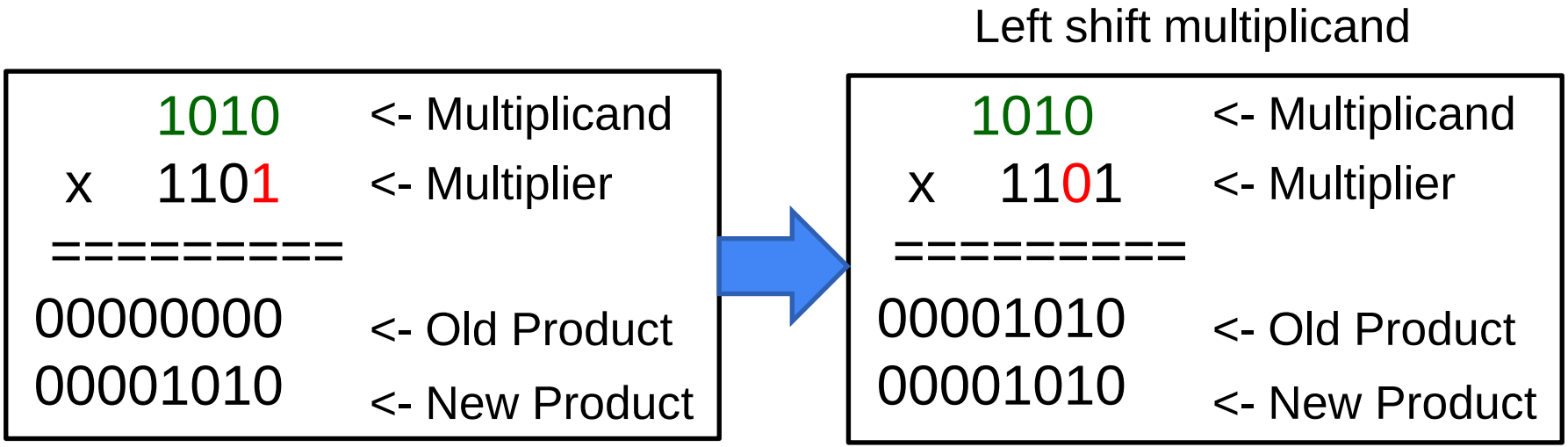
```
          1010
        x 1101
        ========
       1111
          1010
         0000
        1010
       1010
       ==========
       10000010 <- Product
```

Multiplication

- Human method not best for computers
  - Each partial product must be stored -> extra hardware
  - Columns vary in size -> complexity
  - Multiple-digit carries -> complexity
  - Need a simpler method for low-cost multipliers

Multiplication

- Shift & Add Multiply



Multiplication

- Shift & Add Multiply

Left shift multiplicand

```
  1010          <- Multiplicand
x 1101          <- Multiplier
==========
00001010        <- Old Product
00110010        <- New Product
```

Left shift multiplicand

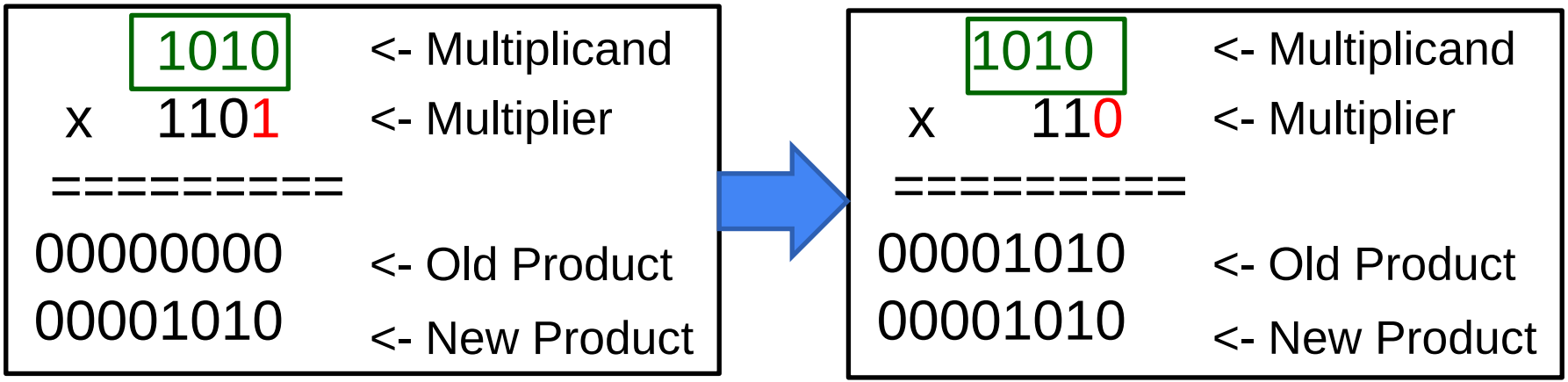
```
  1010          <- Multiplicand
x 1101          <- Multiplier
==========
00110010        <- Old Product
10000010        <- New Product
```

Multiplication

- Shift & Add Multiply
  - Computer multiply also shifts multiplier right so current multiplier bit is at a fixed position, the least significant bit (LSB)

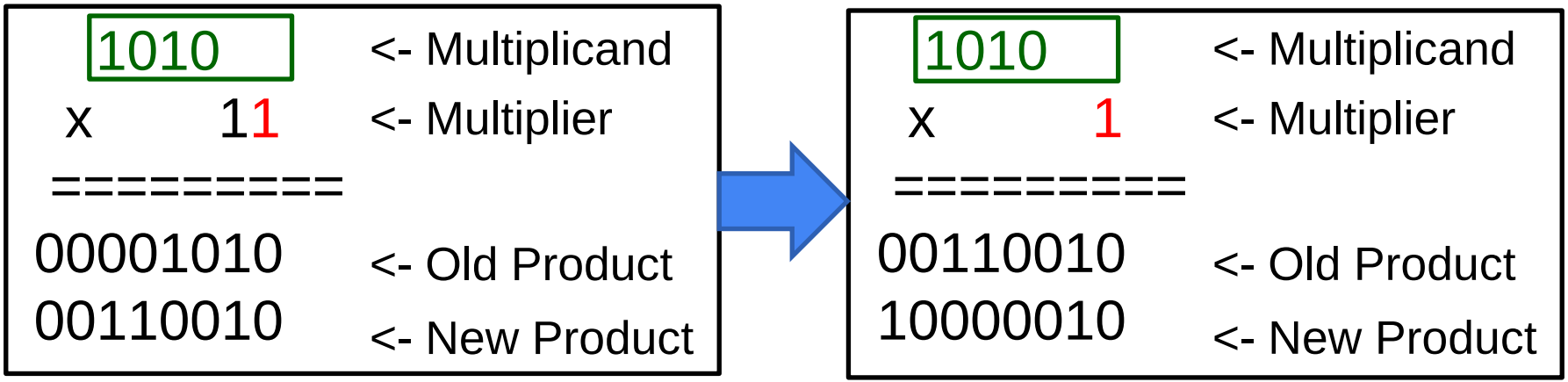
Multiplication

- Shift & Add Multiply



Multiplication

- Shift & Add Multiply



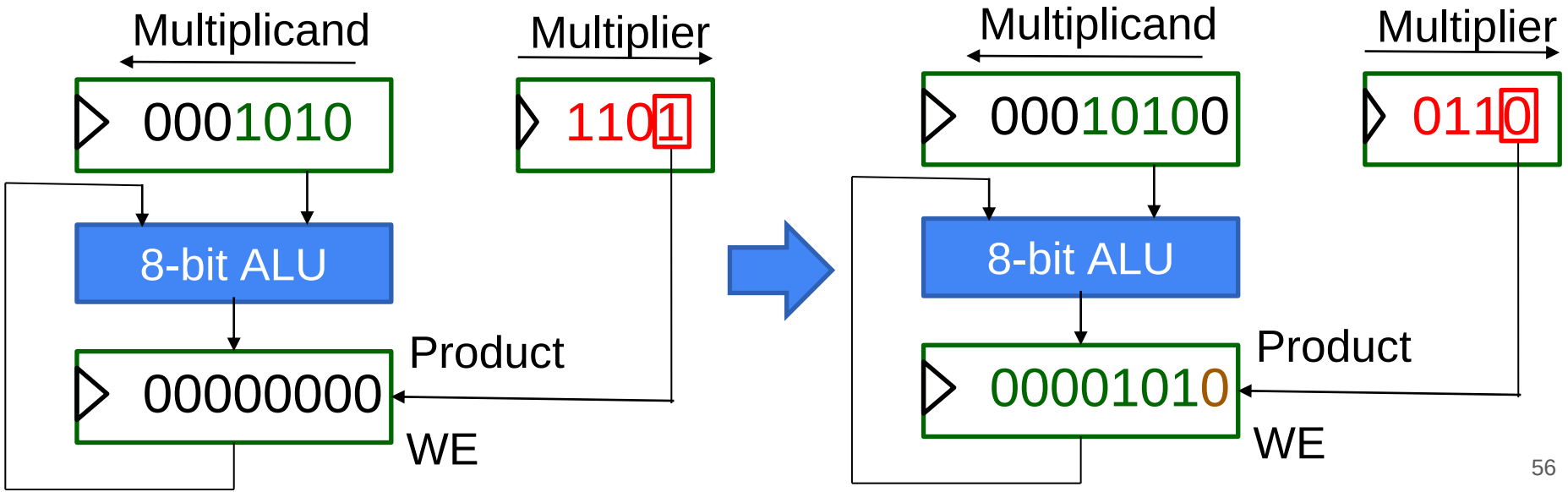
Multiplication

- Shift & Add Multiply
  - Shift & Add Multiply in C programming

```
int product = 0;
for (int i = 0; I < 32; i++)
    if ((multiplier >> i % 2) == 1)
        product = product + multiplicand << i;
```

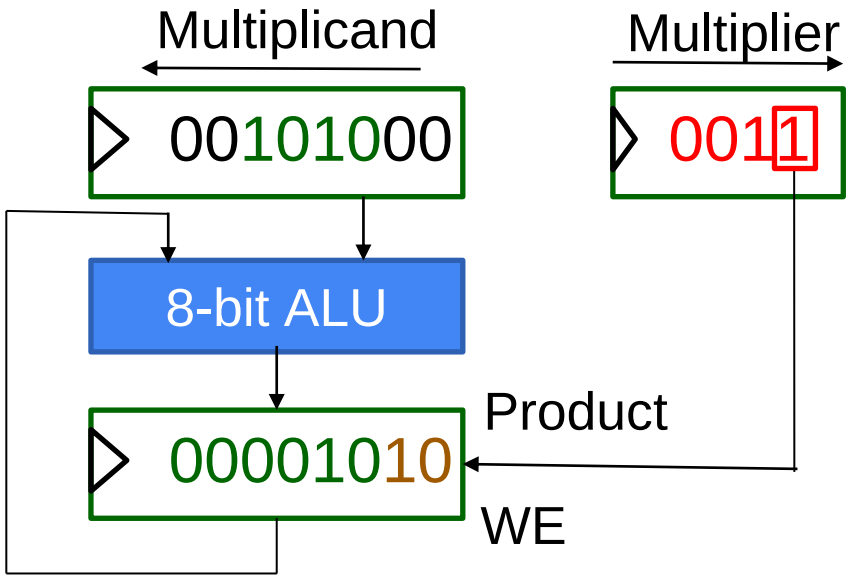
Multiplication

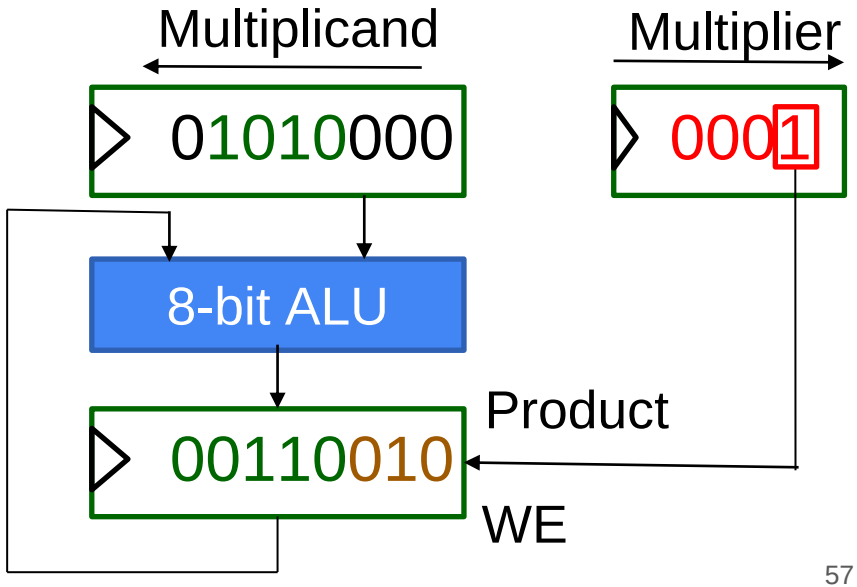
- Simple Shift & Add Multiply Hardware
  - Multiplier LSB is write enable for product latch



Multiplication

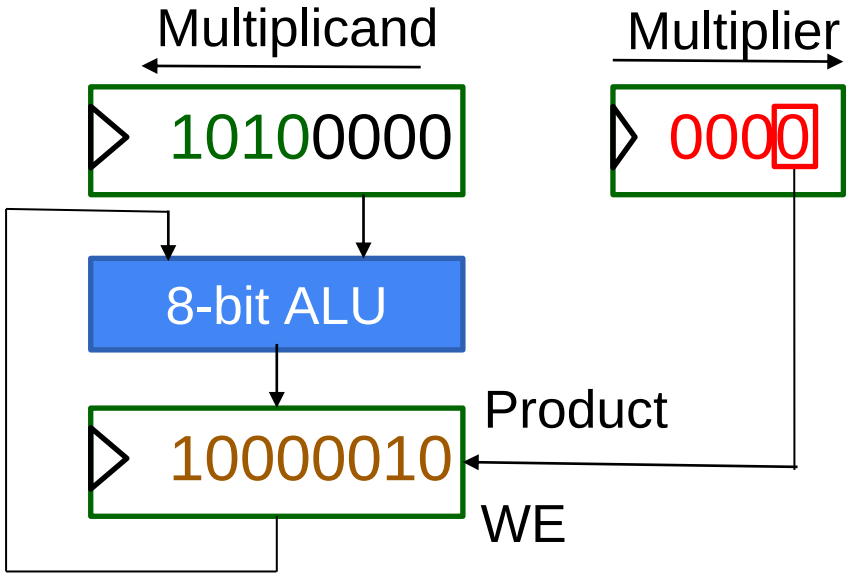
- Simple Shift & Add Multiply Hardware
  - Multiplier LSB is write enable for product latch





Multiplication

- Simple Shift & Add Multiply Hardware
  - Multiplier LSB is write enable for product latch

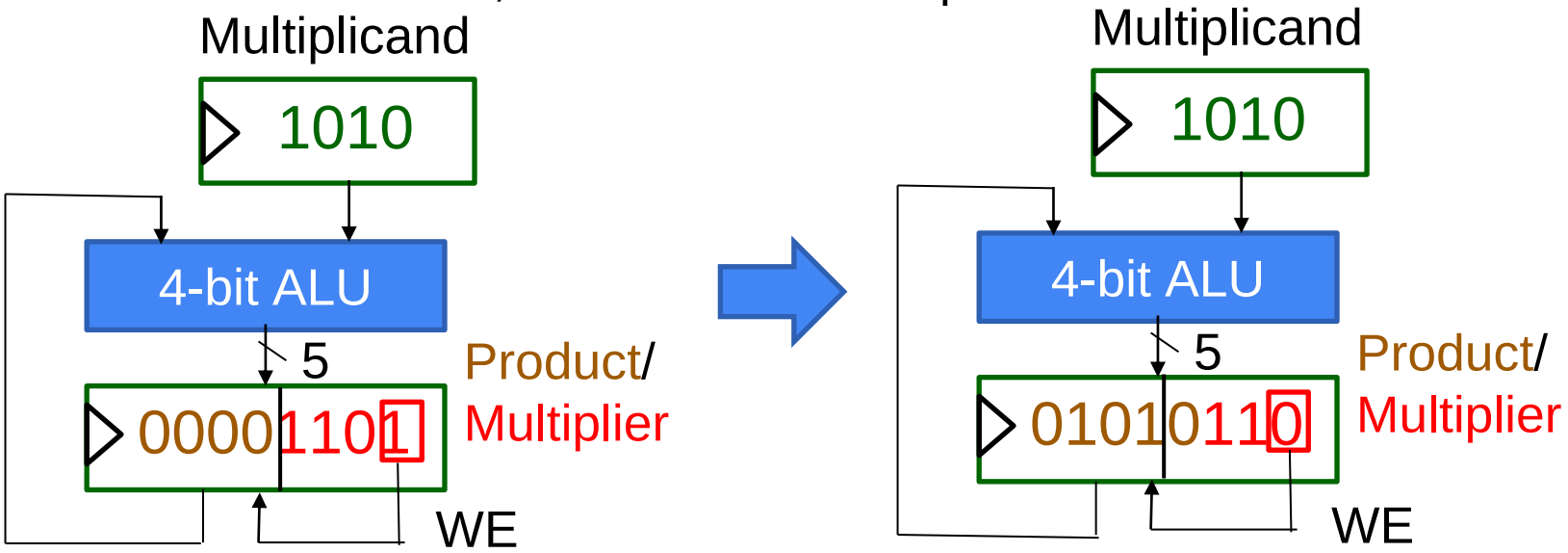


Multiplication

- This simple shift & add hardware
  - Only N significant bits are being summed each cycle, but we are using a 2N-bit adder, a waste
  - Each cycle, one new bit of the product is resolved, while one old bit of the multiplier is discarded
  - Simple multiply shifts Multiplicand left and keep product stationary

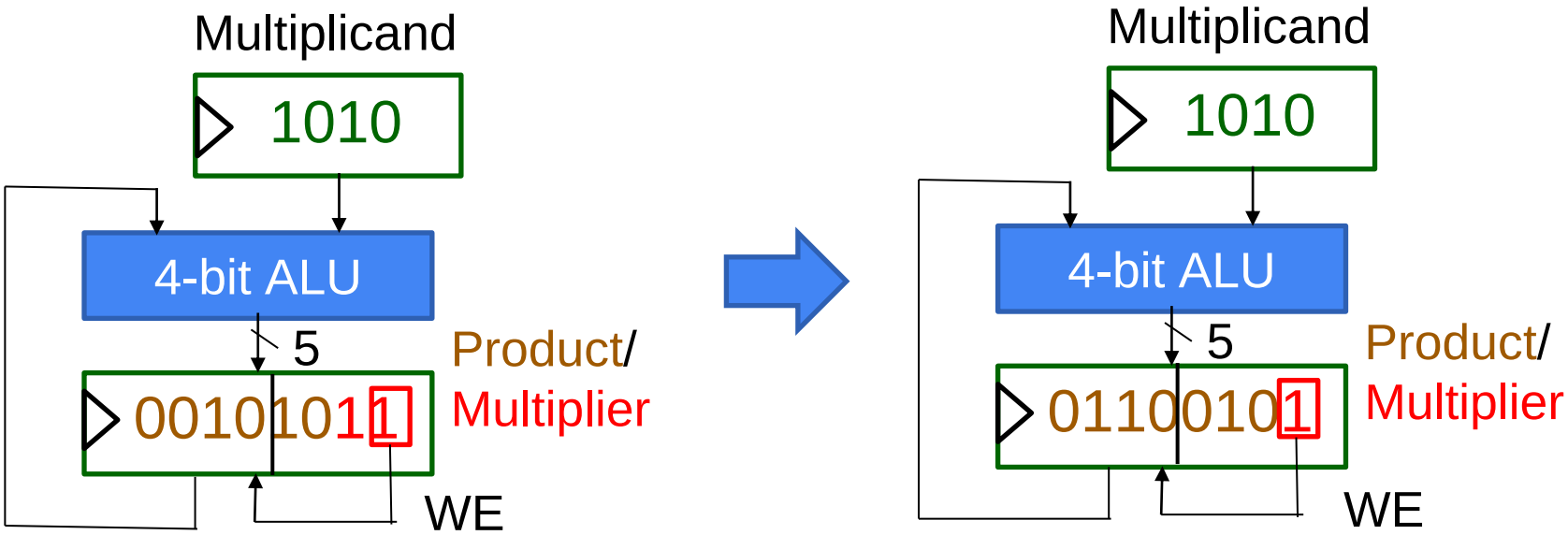
Multiplication

- Refined shift & add hardware
  - ALU input is accept/not accept based on WE
  - When WE = 0, shift but no ALU input



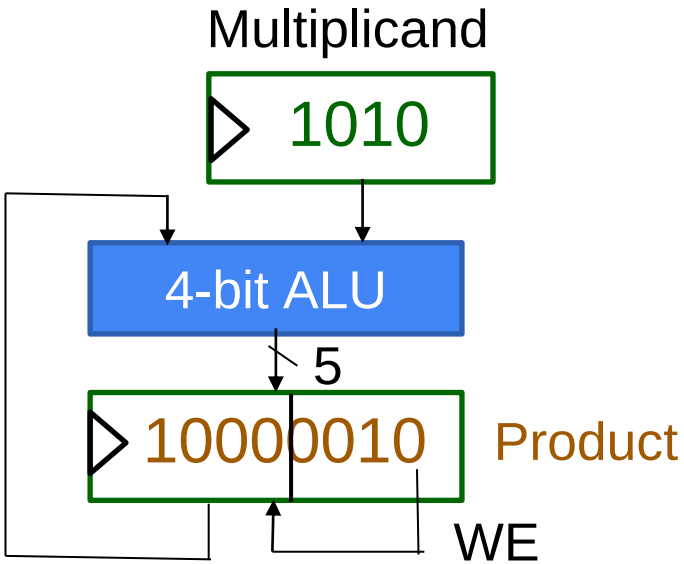
Multiplication

- Refined shift & add hardware
  - ALU input is accept/not accept based on WE



Multiplication

- Refined shift & add hardware
  - Final Result: 10 x 13 = 130



Signed Multiplication

- Shift & Add Multiply only works for positive numbers
- To include negative numbers must:
  - Save XOR of sign bits to get product sign bit
  - Convert multiplier/multiplicand to positive
  - Do shift and add algorithm
  - Negate result if product sign bit is 1

Signed Multiplication

- Booth's algorithm
  - Handle positive/negative number uniformly
  - E.g. 01110_2 (14_{10}) = 10000_2 (16_{10}) – 00010_2 (2_{10})
    = 100002 – 00010$_2$
  - Convert string of 1s into leading +1 and a trialing -1

Signed Multiplication

- Booth's algorithm
  - Identify leading +1s and trailing -1s in multiplier bit position I by looking at multiplier bit i and bit i-1
  - Example
    - 1_{10} = 00010 => 1
    - 1_{10} = 00010 => 11
    - 1_{10} = 00010 => 011
    - 1_{10} = 00010 => 0011
    - 1_{10} = 2 – 1 = 1

		i	i-1
-1	for	1	0
+1	for	0	1
0	for	0	0
0	for	1	1

Signed Multiplication

- Booth's algorithm
  - Identify leading +1s and trailing -1s in multiplier bit position I by looking at multiplier bit i and bit i-1
  - Example
    - -1_{10} = 11110 => 1
    - -1_{10} = 11110 => 01
    - -1_{10} = 11110 => 001
    - -1_{10} = 11110 => 0001
    - -1_{10} = -1

		i	i-1
-1	for	1	0
+1	for	0	1
0	for	0	0
0	for	1	1

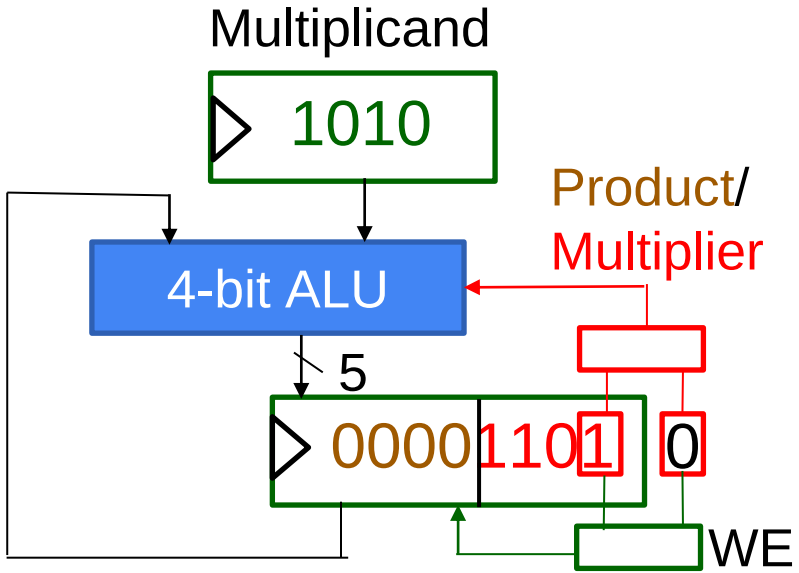
Signed Multiplication

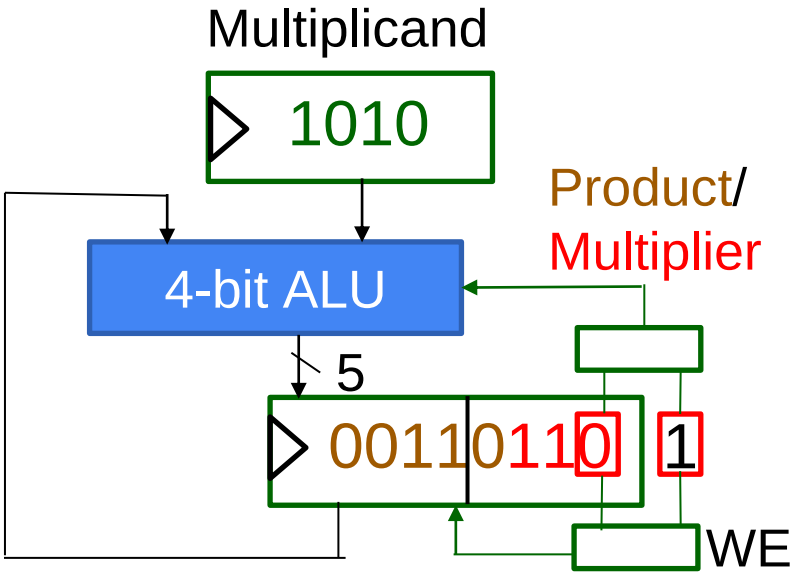
- Booth's algorithm
  - Identify leading +1s and trailing -1s in multiplier bit position I by looking at multiplier bit i and bit i-1
  - Example
    - -6_{10} = 10100 => 0
    - -6_{10} = 10100 => 10
    - -6_{10} = 10100 => 110
    - -6_{10} = 10100 => 1110
    - -6_{10} = -8+4-2 = -6

		i	i-1
-1	for	1	0
+1	for	0	1
0	for	0	0
0	for	1	1

Booth's Algorithm Hardware

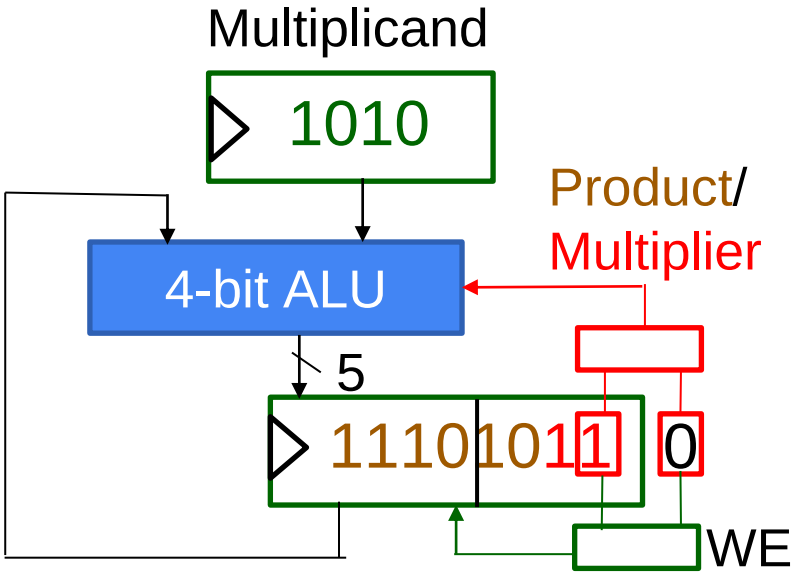
- Use ALU to ADD or SUB based on the trailing -1s, leading 1s from the Multiplier

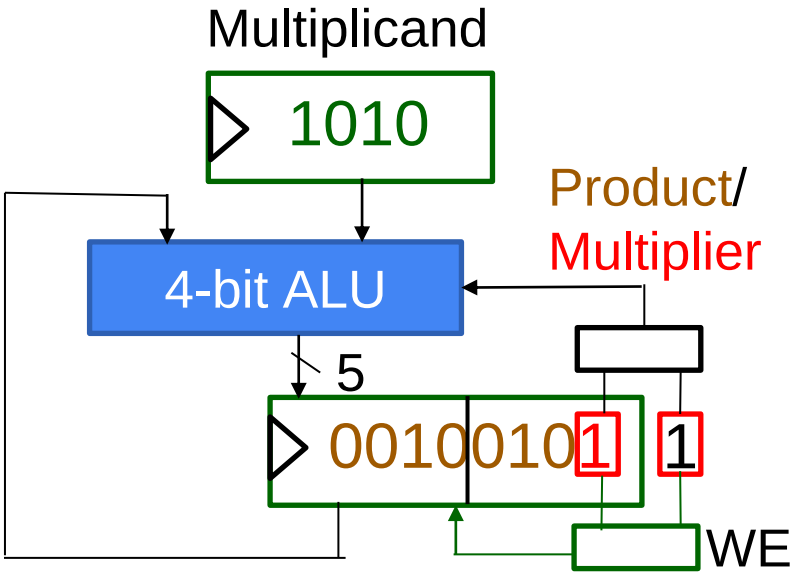




Booth's Algorithm Hardware

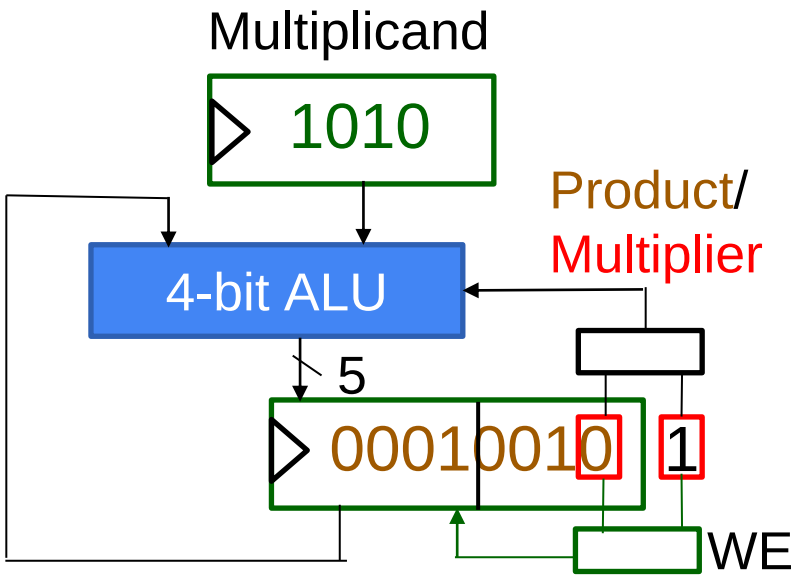
- Use ALU to ADD or SUB based on the trailing -1s, leading 1s from the Multiplier





Booth's Algorithm Hardware

- Final Result: -6 x -3 = 18

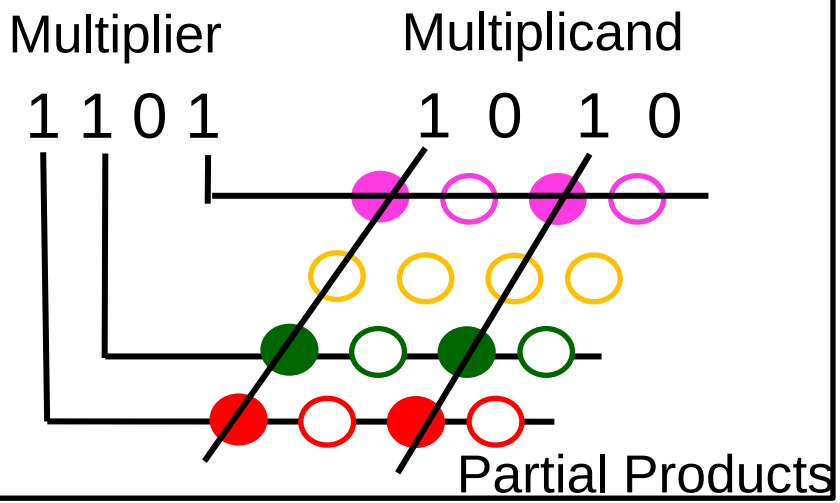


Wallace Tree Multiplier

- Partial products produced sequentially (slow)

```
      1010   Multiplicand
  x   1101   Multiplier
  =========
      1010
     0000
    1010
   1010   Partial Products
```

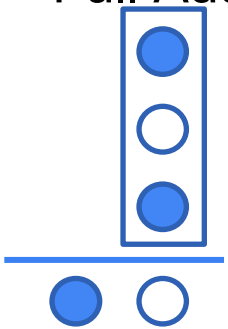
- Partial products produced using array of AND gates (fast)



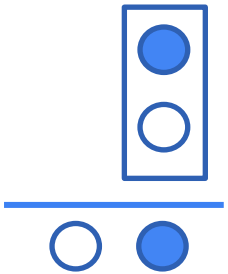
Wallace Tree Multiplier

- Wallace Tree Reducers
  - Full adder cells used to reduce column segment of height 3 to row of width 2
    - Sum and carry out
  - Half adders used selectively for height-3 column that does not need full reduction

Full Adder



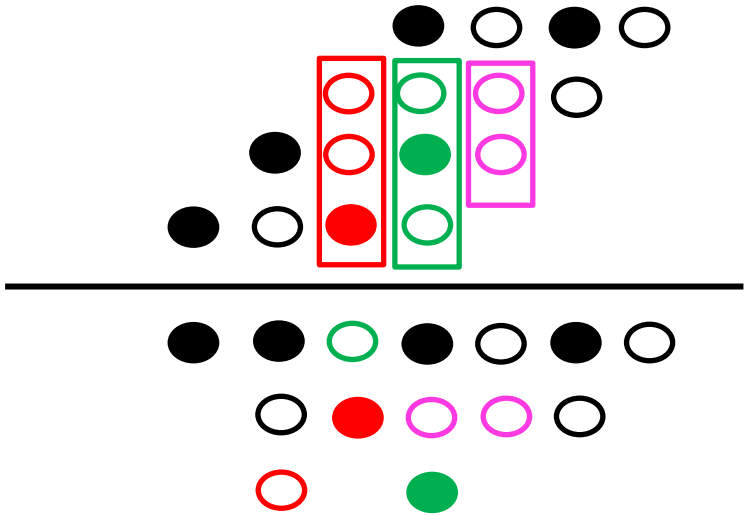
Half Adder



Wallace Tree Multiplier

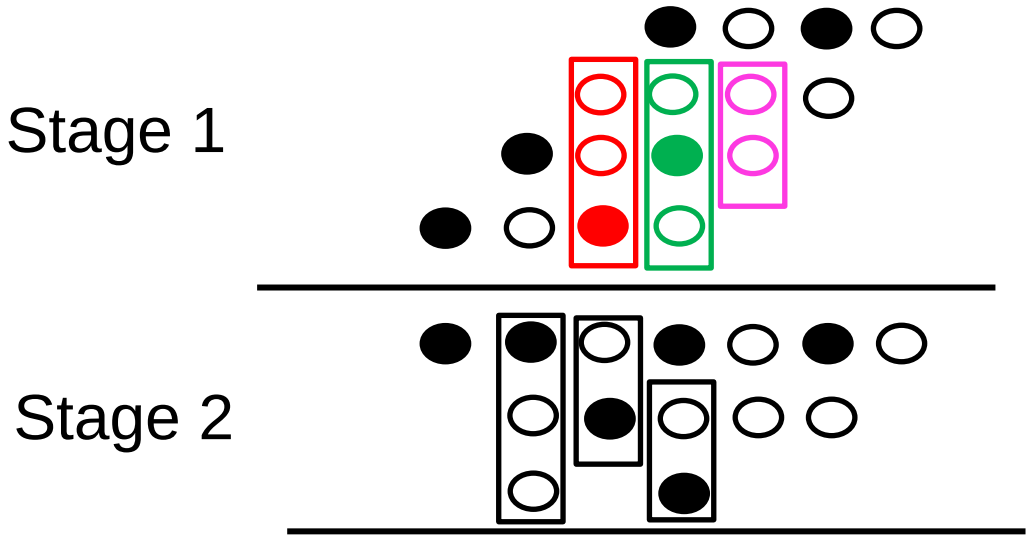
- Wallace Tree: 4 x 4 example

Stage 1



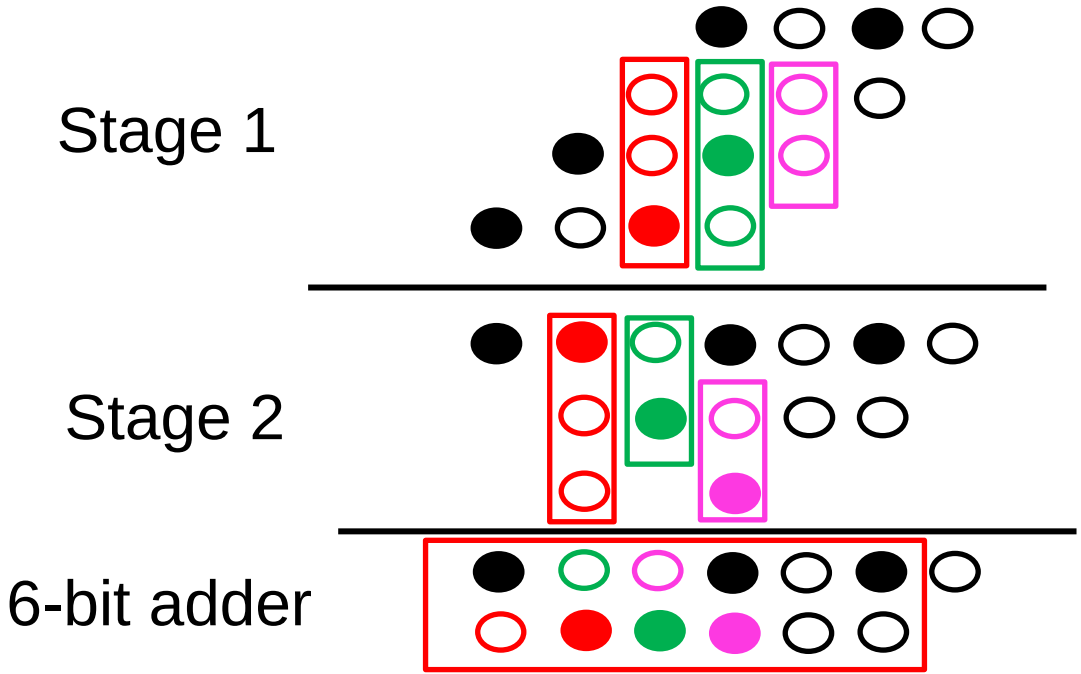
Wallace Tree Multiplier

- Wallace Tree: 4 x 4 example

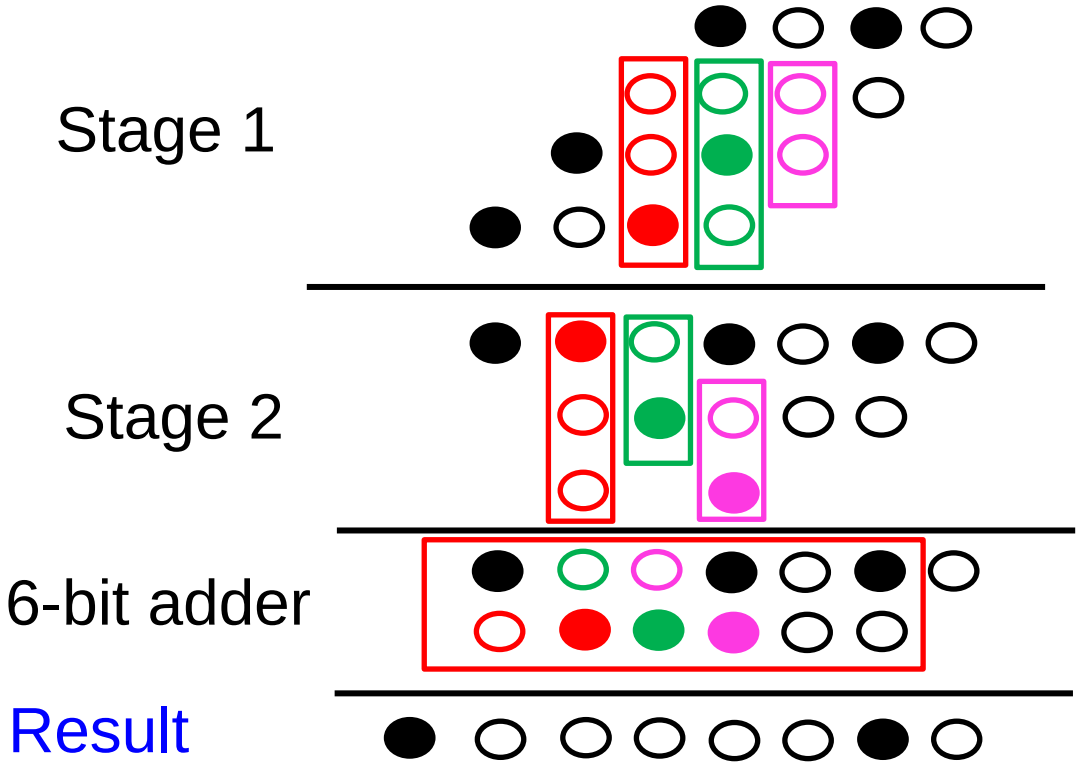


Wallace Tree Multiplier

- Wallace Tree: 4 x 4 example



Wallace Tree Multiplier



Wallace Tree Multiplier

- Wallace tree fast multiply
  - column height is reduced by about 2/3 during each level
- For NxN multiply, an estimate for number of levels

$N \times (2/3)^x = 2$, $(2/3)^x = 2/N$, $x\log_2(2/3) = \log_2(2/N)$,
$x = \log_2(2/N) / \log_2(2/3)$, $x = (\log(2) - \log(N)) / -0.6$,
$x = -1.7 \times (1 - \log_2(N))$

  - 2 levels for N = 4
  - 4 levels for N = 8
  - 7 levels for N = 32

Division

- Consider a long division example

0005678_{10} Quotient

Divisor 1234_{10} | 7006789_{10} Dividend

7006

-6170

8367 Partial Remainder

-7404

…

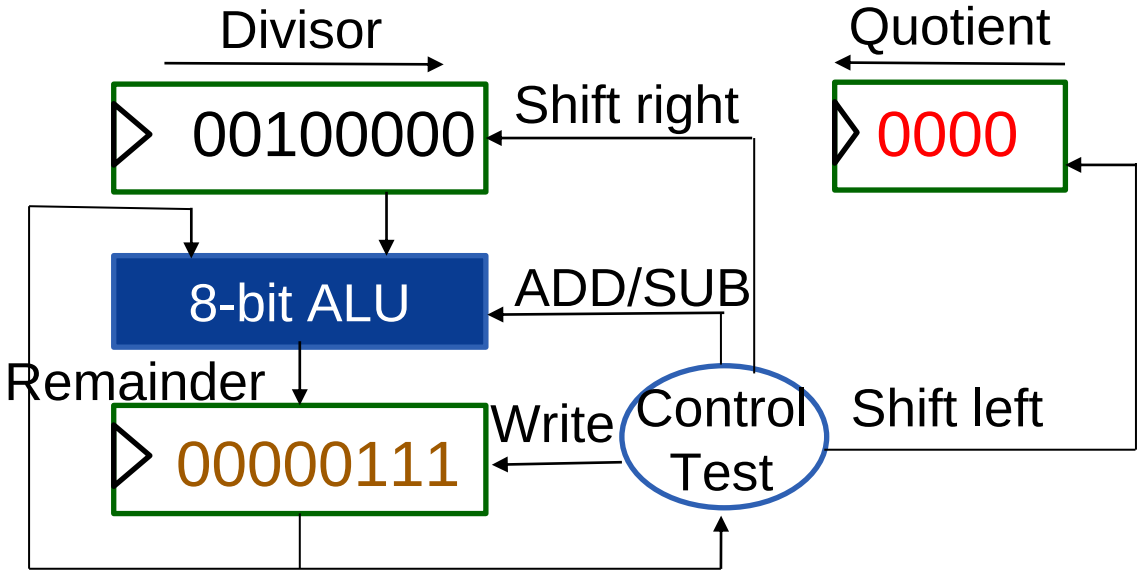
137_{10} Remainder

Division

- Human Division
  - 1. Bring down next digit to partial remainder
  - 2. Compare divisor and partial remainder
    - If partial remainder < divisor: put 0 digit in quotient, goto 1
  - 3. Estimate Quotient digit
  - 4. Multiply divisor by estimated digit
  - 5. Subtract from partial remainder
    - IF result is negative, erase work, decrement estimated digit, goto 4
    - IF result > divisor, erase work, increment estimated digit, goto 4
    - ELSE put estimated digit in quotient, goto 1

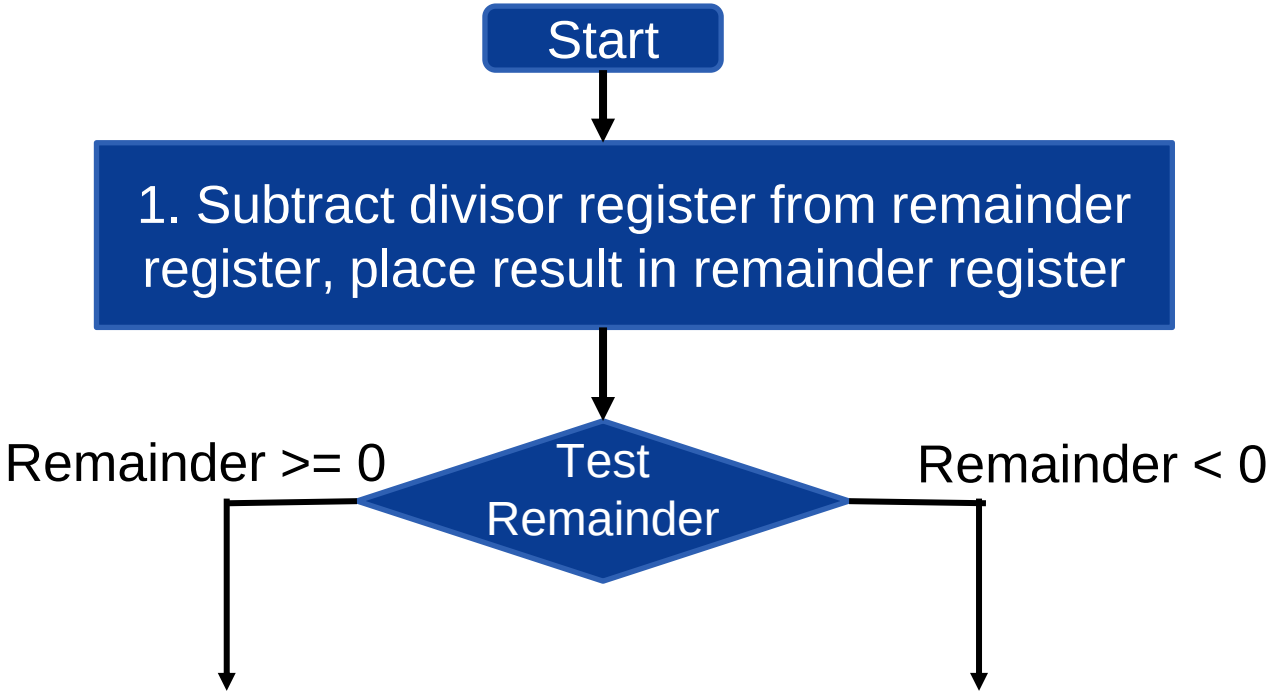
Division

- Simple hardware divider
  - Use ALU to subtract/add shift divisor from remainder
    - Conditionally add to ‘erase’ work after subtract



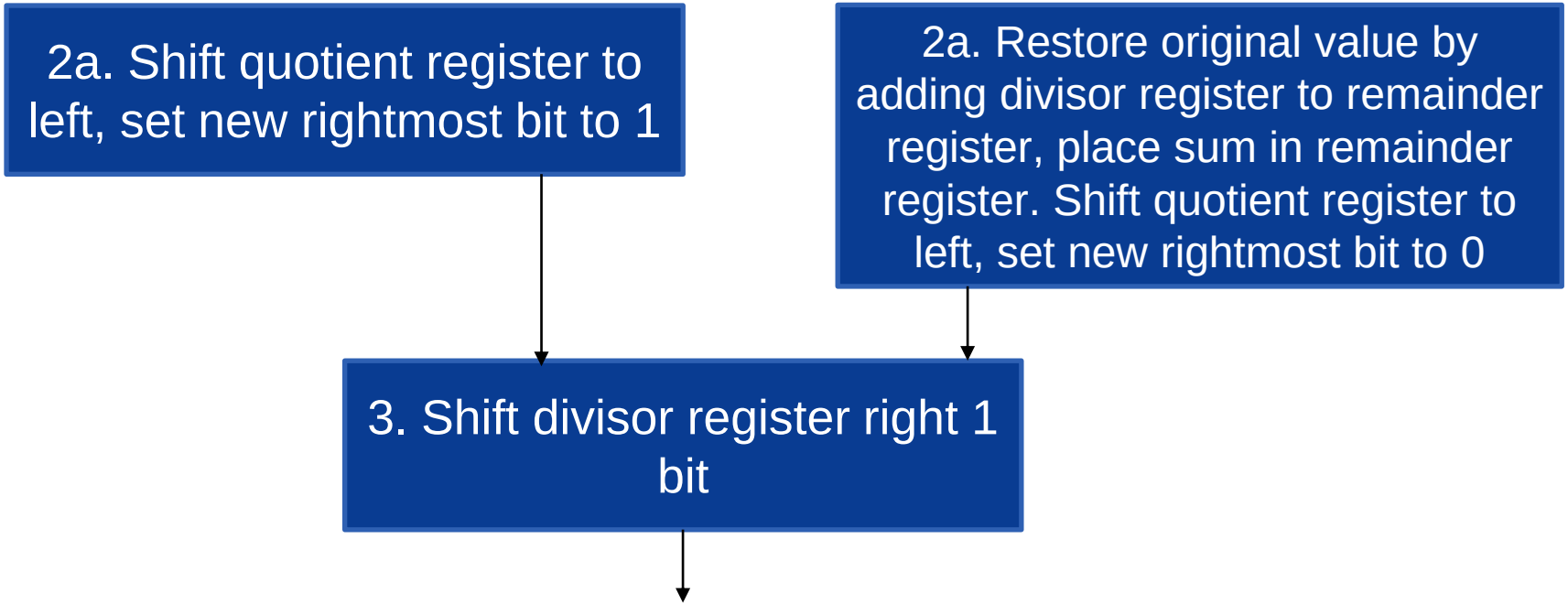
Division

- Simple division algorithm



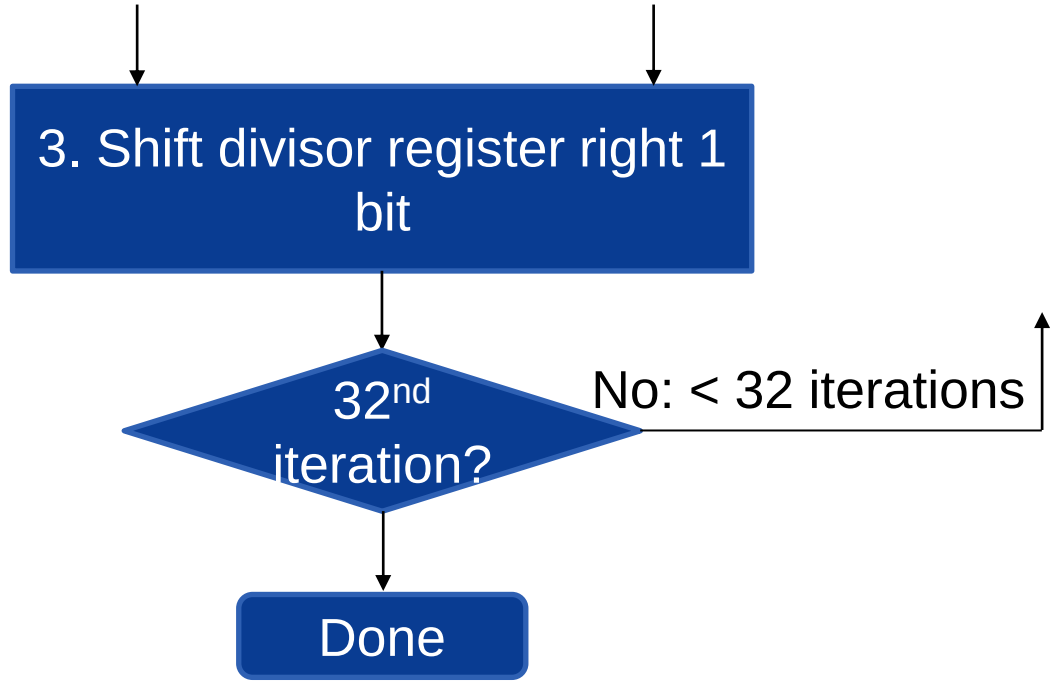
Division

- Simple division algorithm



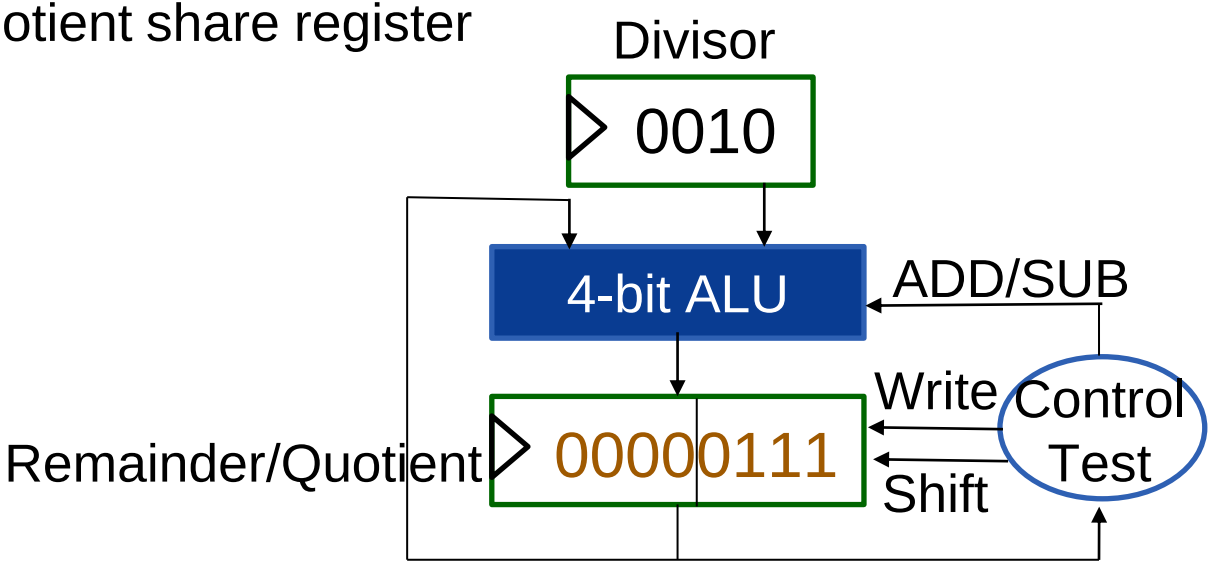
Division

- Simple division algorithm



Division

- Improved hardware divider
  - Divider, ALU are reduced in half
  - Remainder shifts left, divider fixed
  - Remainder/Quotient share register



Division

- Signed division
  - Sign of quotient is XOR of signs of dividend and divisor
  - Sign of remainder matches sign of dividend
    - 122 / 3 = 40 remainder 2
    - -122 / -3 = 40 remainder -2
    - -122 / 3 = -40 remainder -2
    - 122 / -3 = -40 remainder 2

Division

- Fast division
  - So far all ALU operations can be done in $\log_2 N$ time units
    - Carry Look-Ahead Adder: 1 clock cycle
    - Wallace Tree Multiplier: 2 clock cycles
    - Barrel Shifter: 1 clock cycle
  - Best division algorithms take N time units (slow)
    - Resolve 4 bits per iteration rather than 1, and iterations are complex