National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 3: Arithmetic of Computer

CS10014 Computer Organization

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

Acknowledgements and Disclaimer

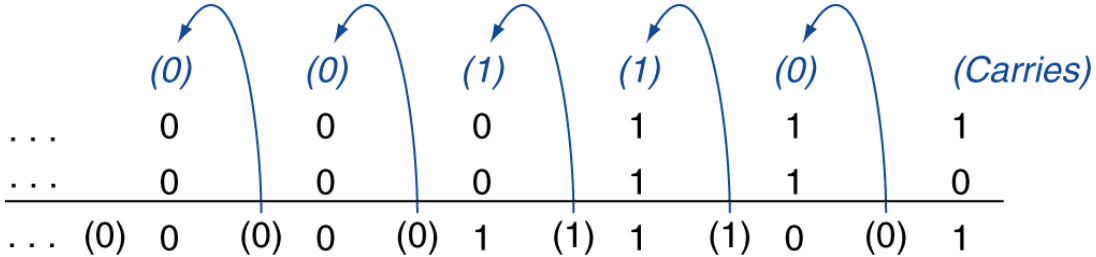
- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - EEC 170 at University of UC Davis
    - https://www.ece.ucdavis.edu/~soheil/private/EEC170/

Outline

- Adder
- Multiplier
- Division
- Floating-point numbers

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (-6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve operands from +ve operand
    - Overflow if result sign is 1

Multiplication

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

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

```
    1010 => 10ten
x   1101 => 13ten
=========
    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

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

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

Multiplication

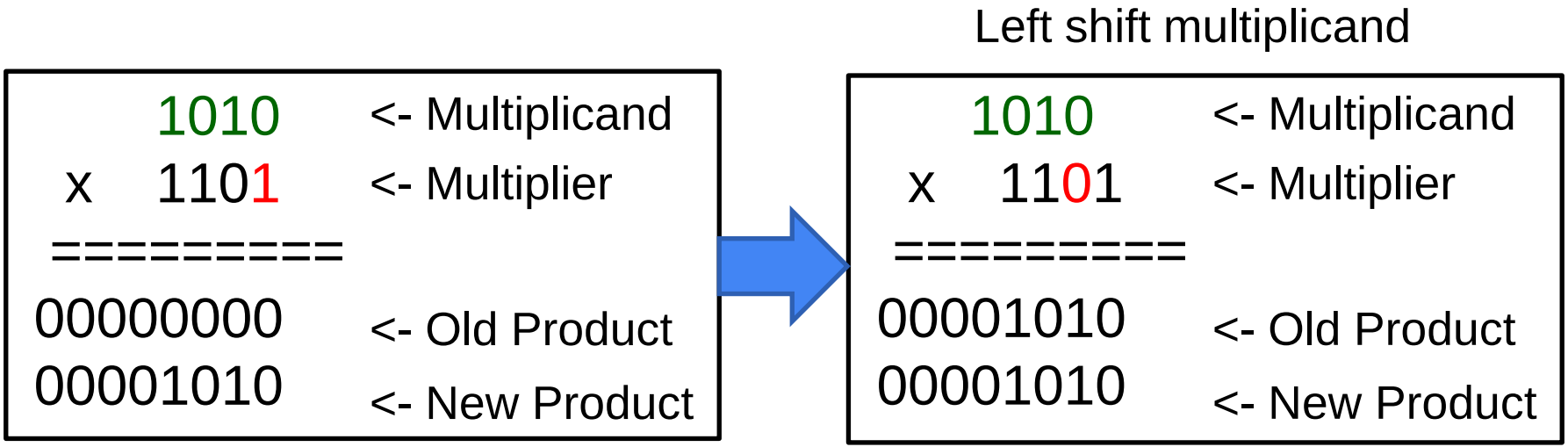
- Length of product is the sum of operand length

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

```
        1010 => 10ten
    x   1101 => 13ten
    ==========
    1111
          1010
         0000
        1010
       1010
    ===========
    10000010 <- Product (130ten)
```

Multiplication

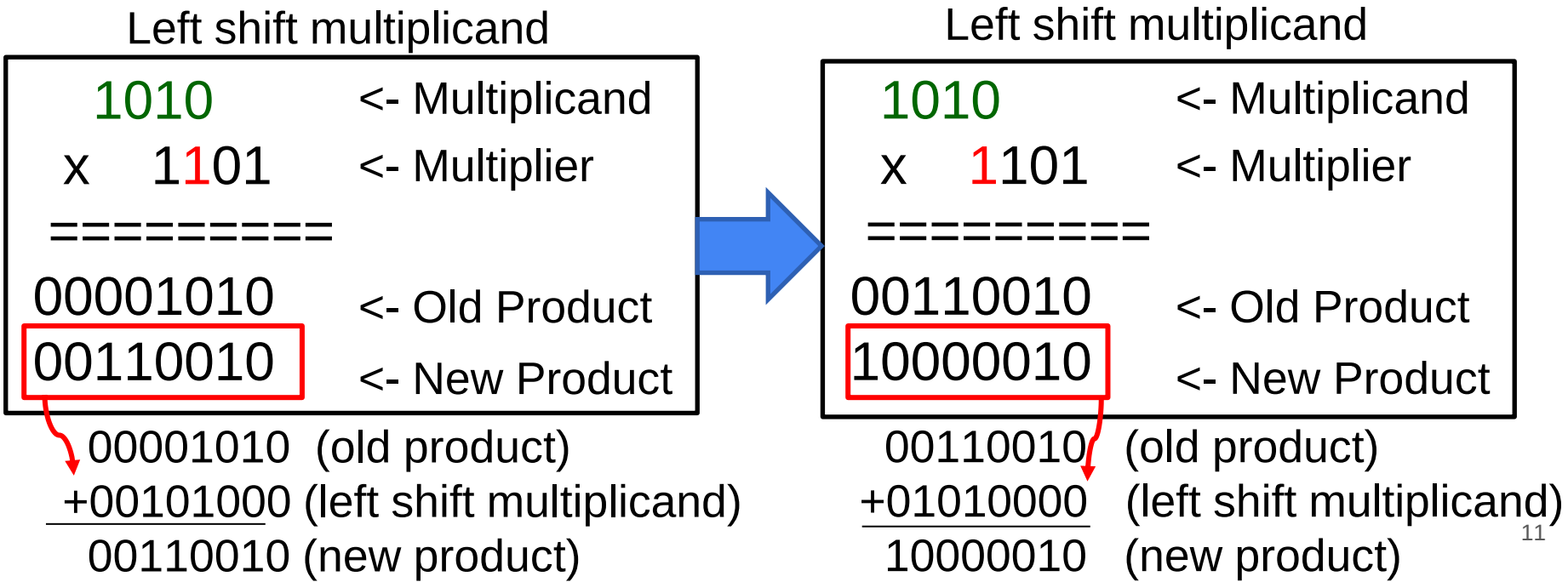
- Shift & Add Multiply



00000000 (old product)
+00001010
00001010 (new product)

Multiplication

- Shift & Add Multiply



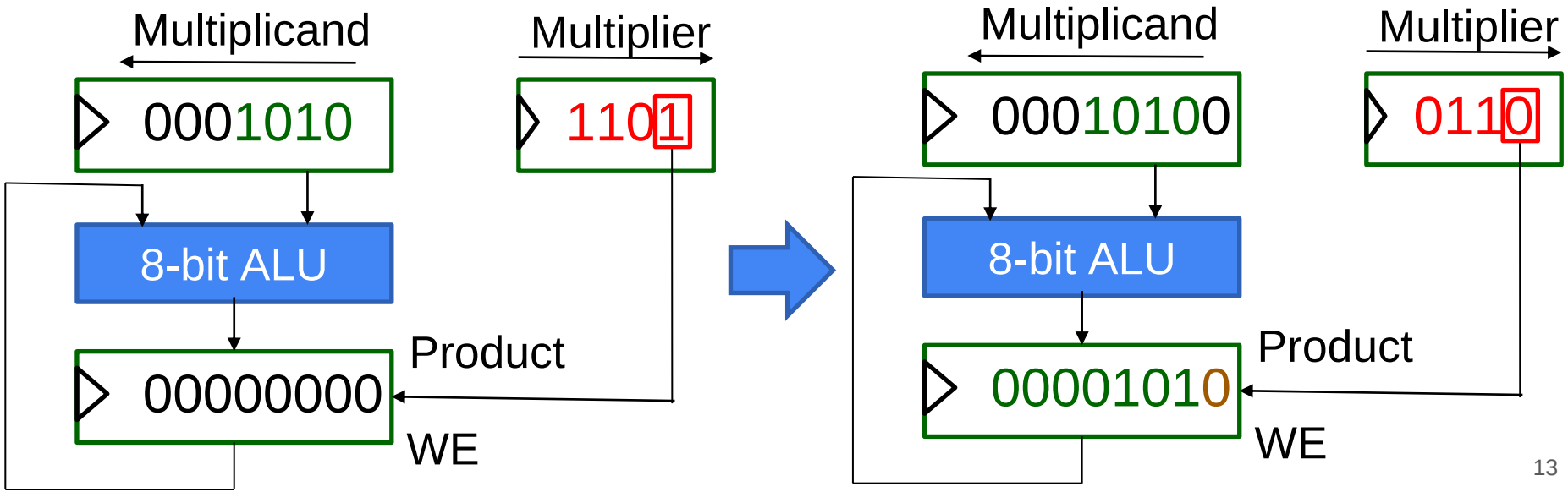
Multiplication

- 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;
```

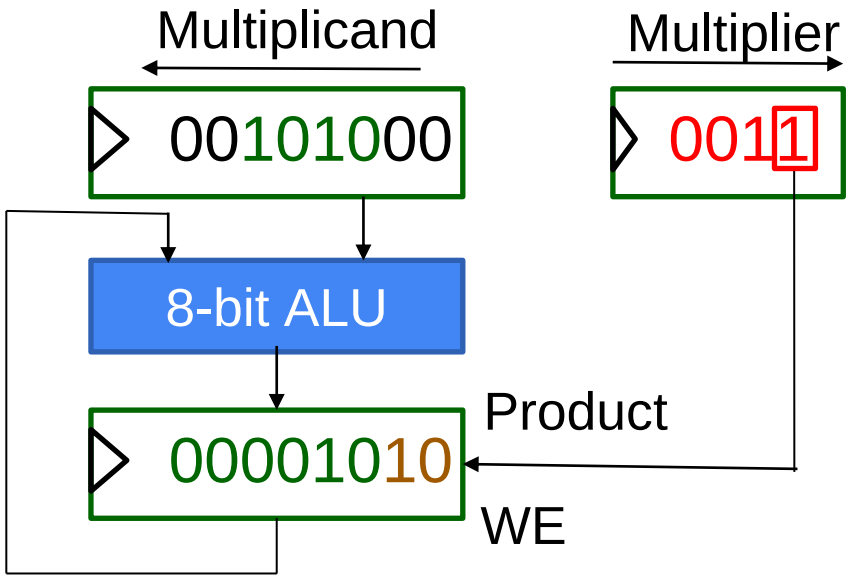
Multiplication

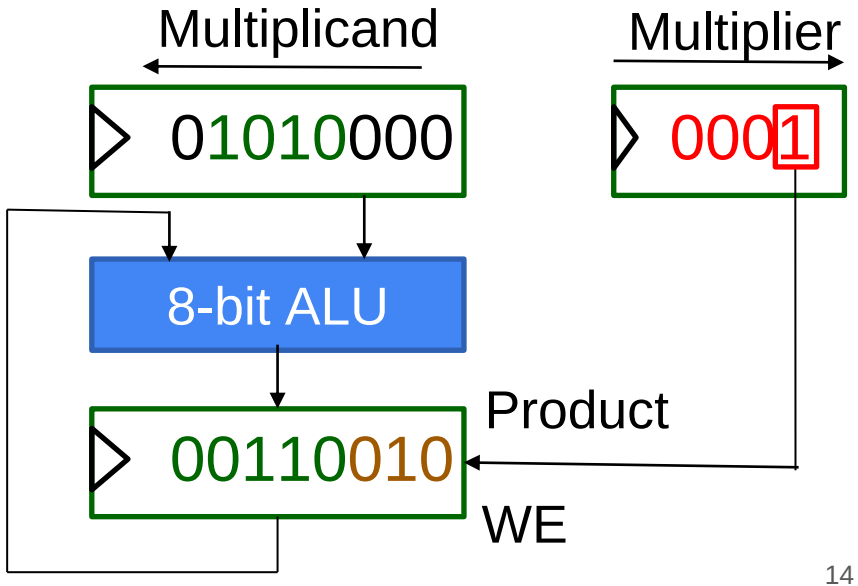
- Simple Shift & Add Multiply Hardware
  - Multiplier LSB is write enable for product latch



Multiplication

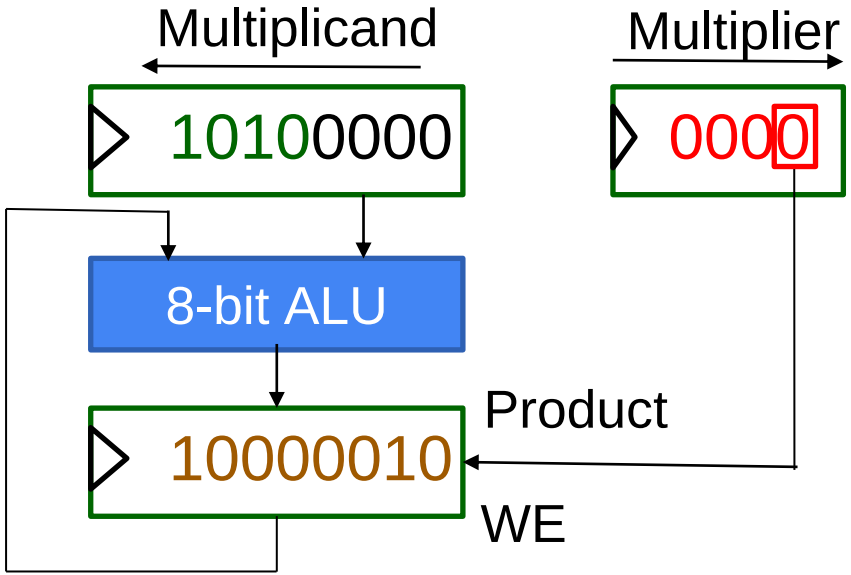
- Simple Shift & Add Multiply Hardware
  - Multiplier LSB is write enable for product latch





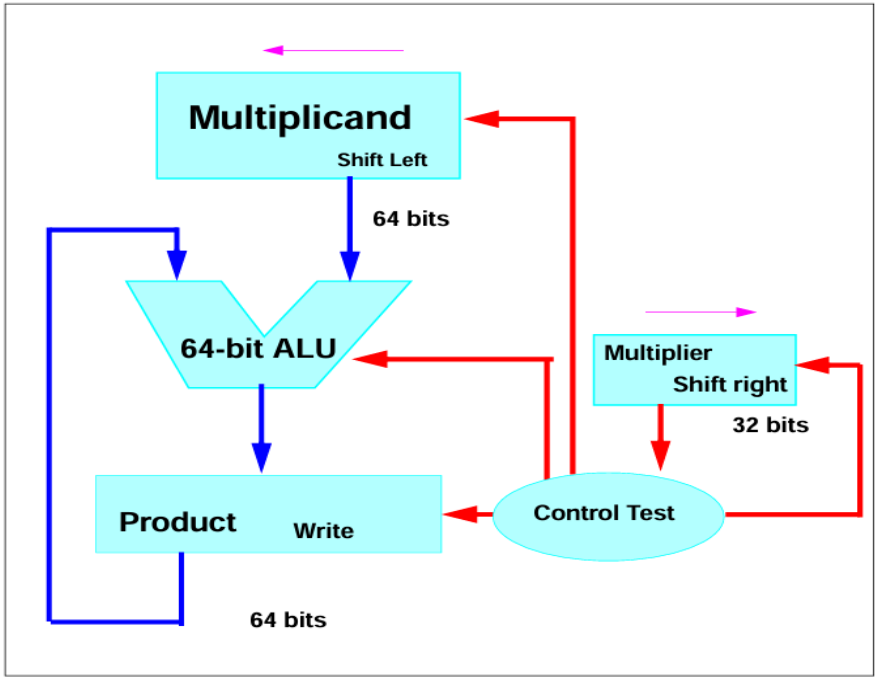
Multiplication

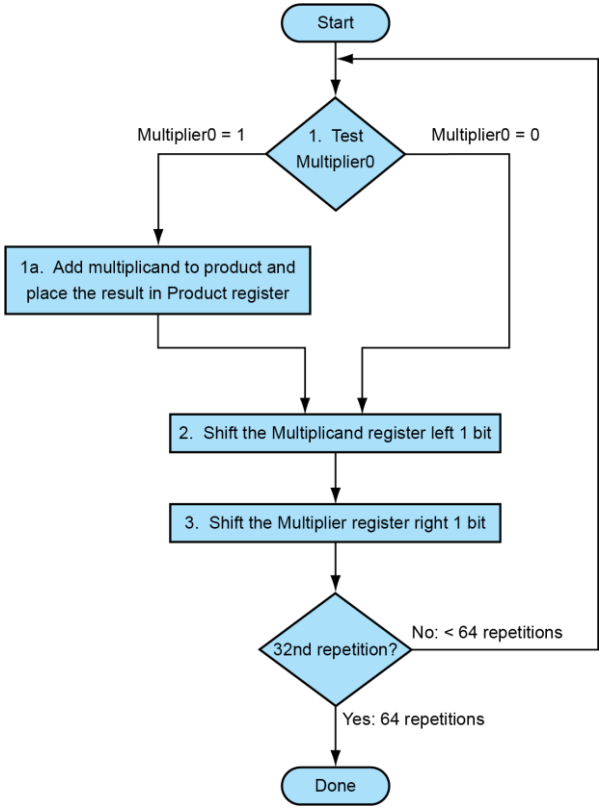
- Simple Shift & Add Multiply Hardware
  - Multiplier LSB is write enable for product latch



First Version of Multiplication Hardware

- Shift & Add Multiply





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

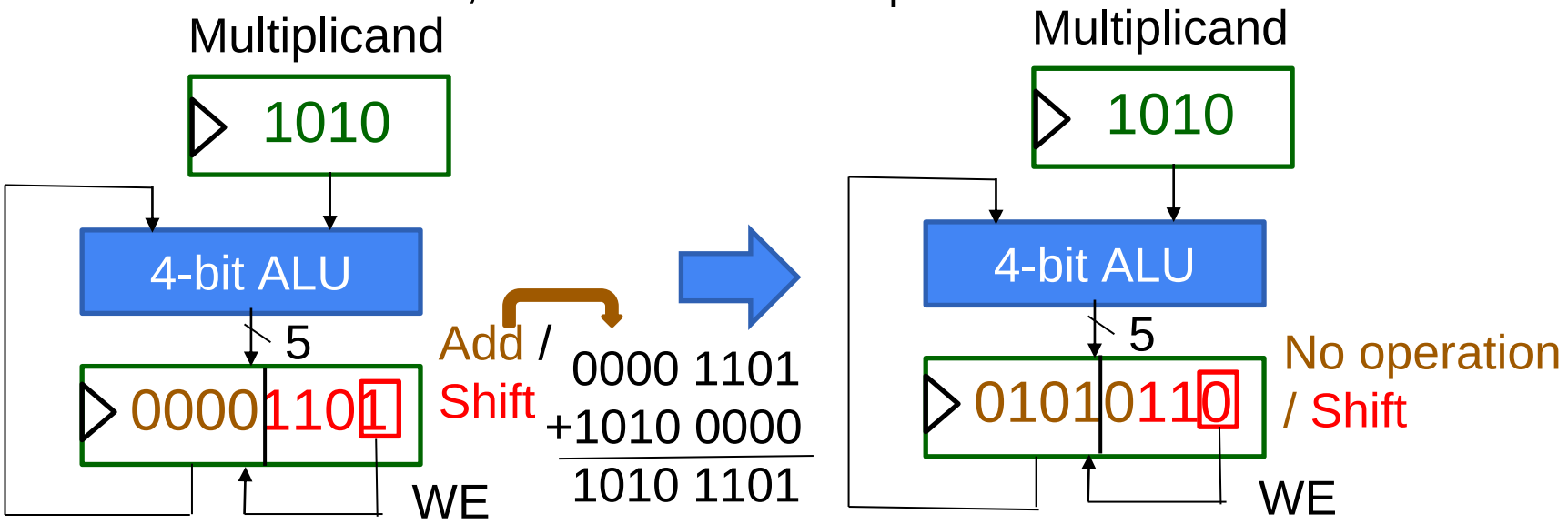
Multiplication

- Refined shift & add hardware (2 (0010_2) x 6 (0110_2))

Iteration	Multiplicand	Step	Product
0	0010	Initial values	0000 0110
1	0010	1: 0 => no operation	0000 0110
	0010	2: shift right product	0000 0011
2	0010	1a: 1=> Prod += Mcand	0010 0011
	0010	2: Shift right product	0001 0001
3	0010	1a: 1=> Prod += Mcand	0011 0001
	0010	2: Shift right product	0001 1000
4	0010	1: 0=> no operation	0001 1000
	0010	2: Shift right product	0000 1100

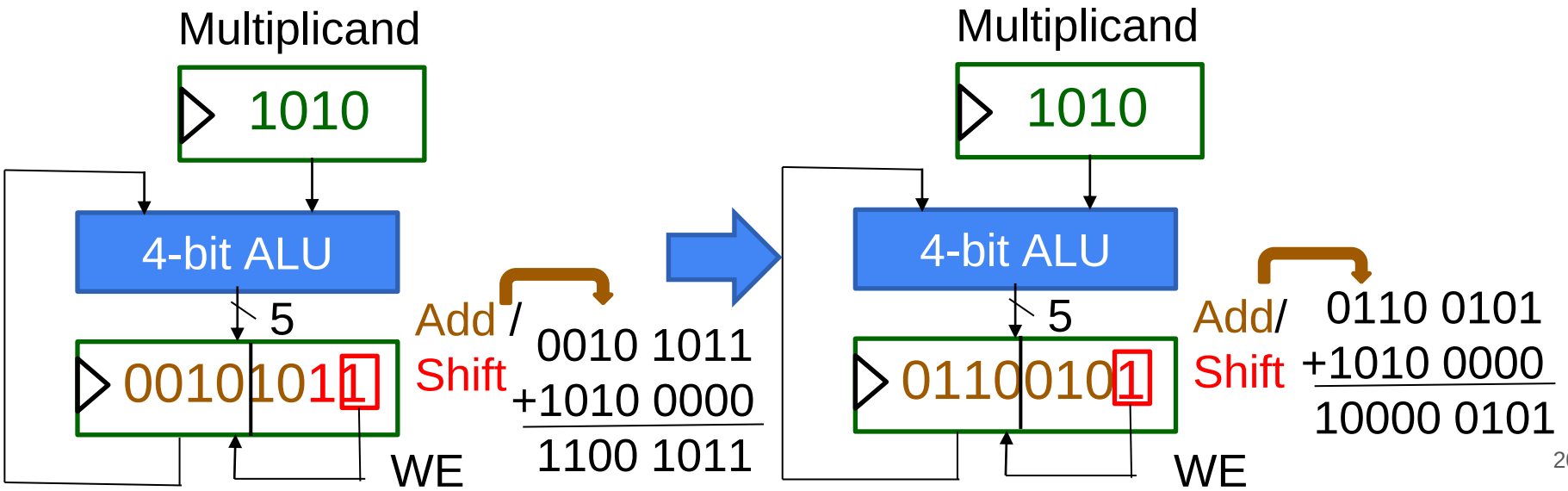
Multiplication

- Refined shift & add hardware (10 (1010_2) x 13 (1101_2))
  - ALU input is accept/not accept based on Wire Enable (WE)
  - When WE = 0, shift but no ALU input



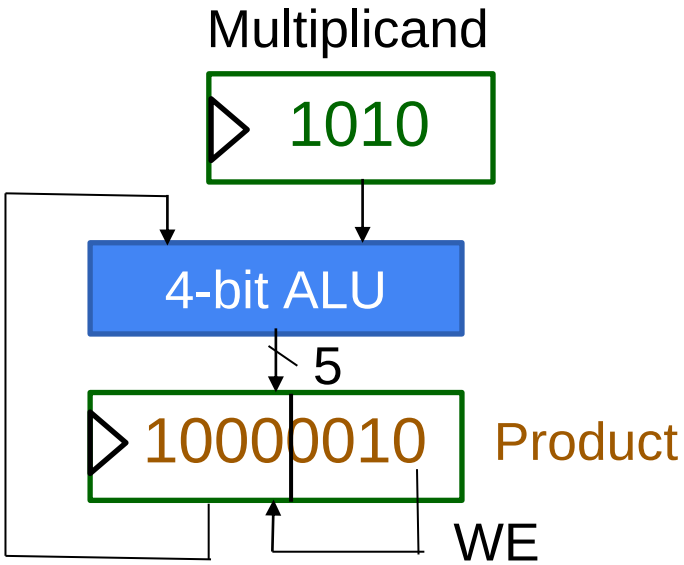
Multiplication

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



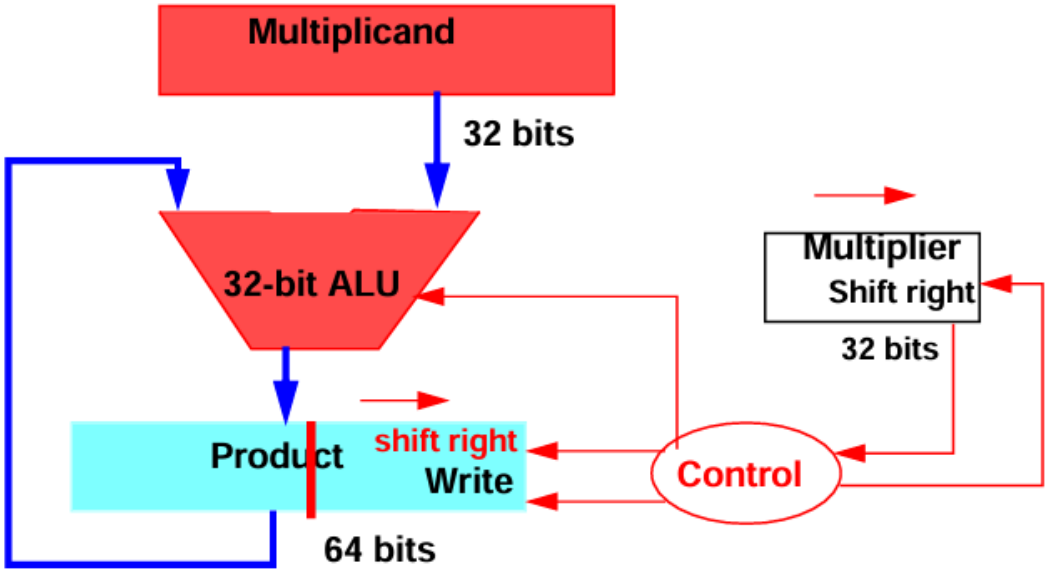
Multiplication

- Refined shift & add hardware
  - Final Result: (10 (1010_2) x 13 (1101_2) = 130 ($1000\ 0010_2$))



Second Version of Multiplication Hardware

- The hardware for this multiplication needs just 32-bit ALU



Second Version MUL Hardware Problems

- 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

Final Version of Multiplication Hardware

- Booth's algorithm
  - Support signed multiplication
  - Handle positive/negative number uniformly
  - E.g. 01110_2 (14_{10}) = 10000_2 (16_{10}) – 00010_2 (2_{10})
    = 10000_2 – 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

2's complement + one additional last bit with 0

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

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

Booth's Algorithm

- Booth's algorithm (2 (0010_2) x 6 (0110_2))

Iteration	Multiplicand	Step	Product
0	0010	Initial values	0000 0110 0
1	0010	1a: 00 => no operation	0000 0110 0
	0010	2: shift right product	0000 0011 0
2	0010	1c: 10=> Prod -= Mcand	1110 0011 0
	0010	2: Shift right product	1111 0001 1
3	0010	1d: 11=> no operation	1111 0001 1
	0010	2: Shift right product	1111 1000 1
4	0010	1b: 01=> Prod += Mcand	0001 1000 1
	0010	2: Shift right product	0000 1100 0

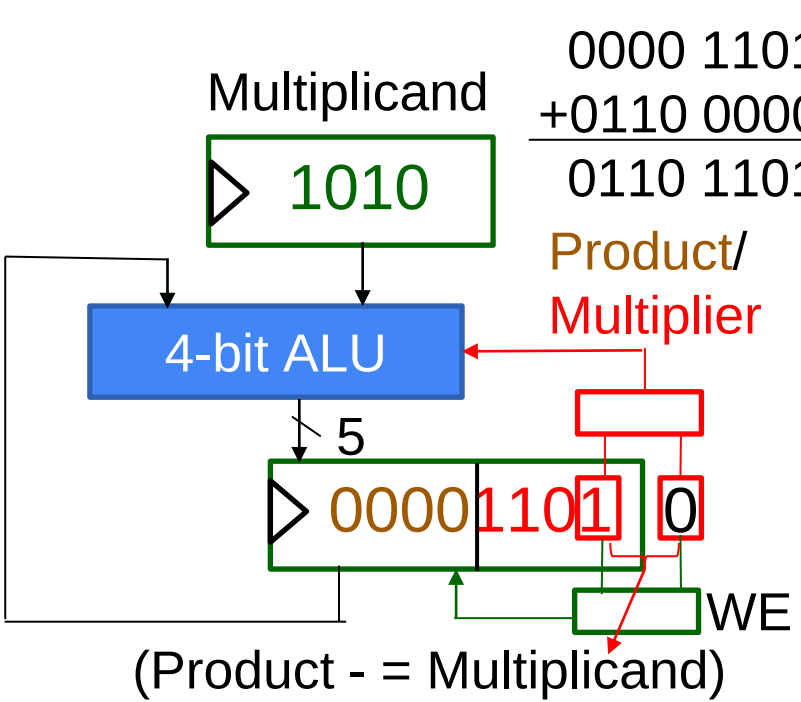
$2_{ten} = 0010_2$
$-2_{ten} = 1110_2$

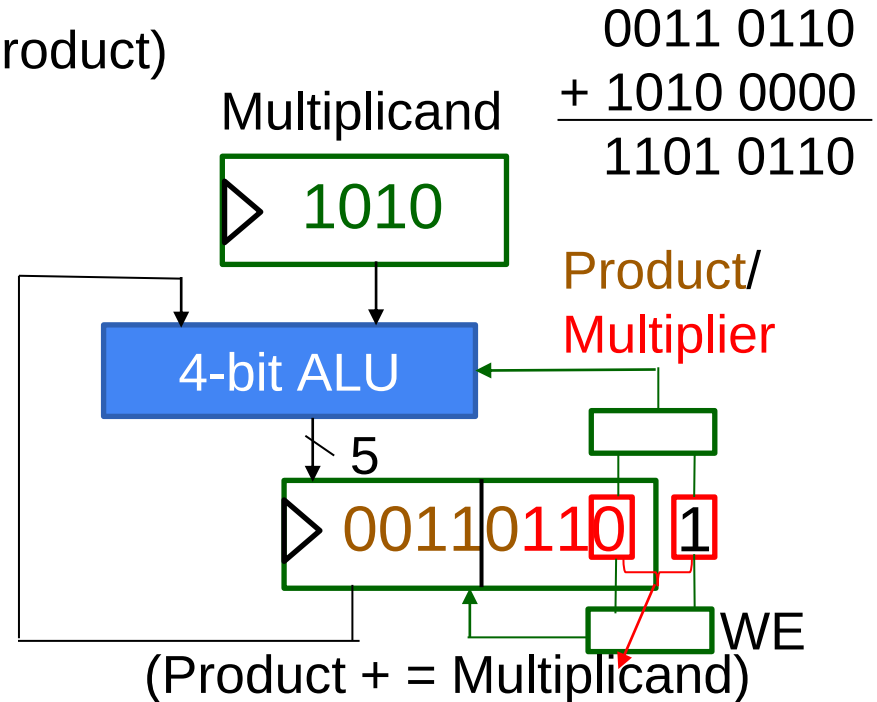
Sign Extension

Booth's Algorithm Hardware

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

- Example: -6 (1010_2) x -3(1101_2)

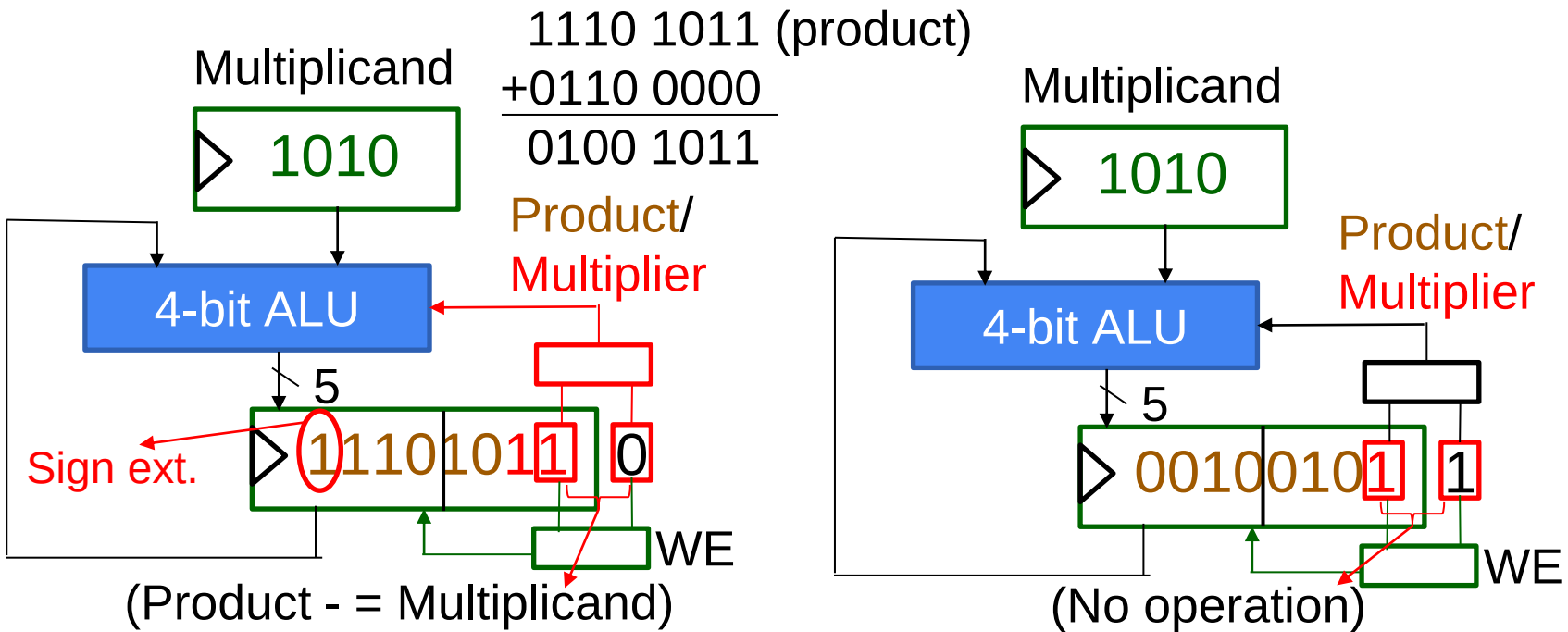




Booth's Algorithm Hardware

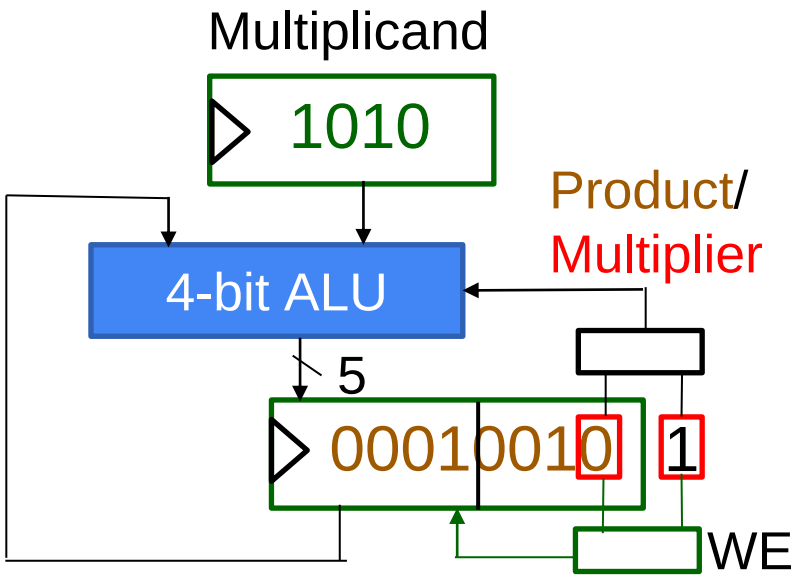
		i	i-1
-1	for	1	0
+1	for	0	1
0	for	0	0
0	for	1	1

- Example: -6 (1010_2) x -3(1101_2)



Booth's Algorithm Hardware

- Final Result: -6 (1010_2) x -3(1101_2) = 18 (0001001_2)



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

Division

Dividend = Quotient x Divisor + Remainder

- Consider a long division example

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

Quotient: 0005678_{10}

7006

-6170

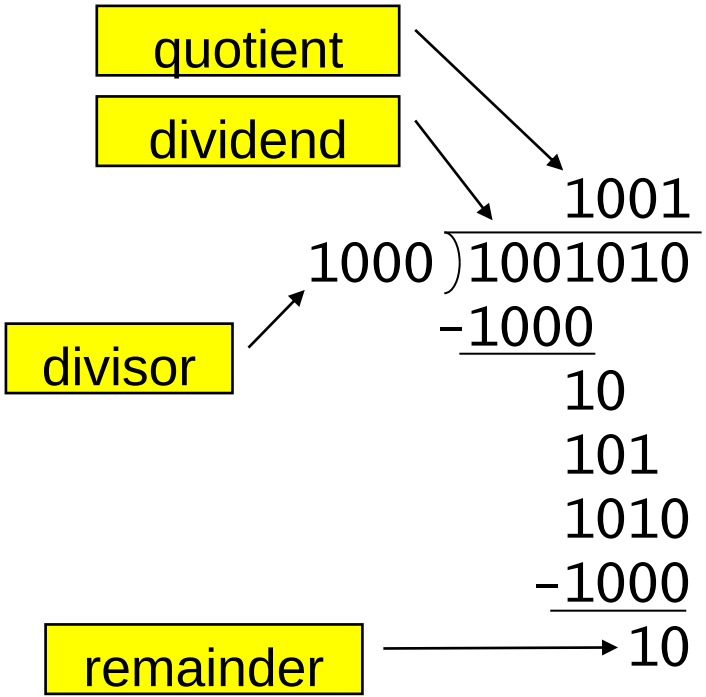
8367 Partial Remainder

-7404

…

Remainder: 137_{10}

Division

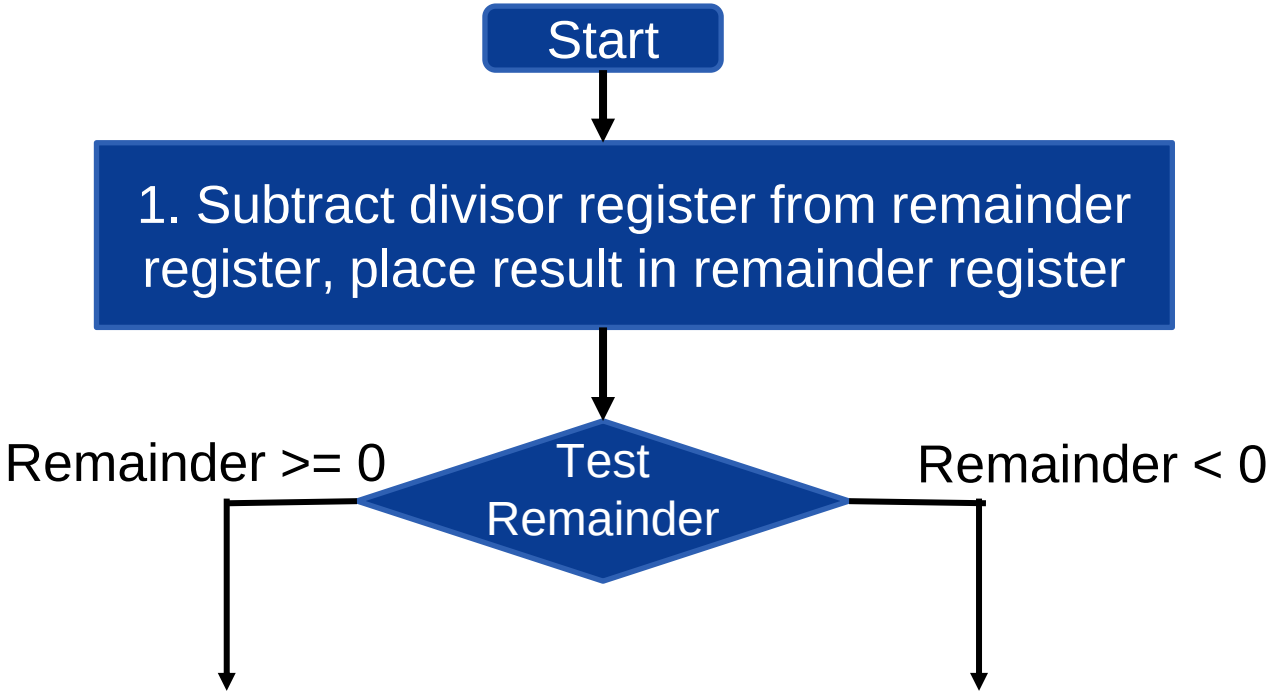


- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

n-bit operands yield *n*-bit quotient and remainder

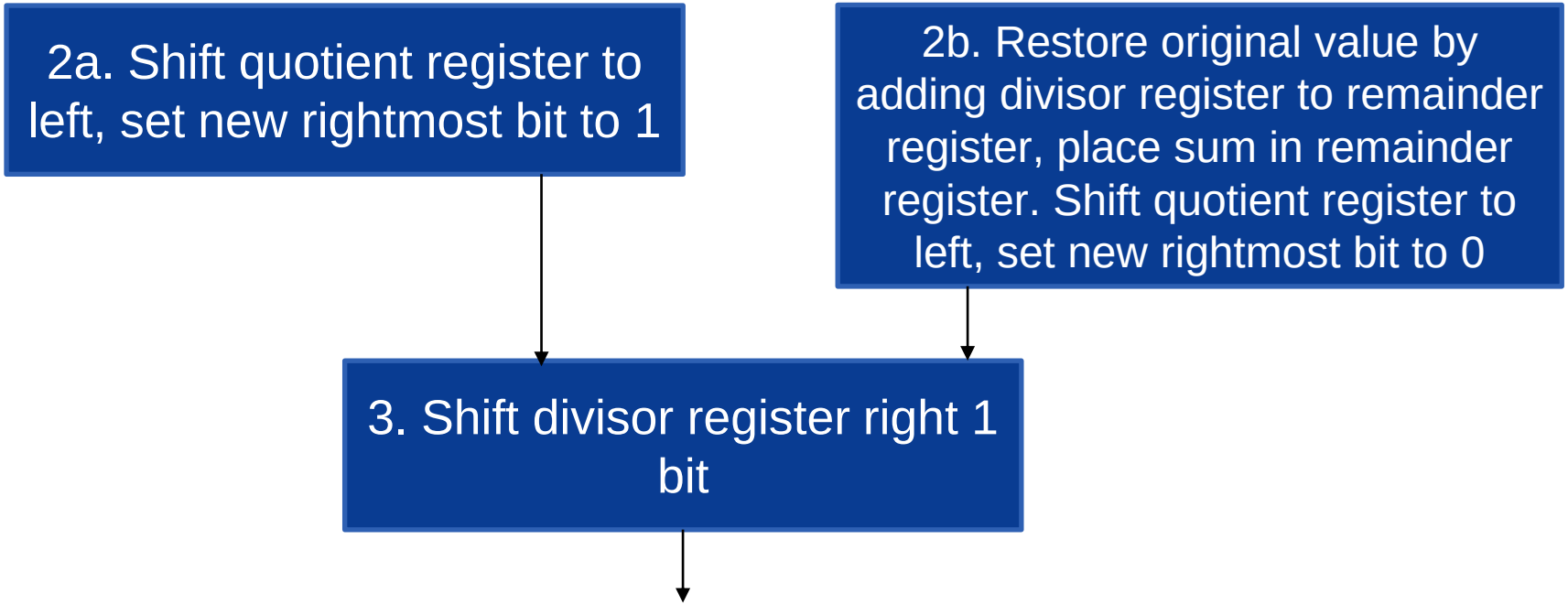
Division

- Simple division algorithm



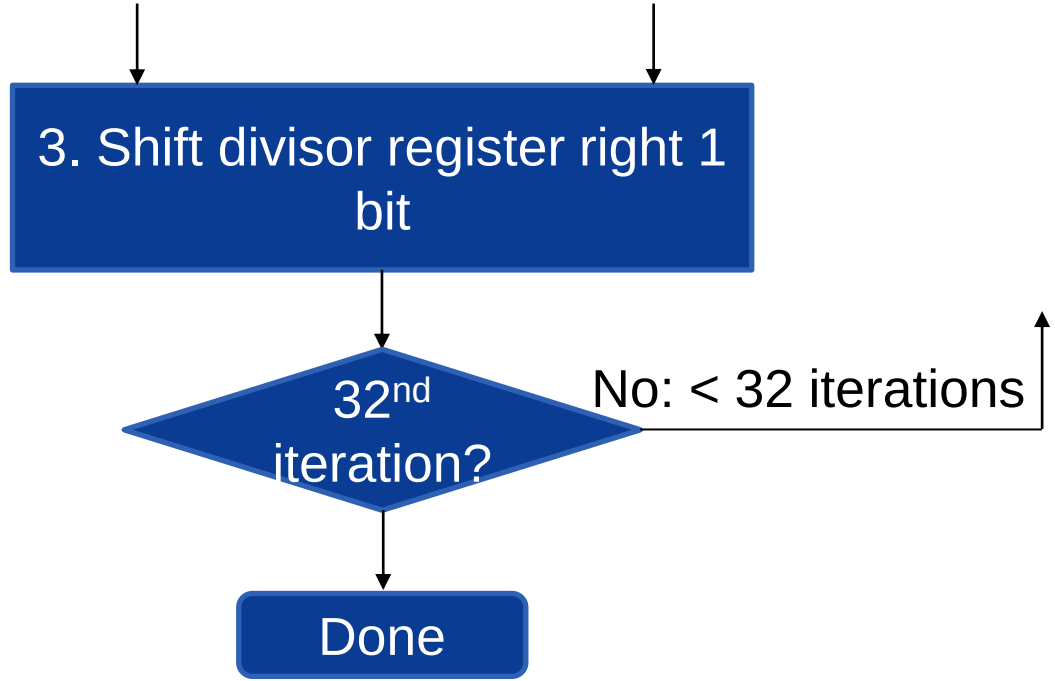
Division

- Simple division algorithm



Division

- Simple division algorithm



Division

LSB of Quotient = 0

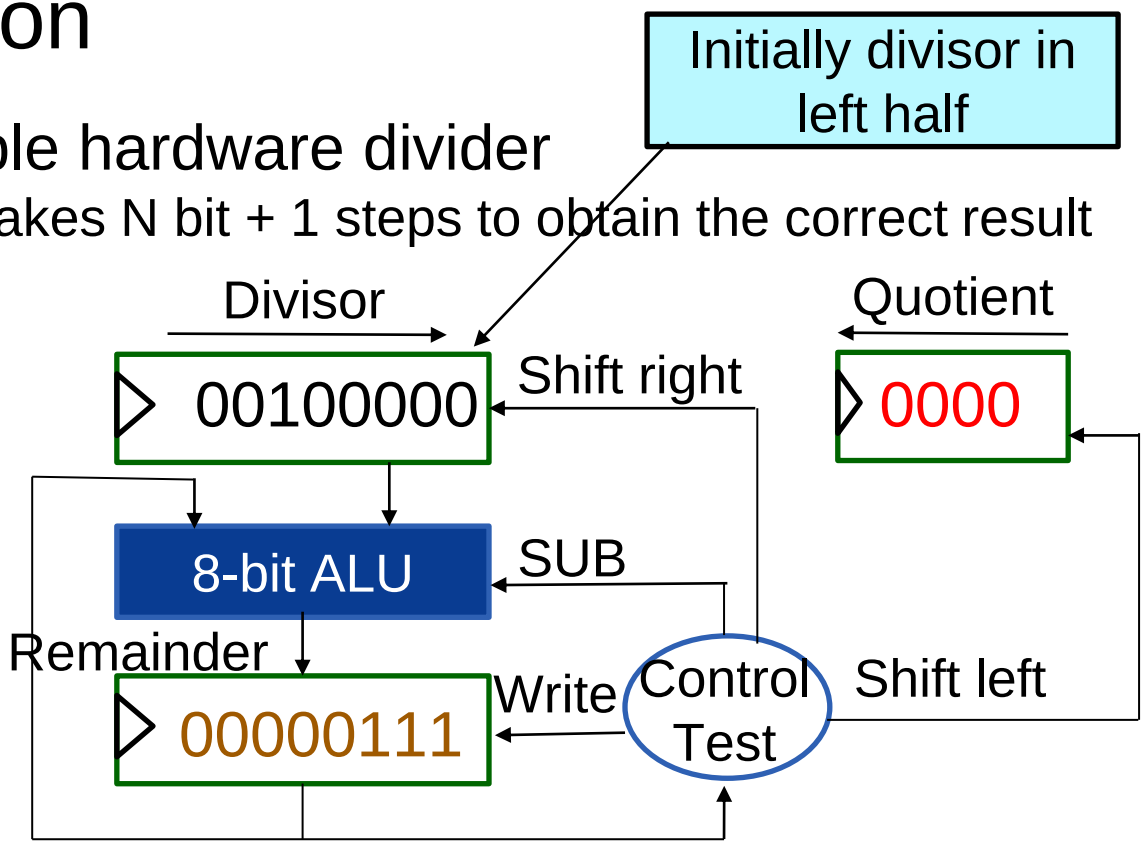
- 4-bit value division: $7_{ten} / 2_{ten} = 0000\ 0111_{two} / 0010_{two}$

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Values	0000	0010 0000	0000 0111
1	1: Rem -= Div	0000	0010 0000	1110 0111
	2b: Rem < 0 => +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem -= Div	0000	0001 0000	1111 0111
	2b: Rem < 0 => +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem -= Div	0000	0000 1000	1111 1111
	2b: Rem < 0=> +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem -= Div	0000	0000 0100	0000 0011
	2a: Rem >=0 => sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem -= Div	0001	0000 0010	0000 0001
	2a: Rem >= 0 => sll Q, Q0 = 1	0011	0000 0010	0000 0001
	1: Shift Div right	0011	0000 0001	0000 0001

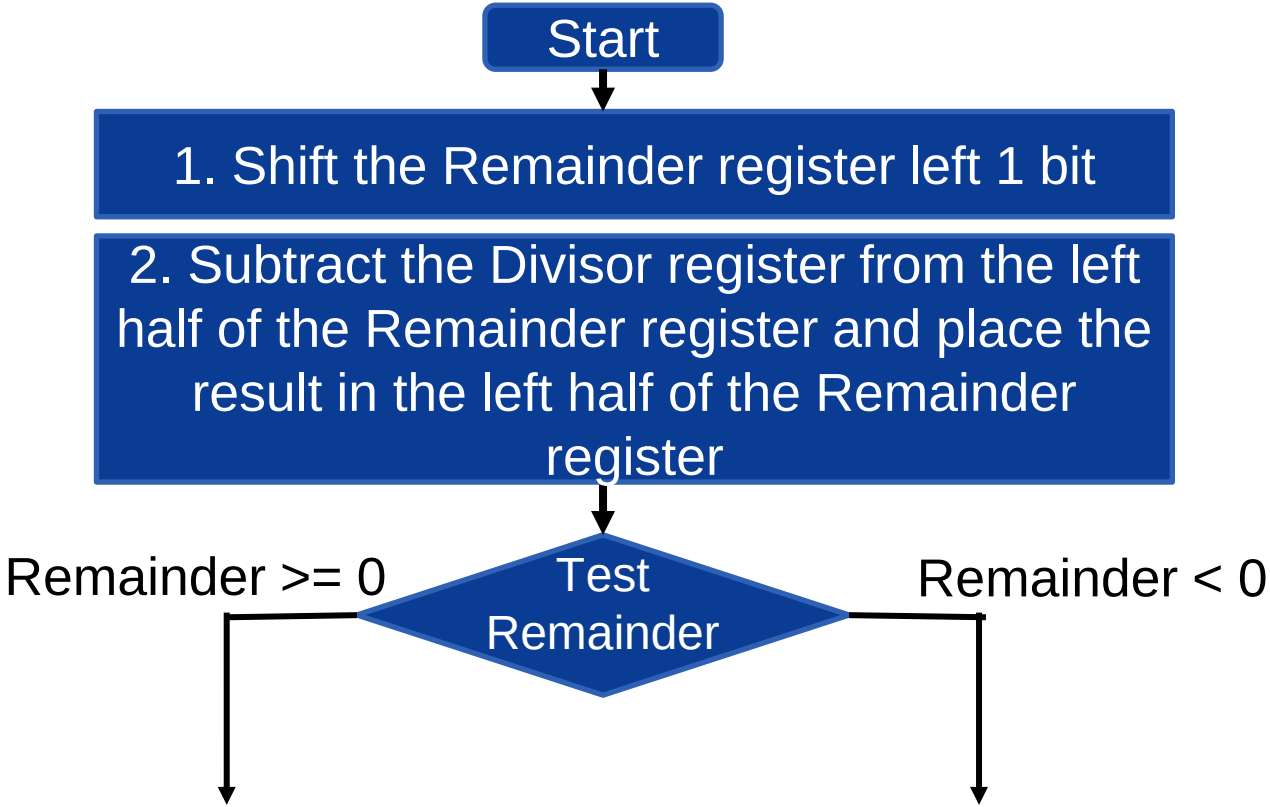
```
  0000 0111
- 0010 0000
-----------
= 0000 0111
+ 1110 0000
-----------
  1110 0111
```

Division

- Simple hardware divider
  - Takes N bit + 1 steps to obtain the correct result



Optimized Divider

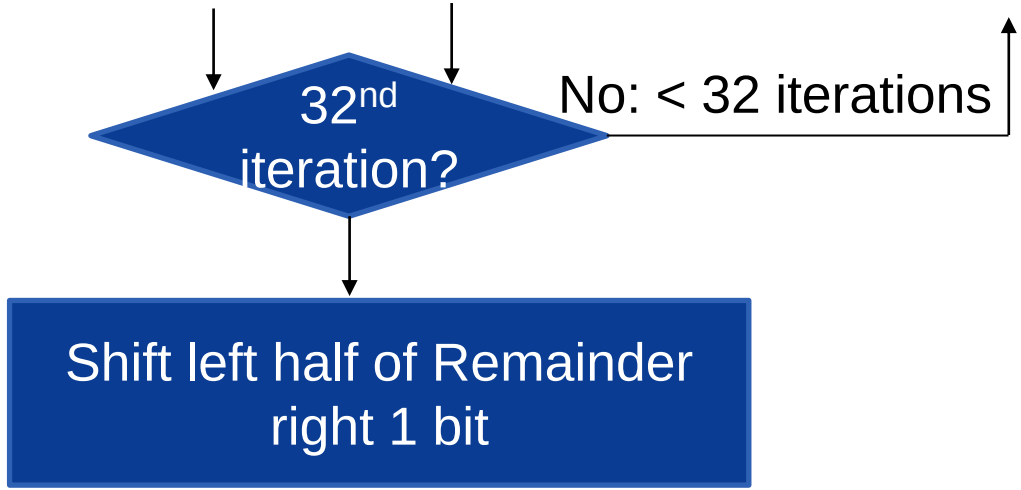


Optimized Divider

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1

3b. Restore original value by adding divisor register to remainder register, place sum in remainder register. Shift quotient register to left, set new rightmost bit to 0

Optimized Divider



Optimized Divider

LSB of Remainder = 0

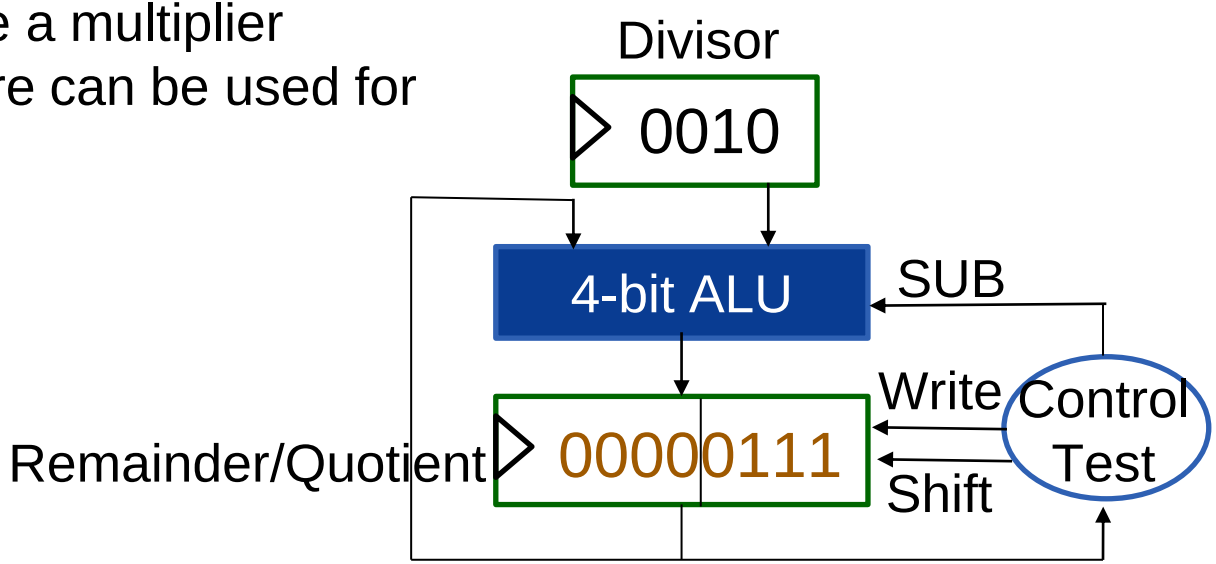
- 4-bit value division: 7_{ten} / 2_{ten} = $0000\ 0111_{two}$ / 0010_{two}

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem -= Div	0010	1110 1110
	3b: Rem < 0 => +Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem -= Div	0010	1111 1100
	3b: Rem < 0 => +Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem -= Div	0010	0001 1000
	3a: Rem>=0 => sll R, R0 = 1	0010	0011 0001
4	2: Rem -= Div	0010	0001 0001
	3a: Rem>= 0 => sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

Quotient

Optimized Divider

- Improved hardware divider
  - Divider, ALU are reduced in half
  - One cycle per partial remainder subtraction
  - Looks a lot like a multiplier
  - Same hardware can be used for both



Right Shift and Division

- 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
```

Number Systems

- Fixed-point notation
  - Has an implied binary point between the integer and fraction bits

0110.1100

Integer bits Fraction bits

$= 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

Number Systems

- Fixed-point notation
  - Signed fixed-point numbers can use either two's complement or sign/magnitude notation
  - E.g. How to represent -2.375 using fixed-point notations with four integer and four fraction bits

(a) 2.375 = 0010.0110 (absolute value)
(b) 1010.0110 (sign and magnitude)
(c) -2.375 = 1101.1010 (two's complement)

Two's complement inverts the bit of the absolute value and adding a 1 to the LSB

Number Systems

- Floating-point number is composed of a
  - Sign
  - Mantissa (M)
  - Base (B)
  - Exponent (E)

Floating-Point Numbers

- Binary point floats to the right of the most significant 1
- Similar to decimal scientific notation

- For example, write 273_{10} in scientific notation:

$$\mathbf{273 = 2.73 \times 10^2}$$

- In general, a number is written in scientific notation as:

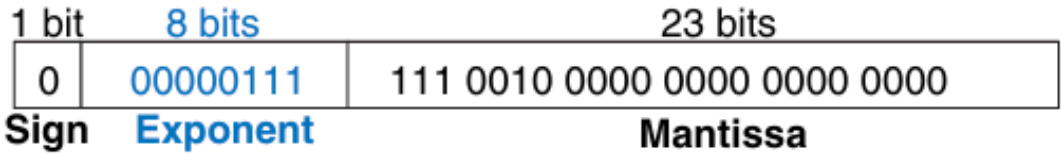
$$\mathbf{\pm M \times B^E}$$

  - **M** = mantissa
  - **B** = base
  - **E** = exponent
  - In the example, M = 2.73, B = 10, and E = 2

Floating-Point Numbers

- IEEE 754 32-bit Floating-point format
  - What is the floating-point representation of decimal number 228?
    - $228_{10} = 11100100_2 = 1.11001_2 \times 2^7$
    - The sign bit is positive (0)
    - The 8 exponent bits give the value 7
    - The remaining 23 bits are mantissa (111 001….)

$$\pm M \times B^E$$



Floating-Point Numbers

- IEEE 754 32-bit Floating-point format
  - The exponent needs to represent both positive and negative exponents
    - Uses a biased exponent, which is the original exponent plus a constant bias
    - 32-bit floating-point uses a bias of 127

$$value = (-1)^{s} \times 1.M \times 2^{E-127}$$

Floating-Point Numbers

- **Single-Precision:**
  - 32-bit
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

- **Double-Precision:**
  - 64-bit
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
  - Actual exponent = 1 – 127 = -126
  - Fraction: 000…00
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - Exponent: 11111110
  - Actual exponent = 254 – 127 = +127
  - Fraction: 111…11
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
  - Actual exponent = 1 – 1023 = -1022
  - Fraction: 000…00
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111110
  - Actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Numbers

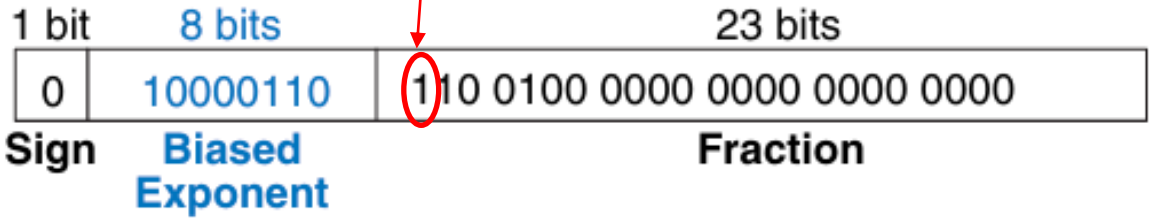
- **How to convert 10.875 to IEEE 754 FP Format ?**
  - Step 1: Write the number in binary
    - $1010.111_2 = 1.010111000\ldots * 2^3$
  - Step 2: Determine Sign/Exponent/Mantissa
    - Sign = Positive ->0
    - Exponent: 3-(-127) = 130 -> $1000\ 0010_2$
    - Mantissa: 1010 1110 0000 0000 0000 0000
  - Step 3: Concatenate
    - 0 100 0001 0 101 0111 0000 0000 0000 0000

S Exponent Mantissa

Floating-Point Numbers

- IEEE 754 32-bit Floating-point format
  - How to represent $1.11001_2 \times 2^7$ in IEEE 754 format?
    - For the exponent 7, the biased exponent is $7 - (-127) = 134 = 10000110_2$
    - With an implicit leading one

$$value = (-1)^S \times 1.M \times 2^{E-127}$$



Floating-Point Numbers

- **How to convert 0xC3CC0000 to decimal?**
  - Step 1: Write the number in binary
    - 1100 0011 1100 1100 0000 0000 0000 0000
    - C 3 C C 0 0 0 0
  - Step 2: Determine Sign/Exponent/Mantissa
    - Sign = Negative ->1
    - Exponent: $1000\ 0111_2$ -> 135-(127) = 8
    - Mantissa: 1001 1000 … -> 1.001 1000 = $1 + 2^{-3} + 2^{-4}$
    - $(1 + 2^{-3} + 2^{-4}) * 2^8 = 2^8 + 2^5 + 2^4 = 304$

The implicit Leading 1

- Our mantissa is guaranteed not to have any leading zeros
  - If we wanted to write $0.234*10^5$, we'd instead write it as $2.34*10^4$
- In binary, every digit is only either 1 or 0
  - Since the MSB can't be 0, it must therefore be 1
  - **If the first bit will always be 1, we don't need to store it!**
  - We can save 1 bit (or alternatively add another bit of precision) to the mantissa by not including the MSB of the mantissa
    - This is known as the implicit 1
    - The resulting mantissa is a "normalized" number

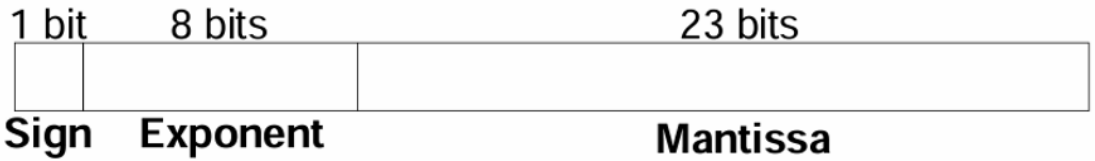
The implicit Leading 1

- **How to convert 10.875 to IEEE 754 FP Format?**
  - Step 1: Write the number in binary
    - $1010.111_2 = 1.010111000\ldots * 2^3$
  - Step 2: Determine Sign/Exponent/Mantissa
    - Sign = Positive ->0
    - Exponent: 3-(-127) = 130 -> $1000\ 0010_2$
    - Mantissa: 0101 1100 0000 0000 0000 000
  - Step 3: Concatenate
    - 0100 0001 0010 1110 0000 0000 0000 0000

S Exponent Mantissa

Takeaway Question

- How to represent -58.25_{10} in the IEEE 754 format with implicit leading 1?



Takeaway Question

- How to represent -58.25_{10} in the IEEE 754 format with implicit leading 1?

Write **-58.25_{10}** in floating point (IEEE 754)

1. Convert magnitude of decimal to binary:

   $58.25_{10} = 111010.01_2$

2. Write in binary scientific notation:

   1.1101001×2^5

3. Fill in fields:

   Sign bit: 1 (negative)
   8 exponent bits: (127 + 5) = 132 = 10000100_2
   23 fraction bits: 110 1001 0000 0000 0000 0000

in hexadecimal: **0xC2690000**

Infinities and NaNs

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

Rounding & Overflow

- **Overflow:** number too large to be represented
- **Underflow:** number too small to be represented
- **Rounding modes:**
  - Down
  - Up
  - Toward zero
  - To nearest

Rounding & Overflow

- **Example:** round 1.100101 (1.578125) to only 3 fraction bits
  - **Down:** 1.100
  - **Up:** 1.101
  - **Toward zero:** 1.100
  - **To nearest:** 1.101 (1.625 is closer to 1.578125 than 1.5 is)

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

Floating-Point Addition

- Consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

Floating-Point Addition

- Consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
  - Since 127≥-4≥-126 => no overflow or underflow
  - The biased exponent would be -4+127 = 123, which is between 1 and 254 (the smallest and largest unreserved biased exponents)
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625
  - The sum is already fits exactly in 4 bits

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware

Floating-Point Multiplication

- Consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
  - Unbiased: -1 + -2 = -3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. Multiply significands
  - We need to keep it 4 bits
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$

$$
\begin{array}{rr}
 & 1.000_2 \\
\times & 1.110_2 \\
\hline
 & 0000 \\
 & 1000 \\
 & 1000 \\
 & 1000 \\
\hline
 & 1110000
\end{array}
$$

Floating-Point Multiplication

- Consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
  - The product is normalized, since 127≥-3≥-126 => no overflow
  - 254≥124 (Bias)≥1 => the exponent fits
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)

Floating-Point Multiplication

- Consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -7/2^5_{ten} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP<-> integer conversion
- Operations usually takes several cycles
  - Can be pipelined

RISC-V FP Instructions

- Separate FP registers: f0, …, f31
  - double-precision
  - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - flw, fld
  - fsw, fsd

RISC-V FP Instructions

- Single-precision arithmetic
  - fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s
    - e.g., fadds.s f2, f4, f6
- Double-precision arithmetic
  - fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d
    - e.g., fadd.d f2, f4, f6

RISC-V FP Instructions

- Single- and double-precision comparison
  - feq.s, flt.s, fle.s
  - feq.d, flt.d, fle.d
  - Result is 0 or 1 in integer destination register
    - Use beq, bne to branch on comparison result
- Branch on FP condition code true or false
  - B.cond

Conclusions

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow