Chapter 3

Arithmetic for Computers

Introduction

- Integers can be represented either in decimal or binary form
- What about fractions and other real numbers?
- What happens if an operation creates a number bigger than can be represented?
- How does hardware really multiply or divide numbers?

Addition and Subtraction

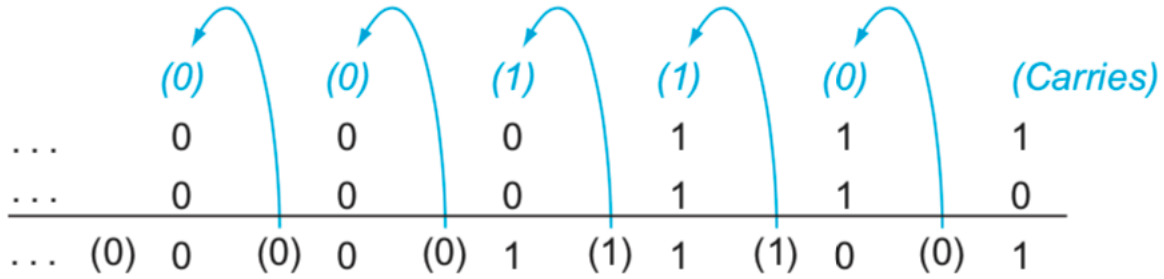
- Addition in computers: Digits are added bit by bit from right to left, with carries passed to the next digit to the left

- Subtraction: use addition; the appropriate operand is simply negated before being added

Addition and Subtraction

Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
+ & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} = 13_{ten}
\end{array}
$$



Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

Addition and Subtraction

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
- & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}
\end{array}
$$

or via addition using the two's complement representation of -6:

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
+ & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}
\end{array}
$$

Addition and Subtraction

- **Overflow occurs** when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word

- **When can overflow occur in addition**
- No overflow can occur when adding positive and negative operands.
- Because the operands and the sum fit in 32 bits
- For example, -10 + 4=- 6.

Addition and Subtraction

- **Overflow occurs** when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word

- **When can overflow occur in subtraction**
- when the signs of the operands are the *same*, overflow cannot occur
- remember that *c* - *a*= *c* + (-*a*) because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign *we end up by adding operands of different signs*

Addition and Subtraction

- Adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully expressed.
- The lack of a 33rd bit means that when overflow occurs

- **Overflow occurs when adding two positive numbers and the sum is negative, or vice versa.**

Addition and Subtraction

- Adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully expressed.
- The lack of a 33rd bit means that when overflow occurs

- **Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result.**

Addition and Subtraction

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

Overflow conditions for addition and subtraction.

Addition and Subtraction

- **What about overflow with unsigned integers?** Unsigned integers are commonly used for memory address where overflow are ignored. The computer designer must provide a way to ignore overflow in some cases and recognize in other cases

- The MIPS solution: two arithmetic instructions to recognize the two choices

  - Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
  - Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Addition and Subtraction

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions `addu`, `addiu`, and `subu`, no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

Addition and Subtraction

- **How to handle overflow?** MIPS detects overflow with an exception, also called an interruption.
- **Exception or interruption is an unscheduled procedure call**

1. The address of the instruction that overflowed is saved in a register
2. The computer jumps to a predefined address to invoke the appropriate routine for that exception.
3. The interrupted address is saved, so that in some situations that program can continue after corrective code is executed

Addition and Subtraction

- MIPS includes a register called the *exception program counter* (EPC) to contain the address of the instruction that caused the exception.
- The instruction *move from system control* (mfc0) is used to **copy EPC into a general-purpose register** so that MIPS software has the option of returning to the offending instruction via a jump register instruction.
- .

Multiplication

- The first operand is called the *multiplicand* and the second the *multiplier*. The final result is called the *product*.
- **Multiplication steps :**

1. **Take the digits of the multiplier one at a time from right to left,**
2. **Multiply the multiplicand by the single digit of the multiplier,**
3. **Shifting the intermediate product one digit to the left of the earlier intermediate products.**

Multiplicand		1000_{ten}
Multiplier	x	1001_{ten}
		1000
		0000
		0000
		1000
Product		1001000_{ten}

Multiplication

- The observation of multiplication
- The number of digits in the product is considerably larger than the number in either the multiplicand or the multiplier.

- In fact, if we ignore the sign bits, the length of the multiplication of **an *n*-bit multiplicand** and **an *m*-bit multiplier** is **a product that is *n* + *m* bits long**. That is, *n* + *m* bits are required to represent all possible products.

- Hence, like add, multiply must cope with overflow because **we frequently want a 32-bit product as the result of multiplying two 32-bit numbers**.

Multiplication

- In the previous example, we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

1. Just place a copy of the multiplicand (1* multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 (0* multiplicand) in the proper place if the digit is 0.

Multiplication

- let's assume that the multiplier is in the 32 bit multiplier register and **the 64 bits product register is initialized to 0.**
- We need to move the multiplicand left one digit each step. Over 32 steps, a 32 bit multiplicand would move 32 bits to left
- Therefore, we need a 64 bit multiplicand register, **initialized with the 32 multiplicand in the right half and zero in the left half**

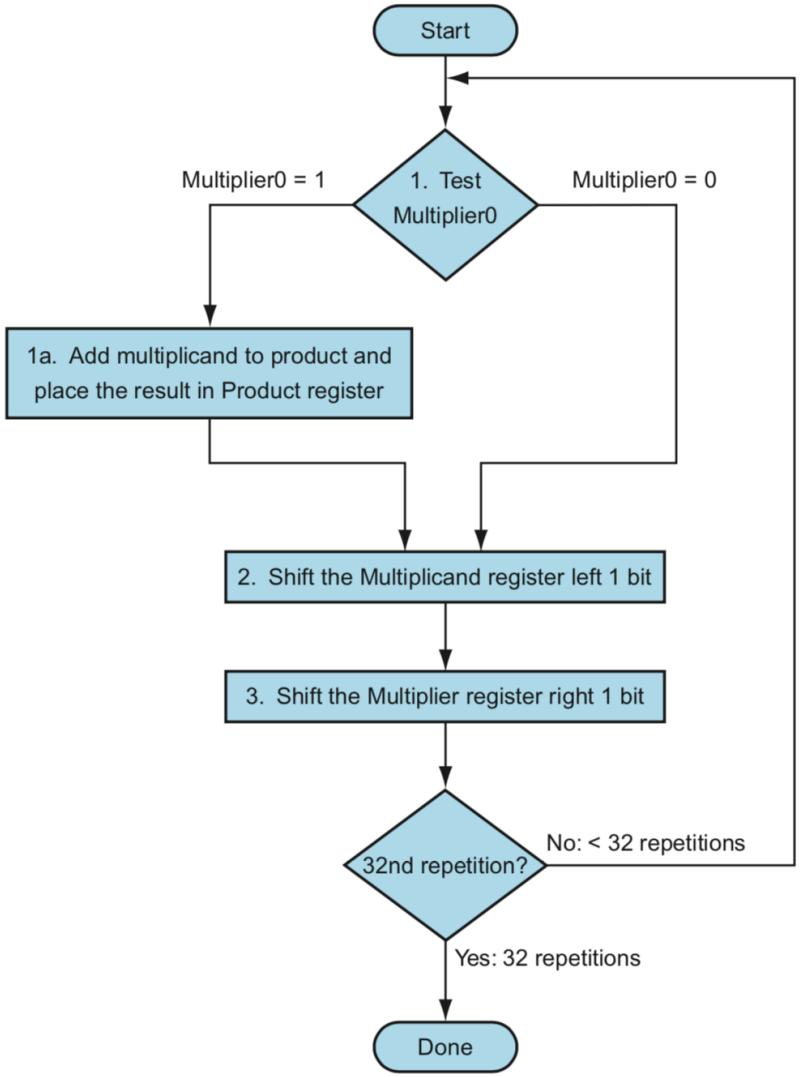
Multiplication

- Three steps needed for each bit

1. The least significant bit of multiplier determines if the multiplicand is added to the product register. **If the least significant bit is 1**, We need to **add multiplicand to product and place the result in Product register**. Otherwise go to step 2
2. **Shift the Multiplicand register left 1 bit**
3. **Shift the Multiplier register right 1 bit**

- These steps are repeated 32 times to obtain the product

Multiplication



Multiplication

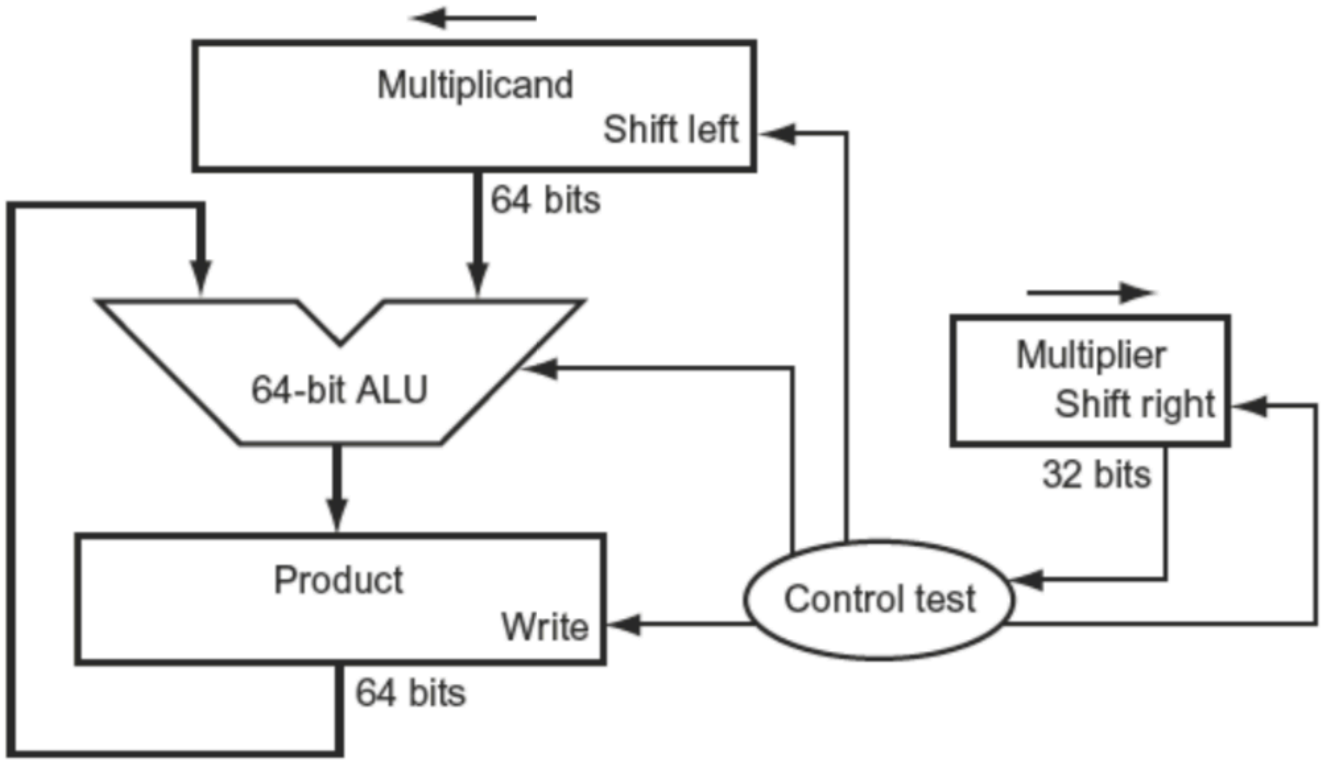


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Multiplication

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

Signed Multiplication

- The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember the original signs
- The algorithms should then be run for 31 iterations, leaving the signs out of the calculation

Division

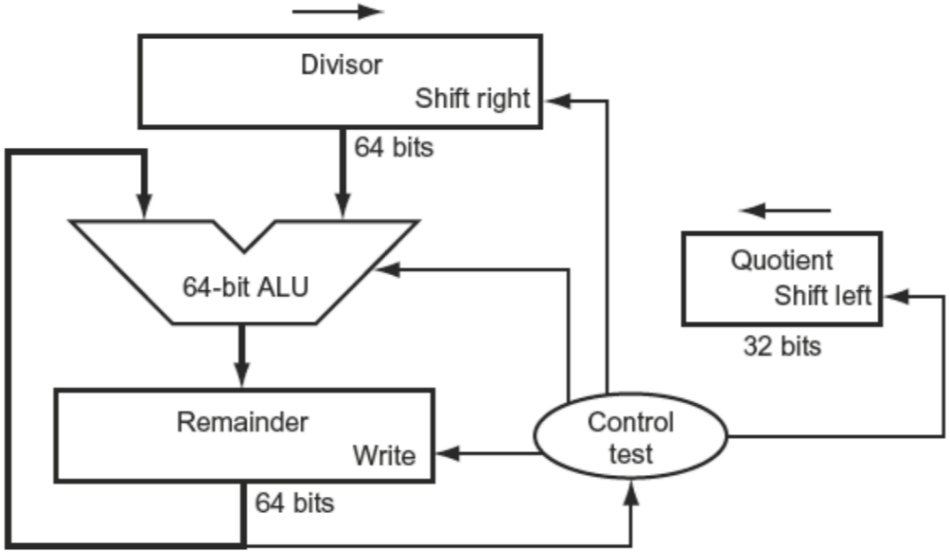
The example is dividing $1{,}001{,}010_{ten}$ by 1000_{ten}:

$$
\begin{array}{rll}
 & 1001_{ten} & \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)\,1001010_{ten}} & \text{Dividend} \\
 & \underline{-1000} & \\
 & 10 & \\
 & 101 & \\
 & 1010 & \\
 & \underline{-1000} & \\
 & 10_{ten} & \text{Remainder}
\end{array}
$$

- 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:

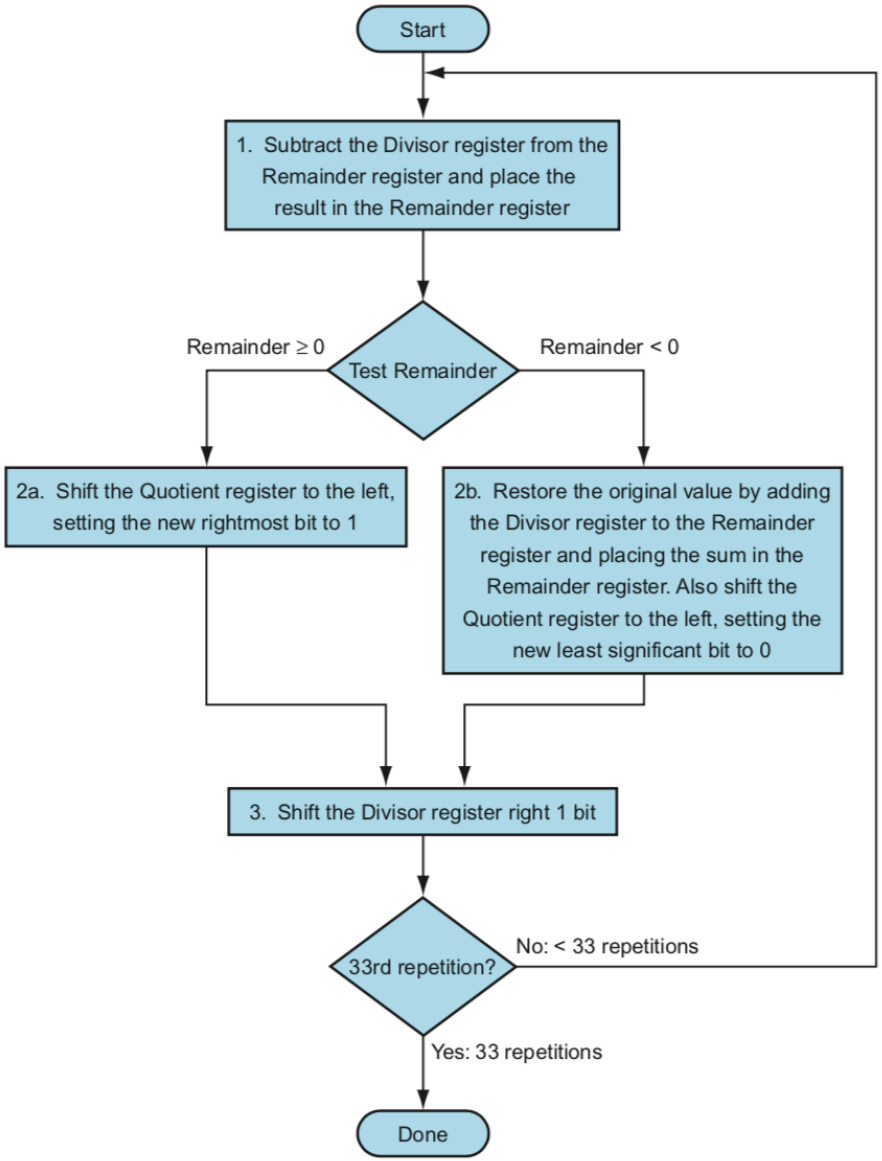
$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

A Division Algorithm and Hardware



- We start with the 32-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 64-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.

A Division Algorithm and Hardware



A Division Algorithm and Hardware

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}.

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, sll 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, sll 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, sll 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 ⇒ sll 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 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Signed Division

- Remember the signs of the divisor and dividend and then negate the quotient if the sign disagree

Reading assignment

- Read 3.1, 3.2, 3.3, and 3.4 of the textbook
- + Appendix B5