

Integer Multiplication and Division

 **Unsigned Multiplication** Signed Multiplication Faster Multiplication Unsigned Division Signed Division Multiplication and Division in MIPS

Unsigned Multiplication

Paper and Pencil Example:

$$\begin{array}{r} \text{Multiplicand} \quad 11002 = 12 \\ \text{Multiplier} \quad \times 11012 = 13 \\ \hline 1100 \\ 0000 \\ 1100 \\ 1100 \\ \hline \text{Product} \quad 100111002 = 156 \end{array}$$

m-bit multiplicand \times n-bit multiplier = (m+n)-bit product

Accomplished via **shifting** and **addition**

Consumes more time and more chip area

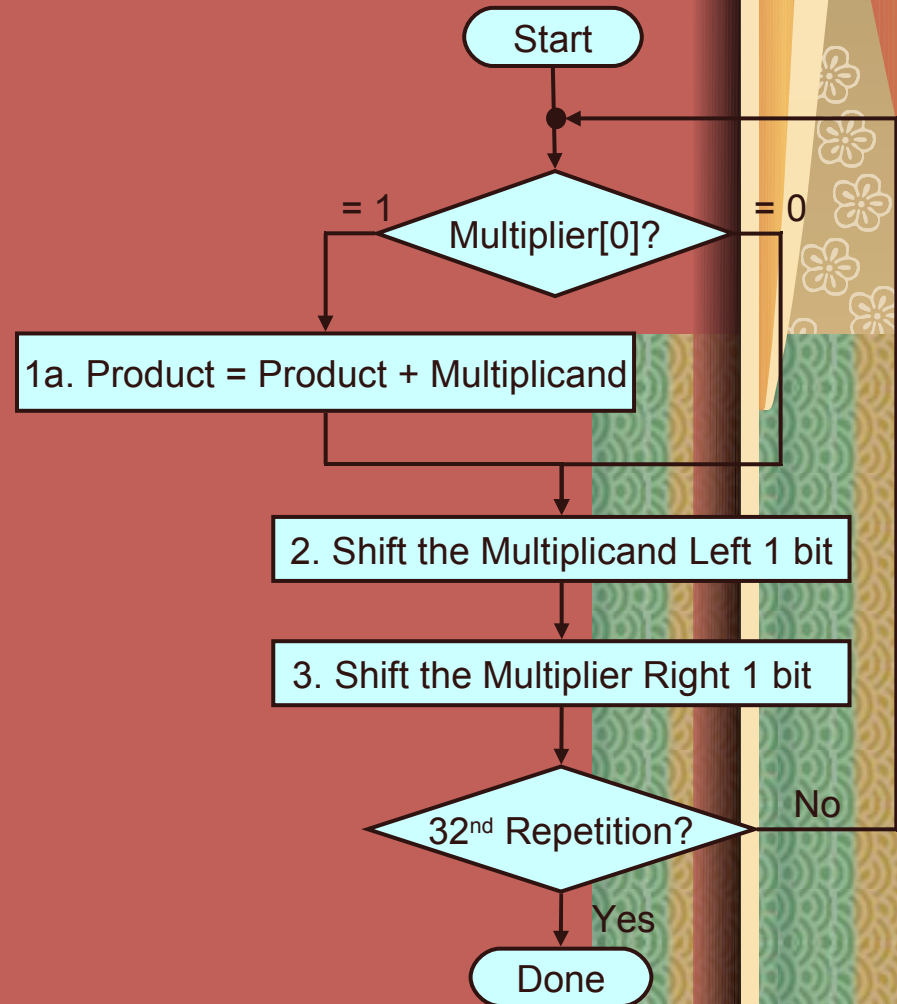
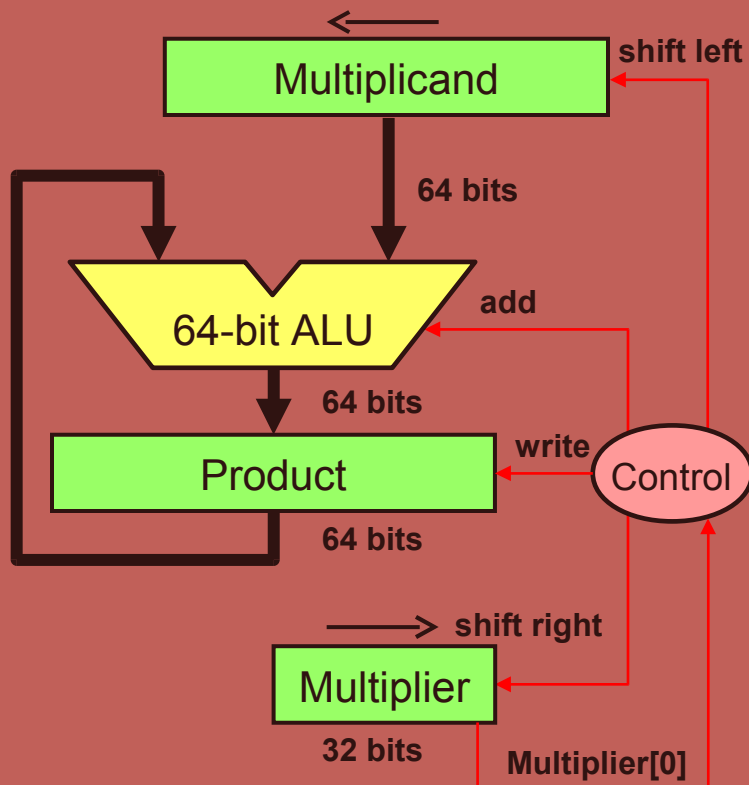
Binary multiplication is easy

$0 \times \text{multiplicand} = 0$

$1 \times \text{multiplicand} = \text{multiplicand}$

Initialize Product = 0

Multiplicand is zero extended



Consider: $1100_2 \times 1101_2$, Product = 10011100_2






4-bit multiplicand and multiplier are used in this example

Multiplicand is zero extended because it is **unsigned**




Iteration		Multiplicand	Multiplier	Product
0	Initialize	00001100	1101	00000000
1	Multiplier[0] = 1 => ADD			+→ 00001100
	SLL Multiplicand and SRL Multiplier	00011000	0110	
2	Multiplier[0] = 0 => Do Nothing			00001100
	SLL Multiplicand and SRL Multiplier	00110000	0011	
3	Multiplier[0] = 1 => ADD			+→ 00111100
	SLL Multiplicand and SRL Multiplier	01100000	0001	
4	Multiplier[0] = 1 => ADD			+→ 10011100
	SLL Multiplicand and SRL Multiplier	11000000	0000	

Signed Multiplication

Version 1 of Signed Multiplication

-  Convert multiplier and multiplicand into positive numbers
 -  If negative then obtain the 2's complement and remember the sign
-  Perform unsigned multiplication
-  Compute the sign of the product
-  If product sign < 0 then obtain the 2's complement of the product

Refined Version:

-  Use the refined version of the unsigned multiplication hardware
-  When shifting right, **extend the sign** of the product
-  If multiplier is negative, the **last step** should be a **subtract**

Case 1: Positive Multiplier

Multiplicand $1100_2 = -4$
Multiplier $\times 0101_2 = +5$

11111100
111100

Product $11101100_2 = -20$

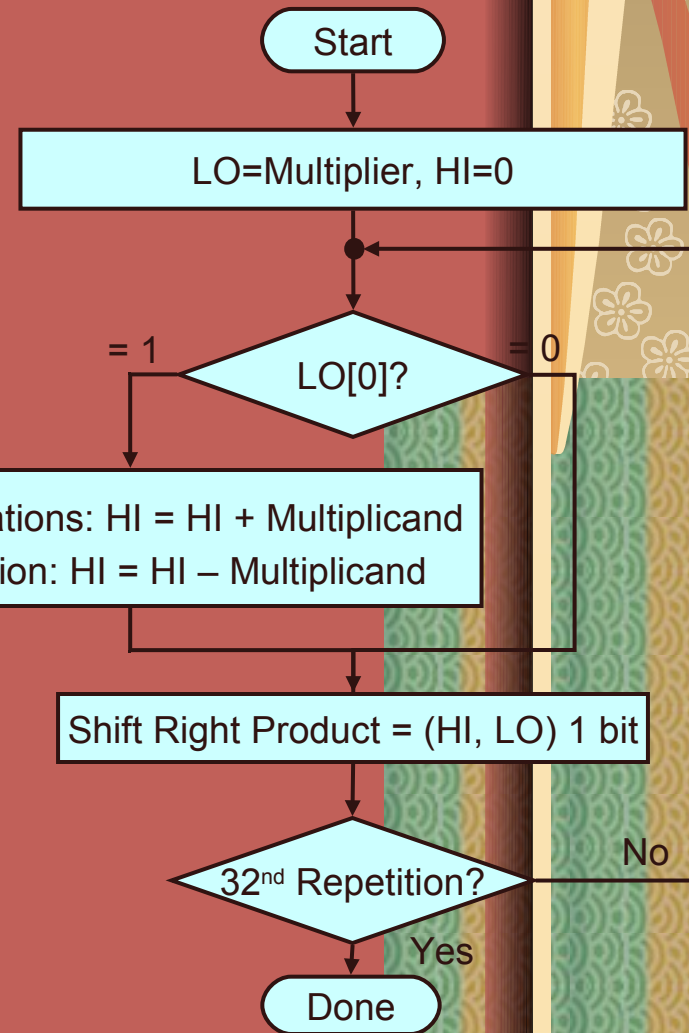
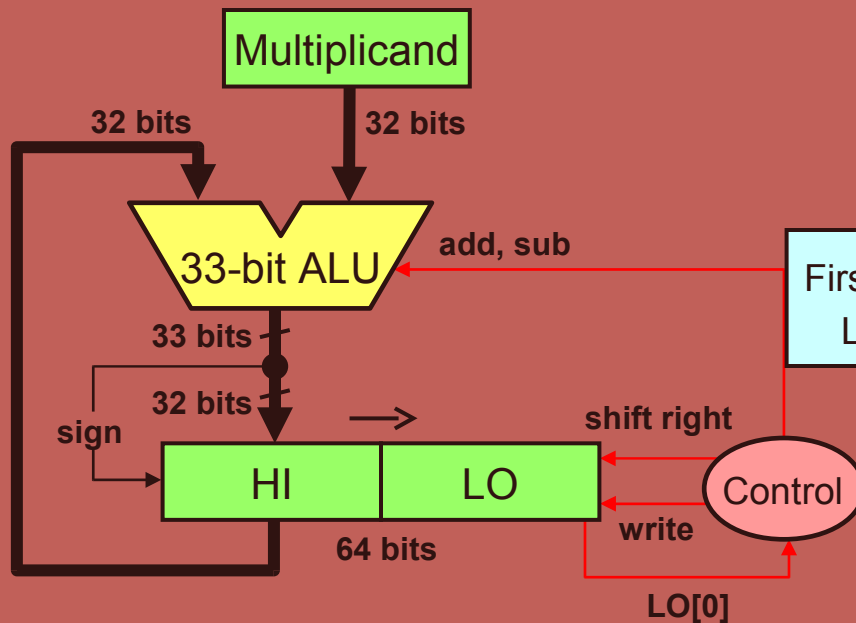
Case 2: Negative Multiplier

Multiplicand $1100_2 = -4$
Multiplier $\times 1101_2 = -3$

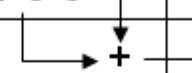
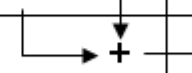
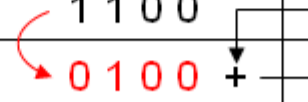
11111100
111100
00100 (2's complement of 1100)

Product $00001100_2 = +12$

Similar to Unsigned Multiplier
 ALU produces a **33-bit** result
 Multiplicand and HI are **sign-extended**
Sign is the sign of the result



Consider: $1100_2 (-4) \times 1101_2 (-3)$, Product = 00001100_2
 Multiplicand and HI are **sign-extended** before addition
 Last iteration: add 2's complement of Multiplicand

Iteration		Multiplicand	Sign	Product = HI, LO
0	Initialize (LO = Multiplier)	1 1 0 0		0 0 0 0 1 1 0 1
1	LO[0] = 1 => ADD		1	1 1 0 0 1 1 0 1
	Shift Product = (HI, LO) right 1 bit	1 1 0 0		1 1 1 0 0 1 1 0
2	LO[0] = 0 => Do Nothing			
	Shift Product = (HI, LO) right 1 bit	1 1 0 0		1 1 1 1 0 0 1 1
3	LO[0] = 1 => ADD		1	1 0 1 1 0 0 1 1
	Shift Product = (HI, LO) right 1 bit	1 1 0 0		1 1 0 1 1 0 0 1
4	LO[0] = 1 => SUB (ADD 2's compl)		0	0 0 0 1 1 0 0 1
	Shift Product = (HI, LO) right 1 bit			0 0 0 0 1 1 0 0

Unsigned Division

Divisor 1011_2

$$\begin{array}{r}
 10011_2 = 19 \\
 \overline{) 11011001_2 = 217} \\
 \underline{-1011} \\
 10 \\
 \underline{101} \\
 1010 \\
 \underline{10100} \\
 -1011 \\
 \underline{1001} \\
 10011 \\
 \underline{-1011} \\
 1000_2 = 8
 \end{array}$$

Quotient

Dividend

Dividend =

Quotient × Divisor

+ Remainder

$217 = 19 \times 11 + 8$

Try to see how big a number can be subtracted, creating a digit of the quotient on each attempt

Binary division is accomplished via shifting and subtraction

$1000_2 = 8$

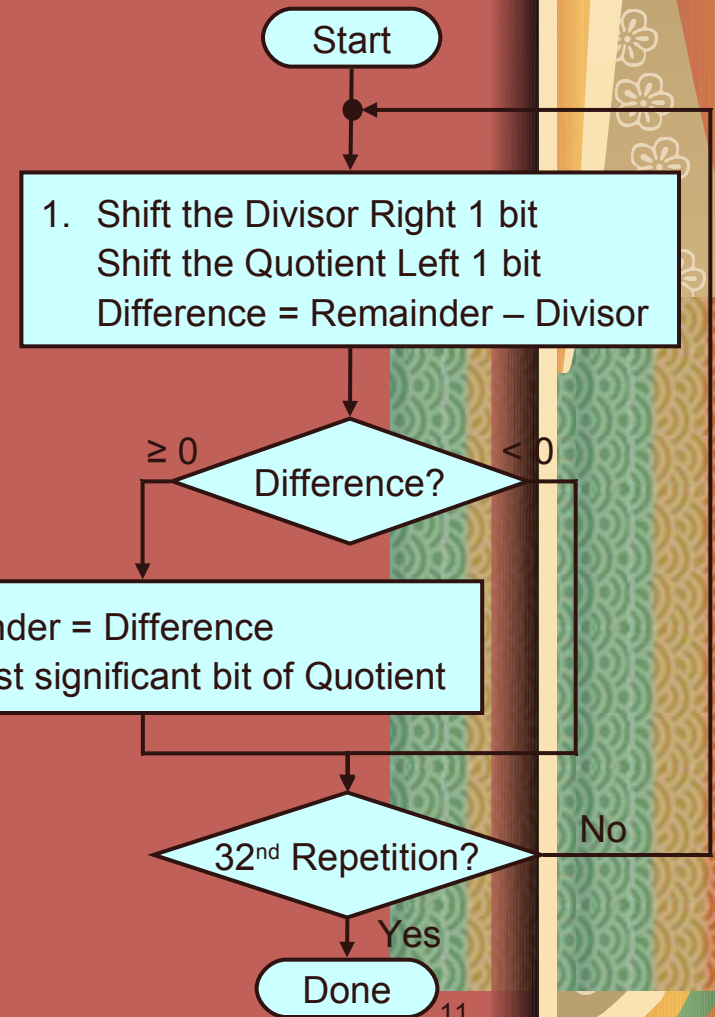
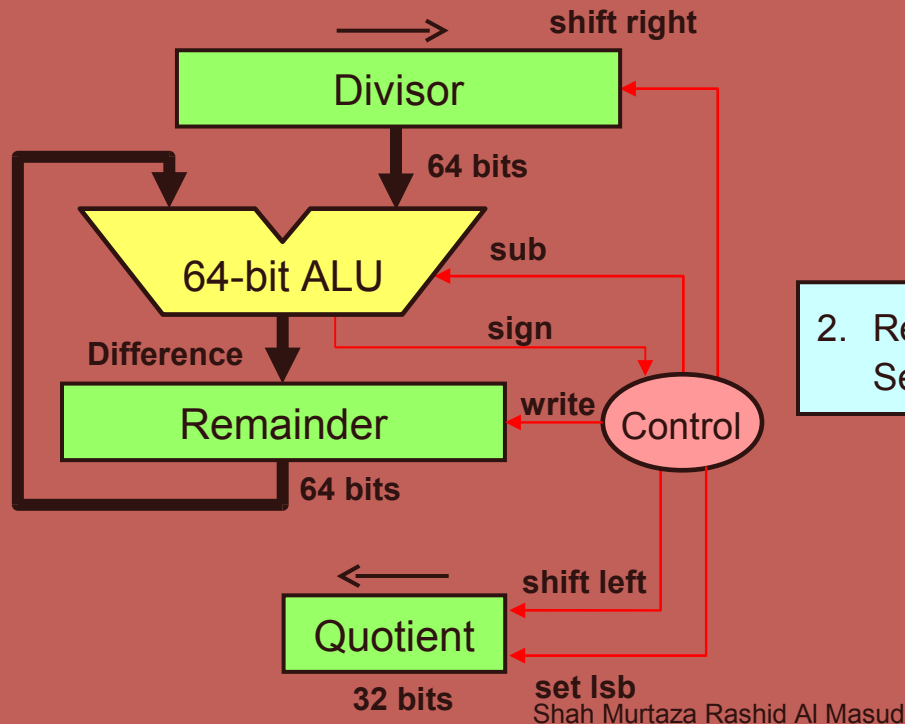
Remainder

First Division Algorithm & Hardware



Initialize:

- Remainder = Dividend (0-extended)
- Load Upper 32 bits of Divisor
- Quotient = 0



Consider: $1110_2 / 0011_2$ (4-bit dividend & divisor)
 Quotient = 0100_2 and Remainder = 0010_2
 8-bit registers for Remainder and Divisor (8-bit ALU)

Iteration		Remainder	Divisor	Difference	Quotient
0	Initialize	0000 1110	0011 0000		0000
1	1: SRL Div, SLL Q, Difference	00001110	00011000	11110110	0000
	2: Diff < 0 => Do Nothing				
2	1: SRL Div, SLL Q, Difference	00001110	00001100	00000010	0000
	2: <u>Rem</u> = Diff, set <u>lsb</u> Quotient	00000010			000 1
3	1: SRL Div, SLL Q, Difference	00000010	00000110	11111100	0010
	2: Diff < 0 => Do Nothing				
4	1: SRL Div, SLL Q, Difference	00000010	00000011	11111111	0100
	2: Diff < 0 => Do Nothing				

Observations on Version 1 of Divide

Version 1 of Division hardware can be optimized

Instead of shifting divisor right,

Shift the remainder register left

Has the same net effect and produces the same results

Reduce Hardware:

Divisor register can be reduced to 32 bits (instead of 64 bits)

ALU can be reduced to 32 bits (instead of 64 bits)

Remainder and Quotient registers can be combined

Refined Division Hardware



Observation:



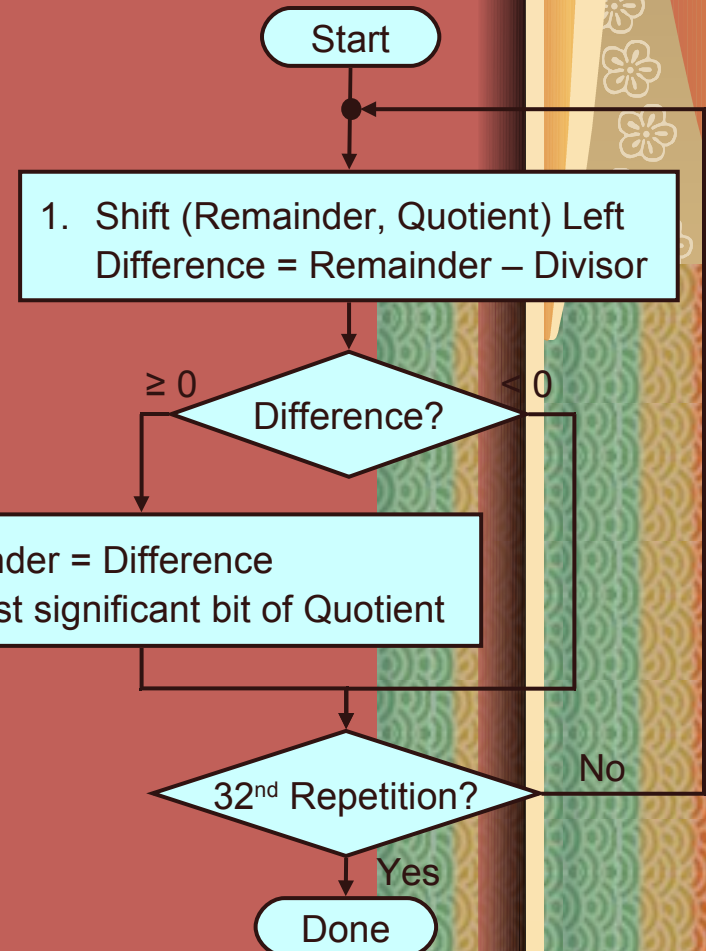
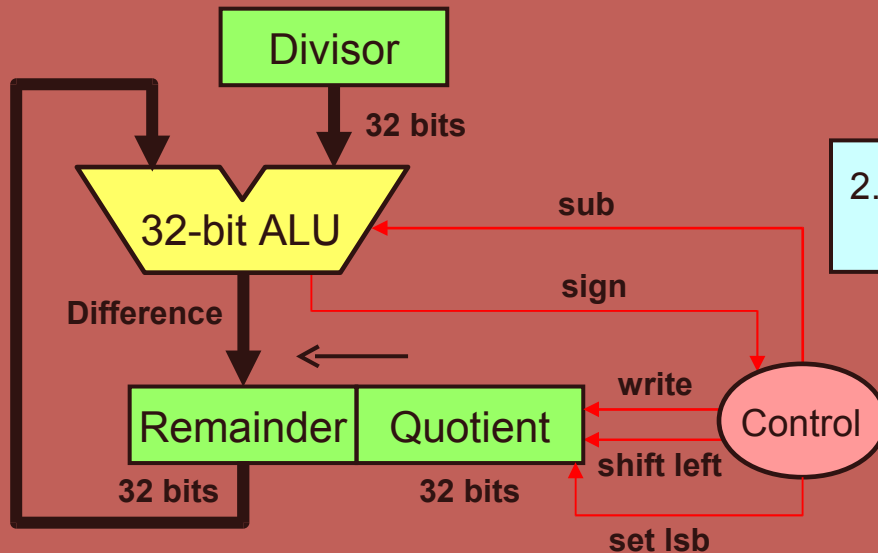
Shifting remainder left does the same as shifting the divisor right



Initialize:



Quotient = Dividend, Remainder = 0



Same Example: $1110_2 / 0011_2$ (4-bit dividend & divisor)

Quotient = 0100_2 and Remainder = 0010_2

4-bit registers for Remainder and Divisor (4-bit ALU)

Iteration		Remainder	Quotient	Divisor	Difference
0	Initialize	0 0 0 0	1 1 1 0	0 0 1 1	
1	1: Shift Left, Difference	0 0 0 1 ← 1 1 0 0		0 0 1 1	1 1 1 0
	2: Diff < 0 => Do Nothing				
2	1: Shift Left, Difference	0 0 1 1 ← 1 0 0 0		0 0 1 1	0 0 0 0
	2: <u>Rem</u> = Diff, set <u>lsb</u> Quotient	0 0 0 0	1 0 0 1		
3	1: Shift Left, Difference	0 0 0 1 ← 0 0 1 0		0 0 1 1	1 1 1 0
	2: Diff < 0 => Do Nothing				
4	1: Shift Left, Difference	0 0 1 0 ← 0 1 0 0		0 0 1 1	1 1 1 1
	2: Diff < 0 => Do Nothing				

Signed Division

- Simplest way is to remember the signs
- Convert the dividend and divisor to positive
 - Obtain the 2's complement if they are negative
- Do the unsigned division
- Compute the signs of the quotient and remainder
 - $\text{Quotient sign} = \text{Dividend sign} \text{ XOR } \text{Divisor sign}$
 - $\text{Remainder sign} = \text{Dividend sign}$
- Negate the quotient and remainder if their sign is negative
 - Obtain the 2's complement to convert them to negative

1. **Positive** Dividend and **Positive** Divisor

Example: $+17 / +3$ Quotient = $+5$ Remainder = $+2$

1. **Positive** Dividend and **Negative** Divisor

Example: $+17 / -3$ Quotient = -5 Remainder = $+2$

1. **Negative** Dividend and **Positive** Divisor

Example: $-17 / +3$ Quotient = -5 Remainder = -2

1. **Negative** Dividend and **Negative** Divisor

Example: $-17 / -3$ Quotient = $+5$ Remainder = -2

The following equation must always hold:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

Multiplication in MIPS

Two Multiply instructions

 `mult $s1,$s2` **Signed multiplication**

 `multu $s1,$s2` **Unsigned multiplication**

32-bit multiplication produces a 64-bit Product

Separate pair of 32-bit registers

 **HI = high-order 32-bit**

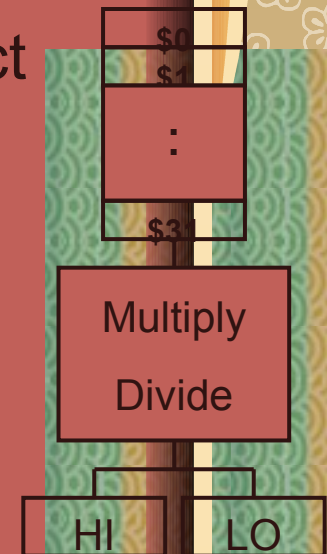
 **LO = low-order 32-bit**

 Result of multiplication is always in HI & LO

Moving data from HI/LO to MIPS registers

 `mfhi Rd` (move from HI to Rd)

 `mflo Rd` (move from LO to Rd)



Division in MIPS

Two Divide instructions

 **div** \$s1,\$s2

Signed division

 **divu** \$s1,\$s2

Unsigned division

Division produces quotient and remainder

Separate pair of 32-bit registers

 **HI = 32-bit remainder**

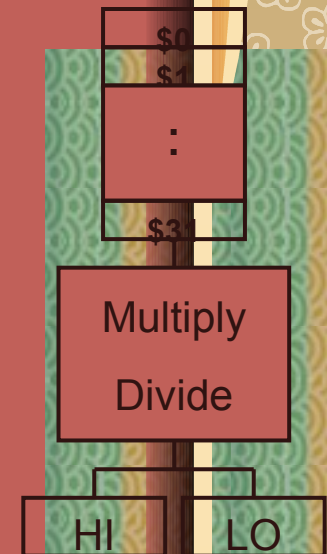
 **LO = 32-bit quotient**

 If divisor is 0 then result is **unpredictable**

Moving data to HI/LO from MIPS registers

 **mthi** Rs (move to HI from Rs)

 **mtlo** Rs (move to LO from Rs)



Booth's multiplication algorithm

- Booth's algorithm involves repeatedly adding one of two predetermined values A and S to a product P , then performing a rightward **arithmetic shift** on P . Let m and r be the multiplicand and multiplier, respectively; and let x and y represent the number of bits in m and r .
- Determine the values of A and S , and the initial value of P . All of these numbers should have a length equal to $(x + y + 1)$.
 - A: Fill the most significant (leftmost) bits with the value of m . Fill the remaining $(y + 1)$ bits with zeros.
 - S: Fill the most significant bits with the value of $(-m)$ in two's complement notation. Fill the remaining $(y + 1)$ bits with zeros.
 - P: Fill the most significant x bits with zeros. To the right of this, append the value of r . Fill the least significant (rightmost) bit with a zero.
- Determine the two least significant (rightmost) bits of P .
 - If they are 01, find the value of $P + A$. Ignore any overflow.
 - If they are 10, find the value of $P + S$. Ignore any overflow.
 - If they are 00, do nothing. Use P directly in the next step.
 - If they are 11, do nothing. Use P directly in the next step.
- **Arithmetically shift** the value obtained in the 2nd step by a single place to the right. Let P now equal this new value.
- Repeat steps 2 and 3 until they have been done y times.
- Drop the least significant (rightmost) bit from P . This is the product of m and r .

Example 1

Find 3×-4 , with $m = 3$ and $r = -4$, and $x = 4$ and $y = 4$:


$A = 0011\ 0000\ 0$

$S = 1101\ 0000\ 0$

$P = 0000\ 1100\ 0$

Perform the loop four times :


 $P = 0000\ 1100\ 0$. The last two bits are 00.


 $P = 0000\ 0110\ 0$. Arithmetic right shift.

 $P = 0000\ 0110\ 0$. The last two bits are 00.


 $P = 0000\ 0011\ 0$. Arithmetic right shift.

 $P = 0000\ 0011\ 0$. The last two bits are 10.

 $P = 1101\ 0011\ 0$. $P = P + S$.

 $P = 1110\ 1001\ 1$. Arithmetic right shift.

 $P = 1110\ 1001\ 1$. The last two bits are 11.

 $P = 1111\ 0100\ 1$. Arithmetic right shift.

 The product is 1111 0100, which is -12 .

Example2

The above mentioned technique is inadequate when the multiplicand is the largest negative number that can be represented (i.e. if the multiplicand has 8 bits then this value is -128). One possible correction to this problem is to add one more bit to the left of A, S and P. Below, we demonstrate the improved technique by multiplying -8 by 2 using 4 bits for the multiplicand and the multiplier:

A = 1 1000 0000 0

S = 0 1000 0000 0

P = 0 0000 0010 0

Perform the loop four times :

P = 0 0000 0010 0. The last two bits are 00.

P = 0 0000 0001 0. Right shift.

P = 0 0000 0001 0. The last two bits are 10.

P = 0 1000 0001 0. $P = P + S$.

P = 0 0100 0000 1. Right shift.

P = 0 0100 0000 1. The last two bits are 01.

P = 1 1100 0000 1. $P = P + A$.

P = 1 1110 0000 0. Right shift.

P = 1 1110 0000 0. The last two bits are 00.

P = 0 1111 0000 0. Right shift.

The product is 11110000 (after discarding the first and the last bit) which is -16 .

Booth's multiplication algorithm

Procedure

If x is the count of bits of the multiplicand, and y is the count of bits of the multiplier :

Draw a grid of three lines, each with squares for $x + y + 1$ bits. Label the lines respectively A (add), S

(subtract), and P (product).

In two's complement notation, fill the first x bits of each line with :

A: the multiplicand

S: the negative of the multiplicand

P: zeroes

Fill the next y bits of each line with :

A: zeroes

S: zeroes

P: the multiplier

Fill the last bit of each line with a zero.

Do both of these steps y times :

1. If the last two bits in the product are...

00 or 11: do nothing.

01: $P = P + A$. Ignore any overflow.

10: $P = P + S$. Ignore any overflow.

2. Arithmetically shift the product right one position.

Drop the last bit from the product for the final result.

Example

Find 3×-4 :

A = 0011 0000 0

S = 1101 0000 0

P = 0000 1100 0

Perform the loop four times :

P = 0000 1100 0. The last two bits are 00.

P = 0000 0110 0. A right shift.

P = 0000 0110 0. The last two bits are 00.

P = 0000 0011 0. A right shift.

P = 0000 0011 0. The last two bits are 10.

P = 1101 0011 0. P = P + S.

P = 1110 1001 1. A right shift.

P = 1110 1001 1. The last two bits are 11.

P = 1111 0100 1. A right shift.

The product is 1111 0100, which is -12.