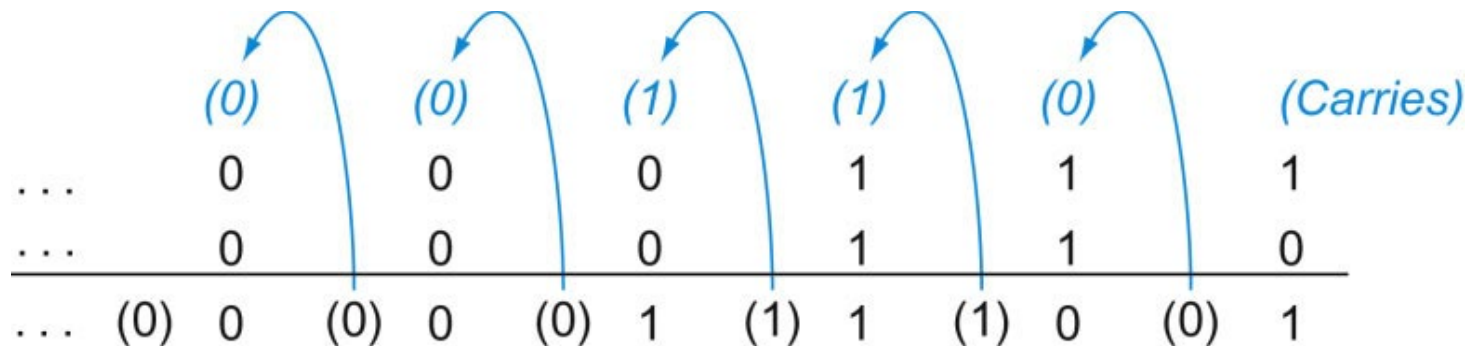


Lecture 9: Addition, Multiplication & Division

- Today's topics:
 - Addition
 - Multiplication
 - Division

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



Source: H&P textbook

Overflows

- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated
- For a signed number, overflow happens when the most significant bit is not the same as every bit to its left
 - 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 – to detect the overflow, other instructions will have to be executed

Multiplication Example

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

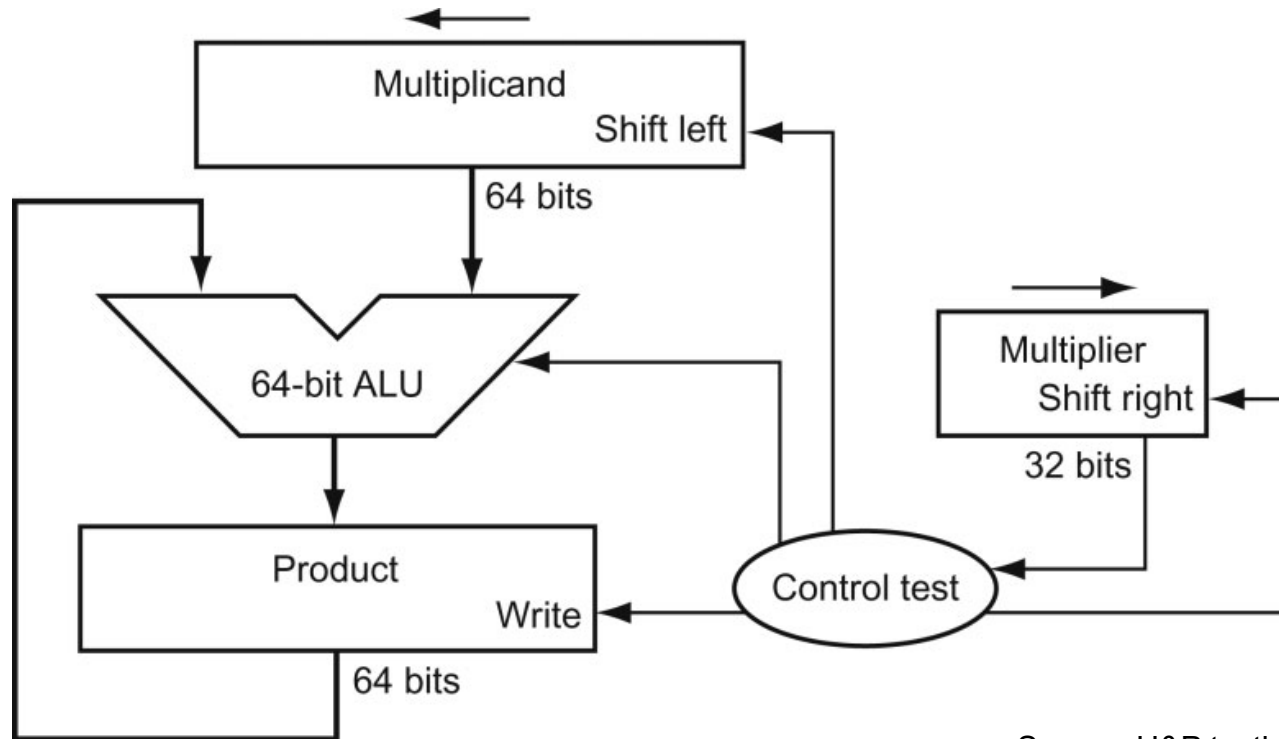
	1000
	0000
	0000
	1000

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

HW Algorithm 1

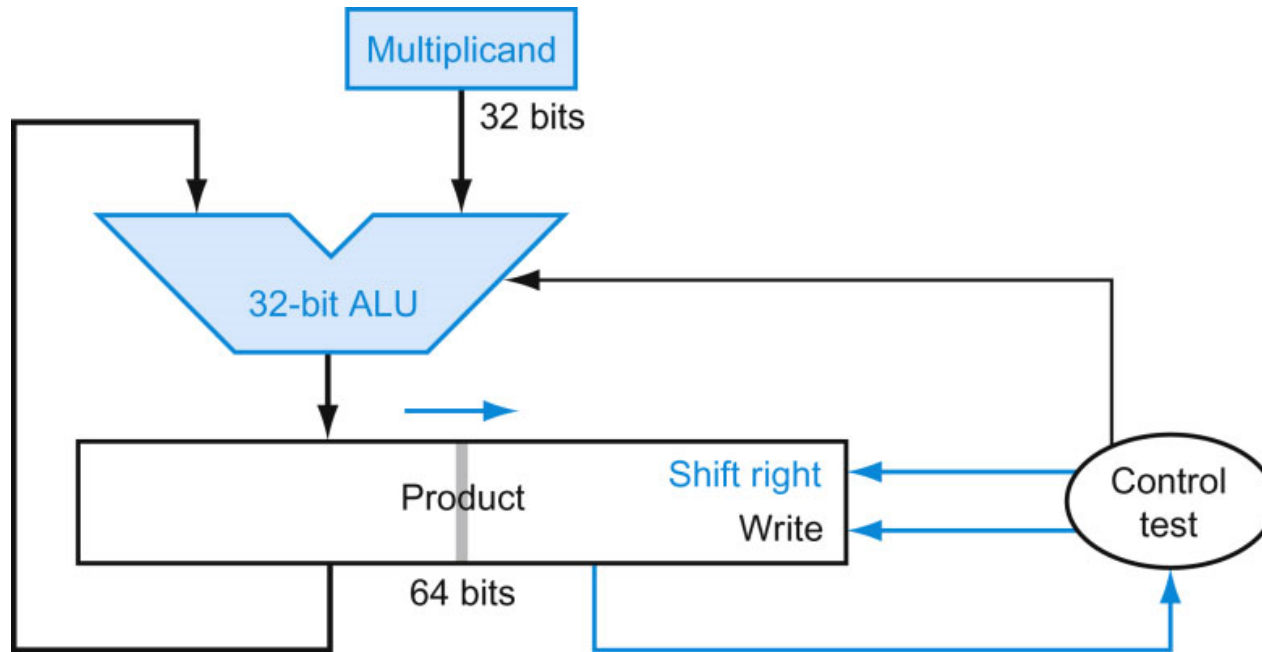


Source: H&P textbook

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

HW Algorithm 2



Source: H&P textbook

- 32-bit ALU and multiplicand is untouched
- the sum keeps shifting right
- at every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register

Notes

- The previous algorithm also works for signed numbers (negative numbers in 2's complement form)
- We can also convert negative numbers to positive, multiply the magnitudes, and convert to negative if signs disagree
- The product of two 32-bit numbers can be a 64-bit number -- hence, in MIPS, the product is saved in two 32-bit registers

MIPS Instructions

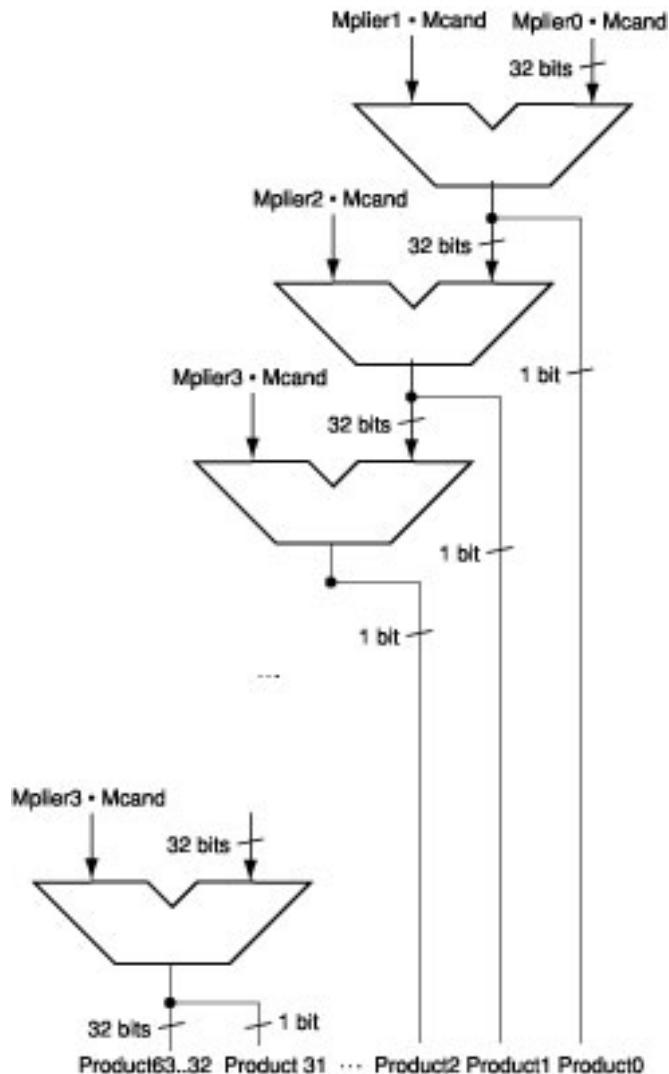
mult \$s2, \$s3 computes the product and stores it in two “internal” registers that can be referred to as **hi** and **lo**

mfhi \$s0 moves the value in **hi** into \$s0

mflo \$s1 moves the value in **lo** into \$s1

Similarly for multu

Fast Algorithm



- The previous algorithm requires a clock to ensure that the earlier addition has completed before shifting
 - This algorithm can quickly set up most inputs – it then has to wait for the result of each add to propagate down – faster because no clock is involved
- Note: high transistor cost

Division

		$\frac{1001_{\text{ten}}}{\phantom{1001010_{\text{ten}}}}$	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 _{ten}		Quotient
Divisor	1000 _{ten}	1001010 _{ten}		Dividend
	0001001010	0001001010	0000001010	0000001010
	100000000000 →	0001000000 →	0000100000 →	0000001000
Quo: 0		000001	0000010	000001001

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