3.3 Division

> "Divide et impera.
> *Latin for "Divide and rule," ancient political maxim cited by Machiavelli, 1532*

The reciprocal operation of multiply is divide, an operation that is even less frequent and even quirkier. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0.

Let's start with an example of long division using decimal numbers to recall the names of the operands and the division algorithm from grammar school. For reasons similar to those in the previous section, we limit the decimal digits to just 0 or 1. The following animation illustrates dividing $1{,}001{,}010_{ten}$ by 1000_{ten}.

PARTICIPATION ACTIVITY 3.3.1: Division example.

Start ☐ 2x speed

$$\begin{array}{rrl} & 1\,0\,0\,1_{ten} & \text{Quotient} \\ \text{Divisor}\ 1\,0\,0\,0_{ten} &) \, 1\,0\,0\,1\,0\,1\,0_{ten} & \text{Dividend} \\ & -\,1\,0\,0\,0 & \\ & \quad 1\,0 & \\ & \quad 1\,0\,1 & \\ & \quad 1\,0\,1\,0 & \\ & \quad -\,1\,0\,0\,0 & \\ & \quad\quad 1\,0_{ten} & \text{Remainder} \end{array}$$

Divide's two operands, called the **dividend** and **divisor**, and the result, called the **quotient**, are accompanied by a second result, called the **remainder**. Here is another way to express the relationship between the components:

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

where the remainder is smaller than the divisor. Infrequently, programs use the divide instruction just to get the remainder, ignoring the quotient.

Dividend: A number being divided.

Divisor: A number that the dividend is divided by.

Quotient: The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

Remainder: The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

The basic division algorithm from grammar school tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt. Our carefully selected decimal example uses just the numbers 0 and 1, so it's easy to figure out how many times the divisor goes into the portion of the dividend: it's either 0 times or 1 time. Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division.

PARTICIPATION ACTIVITY 3.3.2: Binary division.

Consider $103_{ten} \div 11_{ten}$, or $1100111_{two} \div 1011_{two}$. Fill in the missing values.

1) The value placed in the quotient is ____.

$$1011\overline{)1100111}^{\,?}$$

- ○ 0
- ○ 1

2) The value placed in the quotient is ____.

$$1011\overline{)1100111}^{\,?}$$

- ○ 0
- ○ 1

3) The result of the subtraction is ____.

$$\begin{array}{l} \quad\quad 1 \\ 1011\overline{)1100111} \\ \underline{-\,1011} \\ \quad ? \end{array}$$

- ○

0

1

4) The value placed in the quotient is ____.

```
      1???
1011)1100111
   -1011
      1111
```

111

001

5) The remainder is ____.

```
      1001
1011)1100111
   -1011
      1111
    -1011
       ???
```

000

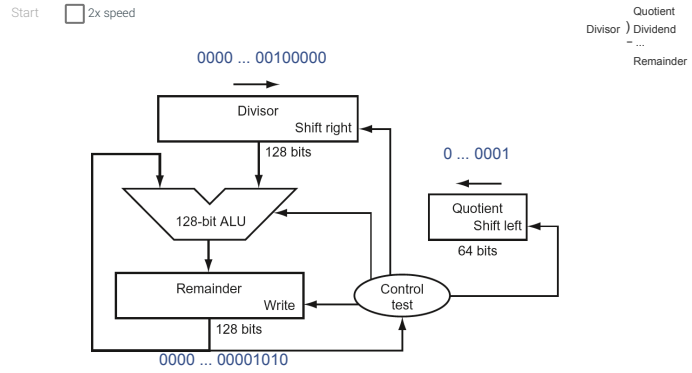
100

Let's assume that both the dividend and the divisor are positive and hence the quotient and the remainder are nonnegative. The division operands and both results are 64-bit values, and we will ignore the sign for now.

A division algorithm and hardware

The figure below shows hardware to mimic our grammar school algorithm. We start with the 64-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 128-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

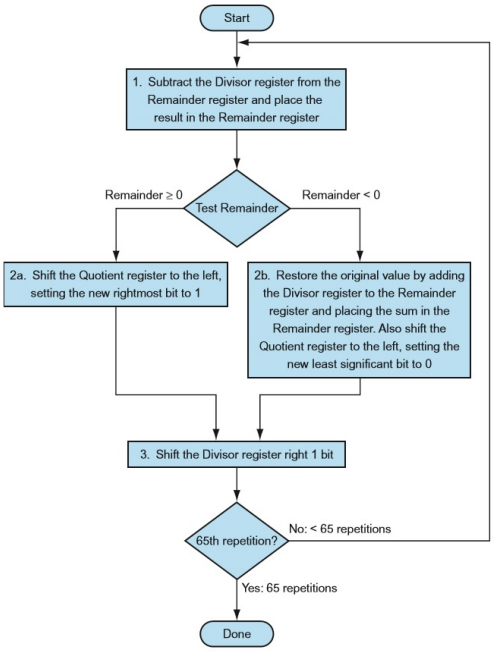
PARTICIPATION ACTIVITY 3.3.3: First version of the division hardware for 64-bit divisor and dividend (COD Figure 3.8).



The figure below shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed comparison. If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right, and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations complete.

Figure 3.3.1: A division algorithm, using the hardware in the above figure (COD Figure 3.9).

If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 65 times.



PARTICIPATION ACTIVITY 3.3.4: Division algorithm and hardware.

1) Each step of the division algorithm shifts the Divisor register 1 bit to the ____.
   - ○ right
   - ○ left

2) If the Remainder is negative, a ____ is shifted into the least significant bit of the Quotient register.
   - ○ 0
   - ○ 1

3) If the Remainder is negative, then the original Remainder value is restored by adding ____ to the Remainder.
   - ○ Control
   - ○ the Divisor

4) Each iteration of the division algorithm consists of ____ basic steps.
   - ○ 3
   - ○ 8
   - ○ 65

Example 3.3.1: A divide algorithm.

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

Answer

The figure below shows the value of each register for each of the steps, with the quotient being 3_{ten} and the remainder 1_{ten}. Notice that the test in step 2 of whether the remainder is positive or negative simply checks whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes $n + 1$ steps to get the proper quotient and remainder.

Figure 3.3.2: Division example using the algorithm in the above figure (COD Figure 3.10).

The bit examined to determine the next step is circled in color.

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, LSL 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, LSL 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, LSL 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 ⟹ LSL 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 ⟹ LSL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

PARTICIPATION ACTIVITY 3.3.5: Divide example using the division algorithm.

Consider the table in the above figure.

1) The initial 8-bit value of Divisor is ____.

Check Show answer

2) The initial 8-bit value of Remainder is ____.

Check Show answer

3) Iteration 1, step 1 subtracts the Divisor from the Remainder and places the result in the Remainder. The Remainder's updated value is ____.

Check Show answer

4) Iteration 1, step 2b restores the Remainder's value to ____.

Check Show answer

5) At the end of iteration 4, the 4-bit value of Quotient is ____.

Check Show answer

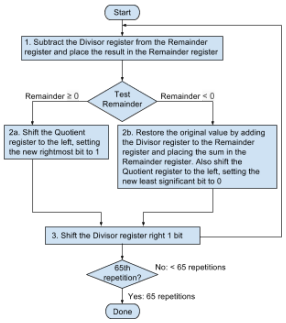
6) The divide operation results in a 4-bit Quotient of 0011 and an 8-bit Remainder of ____.

Check Show answer

PARTICIPATION ACTIVITY 3.3.6: Division algorithm steps and register values.

Consider the divison of $13_{10} \div 4_{10}$, or $1101_2 \div 0100_2$. Fill in the missing values for each of the steps labeled according to COD Figure 3.9 (A division algorithm ...). A copy of the division algorithm figure is shown below to the right.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0100 0000	0000 1101
1	1: Rem = Rem - Div	0000	0100 0000	1100 1101
	(a)	0000	0100 0000	(b)
	3: Shift Div right	0000	0010 0000	0000 1101
2	1: Rem = Rem - Div	0000	0010 0000	1110 1101
	2b: Rem < 0 ⇒ +Div, LSL Q, Q0 = 0	0000	0010 0000	0000 1101
	3: Shift Div right	(c)	0001 0000	0000 1101
3	1: Rem = Rem - Div	0000	0001 0000	1111 1101
	2b: Rem < 0 ⇒ +Div, LSL Q, Q0 = 0	0000	0001 0000	0000 1101
	3: Shift Div right	0000	(d)	0000 1101
4	1: Rem = Rem - Div	0000	0000 1000	0000 0101
	(e)	(f)	0000 1000	0000 0101
	3: 3: Shift Div right	0001	0000 0100	0000 0101
5	1: Rem = Rem - Div	0001	0000 0100	0000 0001
	2a: Rem ≥ 0 ⇒ LSL Q, Q0 = 1	0011	0000 0100	0000 0001
	3: 3: Shift Div right	0011	0000 0010	0000 0001



2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1 0000 1101 2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0

0000 1000 0001 0000

(a)

(b)

(c)

(d)

(e)

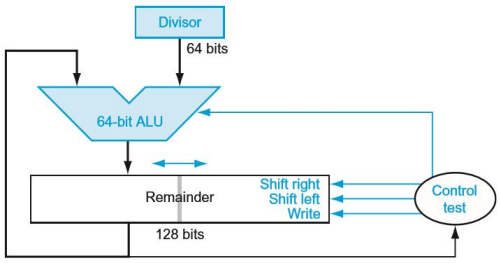
(f)

Reset

This algorithm and hardware can be refined to be faster and cheaper. The speedup comes from shifting the operands and the quotient simultaneously with the subtraction. This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders. The figure below shows the revised hardware.

Figure 3.3.3: An improved version of the division hardware (COD Figure 3.11).

The Divisor register, ALU, and Quotient register are all 64 bits wide, with only the Remainder register left at 128 bits. Compared to COD Figure 3.8 (First version of the division hardware), the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in COD Figure 3.5 (Refined version of the multiplication hardware), the Remainder register should really be 129 bits to make sure the carry out of the adder is not lost.)



PARTICIPATION ACTIVITY 3.3.7: Refined division hardware.

Refer to the improved version of the division hardware (COD Figure 3.11).

1) The improved version of the division hardware does not require a quotient to perform a divide operation.
   - True
   - False

2) The improved version of the division hardware halves the width of the adder and divisor.
   - True
   - False

3) The speedup comes from reducing the size of the registers and ALU.
   - True
   - False

Signed division

So far, we have ignored signed numbers in division. The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.

Elaboration

The one complication of signed division is that we must also set the sign of the remainder. Remembe must always hold:

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

To understand how to set the sign of the remainder, let's look at the example of dividing all the combi first case is easy:

$$+7 \div +2 : \text{Quotient} = +3, \text{ Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$$-7 \div +2 : \text{Quotient} = -3$$

Rewriting our basic formula to calculate the remainder:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (- \\ &= -7 - (-6) = -1\end{aligned}$$

So,

$$-7 \div +2 : \text{Quotient} = -3, \text{Remainder} = -1$$

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

The reason the answer isn't a quotient of −4 and a remainder of +1, which would also fit this formula, quotient would then change depending on the sign of the dividend and the divisor! Clearly, if

$$-(x \div y) \neq (-x) \div y$$

programming would be an even greater challenge. This anomalous behavior is avoided by following t remainder must have identical signs, no matter what the signs of the divisor and quotient.

We calculate the other combinations by following the same rule:

$$+7 \div -2 : \text{Quotient} = -3, \text{Remainder} = +1$$
$$-7 \div -2 : \text{Quotient} = +3, \text{Remainder} = -1$$

Thus, the correctly signed division algorithm negates the quotient if the signs of the operands are op nonzero remainder match the dividend.

Faster division

Moore's Law applies to division hardware as well as multiplication, so we would like to be able to speed up division by throwing hardware at it. We used many adders to speed up multiply, but we cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 64 partial products immediately.

There are techniques to produce more than one bit of the quotient per step. The *SRT division* technique tries to **predict** several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. It relies on subsequent steps to correct wrong predictions. A typical value today is 4 bits. The key is guessing the value to subtract. With binary division, there is only a single choice. These algorithms use 6 bits from the remainder and 4 bits from the divisor to index a table that determines the guess for each step.

The accuracy of this fast method depends on having proper values in the lookup table. The *Fallacy* in COD Section 3.10 (Fallacies and pitfalls) shows what can happen if the table is incorrect.

Divide in LEGv8

You may have already observed that the same sequential hardware can be used for both multiply and divide in COD Figures 3.5 (Refined version of the multiplication hardware) and 3.11 (An improved version of the division hardware). The only requirement is a 128-bit register that can shift left or right and a 64-bit ALU that adds or subtracts.

To handle both signed integers and unsigned integers, LEGv8 has two instructions: *signed divide* (`SDIV`) and *divide unsigned* (`UDIV`).

Summary

The common hardware support for multiply and divide allows LEGv8 to provide a single pair of 64-bit registers that are used both for multiply and divide. We accelerate division by predicting multiple quotient bits and then correcting mispredictions later, the figure below summarizes the enhancements to the LEGv8 architecture for COD Section 3.3 (Multiplication) and COD Section 3.4 (Division).

Figure 3.3.4: LEGv8 core architecture (COD Figure 3.12).

LEGv8 assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD X1, X2, X3	X1 = X2 + X3	Three register operands
	subtract	SUB X1, X2, X3	X1 = X2 - X3	Three register operands
	add immediate	ADDI X1, X2, 20	X1 = X2 + 20	Used to add constants
	subtract immediate	SUBI X1, X2, 20	X1 = X2 - 20	Used to subtract constants
	add and set flags	ADDS X1, X2, X3	X1 = X2 + X3	Add, set condition codes
	subtract and set flags	SUBS X1, X2, X3	X1 = X2 - X3	Subtract, set condition codes
	add immediate and set flags	ADDIS X1, X2, 20	X1 = X2 + 20	Add constant, set condition codes
	subtract immediate and set flags	SUBIS X1, X2, 20	X1 = X2 - 20	Subtract constant, set condition codes
	multiply	MUL X1, X2, X3	X1 = X2 × X3	Lower 64-bits of 128-bit product
	signed multiply high	SMULH X1, X2, X3	X1 = X2 × X3	Upper 64-bits of 128-bit signed product
	unsigned multiply high	UMULH X1, X2, X3	X1 = X2 × X3	Upper 64-bits of 128-bit unsigned product
	signed divide	SDIV X1, X2, X3	X1 = X2 / X3	Divide, treating operands as signed
	unsigned divide	UDIV X1, X2, X3	X1 = X2 / X3	Divide, treating operands as unsigned
Data transfer	load register	LDUR X1, [X2,40]	X1 = Memory[X2 + 40]	Doubleword from memory to register
	store register	STUR X1, [X2,40]	Memory[X2 + 40] = X1	Doubleword from register to memory
	load signed word	LDURSW X1, [X2,40]	X1 = Memory[X2 + 40]	Word from memory to register
	store word	STURW X1, [X2,40]	Memory[X2 + 40] = X1	Word from register to memory
	load half	LDURH X1, [X2,40]	X1 = Memory[X2 + 40]	Halfword memory to register
	store half	STURH X1, [X2,40]	Memory[X2 + 40] = X1	Halfword register to memory
	load byte	LDURB X1, [X2,40]	X1 = Memory[X2 + 40]	Byte from memory to register
	store byte	STURB X1, [X2,40]	Memory[X2 + 40] = X1	Byte from register to memory
	load exclusive register	LDXR X1, [X2,0]	X1 = Memory[X2]	Load; 1st half of atomic swap
	store exclusive register	STXR X1, X3, [X2]	Memory[X2]=X1,X3=0 or 1	Store; 2nd half of atomic swap
	move wide with zero	MOVZ X1,20	X1 = 20 or 20 * 2^{16} or 20 * 2^{32} or 20 * 2^{48}	Loads 16-bit constant, rest zeros
	move wide with keep	MOVK X1,20	X1 = 20 or 20 * 2^{16} or 20 * 2^{32} or 20 * 2^{48}	Loads 16-bit constant, rest unchanged
Logical	and	AND X1, X2, X3	X1 = X2 & X3	Three reg. operands; bit-by-bit AND
	inclusive or	ORR X1, X2, X3	X1 = X2 | X3	Three reg. operands; bit-by-bit OR
	exclusive or	EOR X1, X2, X3	X1 = X2 ^ X3	Three reg. operands; bit-by-bit XOR
	and immediate	ANDI X1, X2, 20	X1 = X2 & 20	Bit-by-bit AND reg with constant
	inclusive or immediate	ORRI X1, X2, 20	X1 = X2 | 20	Bit-by-bit OR reg with constant
	exclusive or immediate	EORI X1, X2, 20	X1 = X2 ^ 20	Bit-by-bit XOR reg with constant
	logical shift left	LSL X1, X2, 10	X1 = X2 << 10	Shift left by constant
	logical shift right	LSR X1, X2, 10	X1 = X2 >> 10	Shift right by constant
Condi-tional branch	compare and branch on equal 0	CBZ X1, 25	if (X1 == 0) go to PC + 4 + 100	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	CBNZ X1, 25	if (X1!= 0) go to PC + 4 + 100	Not equal 0 test; PC-relative
	branch conditionally	B.cond 25	if (condition true) go to PC + 4 + 100	Test condition codes; if true, branch
Uncondi-tional jump	branch	B 2500	go to PC + 4 + 10000	Branch to target address; PC-relative
	branch to register	BR X30	go to X30	For switch, procedure return
	branch with link	BL 2500	X30 = PC + 4; PC + 4 + 10000	For procedure call PC-relative

Hardware/Software Interface

LEGv8 divide instructions ignore overflow, so software must determine whether the quotient is too large. In addition to overflow, division can also result in an improper calculation: division by 0. Some computers distinguish these two anomalous events. LEGv8 software must check the divisor to discover division by 0 as well as overflow.

Elaboration

An even faster algorithm does not immediately add the divisor back if the remainder is negative. It simply adds the dividend to the shifted remainder in the following step, since (r + d) × 2 - d = r - 2 + d × 2 - d = r × 2 + d. This nonrestoring division algorithm, which takes one clock cycle per step, is explored further in the exercises; the algorithm above is called restoring division. A third algorithm that doesn't save the result of the subtract if it's negative is called a nonperforming division algorithm. It averages one-third fewer arithmetic operations.

PARTICIPATION ACTIVITY 3.3.8: LEGv8 multiply and divide instructions.

1) Goal: X6 × X7 (signed multiply high)

`____ X5, X6, X7`

Check **Show answer**

2) Goal: X6 / X7 (unsigned division)

`____ X5, X6, X7`

Check **Show answer**

PARTICIPATION ACTIVITY 3.3.9: LEGv8 division.

1) The division hardware supports signed division.
   - ○ True
   - ○ False

2) Faster divison, like faster multiplication, can be achieved by increasing the number of ALUs.
   - ○ True
   - ○ False

3) The divide operations, `SDIV` and `UDIV`, detect overflow and division by 0.
   - ○ True
   - ○ False

Provide feedback on this section