Lecture 8: Multiplication

Spring 2023
Jason Tang

Topics

- Multiplying unsigned numbers
- Hardware designs for multipliers
- Booth's algorithm

Unsigned Integer Multiplication, Longhand

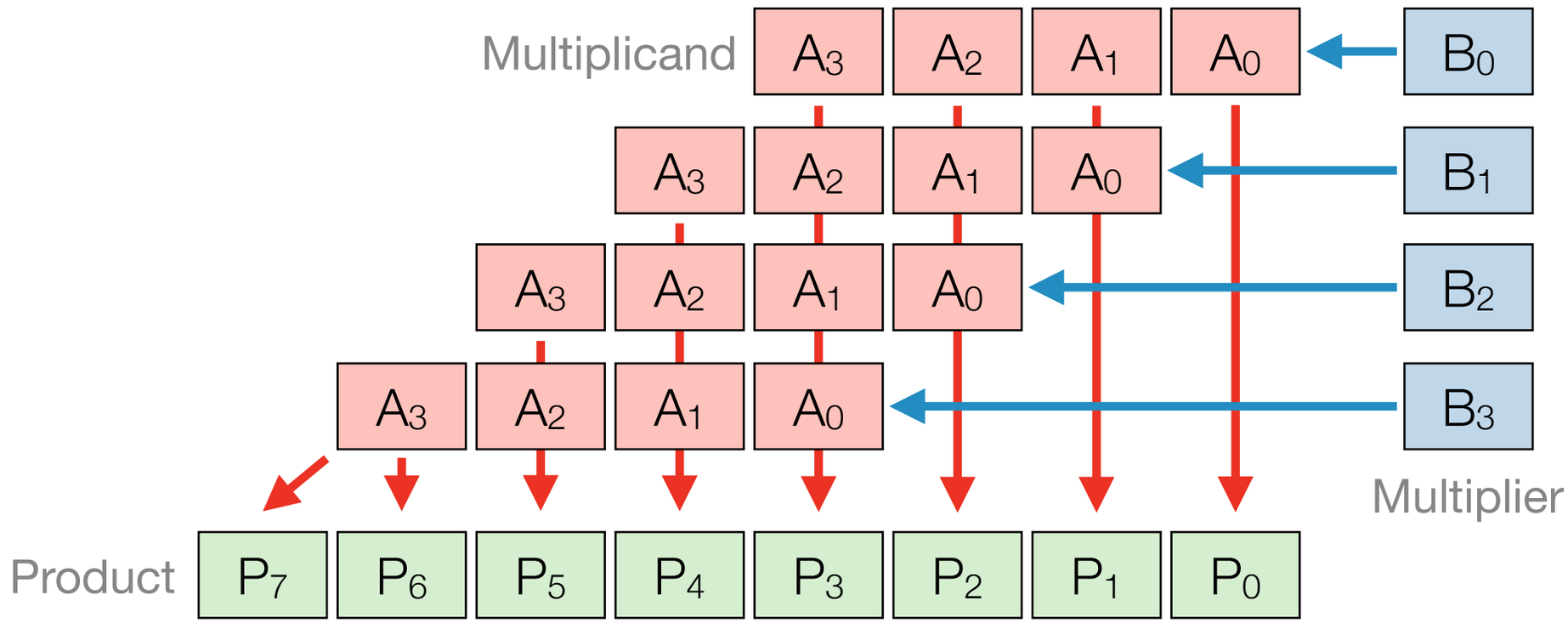
- Multiplying binary values is like multiplying decimals by hand:

Multiplicand		1	0	0	0	$= 8_{10}$
Multiplier	×	1	0	0	1	$= 9_{10}$
		1	0	0	0	
	0	0	0	0		
0	0	0	0			
+ 1	0	0	0			
Product 0	1	0	0	1	0	0	0	$= 72_{10}$

- Given **m** bits × **n** bits, the product requires **m** + **n** bits of storage

- Binary multiplication is easy: 0 → 0, 1 → replicate multiplicand

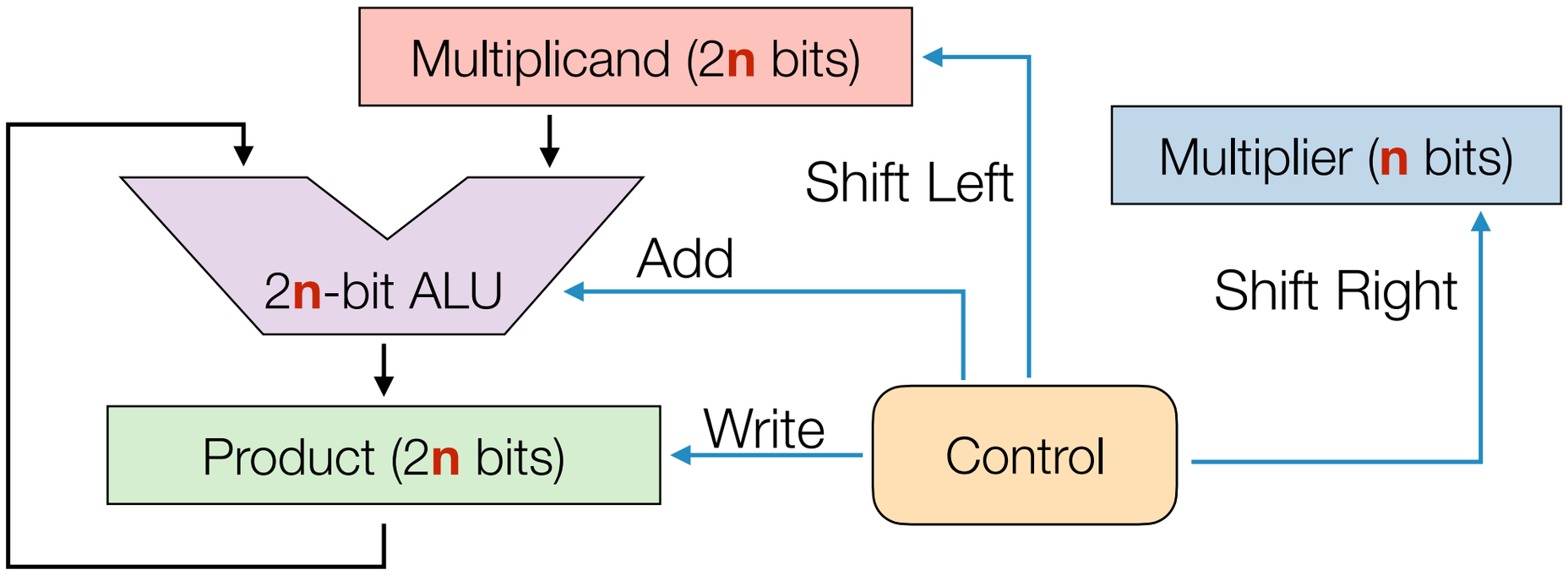
Unsigned Combinational Multiplier



- Stage **i** accumulates **A** × 2^{i}, if B_i is 1
- At each stage, shift **A** left by 1 bit position
- Use next bit in **B** to determine when to add in shifted **A** to partial product **P**

Unsigned Shift-Add Multiplier (version 1)

- Store **n**-bit multiplicand in a register twice its size, towards LSB
- For each step, shift multiplicand left and multiplier right
- Initialize 2**n**-bit product register to zero
- Control decides when to shift and when to write new value into product register



Example of Shift-Add Multiplier (version 1)

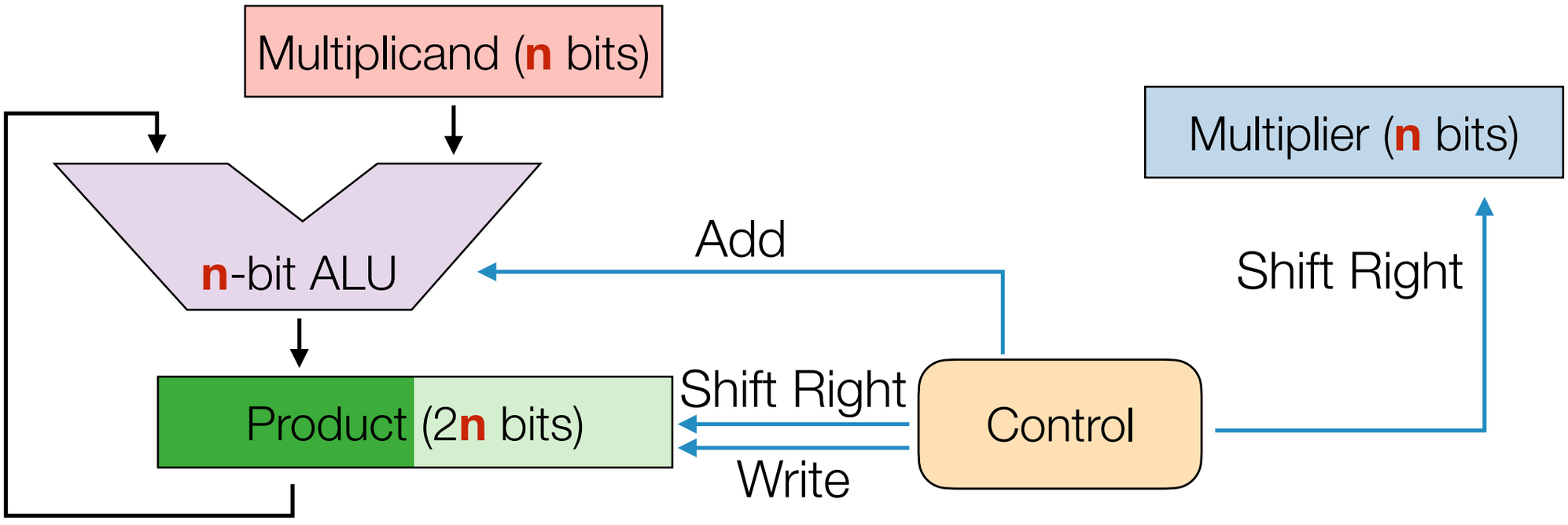
- Multiplying two **n**-bit numbers has up to 2**n**^{2} bit additions, mostly for adding zeroes
  - If LSB of multiplier is 1, then add multiplicand to product; else do nothing
  - Logical shift multiplier right
  - Shift multiplicand left
  - Repeat **n** times

Example: $3_{10} \times 2_{10}$, 4-bits

Iteration	Operations	Product	Multiplier	Multiplicand
0	Initial	0000 0000	0011	0000 0010
1a	Add	0000 0010	0011	0000 0010
1b	Logical shift multiplier right	0000 0010	0001	0000 0010
1c	Shift multiplicand left	0000 0010	0001	0000 0100
after 2	Add, shift, shift	0000 0110	0000	0000 1000
after 3	Shift, shift	0000 0110	0000	0001 0000
after 4	Shift, shift	0000 0110	0000	0010 0000

Unsigned Shift-Add Multiplier (version 2)

- Because half of multiplicand register is filled with zeroes, 2**n**-bit ALU is wasteful
- Use only **n**-bit multiplicand, **n**-bit ALU; keep 2**n**-bit product
- Result of ALU is written to **upper half** of product register
- After each step, **shift right** the product register



Example of Shift-Add Multiplier (version 2)

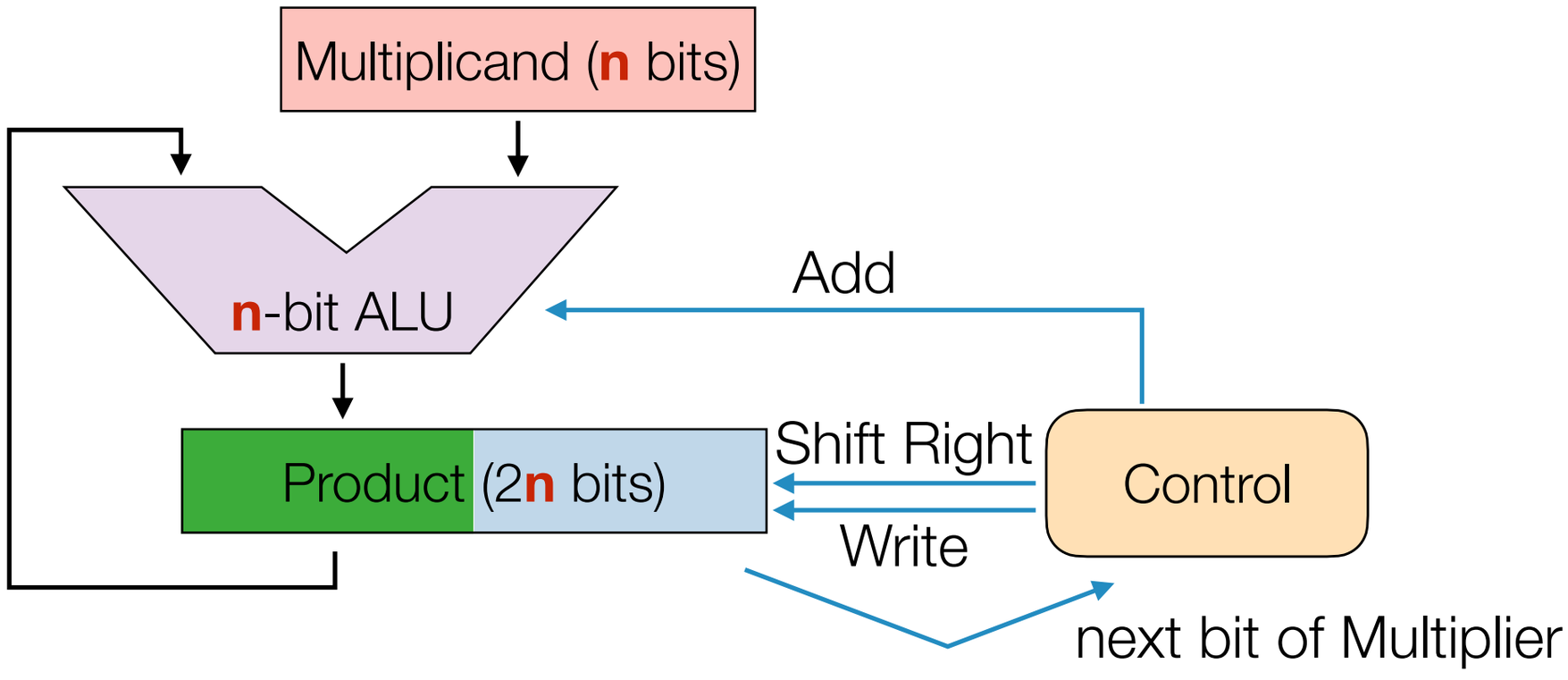
- Multiplying two **n**-bit numbers has up to $\mathbf{n}^2$ bit additions
- If LSB of multiplier is 1, then add multiplicand to upper half of product; else do nothing
  - Logical shift multiplier right
  - Logical shift product right
  - Repeat **n** times

Example: $3_{10} \times 2_{10}$, 4-bits

Iteration	Operations	Product	Multiplier	Multiplicand
0	Initial	0000 0000	0011	0010
1a	Add	0010 0000	0011	0010
1b	Logical shift multiplier right	0001 0000	0001	0010
1c	Logical shift product right	0001 0000	0001	0010
after 2	Add, shift, shift	0001 1000	0000	0010
after 3	Shift, shift	0000 1100	0000	0010
after 4	Shift, shift	0000 0110	0000	0010

Unsigned Shift-Add Multiplier (version 3)

- At startup, bottom half of product register will be shifted away and was never used
- Combine multiplier and product registers, storing multiplier in bottom half of register
- ALU still stores result in upper half of product register



Example of Shift-Add Multiplier (version 3)

- Multiplying two **n**-bit numbers has up to $\mathbf{n}^2$ bit additions
- If LSB of product is 1, then add multiplicand to upper half of product; else do nothing
  - Logical shift product right
  - Repeat **n** times

Example: $3_{10} \times 2_{10}$, 4-bits

Iteration	Operations	Product	Multiplicand
0	Initial	0000 0011	0010
1a	Add	0010 0011	0010
1b	Logical shift product right	0001 0001	0010
after 2	Add, shift	0001 1000	0010
after 3	Shift	0000 1100	0010
after 4	Shift	0000 0110	0010

Signed Integer Multiplication

- Easiest method is to multiply by the absolute (positive) values, then remember to complement product if product is to be negative
  - For **n**-bit terms, run **n**-1 steps, then apply sign bit afterwards
- But attempts to multiply negative numbers directly **do not work for all cases**

Example: $-7_{10} \times 2_{10}$, 4-bits

Iteration	Operations	Product	Multiplicand
0	Initial	0000 0010	1001
1	Shift	0000 0001	1001
2	Add, shift	1100 1000	1001
3	Shift	1110 0100	1001
4	Shift	1111 0110	1001

Counter-Examples: Negative Multiplier

Example: $-7_{10} \times -2_{10}$, no initial sign extension

Iteration	Operations	Product	Multiplicand
0	Initial	0000 1110	1001
1	Shift	0000 0111	1001
2	Add, shift	1100 1011	1001
3	Add, shift	0010 1101	1001
4	Add, shift	0101 1110	1001

- Overflow in step 3's *add*
- Final product is 94_{10}

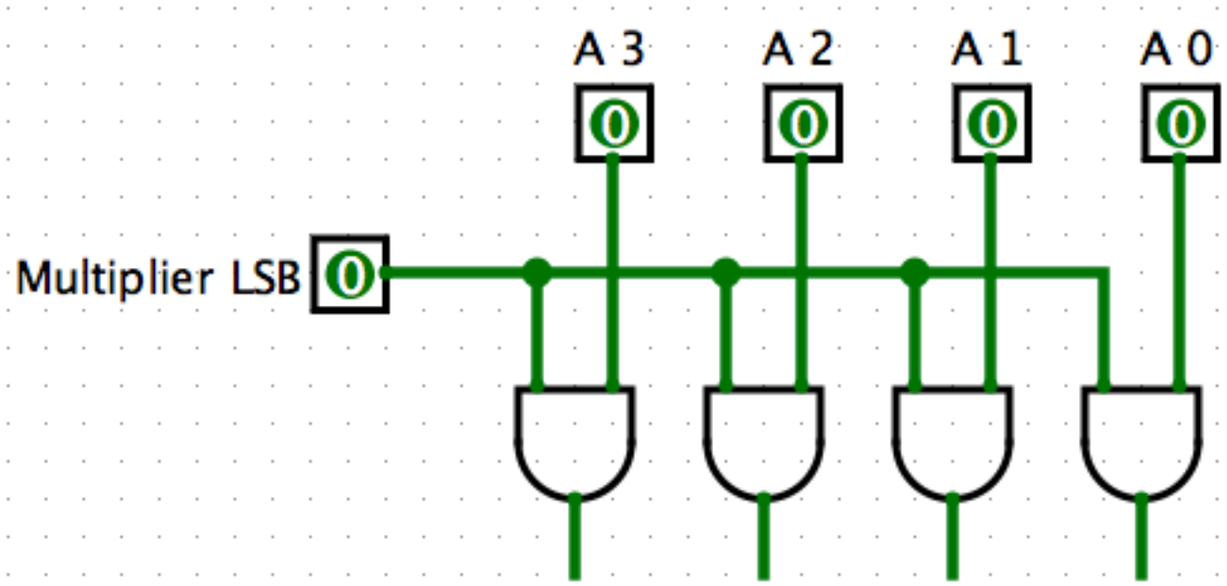
Example: $-7_{10} \times -2_{10}$, initial sign extension

Iteration	Operations	Product	Multiplicand
0	Initial	1111 1110	1001
1	Shift	1111 1111	1001
2	Add, shift	1100 0011	1001
3	Add, shift	0010 1101	1001
4	Add, shift	1101 1101	1001

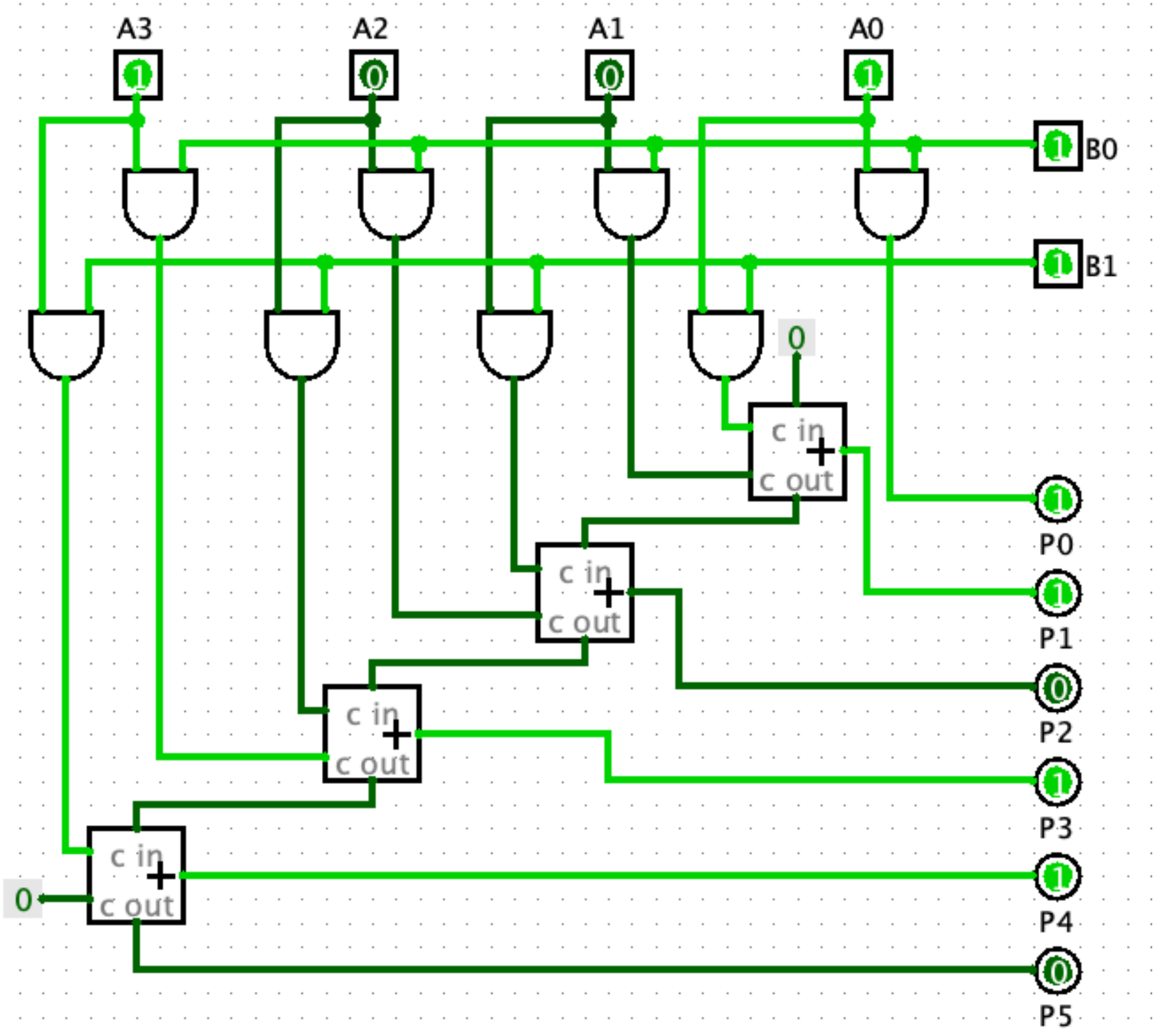
- Overflow in step 2's and 3's *add*
- Final product is -35_{10}

Decreasing Multiplication Times

- Whether to add multiplicand or not is based upon multiplier's individual bits
- As an alternative to deciding to add or not, always add either multiplicand or zero, by ANDing each multiplicand bit by multiplier's LSB
- Less hardware needed
- Decreased gate propagation delay



Implementing Unsigned Combinatorial Multiplier



- Possible optimizations:
  - Use half-adders
  - Use lookahead adders
- What are the advantages and disadvantages of this combinatorial multiplier versus a shift-add multiplier?

Intel Multiplication Implementation

- x86 has two instructions for multiplication: **`MUL`** for unsigned, **`IMUL`** for signed

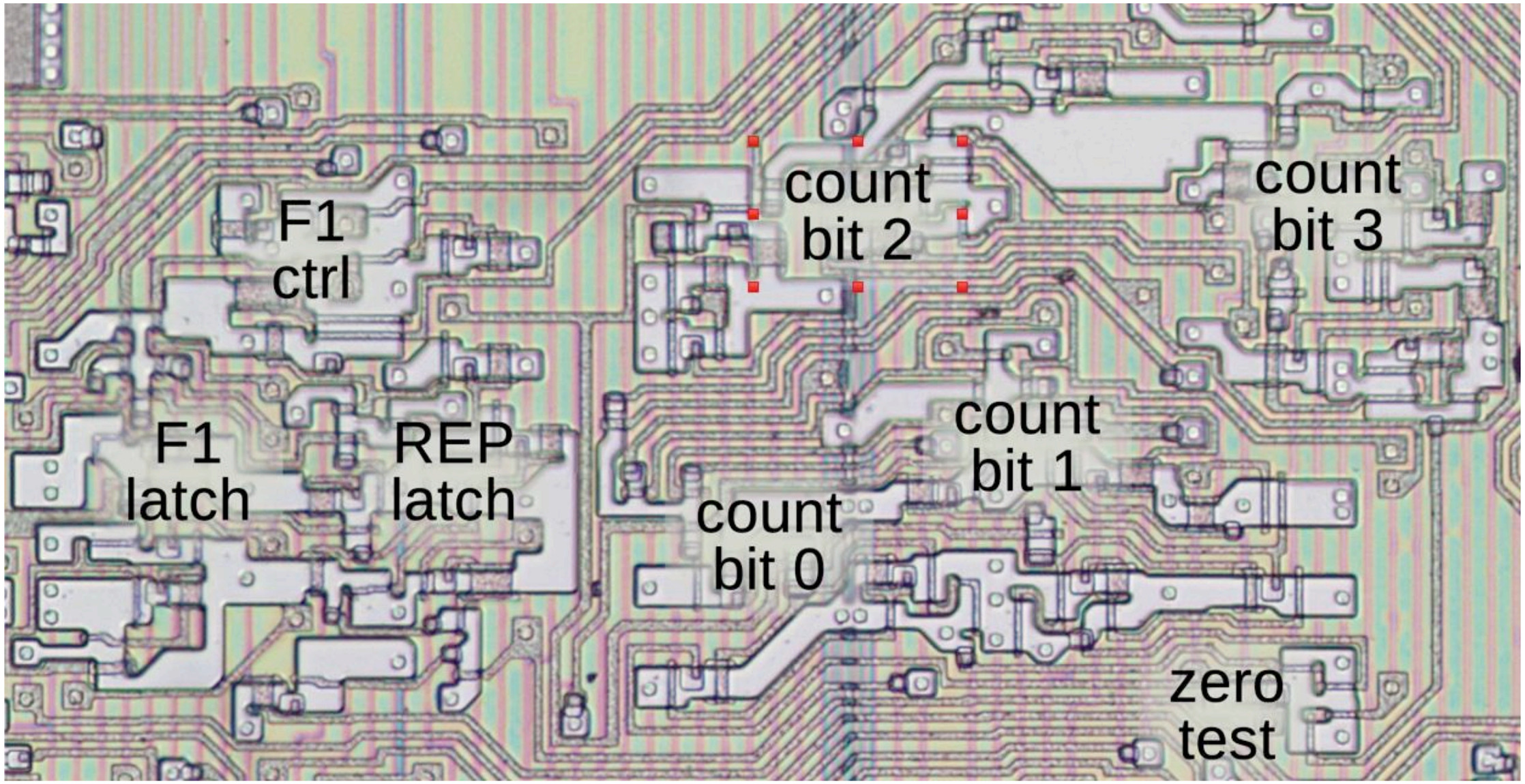
- Original implementation used a shift-add algorithm, taking multiple clock cycles

Operation	Duration
Unsigned 8-bit Multiply	70 - 77 cycles
Signed 8-bit Multiply	80 - 98 cycles
Unsigned 16-bit Multiply	118 -113 cycles
Signed 16-bit Multiply	128 - 154 cycles

http://www.righto.com/2023/03/8086-multiplication-microcode.html

Intel Multiplication Implementation

- F1 tracks sign, REP is loop counter



Booth's Algorithm

- Faster multiplication algorithm, that takes advantage of **shifting**

- Treats a consecutive sequence of ones as **an addition** and **a subtraction**:

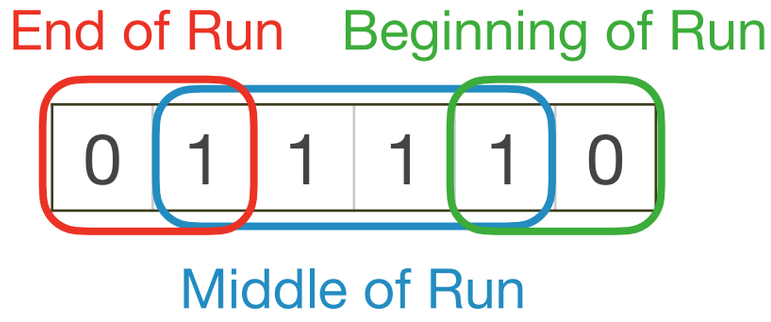
  - Example 1: 4-bit binary sequence $1111_2 = 10000_2 - 1_2$

  - Example 2: 8-bit binary sequence

$$0110\ 1110_2 = 64_{10} + 32_{10} + 8_{10} + 4_{10} + 2_{10}$$
$$= 96_{10} + 14_{10}$$
$$= 110_{10}$$
$$0110\ 1110_2 = 1000\ 0000_2 - 0010\ 0000_2 + 0001\ 0000_2 - 0000\ 0010_2$$
$$= 128_{10} - 32_{10} + 16_{10} - 2_{10}$$
$$= 110_{10}$$

Booth's Algorithm

- **Works for both signed and unsigned numbers**



- Depending on the current and previous bits, do:
  - 00: Middle of string of 0s: nothing
  - 01: End of a string of 1s: add multiplicand to left half of product
  - 10: Beginning of a string of 1s: subtract multiplicand from left half of product
  - 11: Middle of string of 1s: nothing
- Then shift right product register by 1 bit

Example of Booth's Algorithm, Unsigned Integers

- Use current and previous bit to determine operation
- Append an extra 0 digit to LSB of product
- Extend sign when shifting right (arithmetic right shift)
- Discard LSB in the end

Example: $3_{10} \times 2_{10}$, 4-bits

Iteration	Operations	Product	Multiplicand
0	Initial	0000 0011 0	0010
1a	Subtract	1110 0011 0	0010
1b	Arithmetic shift product right	1111 0001 1	0010
after 2	Shift	1111 1000 1	0010
after 3	Add, shift	0000 1100 0	0010
after 4	Shift	0000 0110 0	0010
after 5	Discard LSB	0000 0110	

Example of Booth's Algorithm, Signed Integers

- If multiplicand is the most negative (e.g., -8 given 4 bits), then add an extra MSB to multiplicand and product (sign extended)

- Then discard both extra bits (MSB and LSB) after all operations

- Note: there is a carry-out on step 3, which is discarded

Example: $-8_{10} \times -3_{10}$

Iteration	Operations	Product	Multiplicand
0	Initial	0 0000 1101 0	1 1000
1a	Subtract	0 1000 1101 0	1 1000
1b	Arithmetic shift product right	0 0100 0110 1	1 1000
after 2	Add, shift	1 1110 0011 0	1 1000
after 3	Subtract, Shift	0 0011 0001 1	1 1000
after 4	Shift	0 0001 1000 1	1 1000
after 5	Discard Extra Bits	0001 1000	

Analysis of Booth's Algorithm

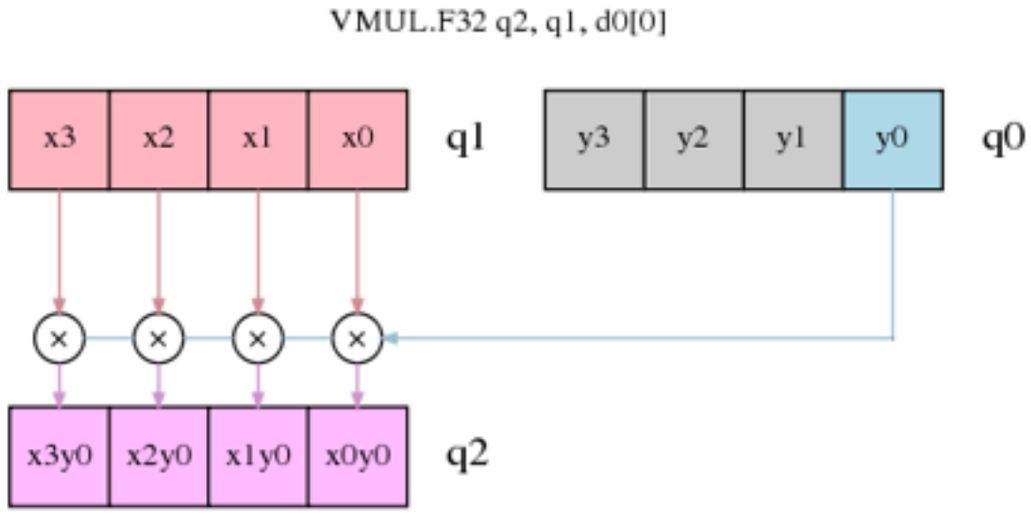
- Very fast given long strings of ones or zeroes
- Same algorithm works for signed and unsigned values
  - No need to invert negative factors
- Inefficient given isolated ones (can be worse than shift-adder)
- Difficult to parallelize in hardware, given variable number of operations
  - Can be improved by processing three bits at a time

https://www.researchgate.net/publication/225315074_A_Comparison_of_Layout_Implementations_of_Pipelined_and_Non-Pipelined_Signed_Radix-4_Array_Multiplier_and_Modified_Booth_Multiplier_Architectures

Other Kinds of Multiplication

- Multiply-accumulate: **a** ← **a** + (**b** × **c**)
  - Used far matrix multiplication, vector dot product, and other common routines
  - Fused multiply-add: when operands are floating point and rounding occurs at the end, instead between intermediate steps

- ARM has NEON instructions for parallel multiplication, including vector-by-scalar multiplication



NEON vector-by-scalar multiplication.

https://community.arm.com/processors/b/blog/posts/coding-for-neon---part-3-matrix-multiplication