3. Multiplication

Multiplication

- Multiplication is much more complicated than addition and subtraction

- **Paper-and-pencil example** (1000_{10} x 1001_{10}):

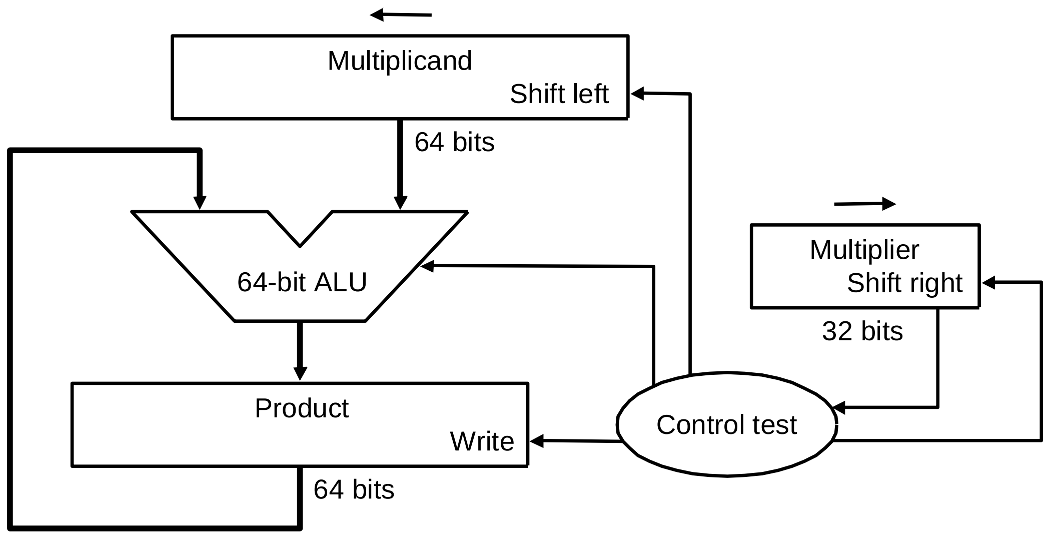
```
Multiplicand                      1000
Multiplier                        1001
--------------------------------------
                                  1000
                                 0000
                                0000
                               1000
--------------------------------------
Product                        1001000
```

- **Observation**:

  Suppose we limit ourselves to using only digits 0 and 1

  If we ignore the sign bits (i.e., unsigned numbers), multiplying an N-bit multiplicand with an M-bit multiplier gives a product that is <u>at most N+M bits long</u>

Sequential Multiplication Hardware – Version 1



- This version follows the flow of paper-and-pencil example
- **One 64-bit ALU and three registers**
  - 64-bit multiplicand register, 64-bit product register, 32-bit multiplier register
- **Operations:**
  - The 32-bit multiplicand starts in the right half of the multiplicand register, and is shifted left 1 bit at each step
  - The multiplier register is shifted right 1 bit at each step
  - The product register is initialized to 0
  - Control decides when to shift the multiplicand and multiplier registers and when to write new values into the product register

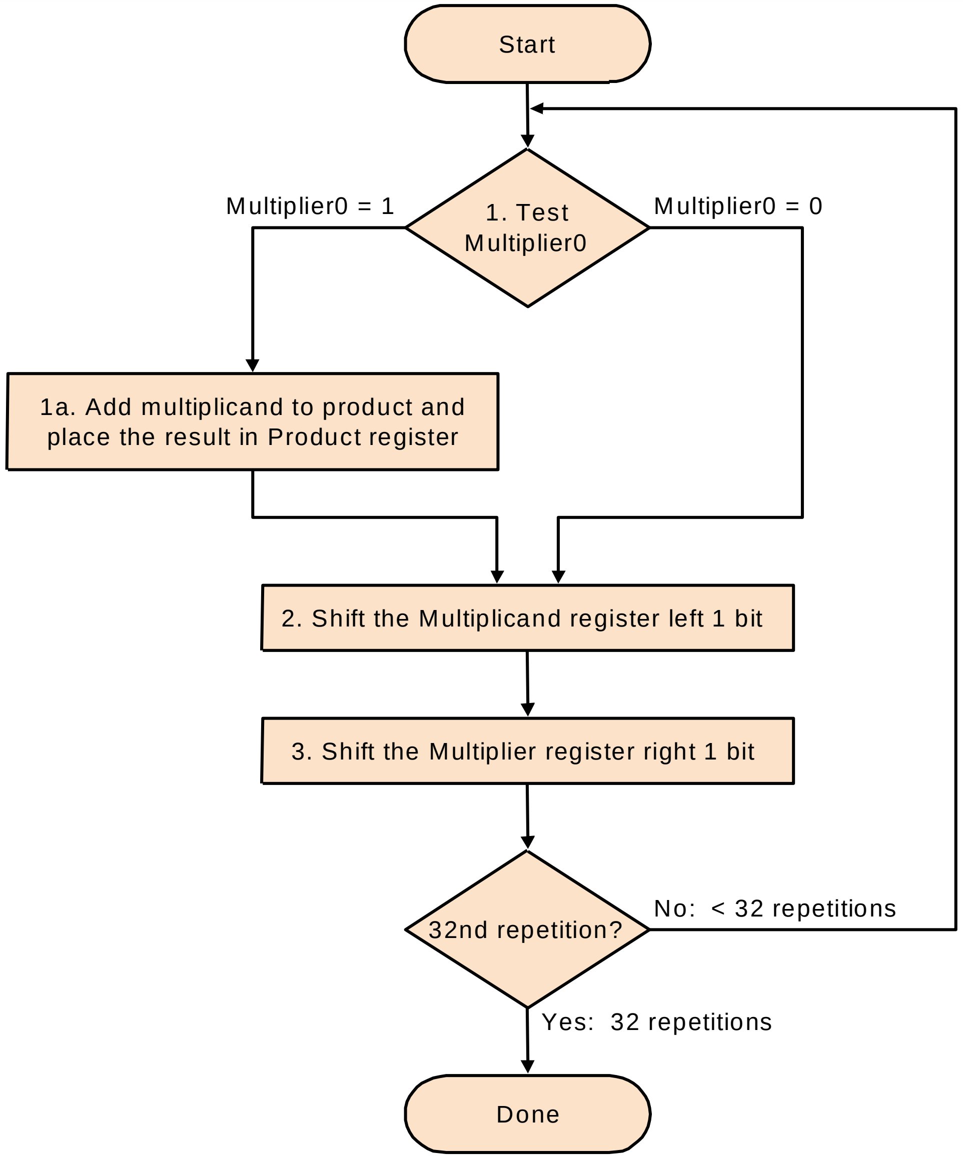
Example for Multiplication Version 1

- 4-bit unsigned integer multiplication: 0010 x 0011

	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

- Color: initial value, recently changed, check bit

Multiplication Algorithm Version 1 (in MIPS processor)


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

Three basic steps needed for each bit

If we need one clock cycle for each step then about 100 clock cycles are needed to multiply two 32-bit numbers

Slow!

Some Observations About Version 1

- Only half of the multiplicand register contains useful bit values
  - Reduce multiplicand register size: 64-bit → 32-bit
- A full 64-bit ALU is wasteful and slow
  - Because half of the adder bits add 0 to the intermediate sum
  - Reduce ALU size: 64-bit → 32-bit
- The multiplicand is shifted left with 0s inserted in the new positions
  - The multiplicand cannot affect the least significant bits of the product after they settle down

Example to Explain Version 2

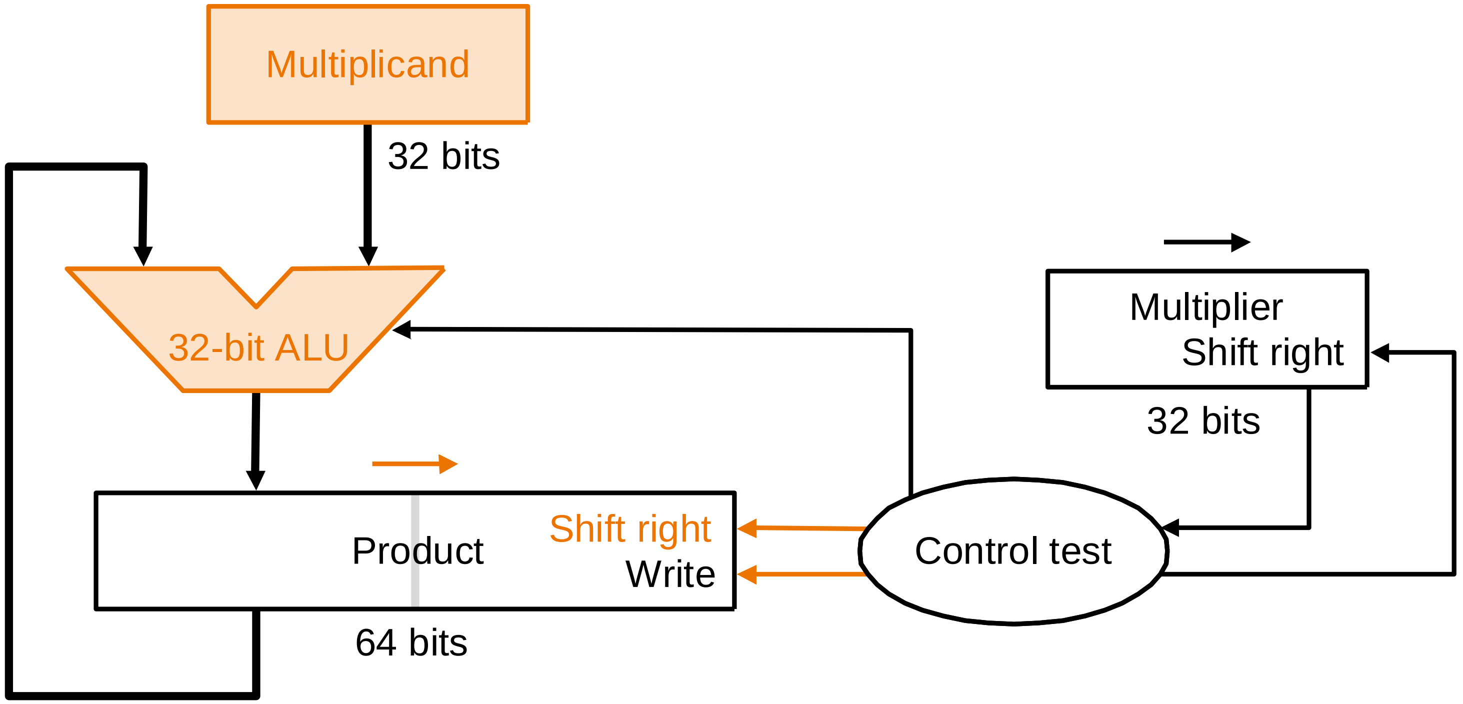
- **Paper-and-pencil example** (1000_{10} x 1001_{10}):

Multiplicand (4-bit)	1000	
Multiplier (4-bit)	1001	
Product (8-bit)	0000	0000
Add multiplicand	1000	
Product	1000	0000
Product (shift right)	0100	0000
Add 0	0000	
Product	0100	0000
Product (shift right)	0010	0000
Add 0	0000	
Product	0010	0000
Product (shift right)	0001	0000
Add multiplicand	1000	
Product	1001	0000
Product (shift right)	0100	1000

Addition on upper 4 bits

Sequential Multiplication Hardware - Version 2

- This version only needs a 32-bit multiplicand register and a 32-bit ALU
- This version shifts "product" instead of "multiplicand"



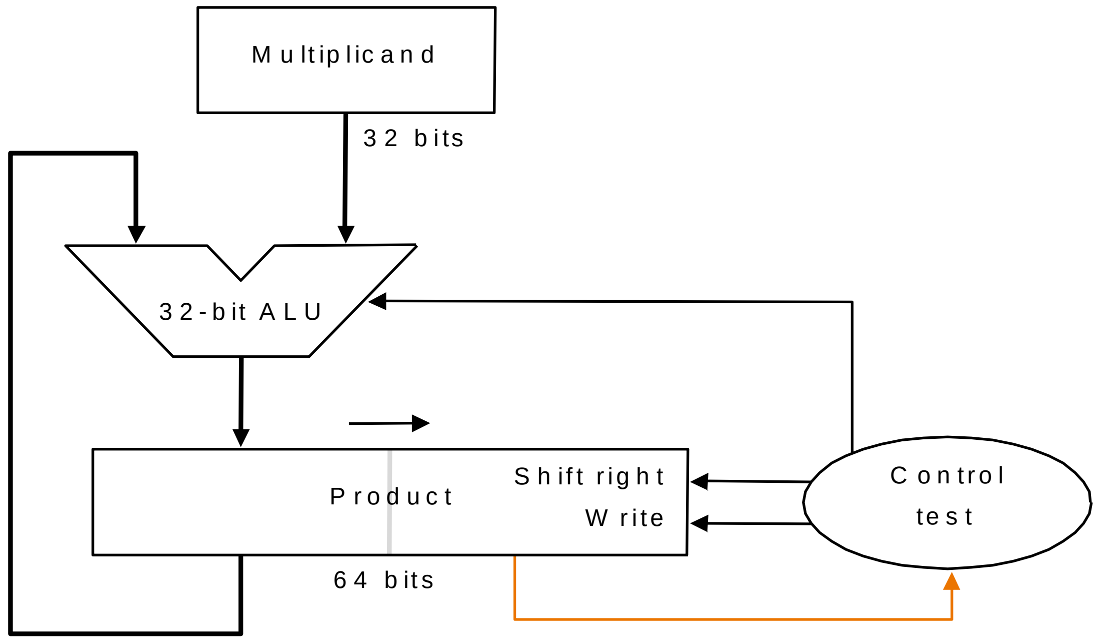
(changes made to previous version are highlighted in orange color)

Some Observations About Version 2

- The number of used bits in the product register increases by 1 bit at each step, from the initial value of 32 to the final value of 64
- The number of used bits in the multiplier register decreases by 1 bit at each step, from the initial value of 32 to the final value of 0
- Hence, the unused bits of the multiplier register can be used for storing part of the product

  More specifically, **the right half of the product register can be combined with the multiplier register to save hardware**

Multiplication Hardware - Refined Version

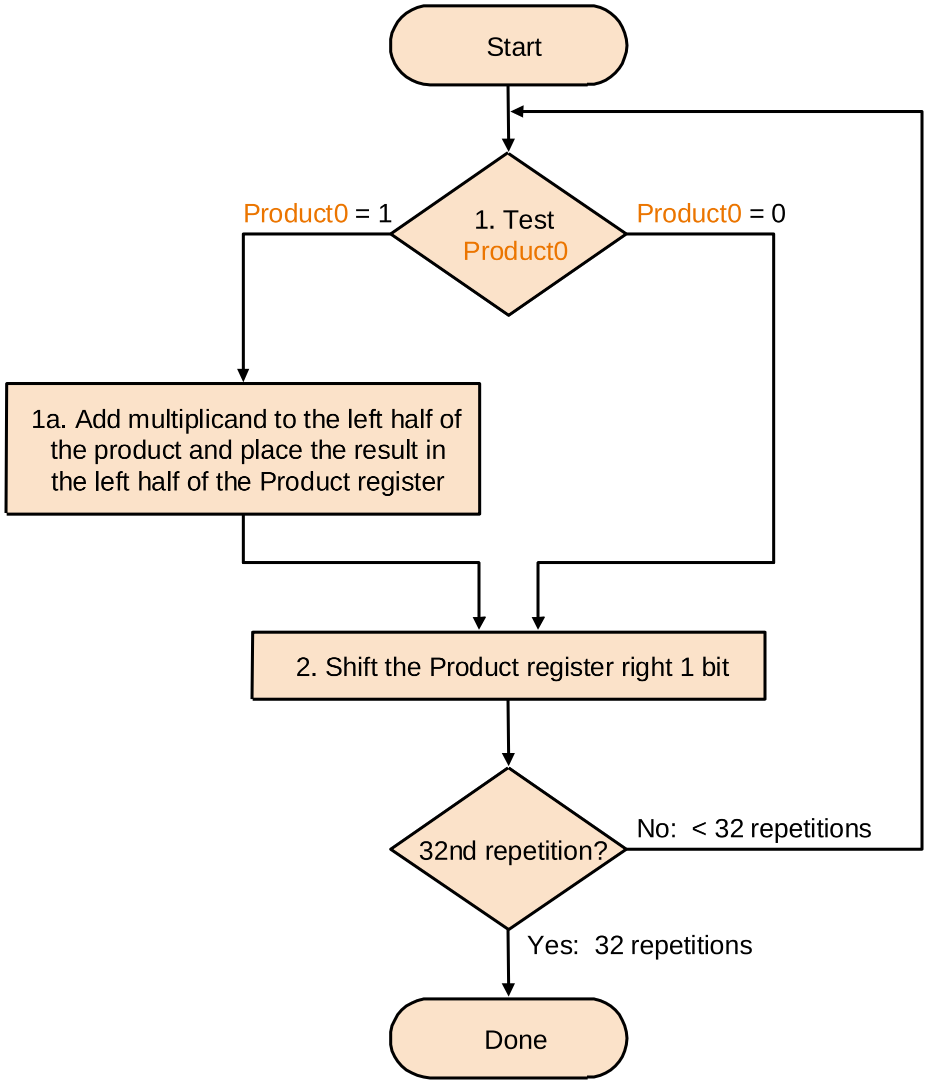


- **32-bit ALU**
- **Two registers**:
  - **Multiplicand register: 32 bits, Product register: 64 bits** (right half also used for storing multiplier)
- **Operations**:

  The right half of the product register is initialized to the multiplier, and its left half is initialized to 0

  The two right-shifts at each step for version 2 are combined into only a single right-shift because the product and multiplier registers have been combined

Multiplication Algorithm - 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 left half of the Product register
2. Shift the Product register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Example

- Multiplication of two 4-bit unsigned numbers (0110 and 0011)

Multiplier

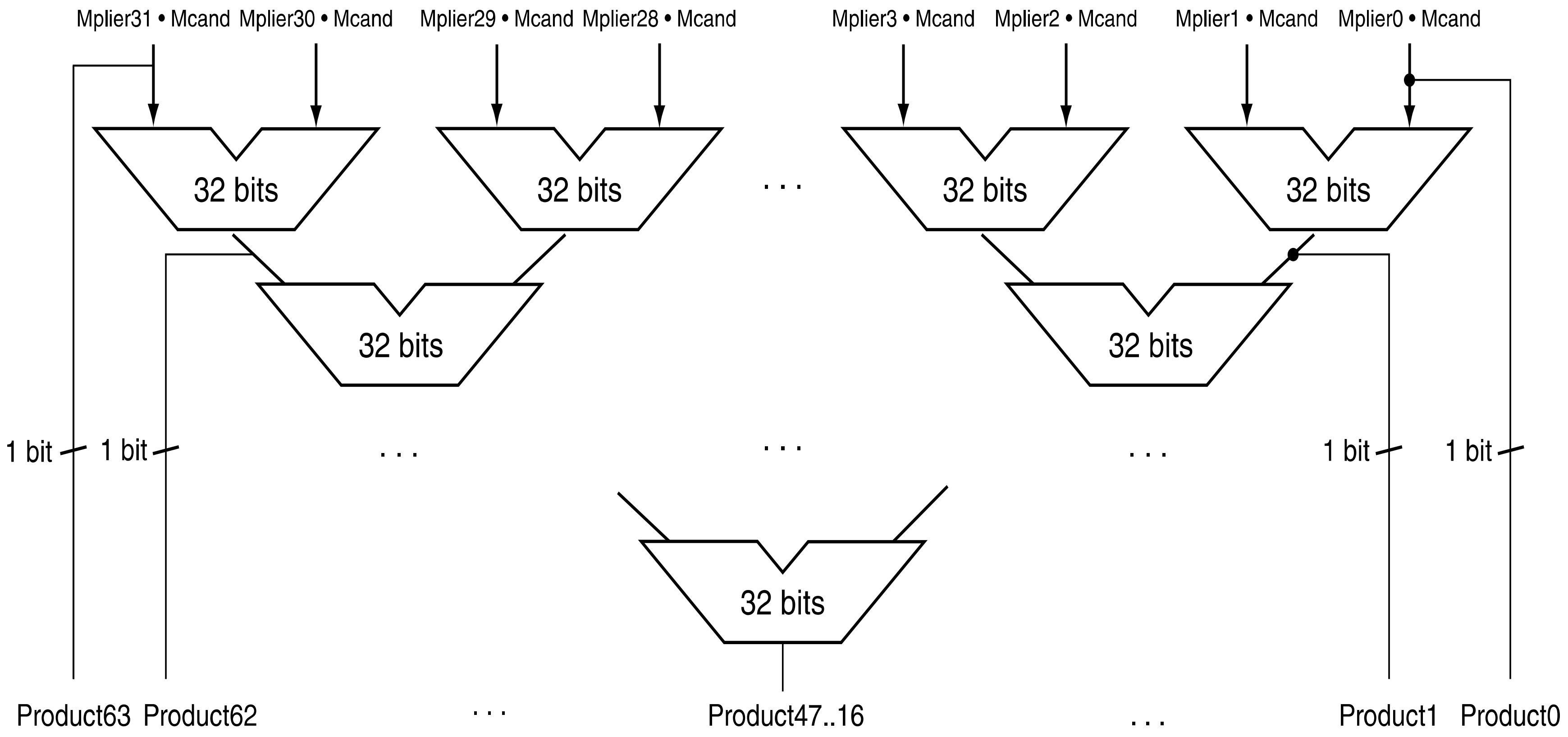
Iteration	Multiplicand (M)	Product (P)	Remark
0	0110	0000 0011	Initial state
1		0110 0011	Left(P) = Left(P) + M
		0011 0001	P = P >> 1
2		1001 0001	Left(P) = Left(P) + M
		0100 1000	P = P >> 1
3		0100 1000	No operation
		0010 0100	P = P >> 1
4		0010 0100	No operation
		0001 0010	P = P >> 1

Signed Multiplication

- If the multiplicand or multiplier is negative, we first negate it to get a positive number
- Use any one of the above methods to compute the product of two positive numbers
- The product should be negated if the original signs of the operands disagree
- **Booth's algorithm**: a more efficient and elegant algorithm for the multiplication of signed numbers (to be covered in tutorial)

Hardware speedup

- Moore's law implies more and more cheaper hardware resources available
- Unroll the for loop and use 31 adders instead of single adder 32 times
- This organization minimizes delay to do 1 Multiply in 5-add time



Multiply in MIPS

- Separate pair of 32-bit registers to contain 64-bit product, `Hi` and `Lo`

- `mult` (multiply) and `multu` (multiply unsigned)

```
mult  $s2, $s3        # Hi, Lo = $s2 x $s3
multu $s2, $s3        # Hi, Lo = $s2 x $s3
```

  Both MIPS multiply instructions ignore overflow

  No overflow if `Hi` is 0 for `multu` or the replicated sign of `Lo` for `mult`

- Fetch the integer 32-bit product

  `mflo` (move from lo) `mflo $s1` `# $s1 = Lo`

  `mfhi` (move from hi) `mfhi $s1` `# $s1 = Hi`

  `mfhi` can transfer `Hi` to a general-purpose register to test for overflow

4. Division

Division

- Division is the reciprocal operation of multiplication
- **Paper-and-pencil example** (1001010_{ten} / 1000_{ten}):

```
                       1001      Quotient
Divisor   1000      |1001010     Dividend
                    -1000000
                     0001010
                     0001010
                     0001010
                       -1000
                          10     Remainder
```

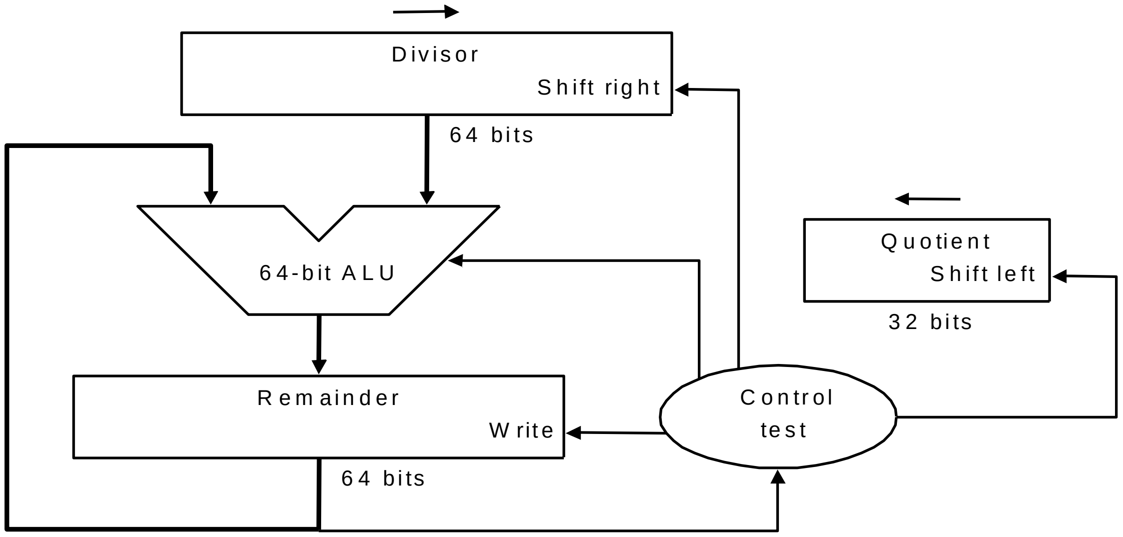
Dividend = Quotient x Divisor + Remainder

Division Example in Binary

- **Paper-and-pencil 4-bit example** (0111_2 / 0010_2):

```
                      00011        Quotient
Divisor   0010      ________
                   |00000111       Dividend
                   -00100000
                   -00010000
                   -00001000
                   -00000100
                    00000011
                   -00000010
                    00000001       Remainder
```

Division Hardware

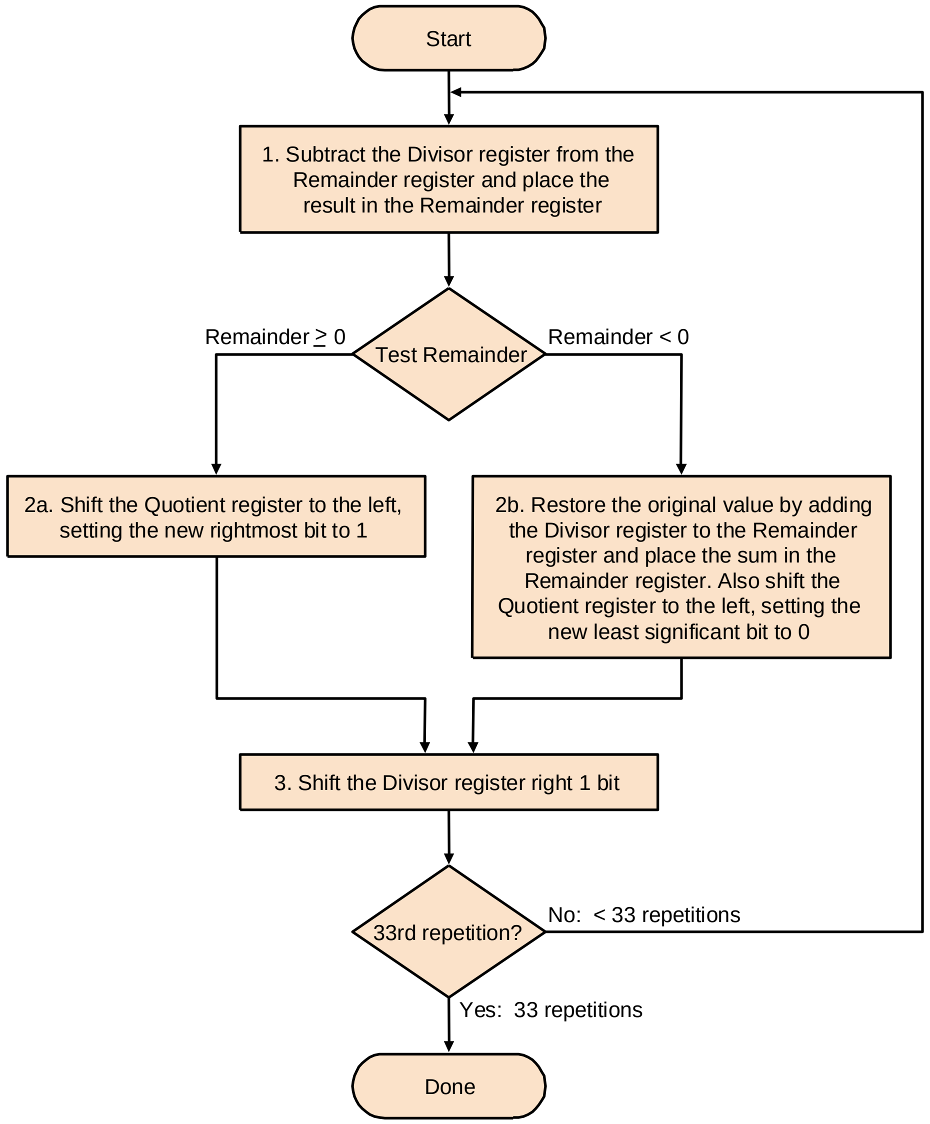


- **64-bit ALU, three registers:**

  Divisor register: 64 bits, Quotient register: 32 bits, Remainder register: 64 bits

- **Operations**:
  - 32-bit divisor starts in the left half of divisor register; is shifted right 1 bit at each step
  - Quotient register is initialized to 0; shifted left 1 bit at each step
  - Remainder register is initialized with the dividend
  - Control decides
    - when to shift the divisor and quotient registers
    - when to write new values into the remainder register

Division Algorithm

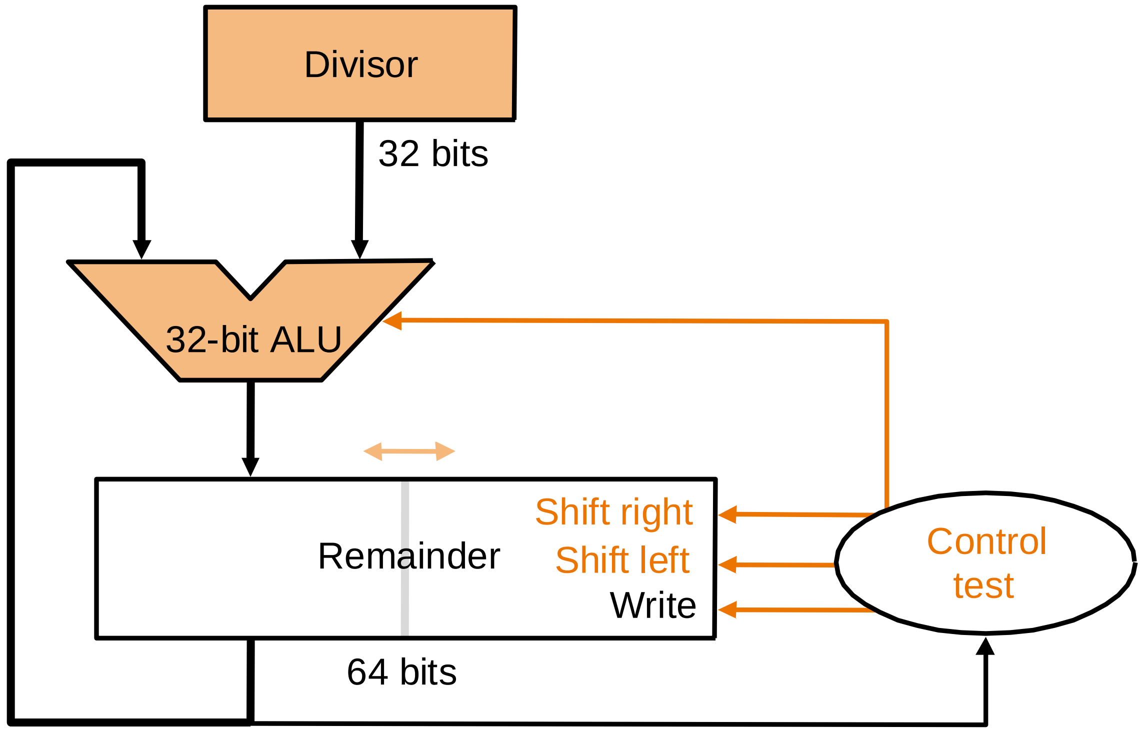

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to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

Some Observations

Like the first version of the multiplication hardware

- At most half of the divisor register has useful information
  - Both the divisor register and ALU could potentially be cut in half
- Shift divisor register to right ⇒ Shift remainder register to left
  - Produce the same alignment
  - But, simplify hardware necessary for the ALU and divisor register
- Combine the remainder and quotient registers

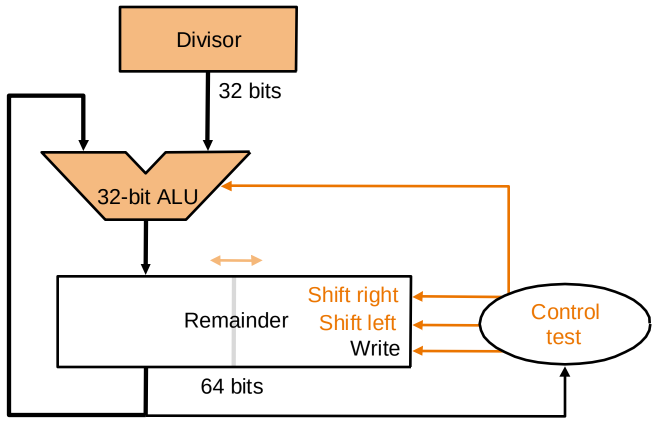
Division Hardware – Refined Version



(changes made to previous version are highlighted in orange color)

Division Hardware – Refined Version

- **32-bit ALU**
- **Two registers**:
  - **Divisor register**: **32 bits**
  - **Remainder register**: **64 bits**
  - (right half also used for storing quotient)



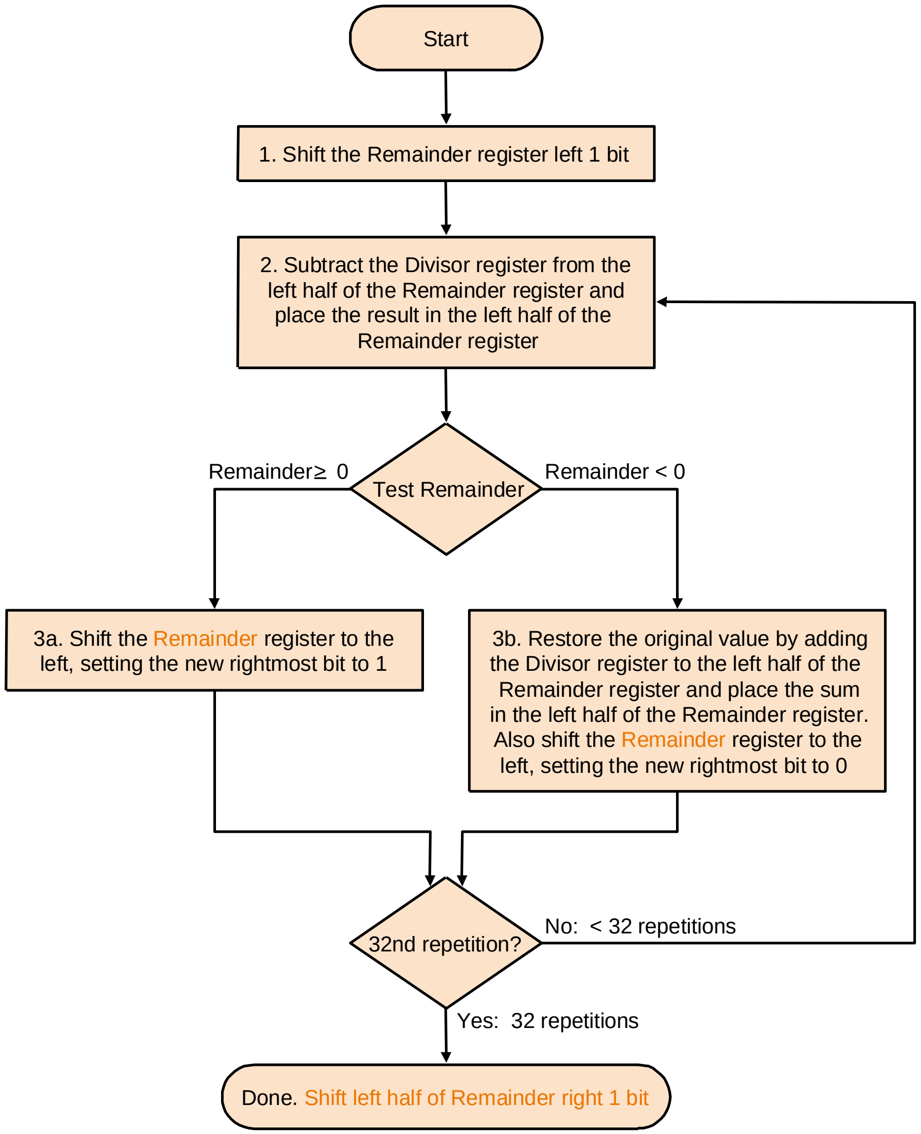
- **Operations**:
  - 32-bit divisor is always subtracted from the left half of remainder register
    - The result is written back to the left half of the remainder register
  - The right half of the remainder register is initialized with the dividend
    - Left shift remainder register by one before starting
  - The new order of the operations in the loop is that the remainder register will be **shifted left one time too many**
    - Thus, **final correction step:** must **right shift back only the remainder** in the left half of the remainder register

Example

- **Paper-and-pencil example** (0111_2 / 0010_2):

```
                          ~~0~~0011      Quotient
Divisor   0010          ___________
                       | 00000111        Dividend
                       -0010
                         00001110
                       -0010
                         00011100
                       -0010
                         00111000
                       -0010
                       ________
                         00011000
                         00110000
                       -0010
                       ________
                         0001        Remainder
```

Division Algorithm – Refined Version


Start
1. Shift the Remainder register left 1 bit
2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register
Remainder≥ 0
Test Remainder
Remainder < 0
3a. Shift the Remainder register to the left, setting the new rightmost bit to 1
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

Example

- Division of a 4-bit unsigned number (0111) by another one (0011)

Iteration	Divisor (D)	Remainder (R)	Remark
0	0011	0000 0111	Initial state
		0000 1110	R = R << 1
1		1101 1110	Left(R) = Left(R) – D
		0000 1110	Undo
		0001 1100	R = R << 1, R_0 = 0
2		1110 1100	Left(R) = Left(R) – D
		0001 1100	Undo
		0011 1000	R = R << 1, R_0 = 0
3		0000 1000	Left(R) = Left(R) – D
		0001 0001	R = R << 1, R_0 = 1
4		1110 0001	Left(R) = Left(R) – D
		0001 0001	Undo
		0010 0010	R = R << 1, R_0 = 0
extra		0001 0010	Left(R) = Left(R) >> 1

Remainder **Quotient**

correction

Signed Division

- Similar to signed multiplication, the signs of the divisor and dividend are checked to determine whether the results (quotient and remainder) should be negated.

- **Two rules to follow**:

  If the signs of the divisor and dividend are different, then the quotient should be negated.

  If the remainder is nonzero, then its sign should be the same as that of the dividend.

- **Example**:

Dividend	Divisor	Quotient	Remainder
+7	+2	+3	+1
−7	+2	−3	−1
+7	−2	−3	+1
−7	−2	+3	−1

Divide in MIPS

- `div` ('divide')
- `divu` ('divide unsigned')

- Examples:

```
div    $s1, $s2     # Lo = $s1 / $s2; Hi = $s1 mod $s2
divu   $s1, $s2     # Lo = $s1 / $s2; Hi = $s1 mod $s2
```

Floating Point Arithmetic (optional)

IEEE 754 Standard for floating point Arithmetic

Single precision:

Exponent / Significand	0	**1 - 254**	255
0	0	$(-1)^S\times(1.F)\times(2)^{E-127}$	$(-1)^S\times(\infty)$
≠ 0	$(-1)^S\times(0.F)\times(2)^{-126}$		non-numbers e.g. 0/0 , $\sqrt{-1}$

Double precision:

Exponent / Significand	0	**1 - 2046**	2047
0	0	$(-1)^S\times(1.F)\times(2)^{E-1023}$	$(-1)^S\times(\infty)$
≠ 0	$(-1)^S\times(0.F)\times(2)^{-1022}$		non-numbers e.g. 0/0 , $\sqrt{-1}$

Floating-Point Addition

- Example: $9.999_{10} \times 10^{1} + 1.610_{10} \times 10^{-1}$
- Assumptions:

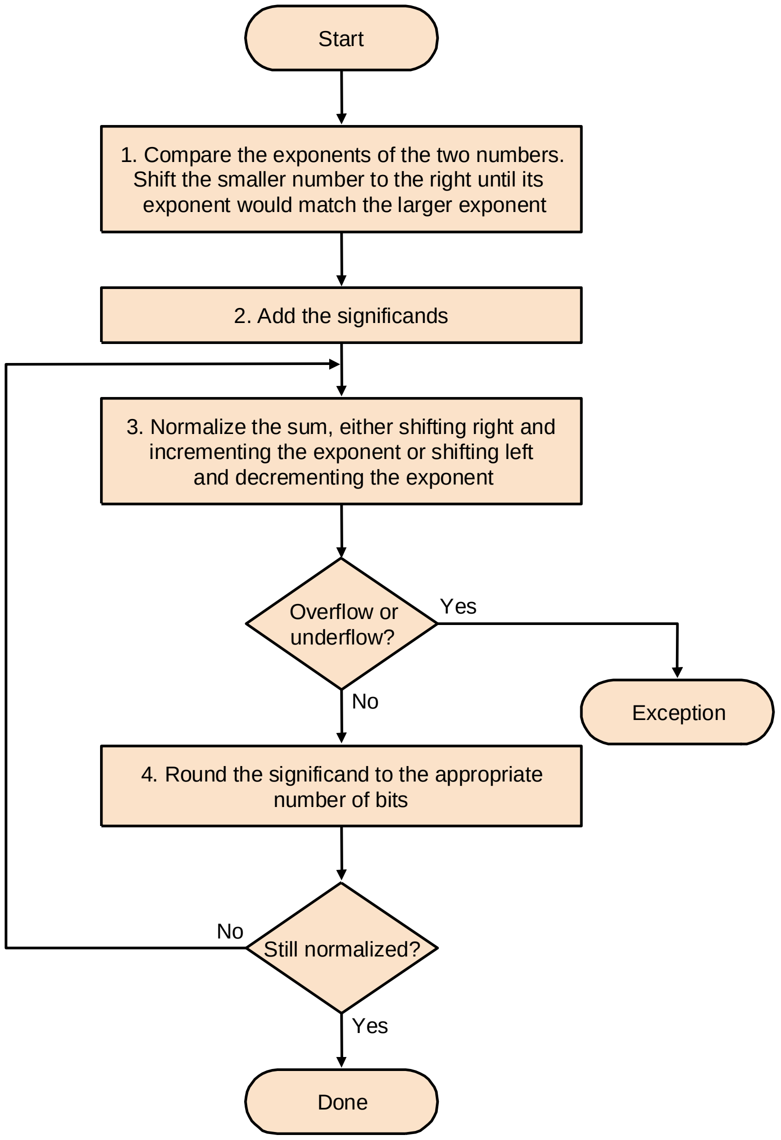
  Significand size = 4 decimal digits

  Exponent size = 2 decimal digits

Algorithm:

1. Align the decimal point of the number that has the smaller exponent
   - e.g. $1.610_{10} \times 10^{-1}$ becomes $0.016_{10} \times 10^{1}$
2. Add the significands of the two numbers together
   - e.g. $9.999_{10} \times 10^{1} + 0.016_{10} \times 10^{1} = 10.015_{10} \times 10^{1}$
3. Normalize the sum
   - e.g. $10.015_{10} \times 10^{1}$ becomes $1.0015_{10} \times 10^{2}$
4. Round the normalized sum
   - e.g. $1.0015_{10} \times 10^{2}$ becomes $1.002_{10} \times 10^{2}$

Floating-Point Addition


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

Binary Example

- Add 0.5_{10} and -0.4375_{10} in binary using the above algorithm
- Assume for simplicity that we only keep 4 bits of precision

- **Answer**:
- $0.5_{10} = 1.000_2 \times 2^{-1}$
- $-0.4375_{10} = -1.110_2 \times 2^{-2}$

1. $-1.110_2 \times 2^{-2} \Rightarrow -0.111_2 \times 2^{-1}$
2. $1.000_2 \times 2^{-1} + (-0.111_2 \times 2^{-1}) = 0.001_2 \times 2^{-1}$
3. $0.001_2 \times 2^{-1} \Rightarrow 1.000_2 \times 2^{-4}$ (no overflow/underflow)
4. $1.000_2 \times 2^{-4}$ (fits in 4 bits, no need for rounding)

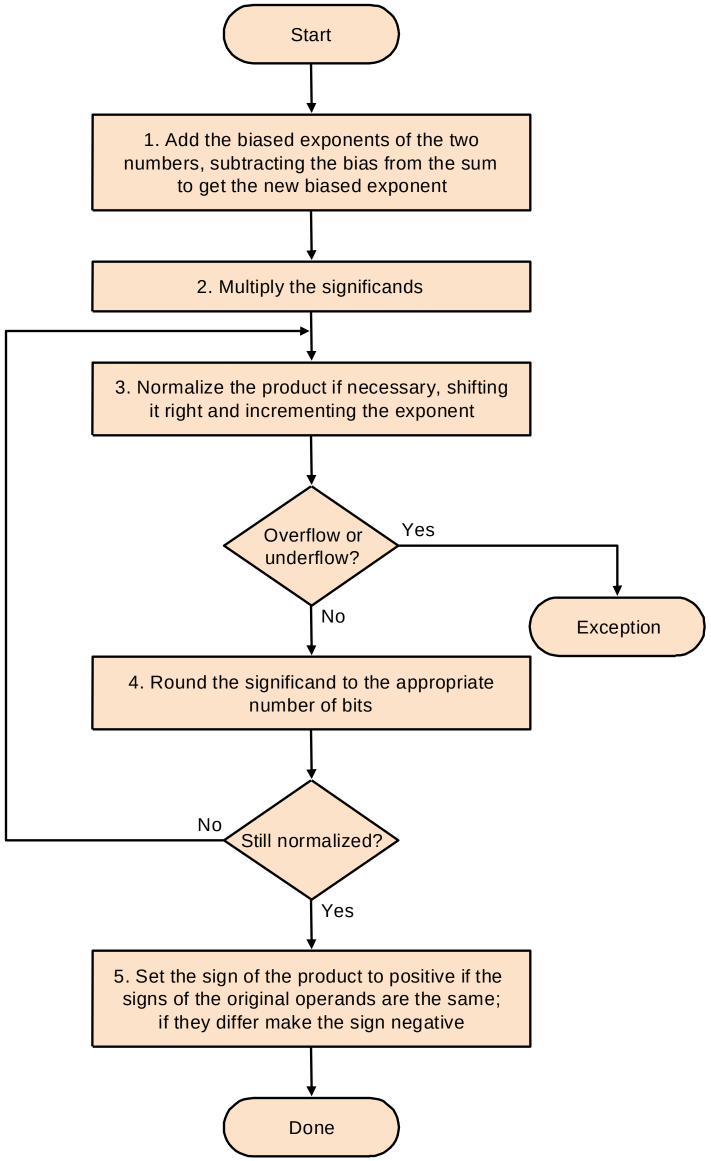
Floating-Point Multiplication

- Example: $(1.110_{10} \times 10^{10}) \times (9.200_{10} \times 10^{-5})$
- Assumptions:
  - Significand size = 4 decimal digits
  - Exponent size = 2 decimal digits

Algorithm:

1. Add the exponents together,
   - new exponent = 10 + (-5) = 5
2. Multiply the significands together
   - new significand = $1.110_{10} \times 9.200_{10} = 10.212_{10}$
3. Normalize the product,
   - $10.212_{10} \times 10^5 \Rightarrow 1.0212_{10} \times 10^6$
4. Round the product
   - $1.0212_{10} \times 10^6 \Rightarrow 1.021_{10} \times 10^6$
5. Find the sign of the product
   - $+1.021_{10} \times 10^6$

Floating-Point Multiplication


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Binary Example

- Multiply 0.5_{10} and -0.4375_{10} in binary using the above algorithm
- Assume for simplicity that we only keep 4 bits of precision

- **Answer**:
- $0.5_{10} = 1.000_2 \times 2^{-1}$
- $-0.4375_{10} = -1.110_2 \times 2^{-2}$

- new exponent = $-1 + (-2) = -3$
- new significand = $1.000_2 \times 1.110_2 = 1.110_2$
- $1.110_2 \times 2^{-3}$ remains unchanged (no overflow/underflow)
- $1.110_2 \times 2^{-3}$ fits in 4 bits (no need for rounding)
- product = $-1.110_2 \times 2^{-3}$

MIPS Instructions for Floating-Point Operations

- MIPS supports IEEE 754 single-precision and double-precision formats

- **Addition**:
  `add.s` ('addition, single'), `add.d` ('addition, double')
- **Subtraction**:
  `sub.s` ('subtraction, single'), `sub.d` ('subtraction, double')
- **Multiplication**:
  `mul.s` ('multiplication, single'), `mul.d` ('multiplication, double')
- **Division**:
  `div.s` ('division, single'), `div.d` ('division, double')
- **Comparison**:
  `c.x.s` ('comparison, single'), `c.x.d` ('comparison, double')
  where `x` may be `eq`, `neq`, `lt`, `le`, `gt`, `ge`
- **Branch**:
  `bc1t` ('branch, true'), `bc1f` ('branch, false')

Floating-Point Register

- MIPS has a FP co-processor
  - Referred to as co-processor 1
  - Has its own floating-point (FP) registers: `$f0`, `$f1`, `$f2`, …
  - These registers are used for either single or double precision
- Separate loads and stores for FP registers: `lwc1` and `swc1`
- Example:
  - load two single precision numbers from memory
  - then, add them and store the sum

```
lwc1    $f4, 4($sp)      # Load 32-bit f.p. number into F4
lwc1    $f6, 8($sp)      # Load 32-bit f.p. number into F6
add.s   $f2, $f4, $f6    # F2 = F4 + F6 single precision
swc1    $f2, 0($sp)      # Store 32-bit f.p. number from F2
```

Accurate Arithmetic

- Floating-point numbers are normally approximations
  - An infinite variety of real numbers exists between 0 and 1
  - No more than 2^{53} can be exactly represented in double precision

- Do the best we can
  - Get floating-point representation close to actual number
  - Keeps 2 extra bits on the right during intermediate additions
    - guard and round
  - Example:
    - $2.56_{10} \times 10^0 + 2.34_{10} \times 10^2$, assume 3 significant decimal digits

With guard and round digits

$$\begin{array}{r} 2.3400_{10} \\ +\,0.0256_{10} \\ \hline 2.3656_{10} \end{array}$$

Guard

round

After rounding $2.37_{10} \times 10^2$

without

$$\begin{array}{r} 2.34_{10} \\ +\,0.02_{10} \\ \hline 2.36_{10} \end{array}$$

Key Concepts to Remember

- **2's complement representation** for signed numbers

- A **32-bit ALU** can be built by connecting 32 1-bit ALUs together
  - **Subtraction** makes use of **addition**
  - **SLT** makes use of **subtraction**
  - A **multiplexor** is used in an ALU to select appropriate result
  - **Carry lookahead adders** better than **ripple carry adders**

- **Multiplication**: through a series of addition and shift operations
- **Division**: through a series of subtraction and shift operations
  - Make sure you understand how the hardware algorithms work

- **Overflow** (a type of **exception**)
  - A result of addition or subtraction
  - Detected by checking the signs of the operands and result

Key Concepts to Remember (2)

- **Floating-point numbers**
  - Representation follows closely the **scientific notation**
  - Almost all computers, including MIPS, follow **IEEE 754 standard**
- In MIPS,
  - **Single-precision** floating-point representation takes 32 bits
  - **Double-precision** floating-point representation takes 64 bits
  - Has a FP **co-processor** and separate **FP registers**
- **Overflow** (**underflow**) in floating-point representation occurs
  - When the exponent is too large (small) to be represented