Computer Organization & Architecture

Moazzam Ali Sahi

Agenda

- Unsigned Arithmetic Continue…
  - Hardware for Addition & Subtraction
  - Multiplication
  - Division

Arithmetic for Computers (Review)

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Integer Addition

		(0)		(0)		(1)		(1)		(0)		(Carries)
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

- 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

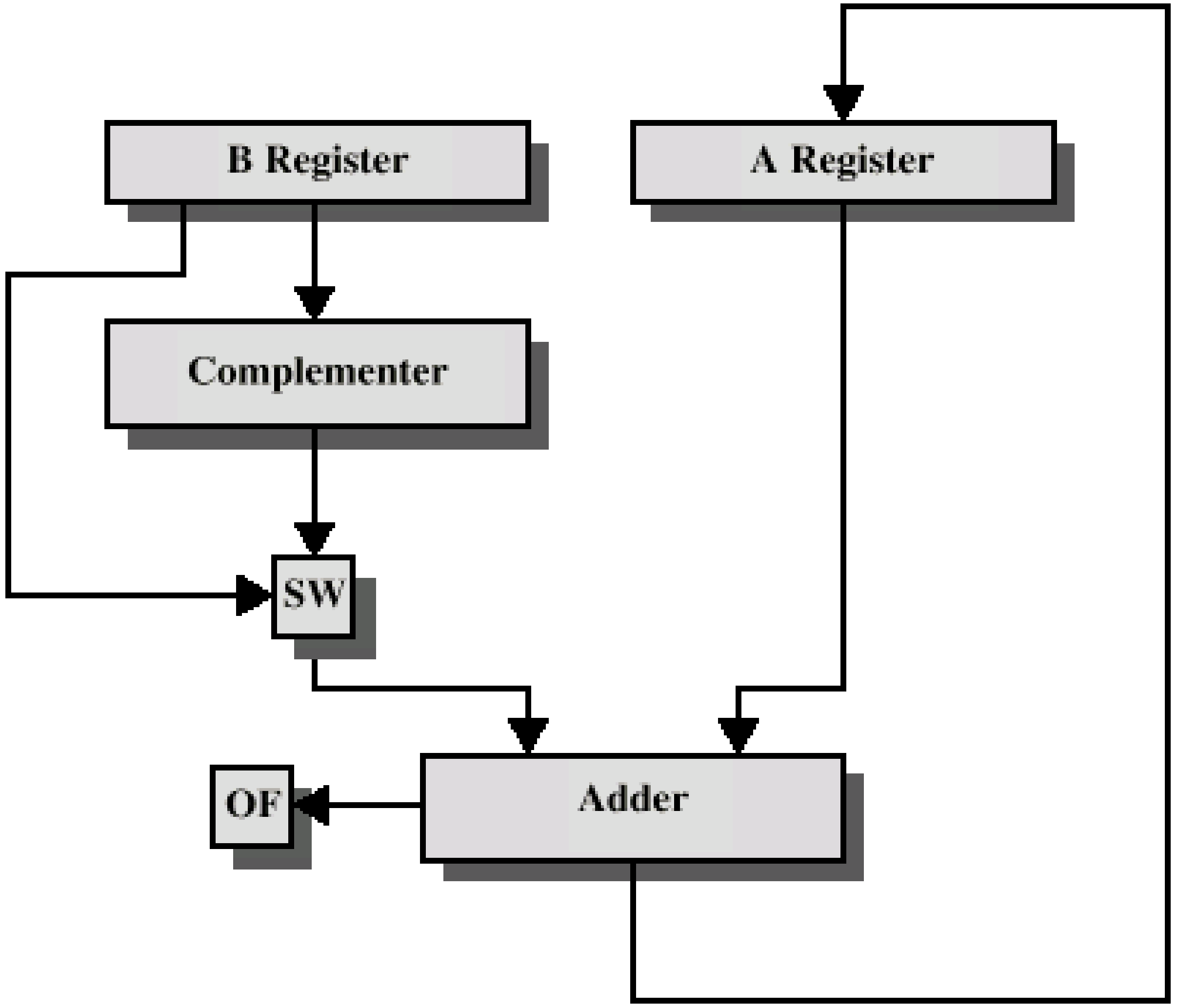
Overflow

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.2 **Overflow conditions for addition and subtraction.**

- Unsigned integers are commonly used for memory addresses where overflows are ignored.

Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

Multiplication

- How about this algorithm:

```
result = 0;
While first number > 0 {
    add second number to result;
    decrement first number;
}
```

- Does it work? What is the complexity?
- Can you think of a better approach?
- Lets do an example 1001 x 100
  - What is this in decimal?

Multiplication – longhand algorithm

- Just like you learned in school
- For each digit, work out partial product (easy for binary!)
- Take care with place value (column)
- Add partial products
- How to do it efficiently?

Example of shift and add multiplication

How many steps?

How do we implement this in hardware?

				1	0	1	1
			x	1	1	0	1
				1	0	1	1
			0	0	0	0	
			0	1	0	1	1
		1	0	1	1		
		1	1	0	1	1	1
	1	0	1	1			
1	0	0	0	1	1	1	1

Multiplication

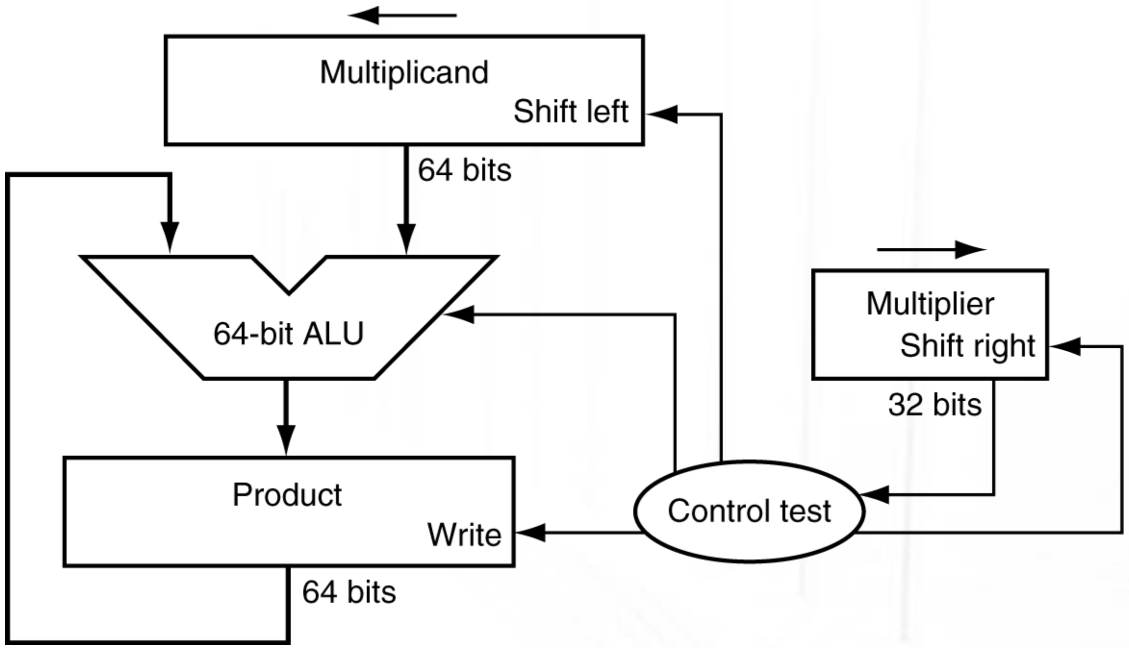
multiplicand

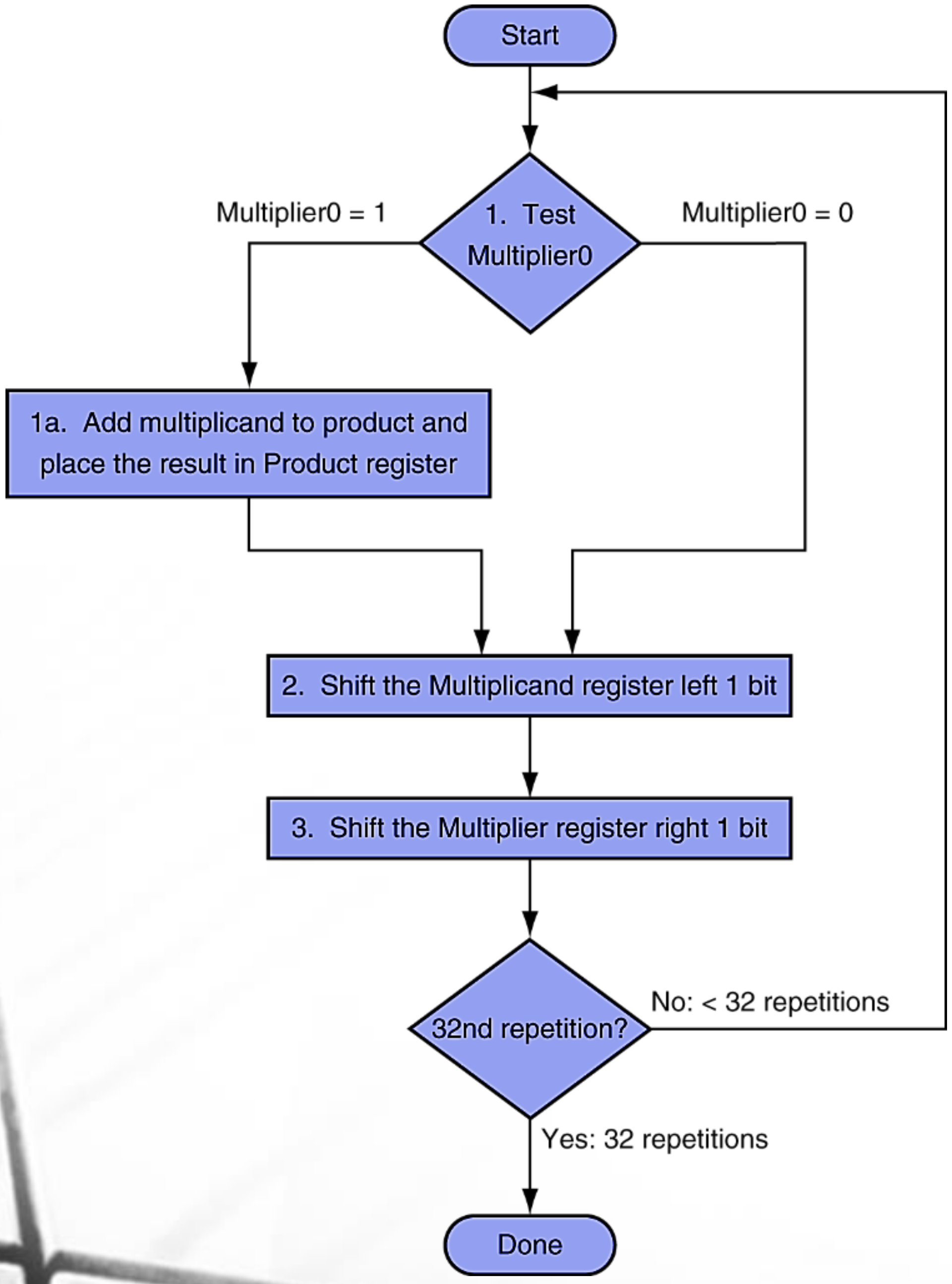
multiplier

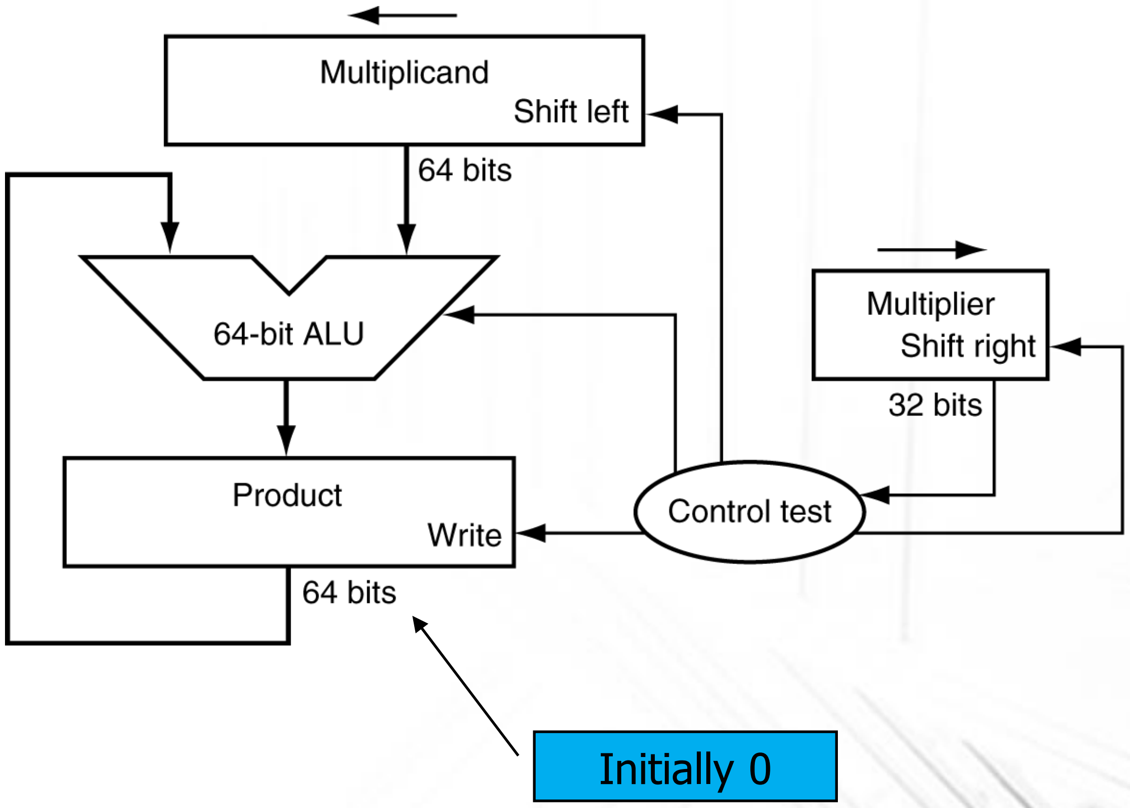
product

```
     1000
×    1001
---------
     1000
    0000
   0000
  1000
---------
  1001000
```

Length of product is the sum of operand lengths




Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done


Multiplicand
Shift left
64 bits
64-bit ALU
Product
Write
64 bits
Multiplier
Shift right
32 bits
Control test
Initially 0

Multiply Example

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

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000

Multiply Example

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

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010

Multiply Example

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

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

Multiply Example

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

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

Multiply Example

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

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

Multiply Example

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

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

Multiply Example

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

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

Multiply Example

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

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

Multiply Example

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

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

Multiply Example

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

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

Multiply Example

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

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

Multiply Example

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

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

Multiply Example

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

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

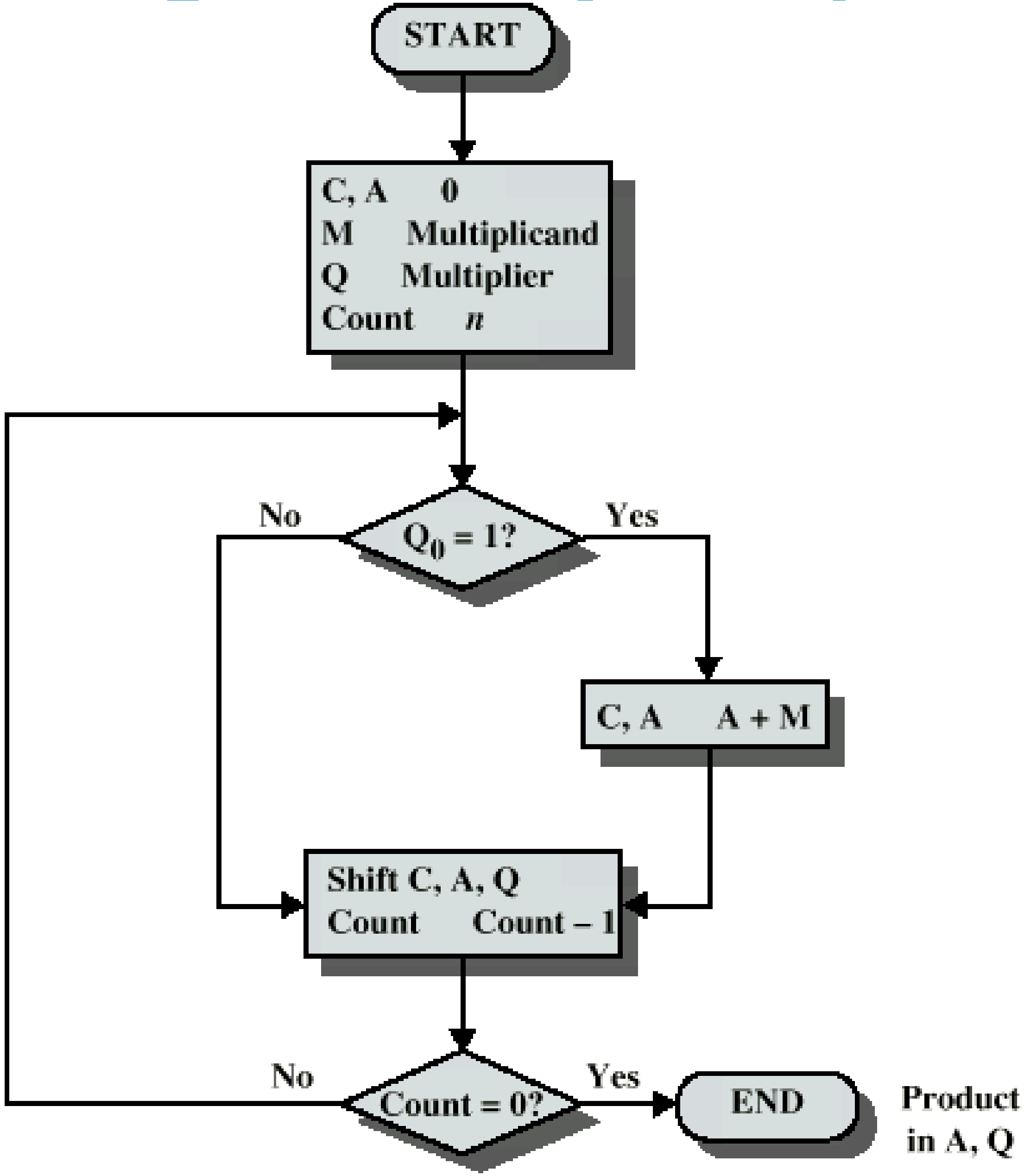
Unsigned Multiplication (12 x 9)

Iteration	Result	Multiplier (Q)	Multiplicand (A)	Operation
0	0000 0000	1100	0000 1001	Initialization
1	0000 0000 0000 0000	1100 0110	0001 0010 0001 0010	Shift left B Shift right Q
2	0000 0000 0000 0000	0110 0011	0010 0100 0010 0100	Shift left B Shift right Q
3	0010 0100 0010 0100 0010 0100	0011 0011 0001	0010 0100 0100 1000 0100 1000	Add B to A Shift left B Shift right Q
4	0110 1100 0110 1100 0110 1100	0001 0001 0000	0100 1000 1001 0000 1001 0000	Add B to A Shift left B Shift right Q

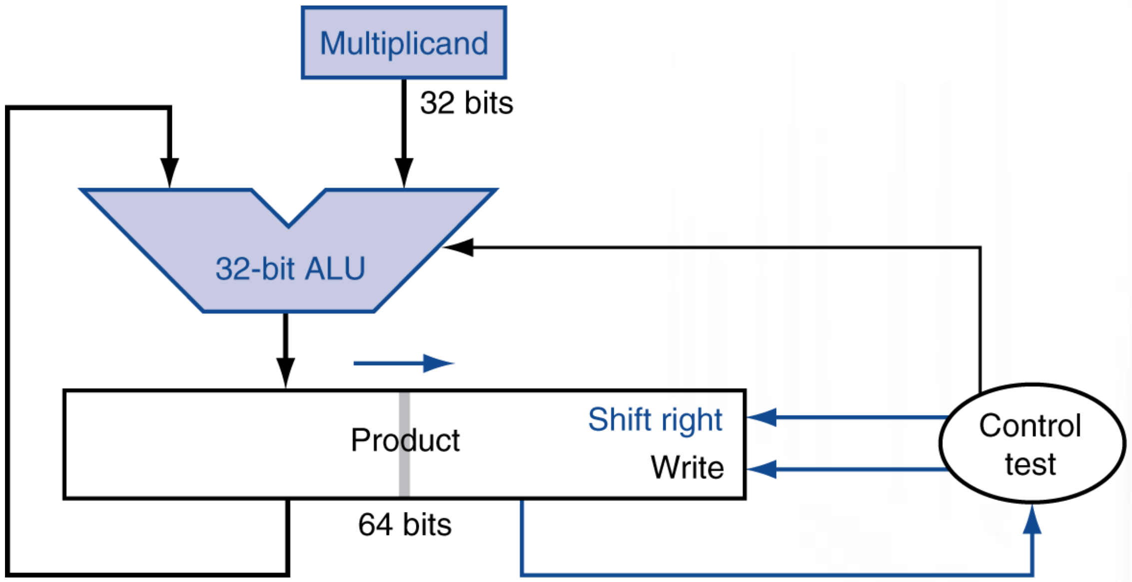
Unsigned Multiplication (12 x 9)

Iteration	Result	Multiplier (Q)	Multiplicand (A)	Operation

Flowchart for Unsigned Binary Multiplication

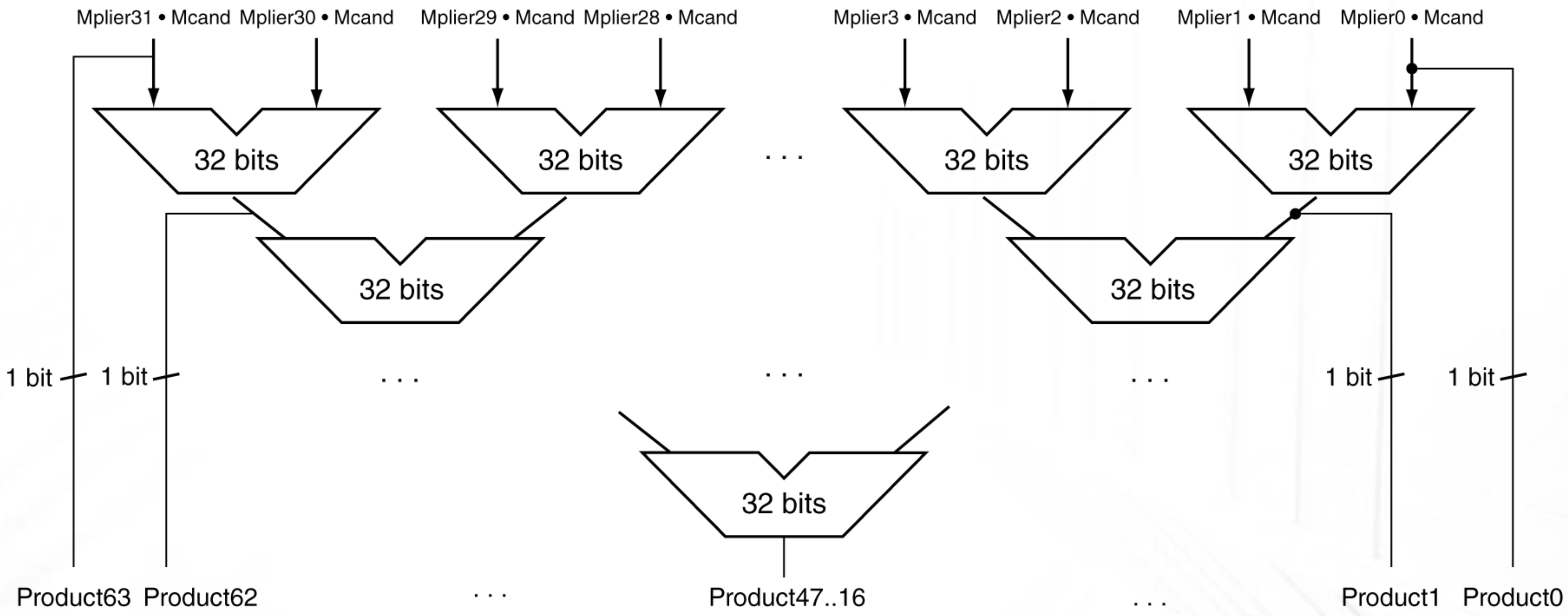


Optimized Multiplier



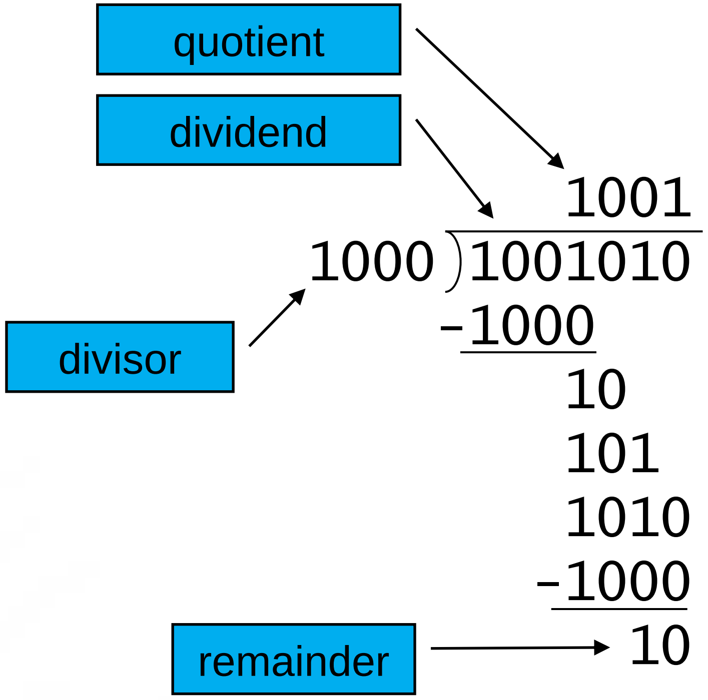
- One cycle per partial-product addition
  - That’s ok, if frequency of multiplications is low

Faster Multiplier



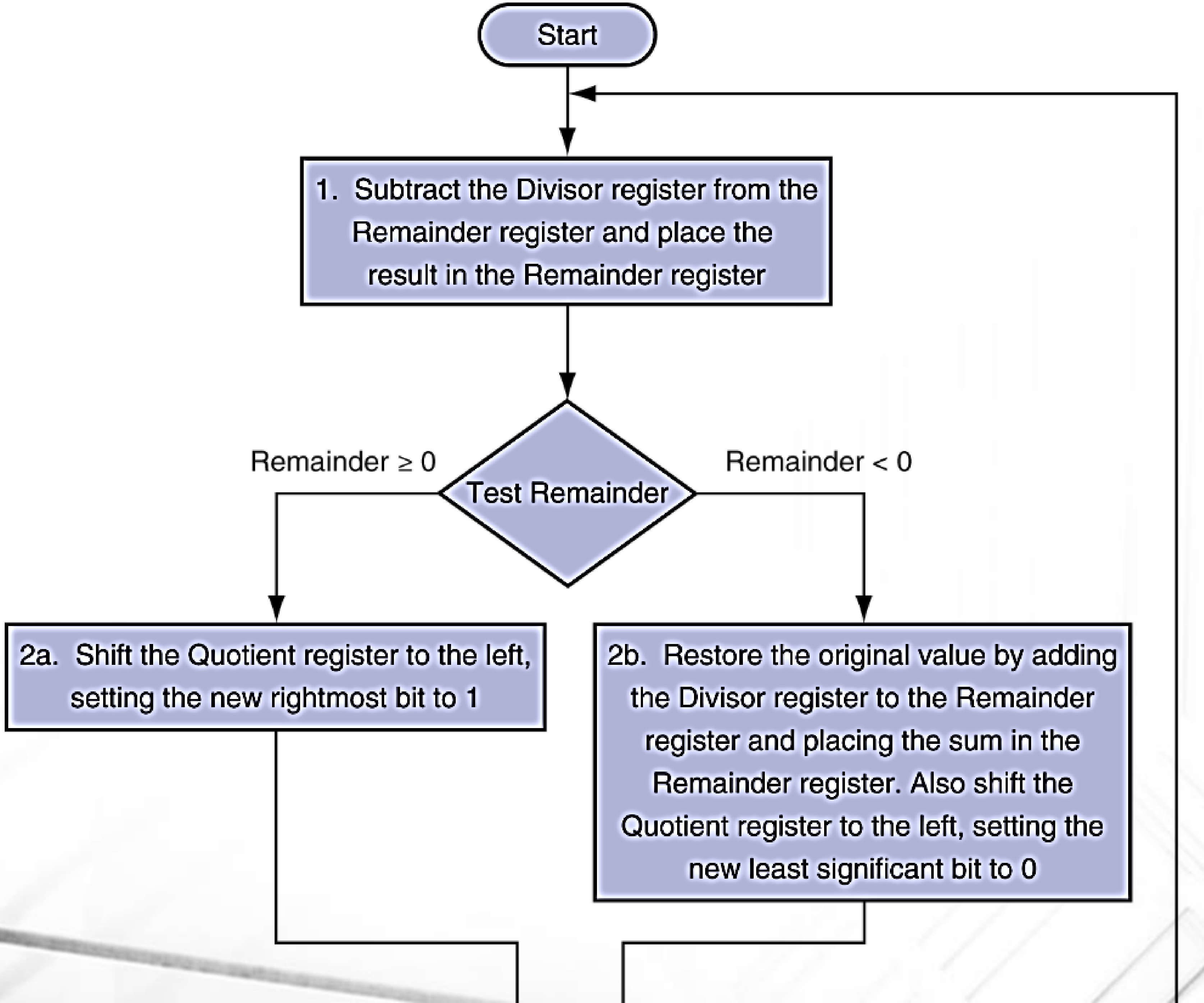
- Can be pipelined
  - Several multiplication performed in parallel

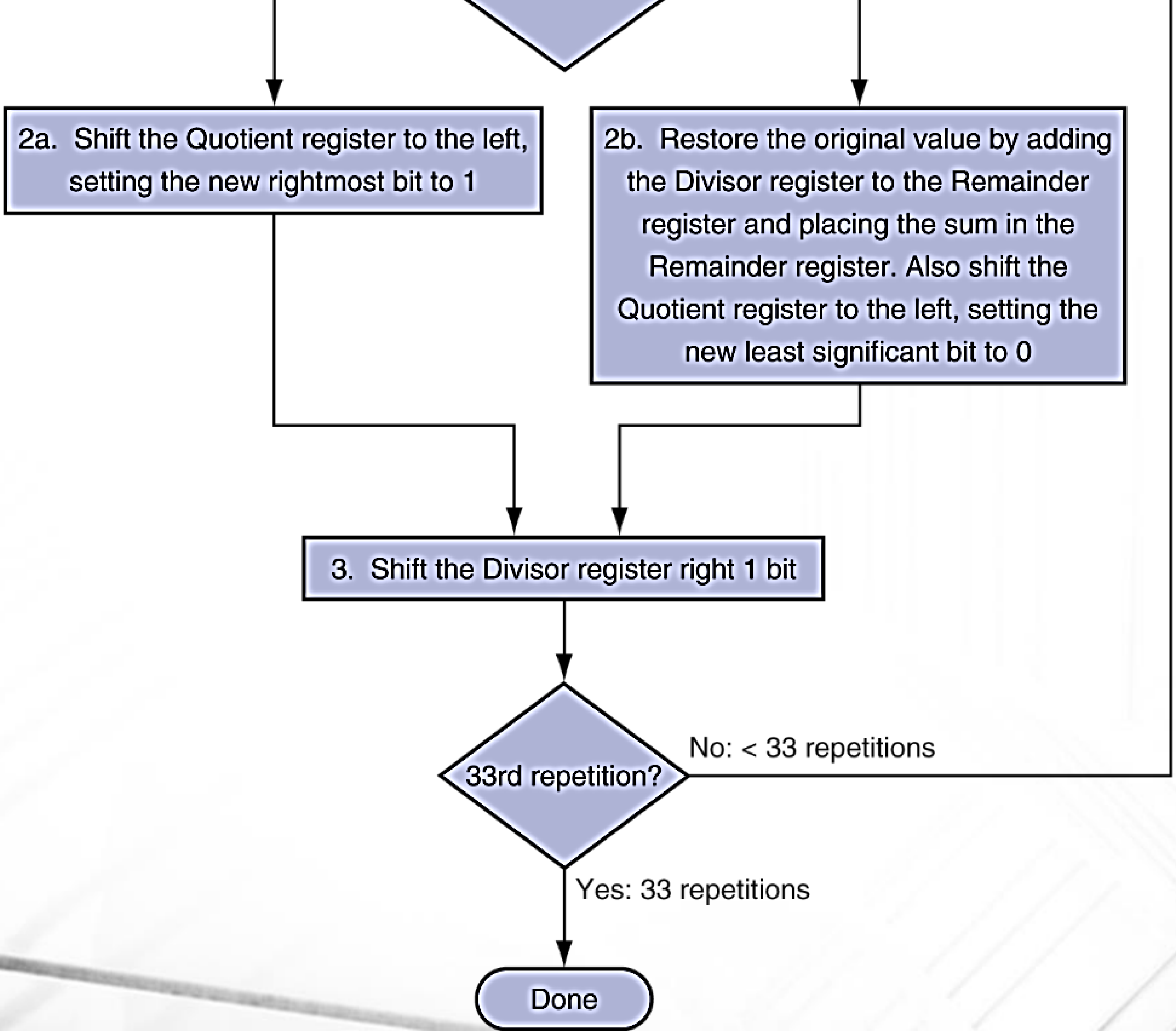
Division

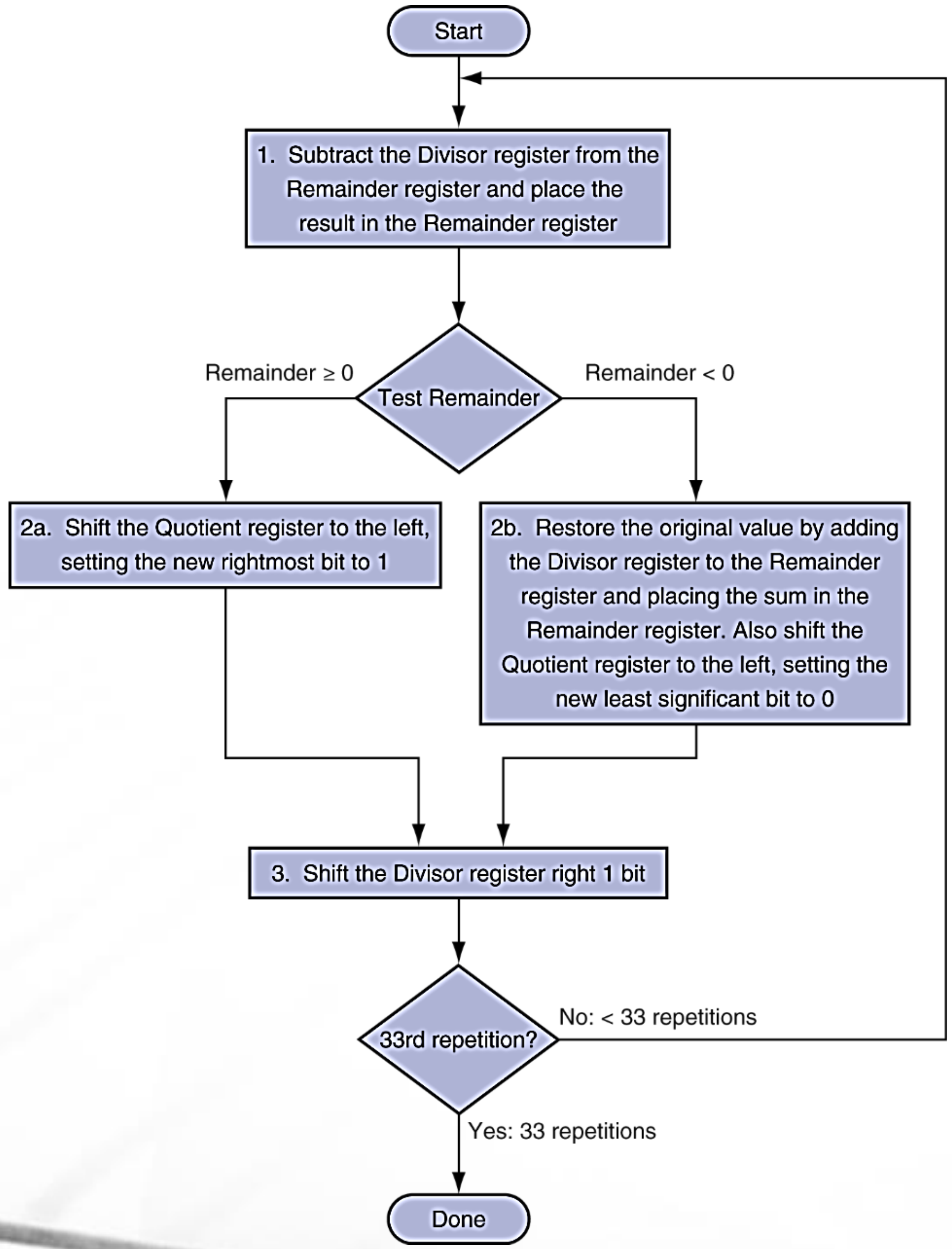


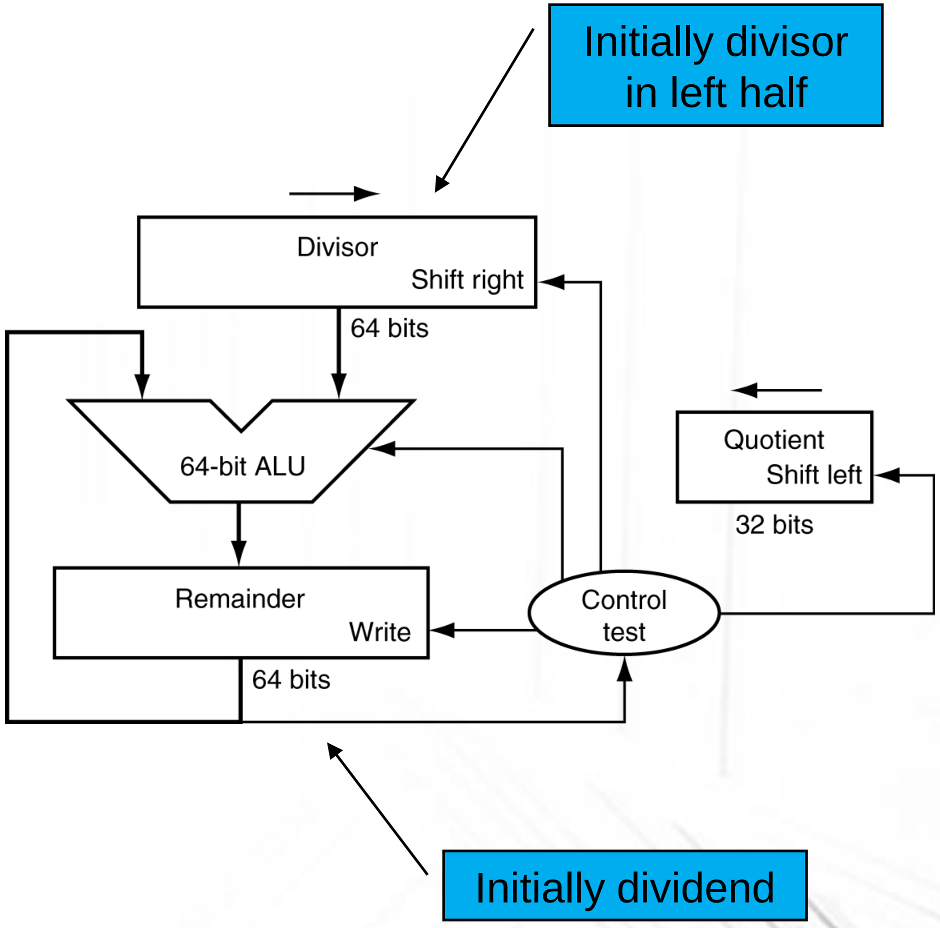
n-bit operands yield *n*-bit quotient and remainder

- 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


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0


2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done


Initially divisor in left half
Divisor
Shift right
64 bits
64-bit ALU
Quotient
Shift left
32 bits
Remainder
Write
64 bits
Control test
Initially dividend

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
	Rem = Rem – Div	0000	0010 0000	1110 0111

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q	0000 0000	0010 0000 0010 0000	1110 0111 0000 0111

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
	Rem = Rem – Div	0000	0001 0000	1111 0111

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q	0000 0000	0001 0000 0001 0000	1111 0111 0000 0111

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
	Rem = Rem – Div	0000	0000 0100	0000 0011

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div Rem >= 0 →, shift 1 into Q	0000 0001	0000 0100 0000 0100	0000 0011 0000 0011

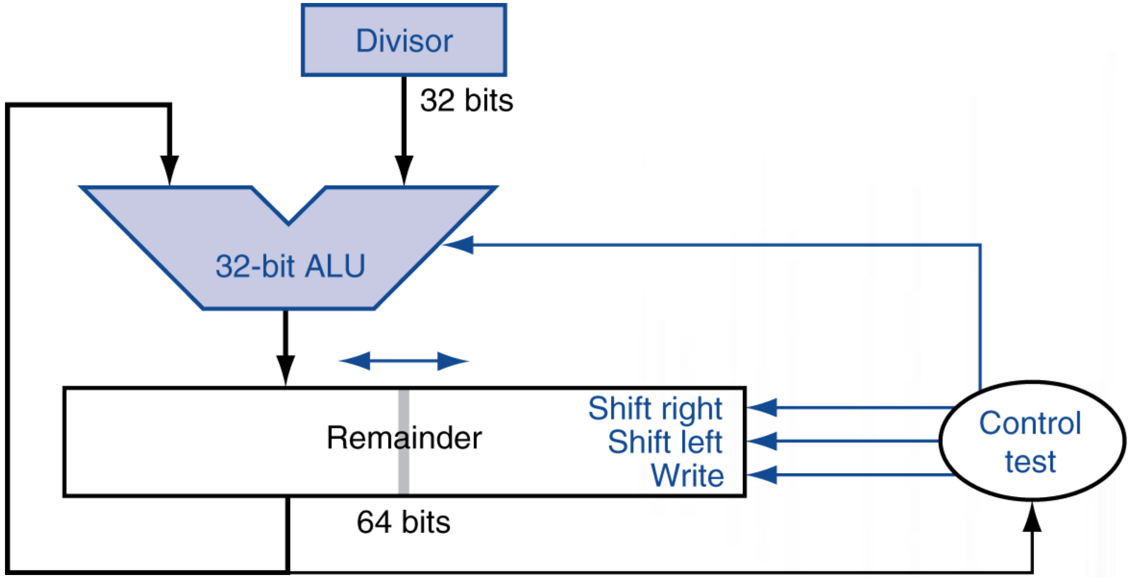
Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div Rem >= 0 →, shift 1 into Q Shift Div right	0000 0001 0001	0000 0100 0000 0100 0000 0010	0000 0011 0000 0011 0000 0011

Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Value	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div Rem >= 0 →, shift 1 into Q Shift Div right	0000 0001 0001	0000 0100 0000 0100 0000 0010	0000 0011 0000 0011 0000 0011
5	Same steps as 4	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

Aside – cost of these operations

- We'd like to be able to finish these operations quickly
  - Usually in one cycle!
- How do we implement add?
  - Remember the 1 bit full adder?
- How many adds do we need for a multiply?
- Specialized logic circuits are used to implement these functionalities quickly (e.g., carry look-ahead adders, loop unrolled multiplication)