

Computer Architecture

Arithmetic for Computers



Unsigned Numbers

- For an N bit system, unsigned numbers are represented from 0 to $2^N - 1$

0000 0000 0000 0000 0000 0000 0000 0000	$_{\text{two}} = 0_{\text{ten}}$
0000 0000 0000 0000 0000 0000 0000 0001	$_{\text{two}} = 1_{\text{ten}}$
0000 0000 0000 0000 0000 0000 0000 0010	$_{\text{two}} = 2_{\text{ten}}$
...	
1111 1111 1111 1111 1111 1111 1111 1110	$_{\text{two}} = 4,294,967,293_{\text{ten}}$
1111 1111 1111 1111 1111 1111 1111 1110	$_{\text{two}} = 4,294,967,294_{\text{ten}}$
1111 1111 1111 1111 1111 1111 1111 1111	$_{\text{two}} = 4,294,967,295_{\text{ten}}$

2's Complement – Signed Numbers

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = 0_{\text{ten}}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}$$

...

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = 2^{31}-1$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = -2^{31}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = -(2^{31} - 1)$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = -(2^{31} - 2)$$

...

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = -1$$

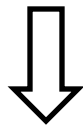
2's Complement – Conversion

- Each number represents the quantity

$$x_{31} - 2^{31} + x_{30} 2^{30} + x_{29} 2^{29} + \dots + x_1 2^1 + x_0 2^0$$

- More conveniently negate each bit and 1 to get the 2's complement.

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 \rightarrow 2^{31} - 1$$



$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 \rightarrow -(2^{31} - 1)$$

Alternative Representations of Negative Numbers

Two's complement is used for signed numbers in every computer today. The following two representations were discarded because they required additional conversion steps before arithmetic could be performed on the numbers

1. sign-and-magnitude: the most significant bit represents +/- and the remaining bits express the magnitude
2. one's complement: $-x$ is represented by inverting all the bits of x

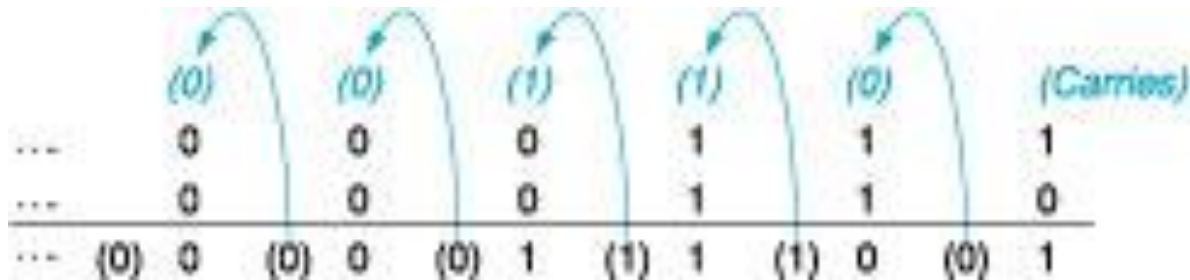
Both representations above suffer from two zeroes

Sign Extension

- In immediate instructions e.g. we have to add a 32 bit number to a 16 bit constant.
- Before performing this operation the constant needs to be converted to 32 bits. This is done by replicating the most significant bit to fill in the additional bits.

Addition and Subtraction

- Addition is similar to decimal arithmetic
- For subtraction, simply add the negative number – hence, subtract $A-B$ involves negating B 's bits, adding 1 and A



Overflows

- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated
- For a signed number, overflow happens
 - when the sum of two positive numbers is a negative result
 - when the sum of two negative numbers is a positive result
 - The sum of a positive and negative number will never overflow
- MIPS allows `addu` and `subu` instructions that work with unsigned integers and never flag an overflow

Multiplication Example

Multiplicand	1000 _{ten}
Multiplier	x 1001 _{ten}

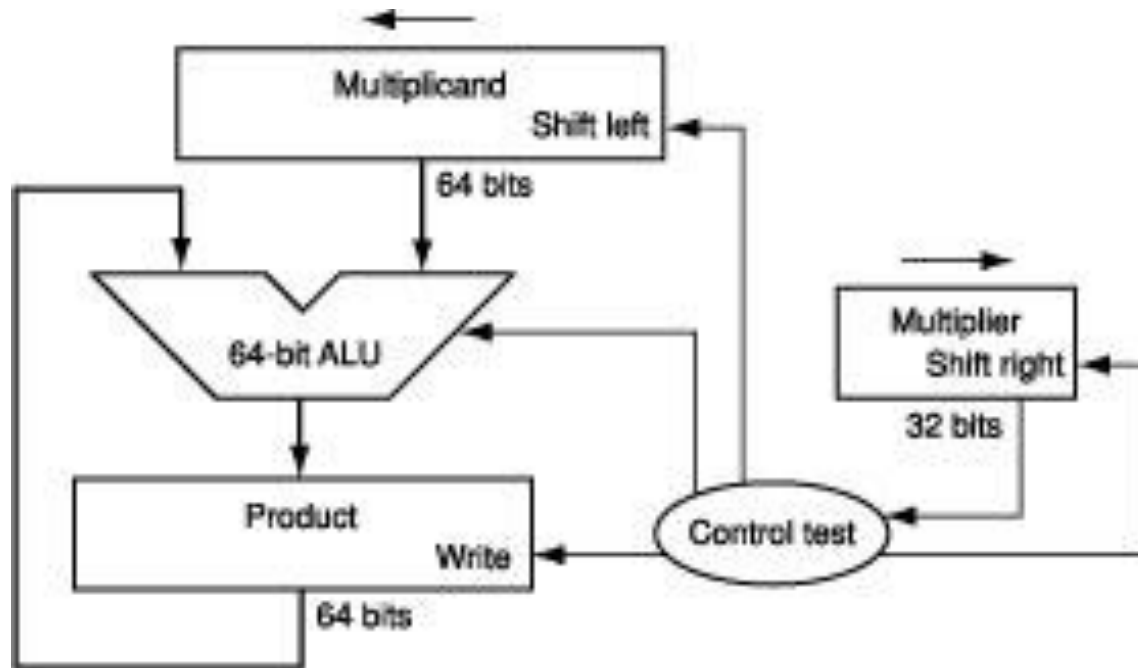
	1000
	0000
	0000
	1000

Product	1001000 _{ten}

In every step

- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product

Multiplication Hardware Algorithm 1



In every step

- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product

Division

		$\frac{1001_{\text{ten}}}{1000_{\text{ten}}}$	Quotient
Divisor	1000_{ten}	1001010_{ten}	Dividend
		$\underline{-1000}$	
		10	
		101	
		1010	
		$\underline{-1000}$	
		10_{ten}	Remainder

At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

Division

		1001 _{two}		Quotient
Divisor	1000 _{two}		1001010 _{two}	Dividend
	01001010		00001010	00001010
	10000000 →	01000000 →	00100000 →	00001000
Quo:	0	01	010	01001

At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

Divide Example

- Divide 7_{ten} ($0000\ 0111_{\text{two}}$) by 2_{ten} (0010_{two})

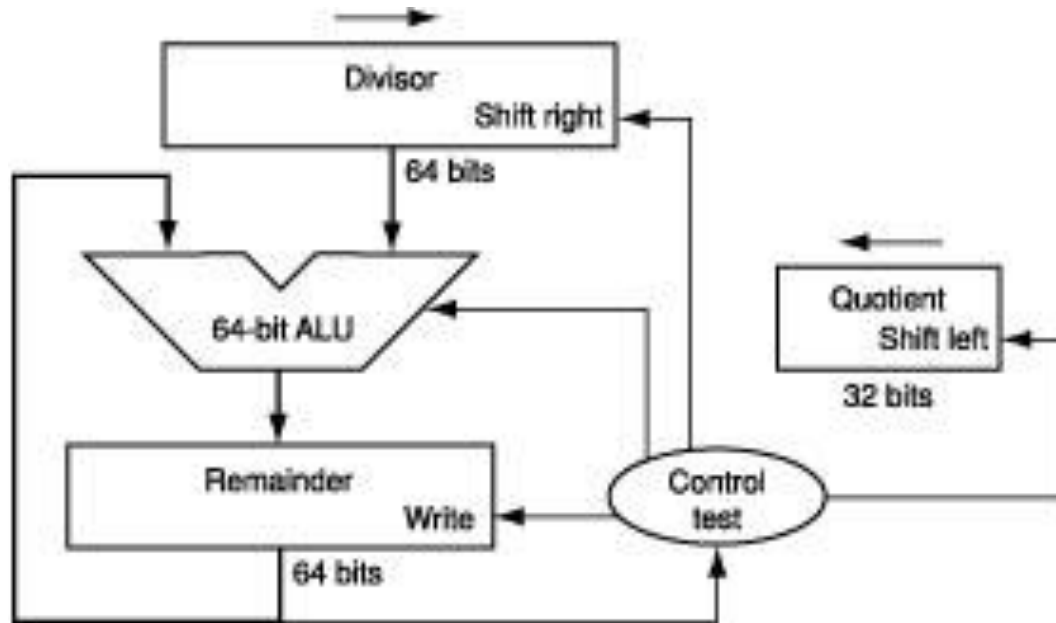
Iter	Step	Quot	Divisor	Remainder
0	Initial values			
1				
2				
3				
4				
5				

Divide Example

- Divide 7_{ten} ($0000\ 0111_{\text{two}}$) by 2_{ten} (0010_{two})

Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 → +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem >= 0 → shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001

Hardware for Division



A comparison requires a subtract; the sign of the result is examined; if the result is negative, the divisor must be added back

Floating Point

- Done in the class