Tikrit University
Electrical Engineering Department

EE-317
Computer Engineering
2024-2025

Arithmetic: for Integers

Jalal Nazar Abdulbaqi, Ph.D.

jalal.abdulbaqi@tu.edu.iq

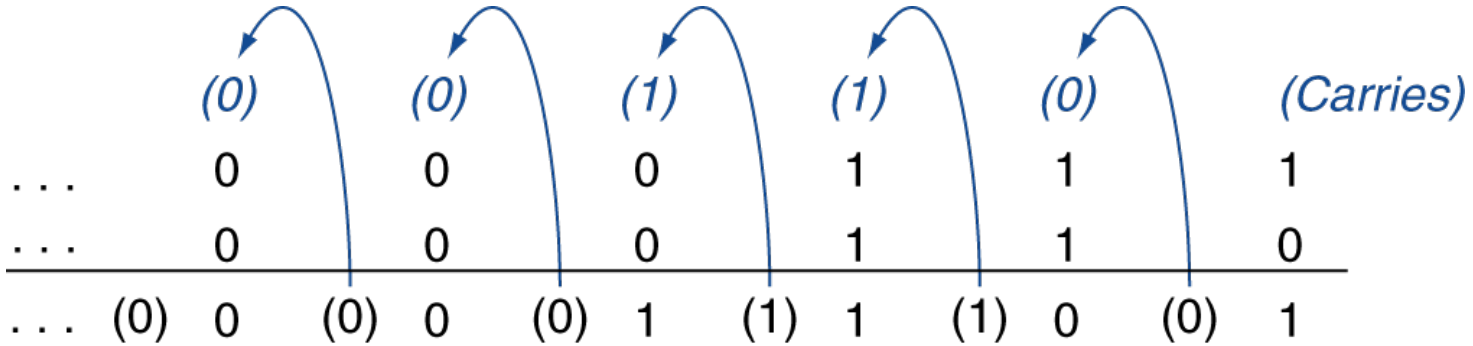
Outline

- Arithmetic on Integers
  - Addition and subtraction
  - Dealing with overflow
  - Multiplication and division

Integer Addition

- Example: 7 + 6

$$7_{ten} = \ldots 0\ 0\ 0\ 1\ 1\ 1_{two}$$
$$6_{ten} = \ldots 0\ 0\ 0\ 1\ 1\ 0_{two} +$$
$$13_{ten} = \ldots 0\ 0\ 1\ 1\ 0\ 1_{two}$$



- Overflow if result out of range (larger than 32-bit)

Integer Subtraction

- Example: 7 – 6

$$
\begin{array}{rl}
7_{ten} & = \ldots 0\ 0\ 0\ 1\ 1\ 1_{two} \\
6_{ten} & = \ldots 0\ 0\ 0\ 1\ 1\ 0_{two} \quad - \\
\hline
1_{ten} & = \ldots 0\ 0\ 0\ 0\ 0\ 1_{two}
\end{array}
$$

two's complement

$$
\begin{array}{rl}
7_{ten} & = \ldots 0\ 0\ 0\ 1\ 1\ 1_{two} \\
-6_{ten} & = \ldots 1\ 1\ 1\ 0\ 1\ 0_{two} \quad + \\
\hline
1_{ten} & = \ldots 0\ 0\ 0\ 0\ 0\ 1_{two}
\end{array}
$$

Names of the Operands

Subtraction	
A	Minuend
−	
B	Subtrahend
=	
C	Difference

Addition	
A	Addend
+	
B	Addend
=	
C	Sum

Overflow in Signed Numbers

Operation	Operand Sign	Result	Overflow
Addition	Different		✗
	Same	$\oplus + \oplus = \ominus$	✓
		$\ominus + \ominus = \oplus$	✓
Subtraction	Different	$\oplus - \ominus = \ominus$	✓
		$\ominus - \oplus = \oplus$	✓
	Same		✗

Overflow in Unsigned Numbers

Operation	Format	Overflow
Addition	A+B = C	C < A (or) C < B
Subtraction	A-B = C	C > A

Multiplication

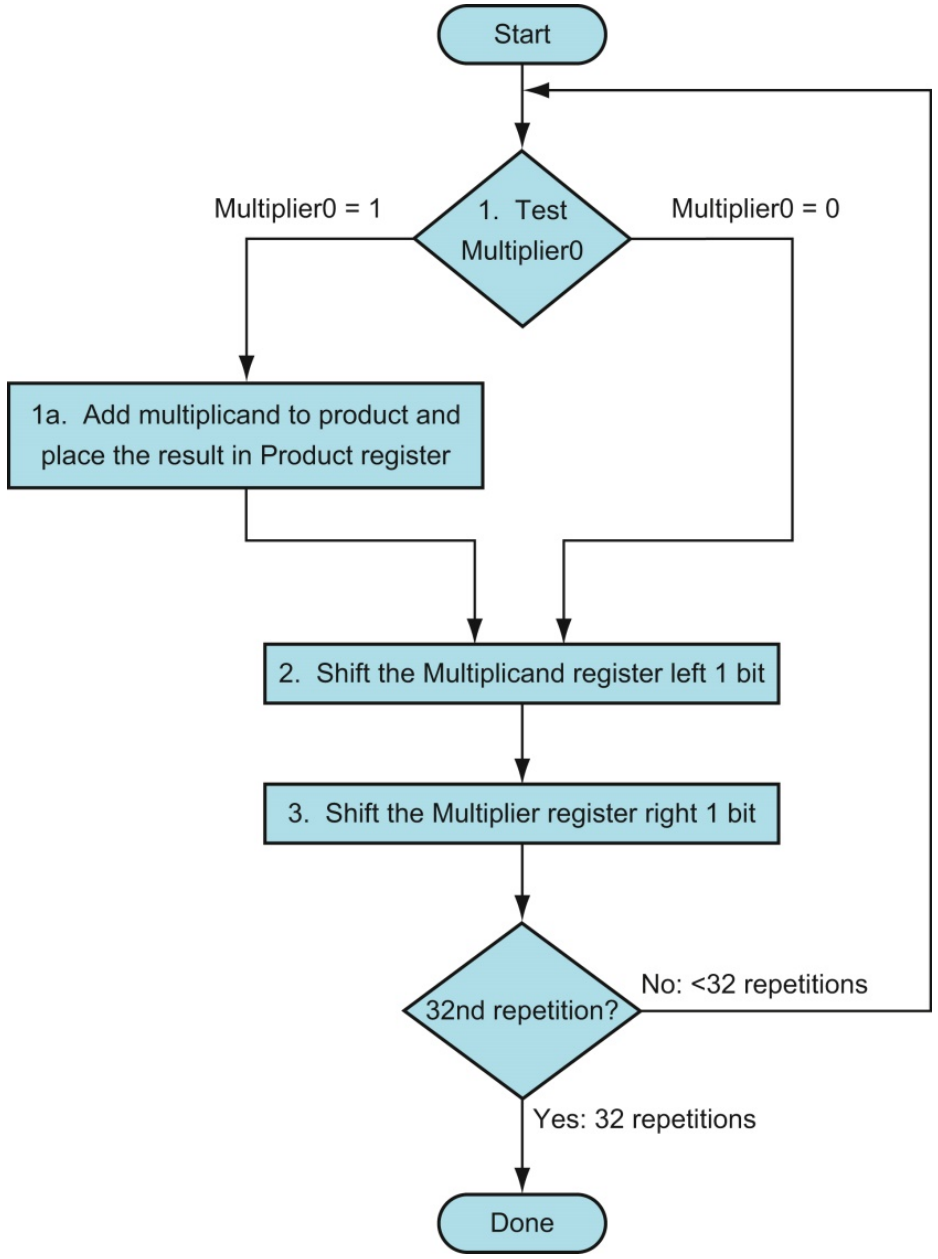
multiplicand

multiplier

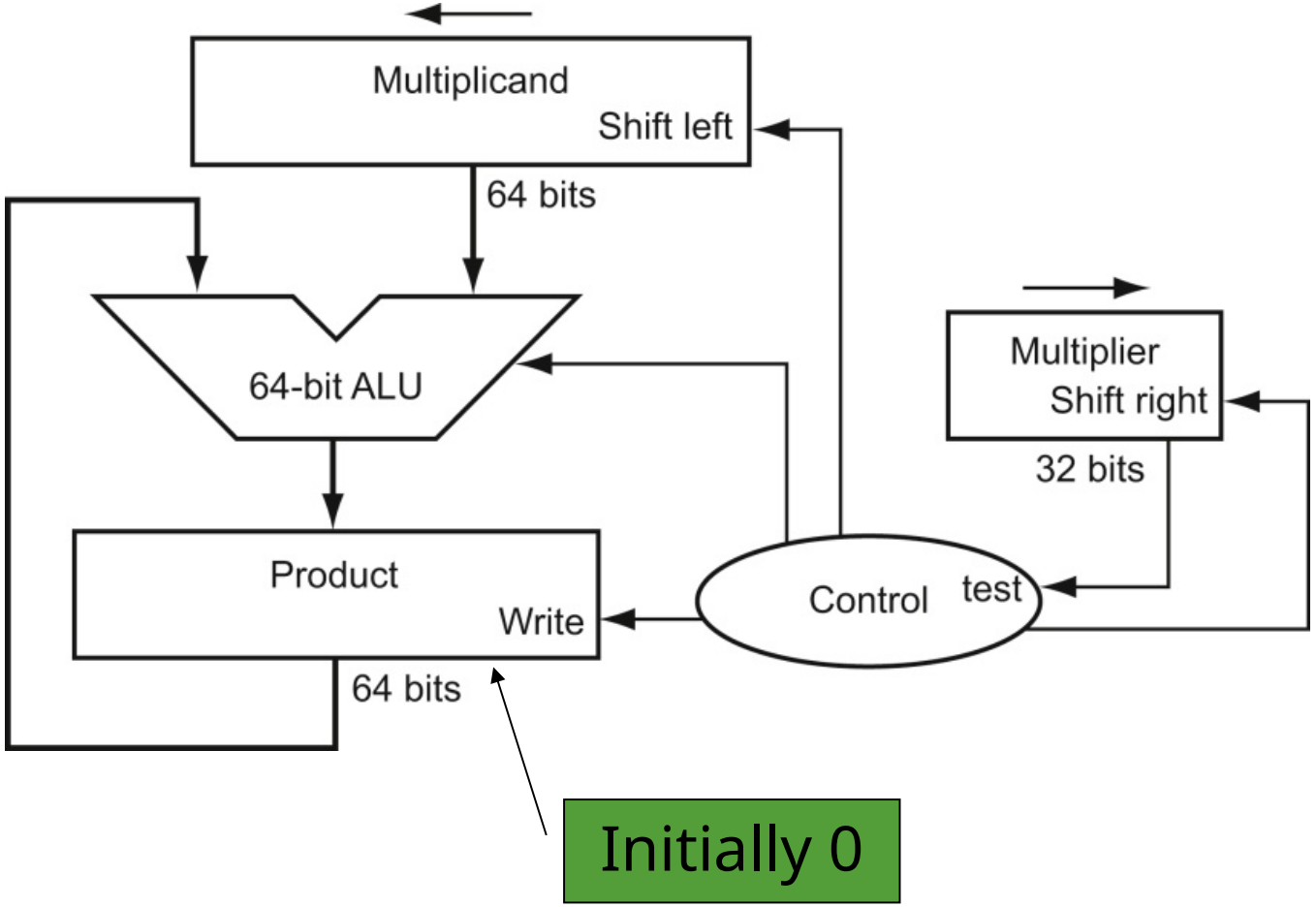
product

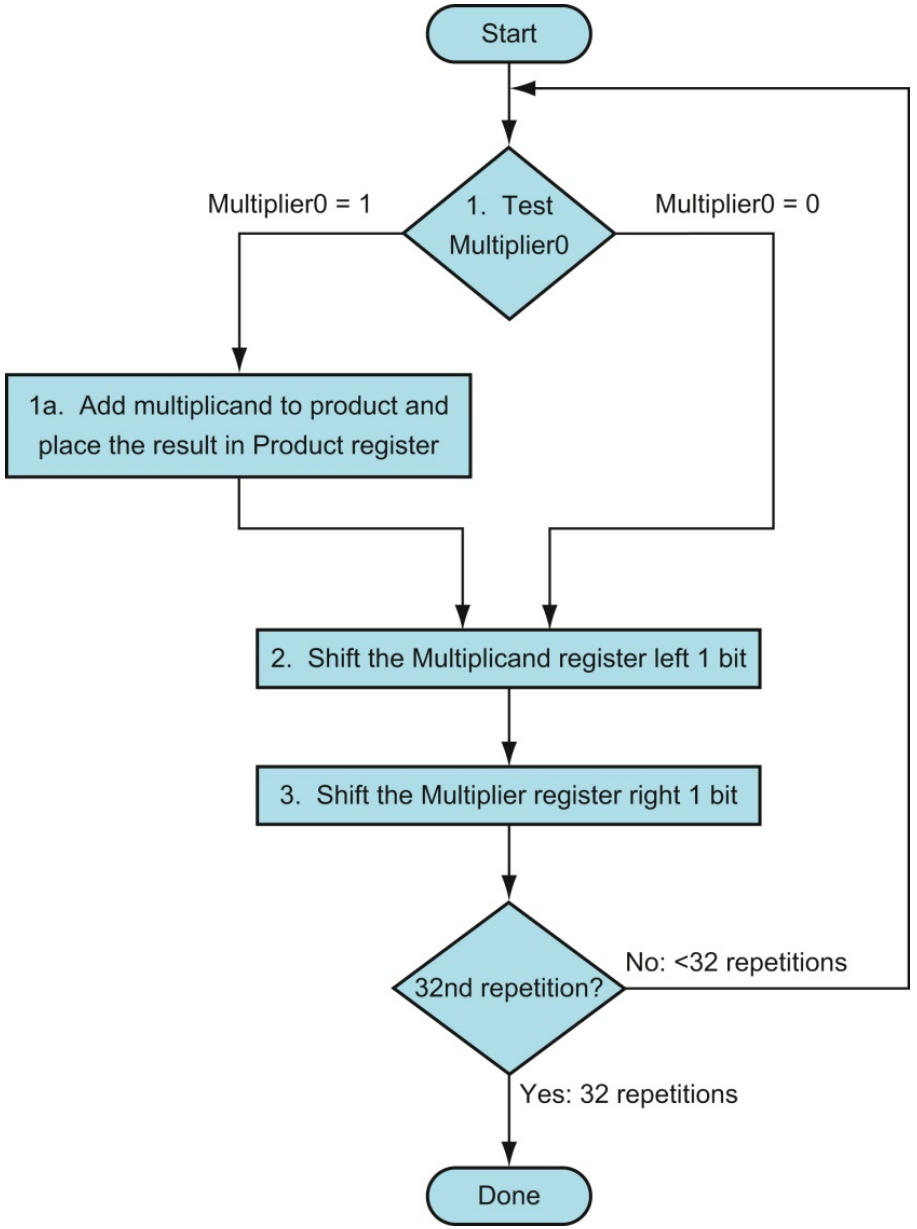
$$
\begin{array}{r}
1000_{ten} \\
\times\ 1001_{ten} \\
\hline
1000 \\
0000 \\
0000 \\
1000 \\
\hline
1001000_{ten}
\end{array}
$$

Length of product is the sum of operand lengths



Multiplication Hardware





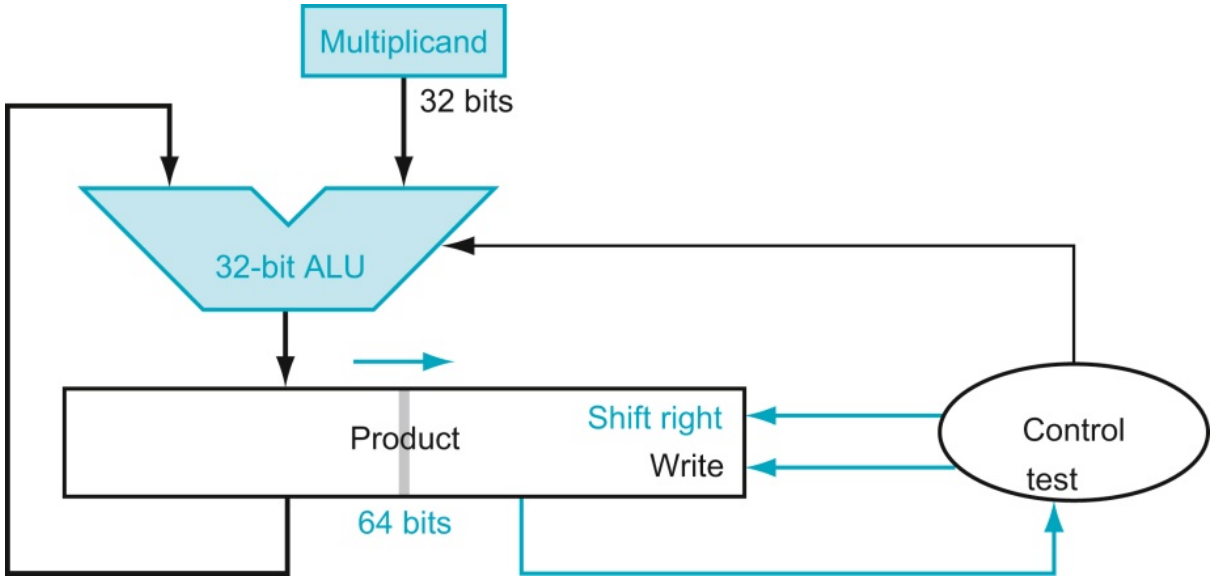
Multiplication Example

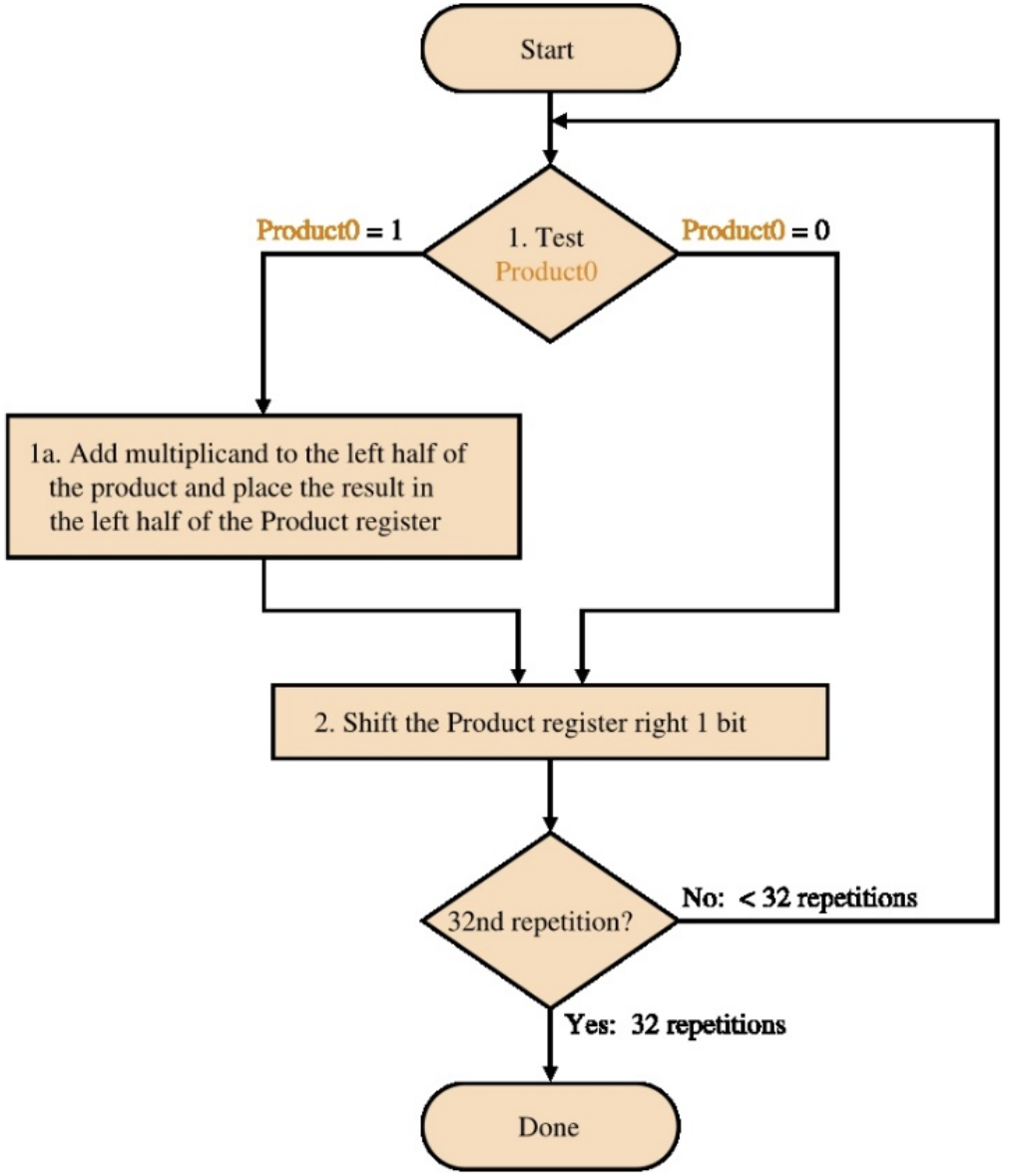
- Using 4-bit numbers to save space, multiply $\mathbf{2_{ten} \times 3_{ten}}$, or $\mathbf{0010_{two} \times 0011_{two}}$.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Optimized Multiplier

- Perform steps in parallel: add/shift





- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

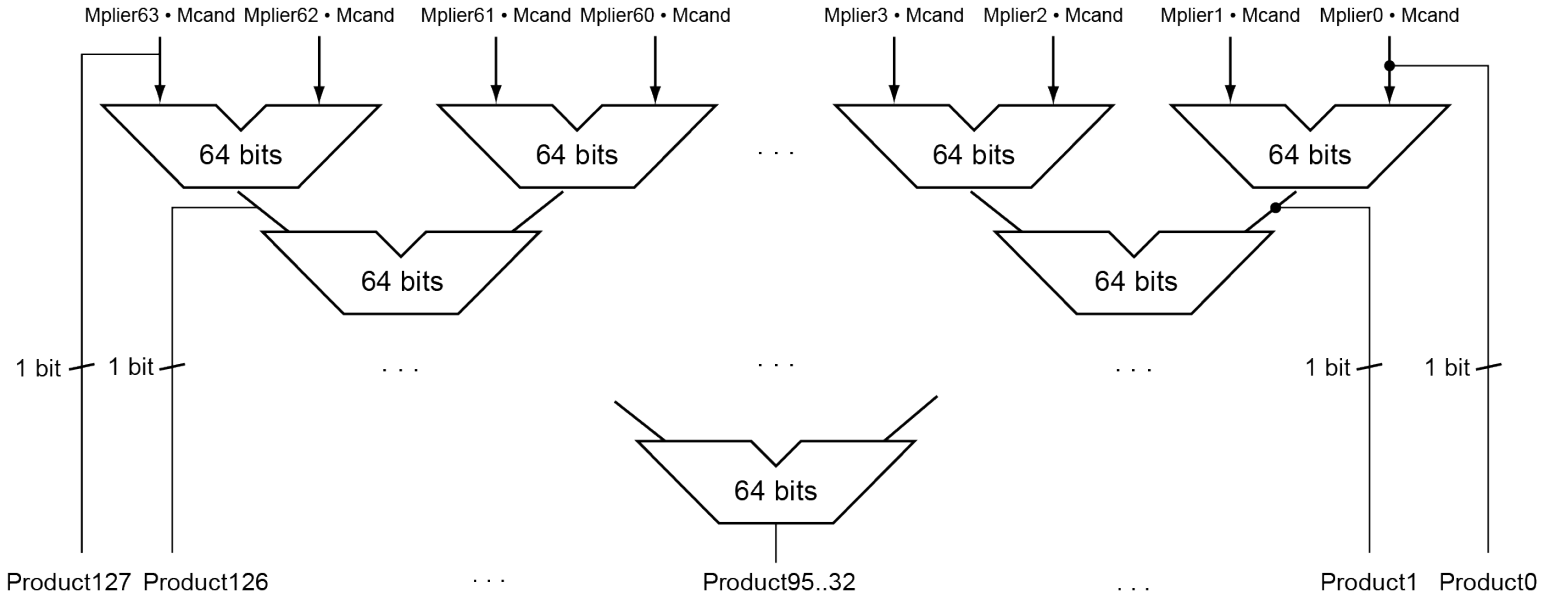
Multiplication Example

- Using 4-bit numbers to save space, multiply $\mathbf{2_{ten} \times 3_{ten}}$, or $\mathbf{0010_{two} \times 0011_{two}}$.

Steps	Action/Operation	Multiplicand	Product/Multiplier
0	initial	0010	0000 001**1**
1	Prod0 = 1 → Prod=Prod+Mcand	0010	0010 0011
	R-shift Product/Multiplier	0010	0001 000**1**
2	Prod0 = 1 → Prod=Prod+Mcand	0010	0011 0001
	R-shift Product/Multiplier	0010	0001 100**0**
3	Prod0 = 0 → NO OP	0010	0001 1000
	R-shift Product/Multiplier	0010	0000 110**0**
4	Prod0 = 0 → NO OP	0010	0000 1100
	R-shift Product/Multiplier	0010	0000 0110

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



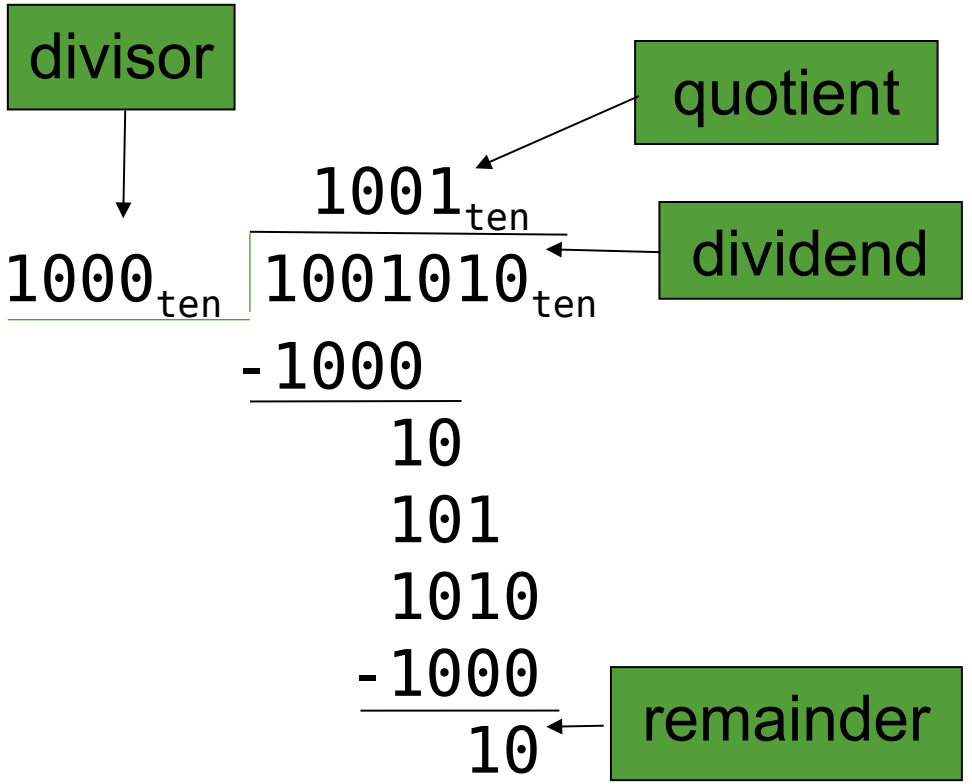
- Can be pipelined
  - Several multiplication performed in parallel

RISC-V Multiplication

- Four multiply instructions:
  - **`mul`**: multiply
    - Gives the lower 64 bits of the product
  - **`mulh`**: multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - **`mulhu`**: multiply high unsigned
    - Gives the upper 64 bits of the product, assuming the operands are unsigned
  - **`mulhsu`**: multiply high signed/unsigned
    - Gives the upper 64 bits of the product, assuming one operand is signed and the other unsigned
- Use **`mulh`** result to check for 64-bit overflow

Division

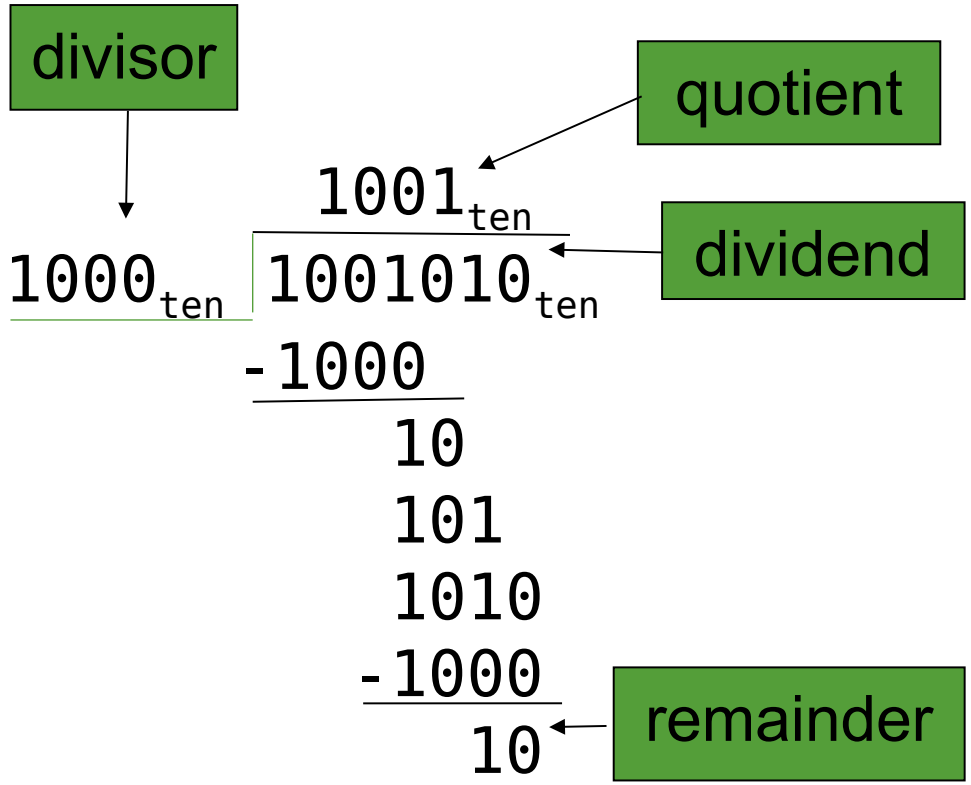
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using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



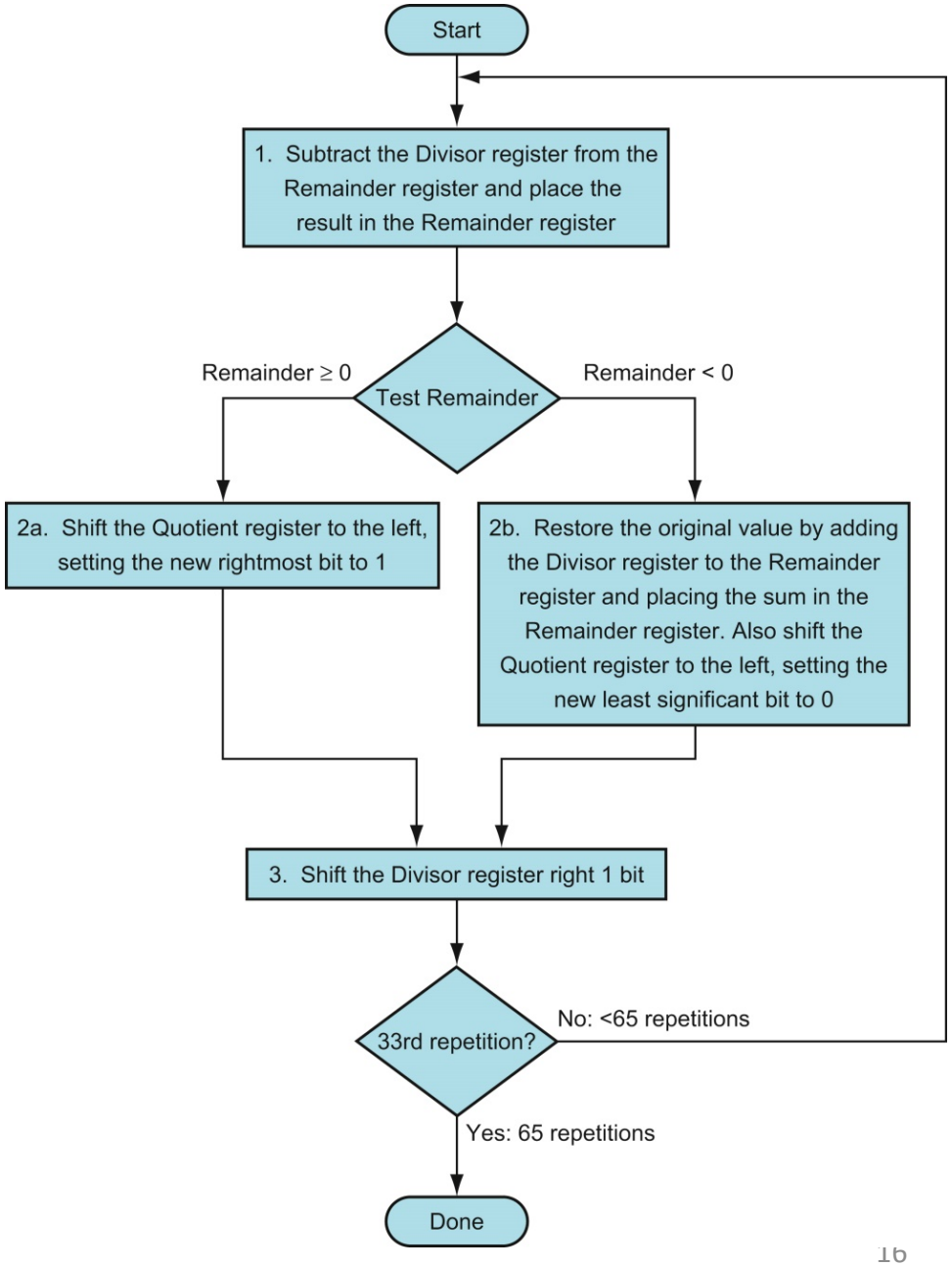
n-bit operands yield *n*-bit quotient and remainder

Dividend = Quotient × Divisor + Remainder

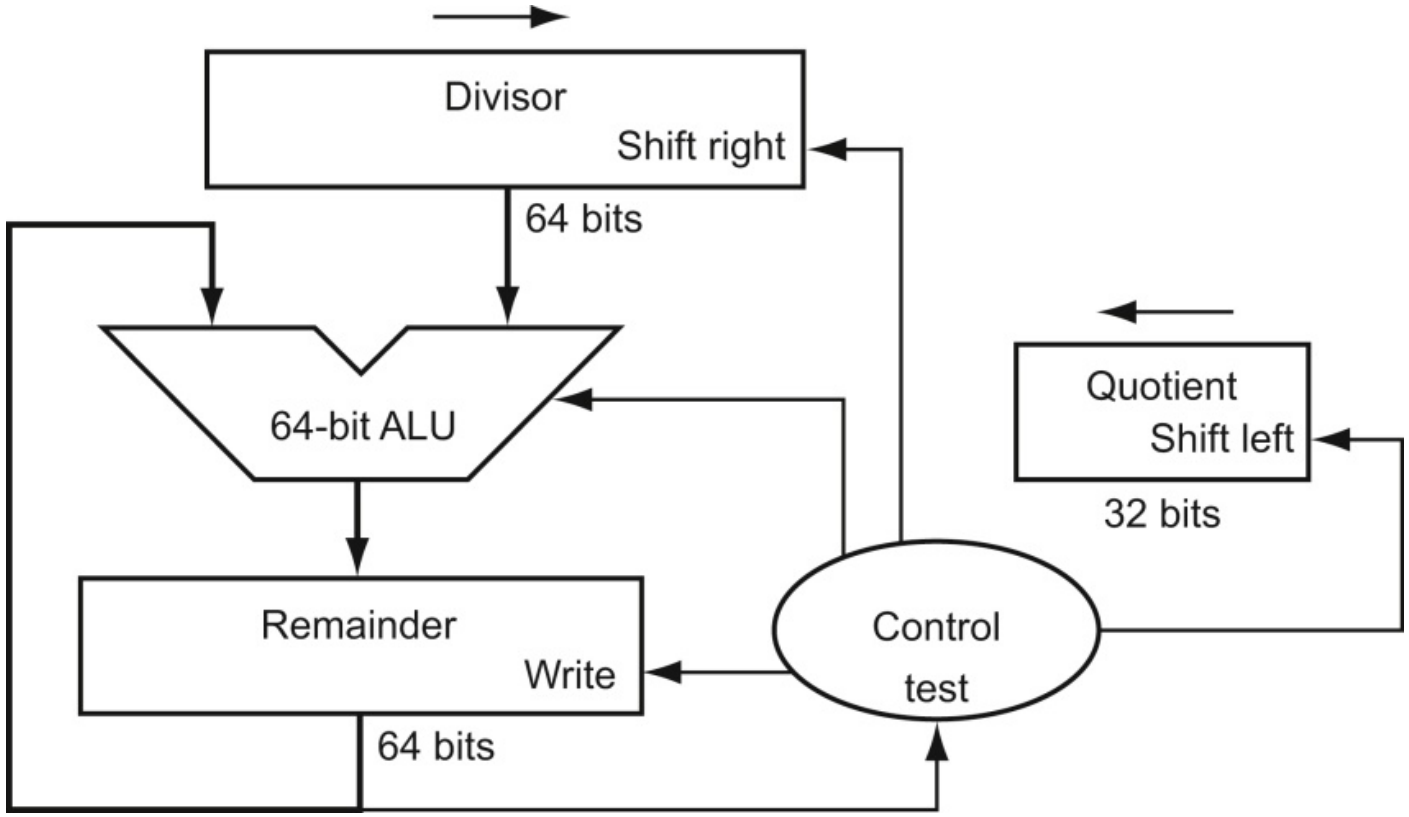
Division Algorithm

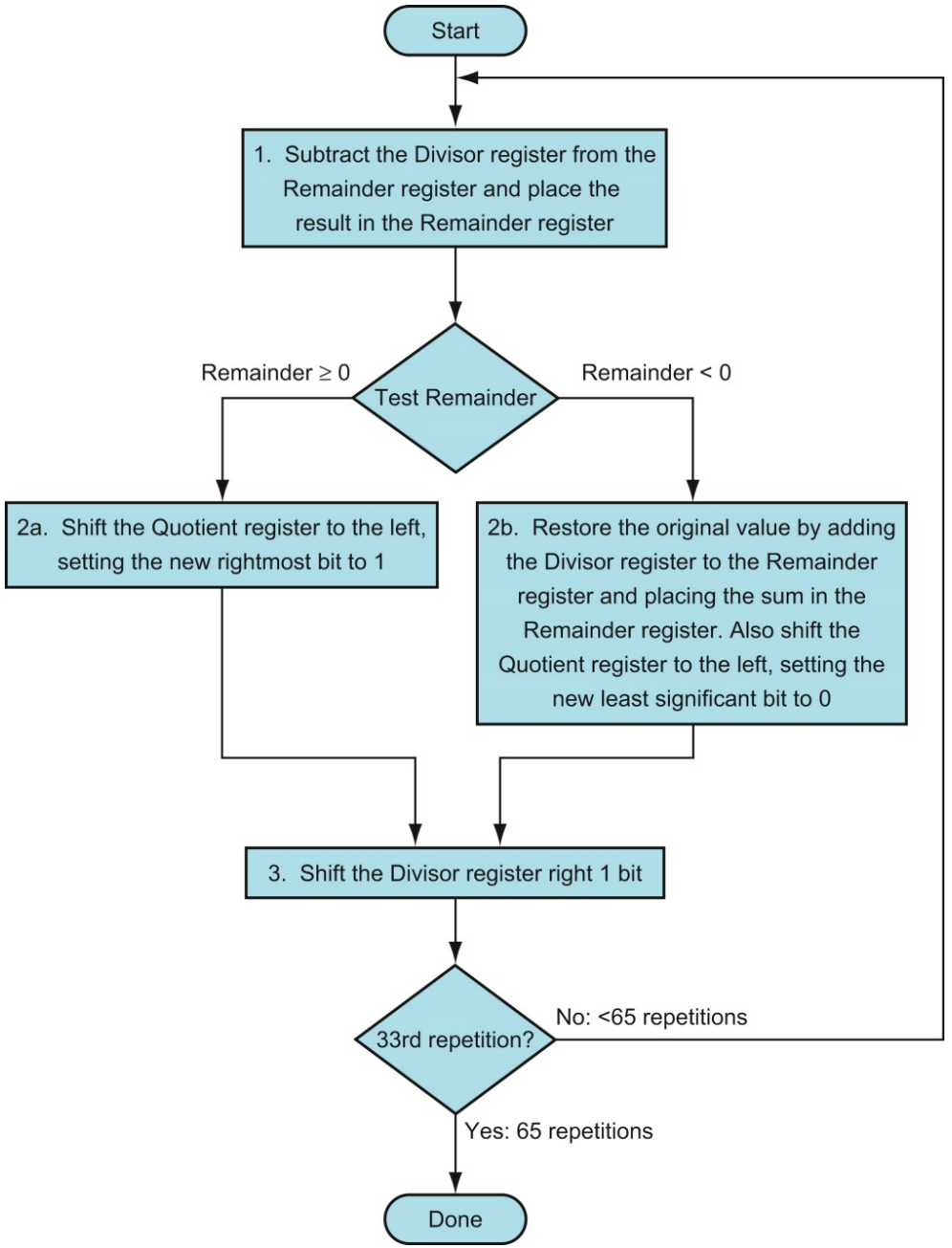


n-bit operands yield *n*-bit quotient and remainder



Division Hardware





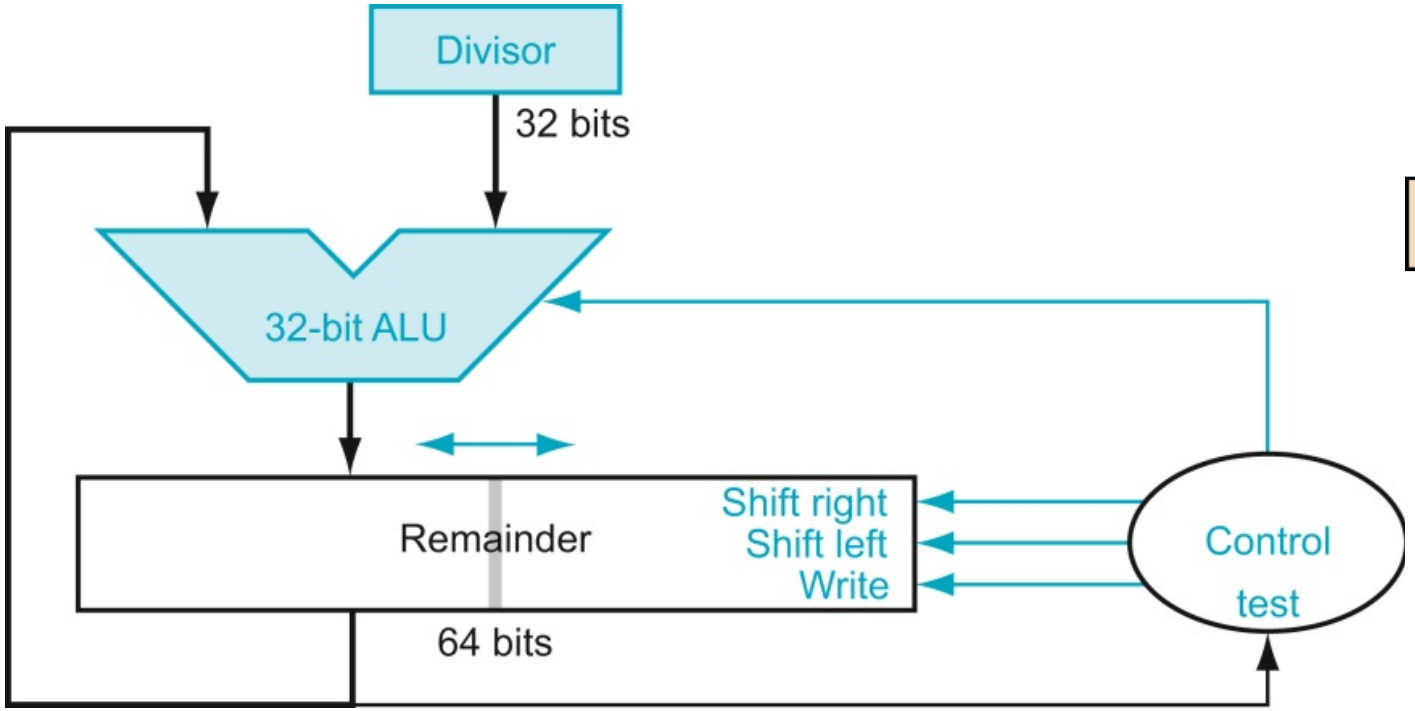
Division Example

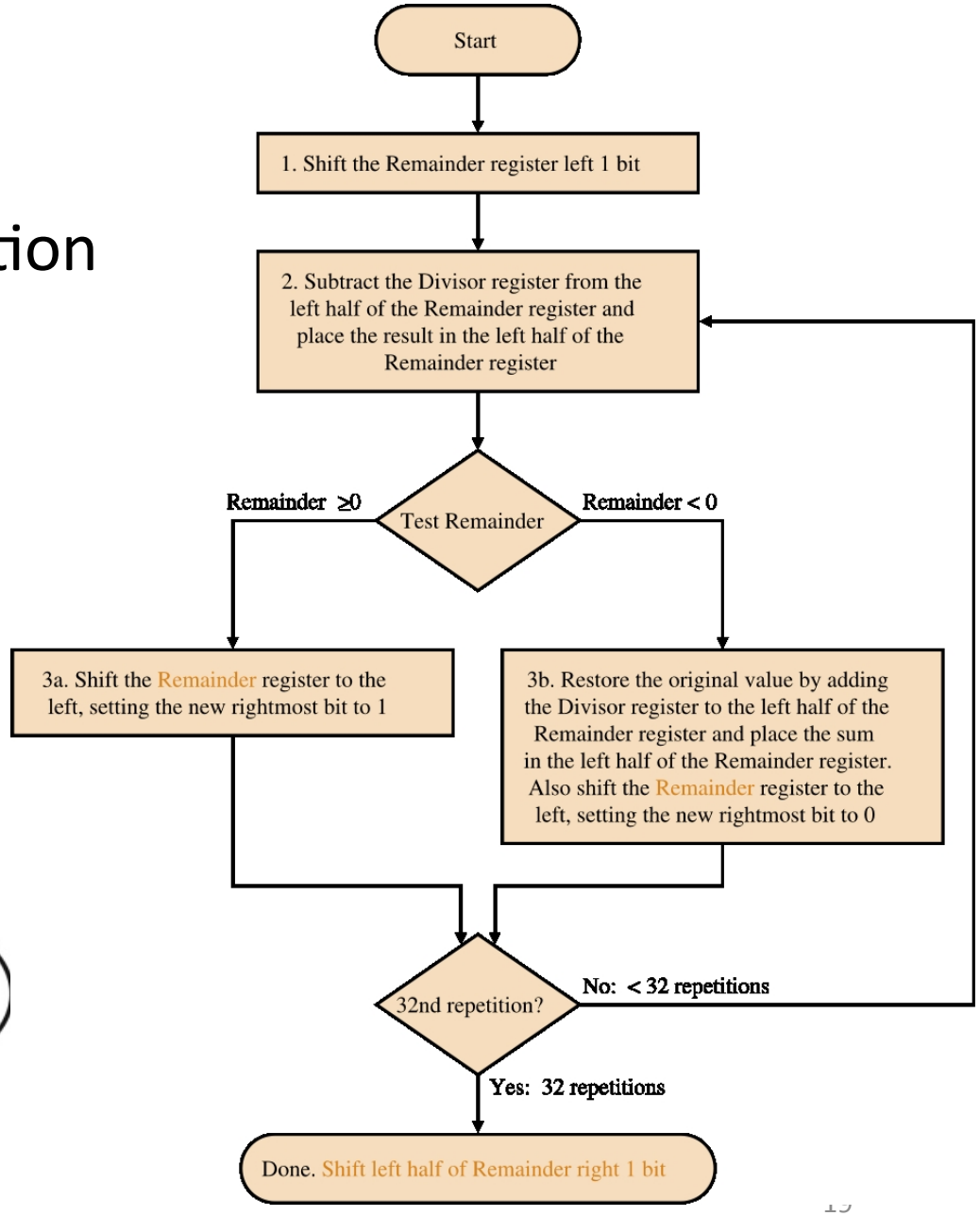
- Using a 4-bit version of the algorithm to save pages, let's try dividing $\mathbf{7}_{ten}$ by $\mathbf{2}_{ten}$, or $\mathbf{0000\ 0111}_{two}$ by $\mathbf{0010}_{two}$.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = 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 = 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 = 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 = 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 = Rem – Div	0001	0000 0010	⓪000 0001
	2a: Rem ≥ 0 ⟹ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Optimized Divider

- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both





Division Example

- Using a 4-bit version of the algorithm to save pages, let's try dividing $\mathbf{7}_{ten}$ by $\mathbf{2}_{ten}$, or $\mathbf{0000\ 0111}_{two}$ by $\mathbf{0010}_{two}$.

Steps	Action/Operation	Divisor	Dividend (Rem. Reg.)
0	initial	0010	0000 0111
	L-shift Rem	0010	0000 1110
1	LRem=LRem-Div	0010	**1**110 1110
	Rem=--ve → LRem=LRem+Div	0010	0000 1110
	L-shift Rem + Dnd0=0	0010	0001 110<u>0</u>
2	LRem = LRem-Div	0010	**1**111 1100
	Rem=--ve → LRem=LRem+Div	0010	0001 1100
	L-shift + Dnd0=0	0010	0011 100<u>0</u>
3	LRem = LRem-Div	0010	**0**001 1000
	L-shift + Dnd0=1	0010	0011 000<u>1</u>
4	LRem = LRem-Div	0010	**0**001 0001
	L-shift + Dnd0=1	0010	0010 001<u>1</u>
5	R-shift Rem	0010	0001 0011

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  - Still require multiple steps

RISC-V Division

- Four instructions:
  - **`div, rem`**: signed divide, remainder
  - **`divu, remu`**: unsigned divide, remainder

- Overflow and division-by-zero don't produce errors
  - Just return defined results
  - Faster for the common case of no error