Computer Organization and Design

Chapter 3 Part 2

Arithmetic for Computers

(Ch. 3 – Sections 3.3 – 3.4)

Integer Multiplication & Division

Integer Multiplication

- Multiplication is more complicated than addition and adds a level of complexity to the design of the ALU
- Multiplication also takes longer increasing the processing time and requires more space in the ALU
- Performance of multiplication (and division) operations is important
- We need to derive a fast method for multiplication, one that also minimizes the need for space in the ALU

Example Binary Multiplication

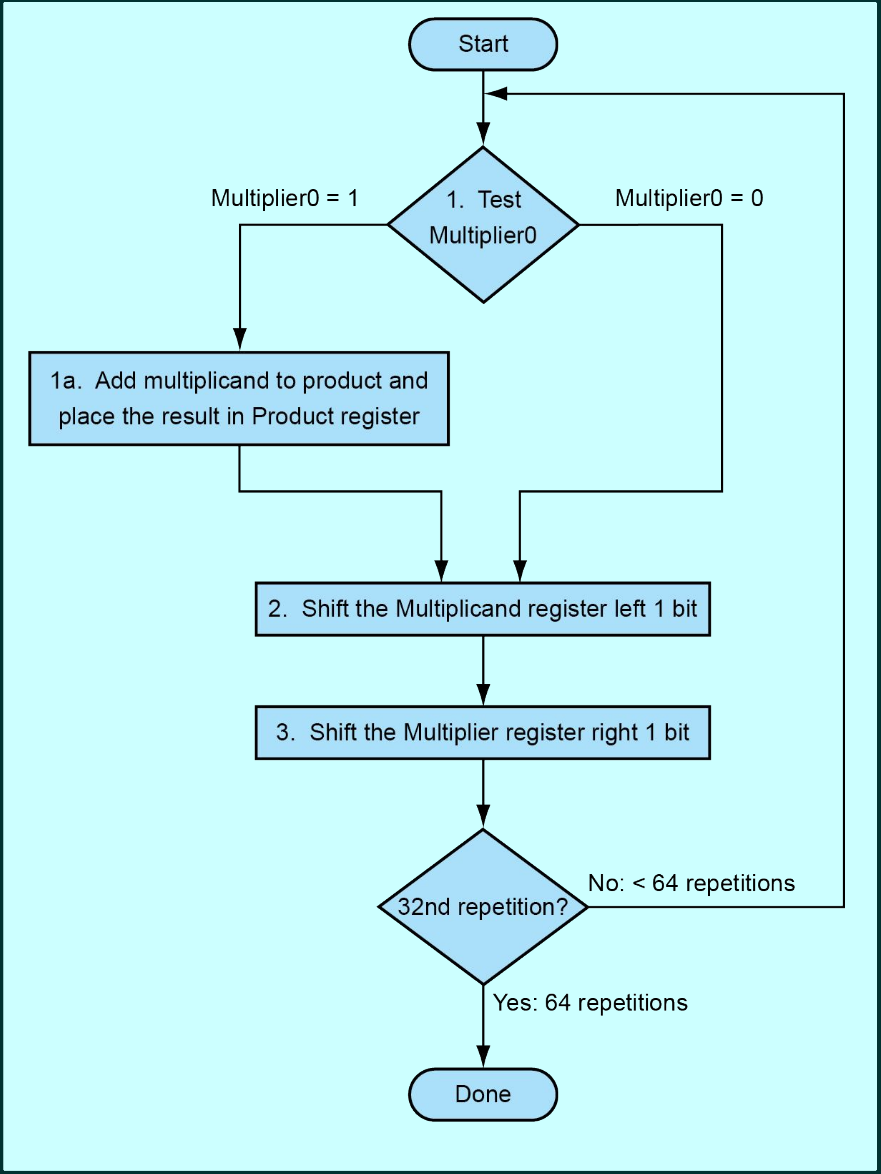
(Two 4-bit unsigned numbers)

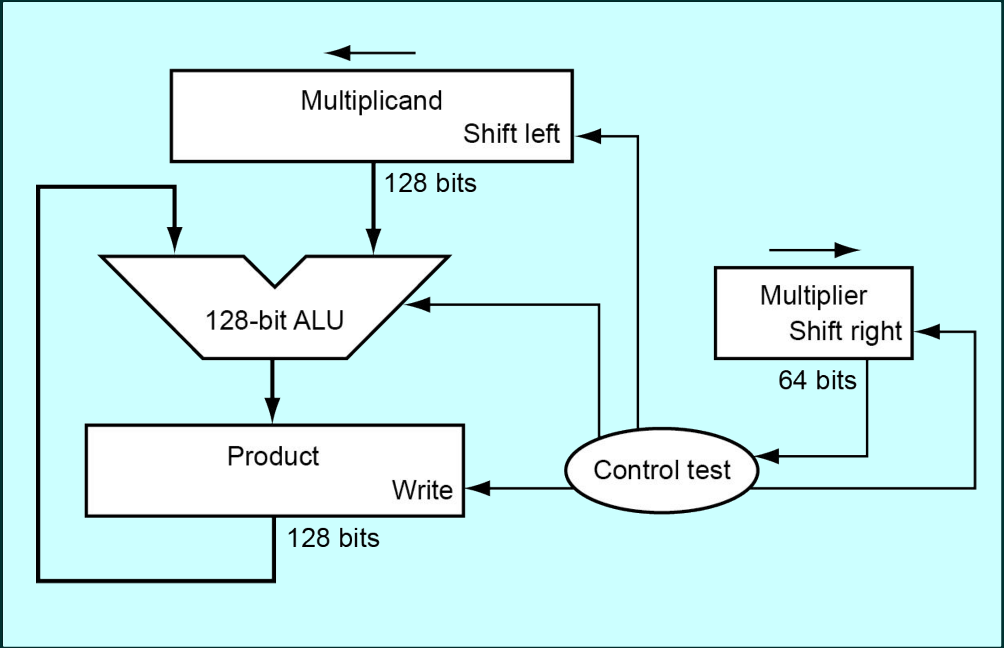
$$
\begin{array}{l rrrrrrrr l}
\text{Multiplicand} & & & & & 1 & 1 & 1 & 1 & (15) \\
\text{Multiplier} & & & & & 1 & 1 & 1 & 1 & (15) \\
\cline{6-9}
 & & & & & 1 & 1 & 1 & 1 & \\
 & & & & 1 & 1 & 1 & 1 & & \Bigg\} \ \text{Partial products} \\
 & & & 1 & 1 & 1 & 1 & & & \\
 & & 1 & 1 & 1 & 1 & & & & \\
\cline{2-9}
\text{Product} & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & (225)
\end{array}
$$

Observations About Integer Multiplication

- m bits x n bits = maximum m + n bit product
- In RISC-V, multiplying two 64-bit numbers yields a 128 bit product
  - Only 64-bit products are valid
  - Multiplication must consider overflow
- Multiplication takes more space in hardware
- More time is required to obtain the product
- As in any computer activity, an algorithm defines the step by step process for the operation

Multiplication Algorithm (First Version)


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: < 64 repetitions
Yes: 64 repetitions
Done



Multiplicand shifts left and multiplier shifts right each iteration.

Add operation occurs only when least significant multiplier bit = 1.

Algorithm Example

Using 4-bit unsigned #s; 2 x 3 = 6; Multiplicand = 0010; Multiplier = 0011

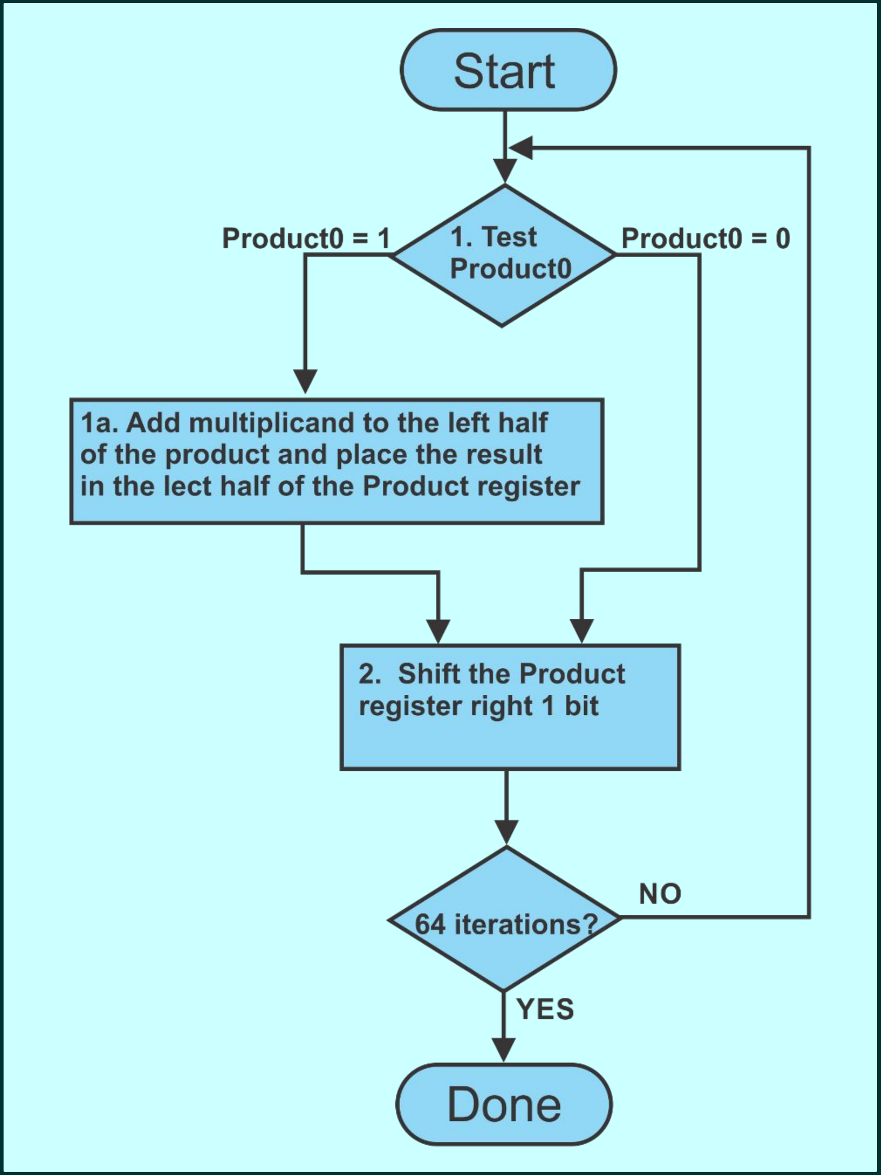
Iter	Step	Multiplier	Multiplicand	Product
0	Initial Values	0011	0000 0010	0000 0000
1	1a. 1 □ prod = prod + mcand	0011	0000 0010	0000 0010
	2. shift left multiplicand	0011	0000 0100	0000 0010
	3. shift right multiplier	0001	0000 0100	0000 0010
2	1a. 1 □ prod = prod + mcand	0001	0000 0100	0000 0110
	2. shift left multiplicand	0001	0000 1000	0000 0110
	3. shift right multiplier	0000	0000 1000	0000 0110
3	1a. 0 □ no-op	0000	0000 1000	0000 0110
	2. shift left multiplicand	0000	0001 0000	0000 0110
	3. shift right multiplier	0000	0001 0000	0000 0110
4	1a. 0 □ no-op	0000	0001 0000	0000 0110
	2. shift left multiplicand	0000	0010 0000	0000 0110
	3. shift right multiplier	0000	0010 0000	0000 0110

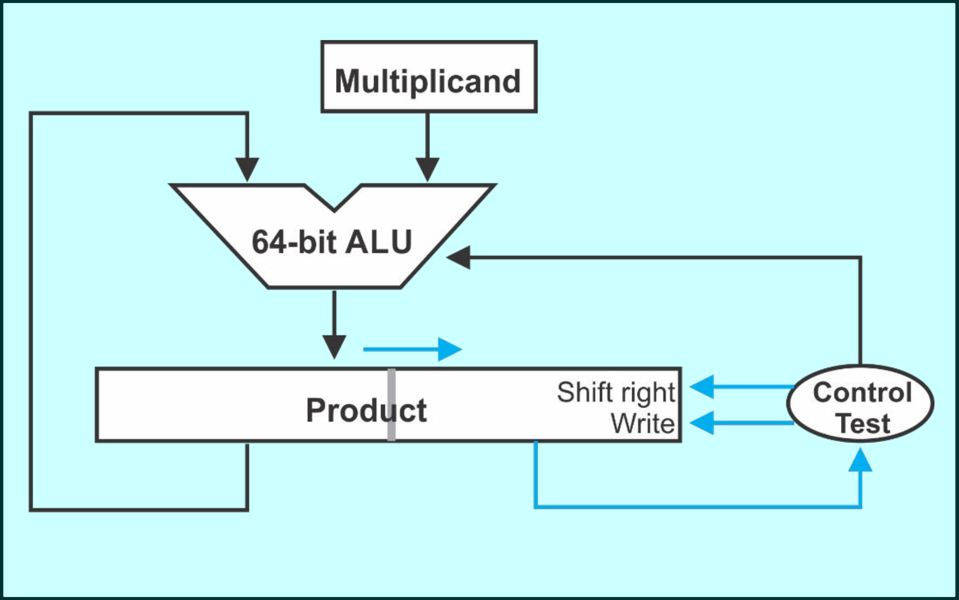
Final Answer

Multiplication Algorithm

- Problems:
  - 3 steps executed 64 times taking 192 cycles if each step takes one clock cycle
  - Wasted cycles if product is calculated in less than 64 iterations
- Refinements include
  - Combining multiplier into unused space in the product which eliminates a separate multiplier register and the step in the algorithm for shifting the multiplier value
  - The multiplier begins in the right half of the product and the LSB is shifted out on each iteration of the algorithm

Multiplication Algorithm and Hardware Design (Refined Version)


Start
1. Test Product0
Product0 = 1
Product0 = 0
1a. Add multiplicand to the left half of the product and place the result in the lect half of the Product register
2. Shift the Product register right 1 bit
64 iterations?
NO
YES
Done



- Multiplier is placed in the right half of the product.
- Multiplicand is added to the left half of the product.
- Product is shifted right each iteration.

Example of Refined Algorithm

Using 4-bit unsigned #s; 2 x 3 = 6; Multiplicand = 0010; Multiplier = 0011

Iter	Step	Multiplicand	Product/Mult
0	Initial Values	0010	0000 0011
1	1a. 1 □ prod = prod + mcand	0010	0010 0011
	2. shift right product	0010	0001 0001
2	1a. 1 □ prod = prod + mcand	0010	0011 0001
	2. shift right product	0010	0001 1000
3	1a. 0 □ no-op	0010	0001 1000
	2. shift right product	0010	0000 1100
4	1a. 0 □ no-op	0010	0000 1100
	2. shift right product	0010	0000 0110

Final Answer

Signed Multiplication

- The previous algorithm works for signed integers
  - Assume working with values of infinite digits but representing them with only 64 bits
    - All values sign extended to 64 bits in registers
  - Shifting steps would need to extend the sign of the product for signed number
  - Shift right arithmetic (sra) instruction will do this
    - Previous most significant bit value (sign bit) is shifted in which extends the sign bit
  - Upon completion of the algorithm, the lower 64 bits will have the correct product

Signed Multiplication Examples
(Using 8-bit byte values)

```
( 10)           00001010
(-10)           11110110
                --------
                00000000
               00001010
              00001010
             00000000
            00001010
           00001010
          00001010
         00001010
        ----------------
(-100)  0000100110011100
```

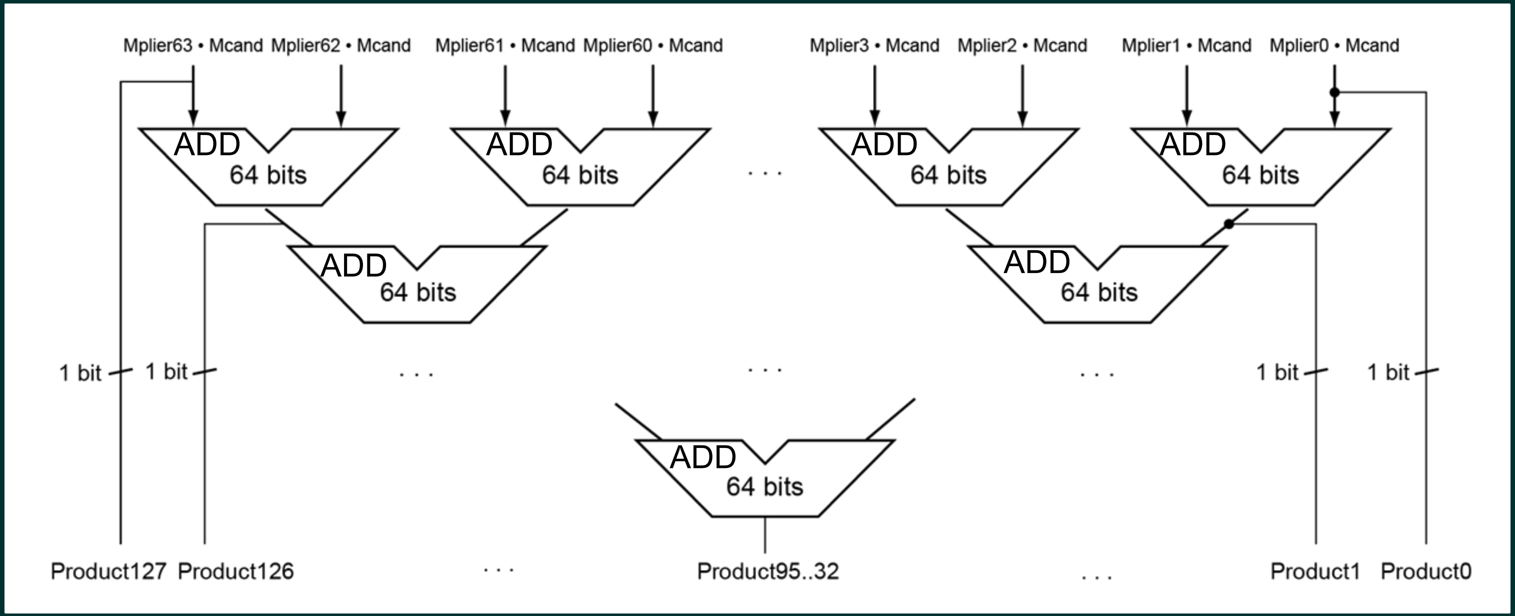
Only the lower
8 bits are valid.

```
(-8)            11111000
(-7)            11111001
                --------
                11111000
               00000000
              00000000
             11111000
            11111000
           11111000
          11111000
         11111000
        ----------------
(56)    1111000100111000
```

Only the lower
8 bits are valid.

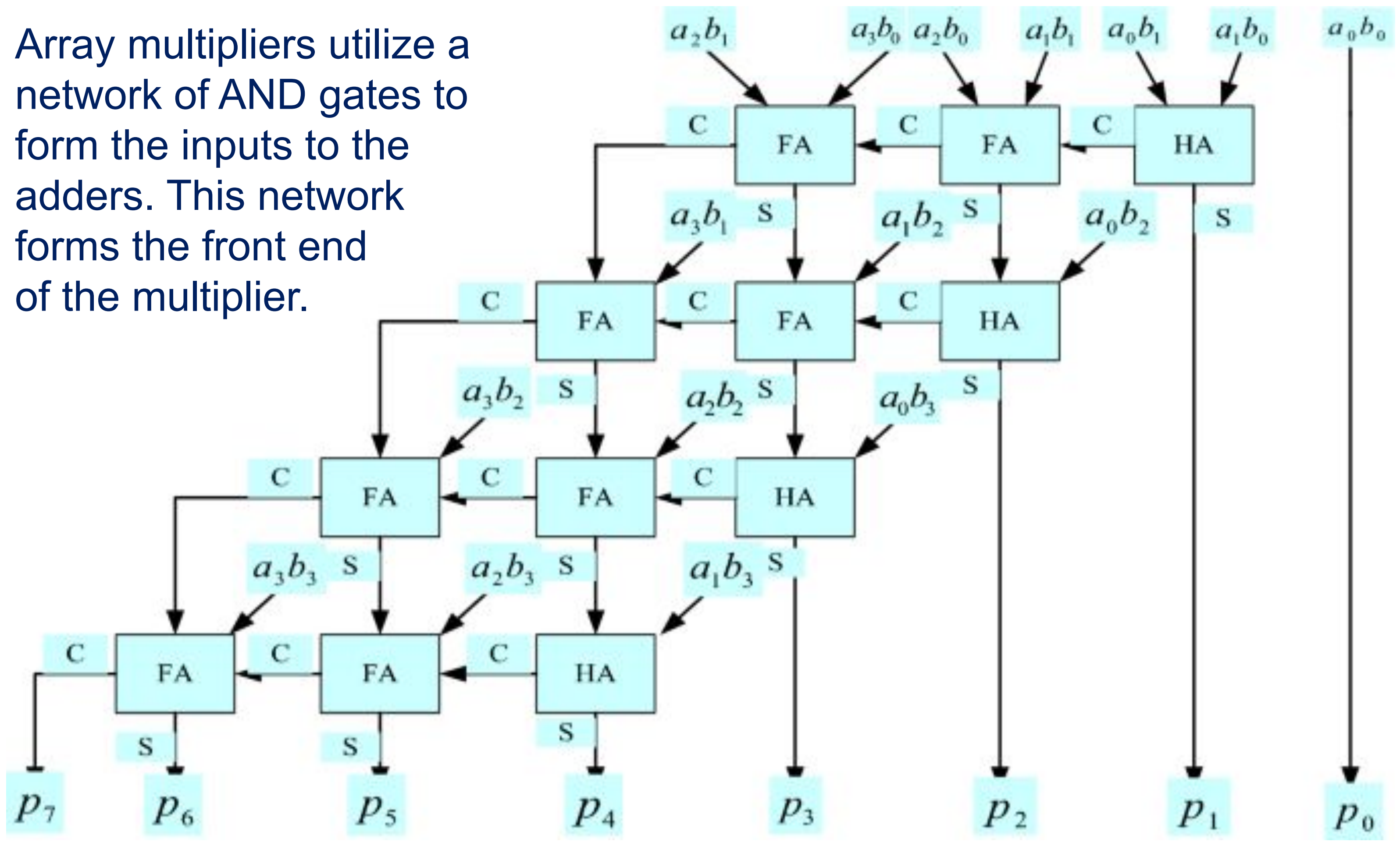
Faster Multiplication

- Use multiple adders in a pipelined configuration
- Costlier but better performance
- Additions performed in parallel by designing a tree structure
  - Rather than waiting for 64 additions in a sequential fashion, time is reduced to 6 add times: $\log_2(64)$



Two’s Complement 4-Bit Array Multiplier

Array multipliers utilize a network of AND gates to form the inputs to the adders. This network forms the front end of the multiplier.



FA = full adder; HA = half adder; a_nb_n = logical AND of bits

This design is easily extended to 8-, 32-, & 64-bits.

RISC-V Multiplication

- To produce a properly signed or unsigned product, RISC-V has four multiply instructions
  - mul (multiply) – produces 64 bit product
  - mulh (multiply high) – produces the upper 64 bits of the 128 bit product if both operands are signed
  - mulhu (multiply high unsigned) – produces the upper 64 bits of the 128 bit product if both operands are unsigned
  - mulhsu (multiply high signed-unsigned) – produces the upper 64 bits of the 128 bit product if one operand is signed and the other is unsigned
- All instructions require 3 operands: destination register & two source registers

RISC-V Multiplication Overflow

- The multiply high instructions can be used to check for overflow in software
  - There is no overflow for 64-bit unsigned multiplication if mulhu’s result is zero
  - There is no overflow for 64-bit signed multiplication if mulhu’s result are copies of the sign bit of mul’s result

Integer Division

- Paper and pencil method (binary division)

```
                         1 0 0 1       (Quotient)
                    ______________
(Divisor)  1 0 0 0 | 1 0 0 1 0 1 0     (Dividend)
                   - 1 0 0 0
                    ________
                         1 0
                         1 0 1
                         1 0 1 0
                       - 1 0 0 0
                        ________
                             1 0       (Remainder)
```

Base 10

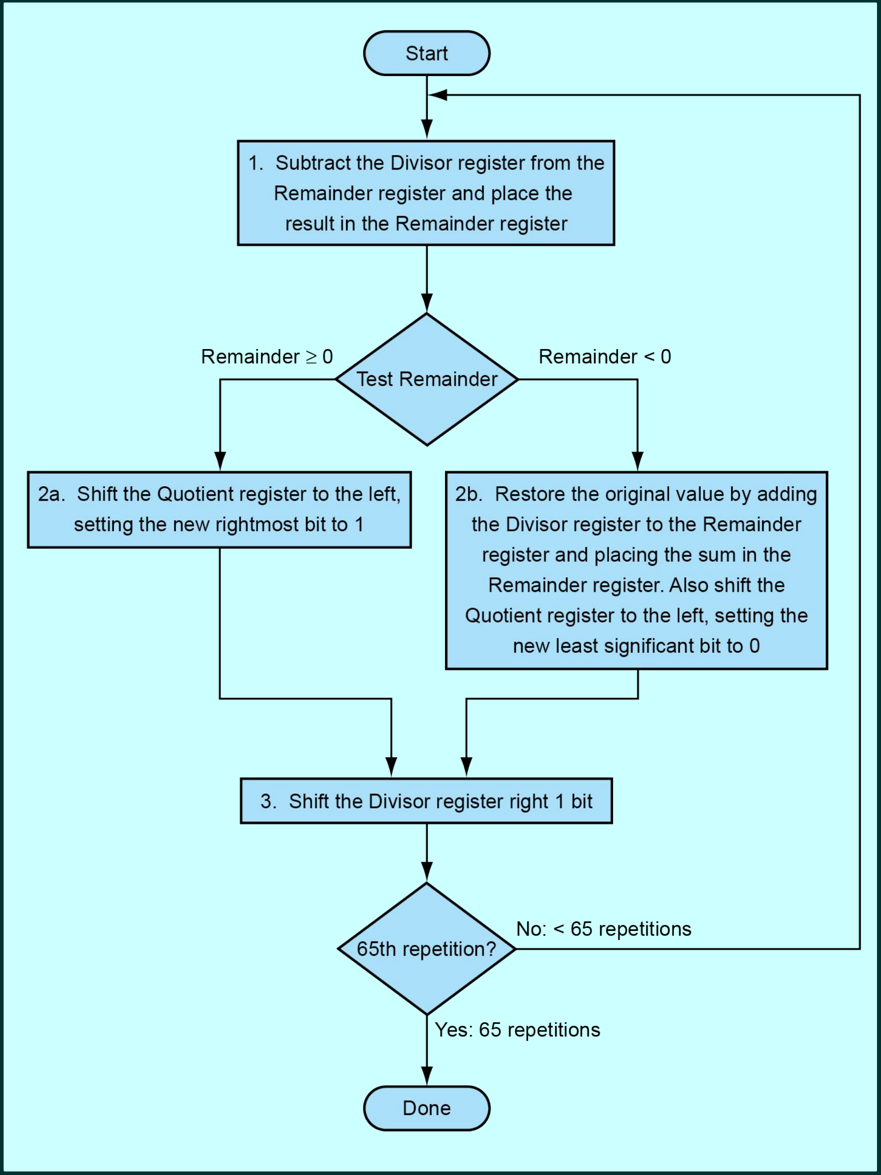
```
       9
 8 ) ‾‾74
     72
     ‾‾
      2
```

- Dividend = Quotient * Divisor + Remainder

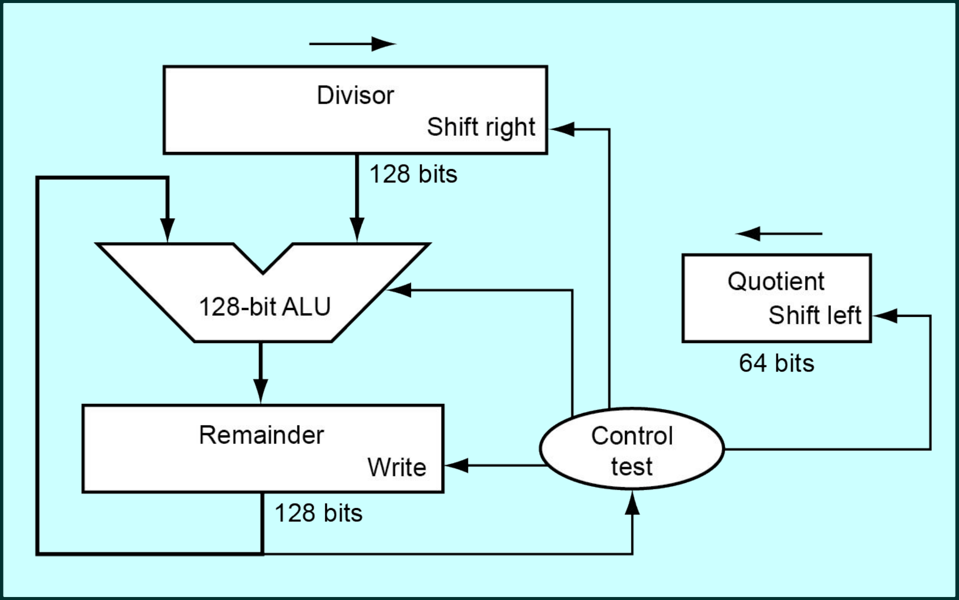
Restoring Division

- Since the processor must determine if the divisor is smaller than the dividend, a subtract operation is required (dividend – divisor)
  - If the result is positive or zero, then a quotient bit of 1 is generated
  - If not, a quotient bit of zero is generated and the subtract operation is reversed to restore the value of the dividend
  - After shifting the divisor, the operation is iterated again
- The basic algorithm is called restoring division because the dividend is restored to its value prior to the subtract operation

Division Algorithm (First Version)



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
65th repetition?
No: < 65 repetitions
Yes: 65 repetitions
Done



Divisor placed in left half of divisor.

Dividend placed in remainder.

Divisor shifts right and quotient shifts left on each iteration.

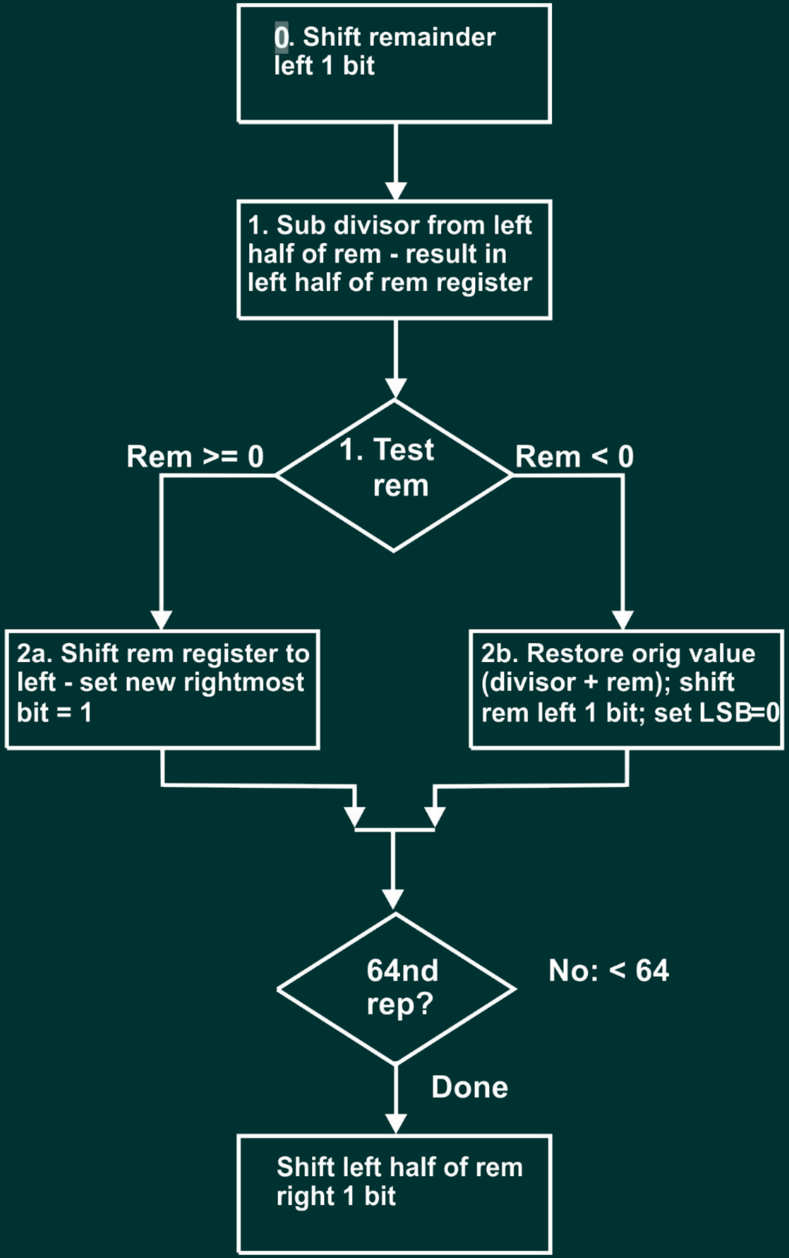
Algorithm Example (13 ÷ 5 = 2 R 3)

Iteration	Step	Quotient	Divisor	Remainder
0	Initialize values	0000	0101 0000	0000 1101
1	1. Rem = Rem – Div	0000	0101 0000	1011 1101
	2b. R < 0; +Div; sll Q; Q_0=0	0000	0101 0000	0000 1101
	3. Shift Div right	0000	0010 1000	0000 1101
2	1. Rem = Rem – Div	0000	0010 1000	1110 0101
	2b. R < 0; +Div; sll Q; Q_0=0	0000	0010 1000	0000 1101
	3. Shift Div right	0000	0001 0100	0000 1101
3	1. Rem = Rem – Div	0000	0001 0100	1111 1001
	2b. R < 0; +Div; sll Q; Q_0=0	0000	0001 0100	0000 1101
	3. Shift Div right	0000	0000 1010	0000 1101
4	1. Rem = Rem – Div	0000	0000 1010	0000 0011
	2a. R >= 0; sll Q; Q_0=1	0001	0000 1010	0000 0011
	3. Shift Div right	0001	0000 0101	0000 0011
5	1. Rem = Rem- Div	0001	0000 0101	1111 1110
	2b. R < 0; +Div; sll Q; Q_0=0	0010	0000 0101	0000 0011
	3. Shift Div right	0010	0000 0010	0000 0011

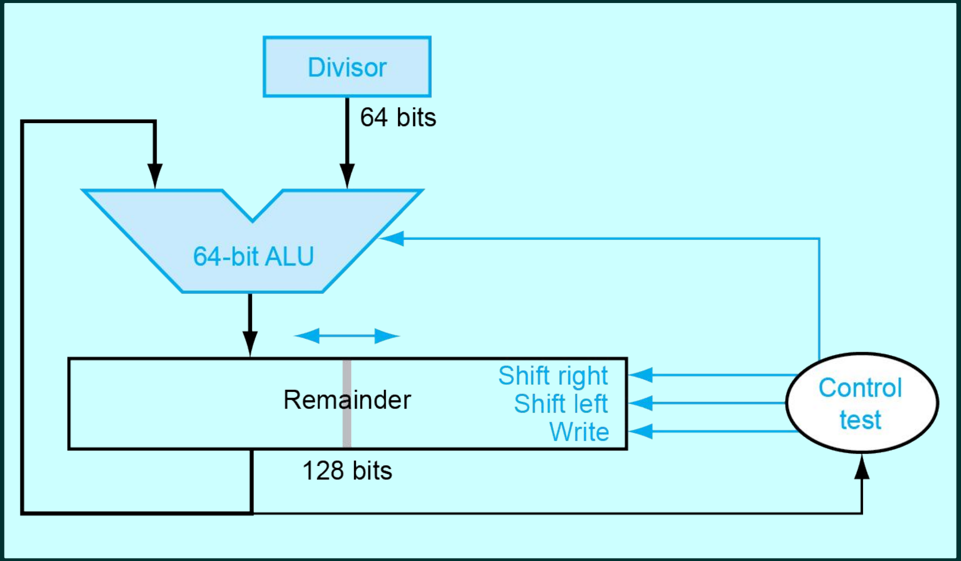
= 2

= 3

Refined Division Algorithm & Hardware



0. Shift remainder left 1 bit
1. Sub divisor from left half of rem - result in left half of rem register
1. Test rem
Rem >= 0
Rem < 0
2a. Shift rem register to left - set new rightmost bit = 1
2b. Restore orig value (divisor + rem); shift rem left 1 bit; set LSB=0
64nd rep?
No: < 64
Done
Shift left half of rem right 1 bit



Similar to multiplication, the quotient and remainder values are combined into the remainder register to simplify the algorithm reducing the number of steps and save space by not having a separate dividend register.

Refined Division Example (14 ÷ 4 = 3 R2)

Iteration	Step	Divisor	Remainder/Quotient
0	Initialize values	0100	0000 1110
	Shift Rem left 1 bit	0100	0001 1100
1	2. Rem = Rem – Div	0100	1101 1100
	3b. R < 0; +Div; sll R; R_0 = 0	0100	0011 1000
2	2. Rem = Rem – Div	0100	1111 1000
	3b. R < 0; +Div; sll R; R_0 = 0	0100	0111 0000
3.	2. Rem = Rem – Div	0100	0011 0000
	3a. R >= 0; sll R; R_0 = 1	0100	0110 0001
4.	2. Rem = Rem – Div	0100	0010 0001
	3a. R >= 0; sll R; R_0 = 1	0100	0100 0011
	Shift left half of Rem right 1 bit	0100	0010 0011

2 3

Signed Division

- Remember the signs of the divisor and dividend and negate the quotient if the signs disagree
- Dividend and remainder **must** have the same signs no matter what the signs of the divisor and quotient
- Division formula must prove true for all values (Dividend = Quotient x Divisor + Remainder)

+7 ÷ +2 = +3 R + 1 -7 ÷ +2 = -3 R -1

+7 ÷ -2 = -3 R +1 -7 ÷ -2 = +3 R -1

Faster Division Algorithm

- Many processors today use the SRT division
  - Named for its creators (Sweeney, Robertson, and Tocher)
- Uses a lookup table based on the dividend and the divisor to predict several quotient bits generated per step
  - Predictive requiring subsequent steps to correct wrong predictions
  - Accuracy depends on proper values in the lookup table
  - This lookup table is contained in special storage within the hardware of the ALU.
- Used on various Intel CPUs

Contemporary Division Implementations

- Newton-Raphson iteration provides a high-speed method for performing division.
  - The algorithm begins with an initial approximation to the reciprocal of the divisor
  - This value is iteratively refined until a specified accuracy is achieved.
  - Used in some Digital Signal Processors
- Goldschmidt division uses an iterative process of repeatedly multiplying both the dividend and divisor by a common factor F_i, chosen such that the divisor converges to 1.
  - This causes the dividend to converge to the sought quotient Q
  - Used in AMD processors

RISC-V Division

- RISC-V provides four division instructions
  - div (divide) – signed division
  - divu (divide unsigned) – unsigned division
  - rem (remainder) – signed remainder
  - remu (remainder unsigned) – unsigned remainder
- Overflow is ignored by divide instructions
  - Must check in software if quotient is too large
- Potential errors
  - Divide by zero – must check prior to hardware divide
  - Some division algorithms include this as the first step
  - Assembly language programmer must include in code check for zero divisor before every division instruction

Divide by Zero

- RISC-V doesn't raise an exception on divide by zero.
- The result of dividing by zero is all 1s in destination register
  - For unsigned numbers, this is the largest integer
  - For signed numbers, this is -1.
- The check for divide by zero must be done in software prior to any divide instruction
  - Added by compiler or included as an instruction by assembly language programmer

```
beq     t0, zero, divzero # t0 = denominator
div t2, t1, t0          # divide
```

  - Label divzero will be code to generate exception