BCS361: Computer Architecture

Arithmetic for Computers

Computer Science

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

2's Complement – Signed Numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} – 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} – 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

2's Complement – Conversion

- Each number represents the quantity

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

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

0111 1111 1111 1111 1111 1111 1111 1111 → $2^{31} - 1$

⇩

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

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

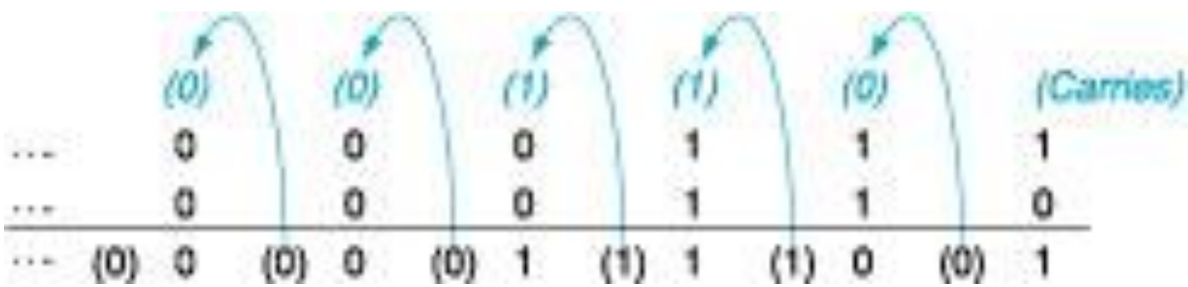
Both representations above suffer from two zeroes

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

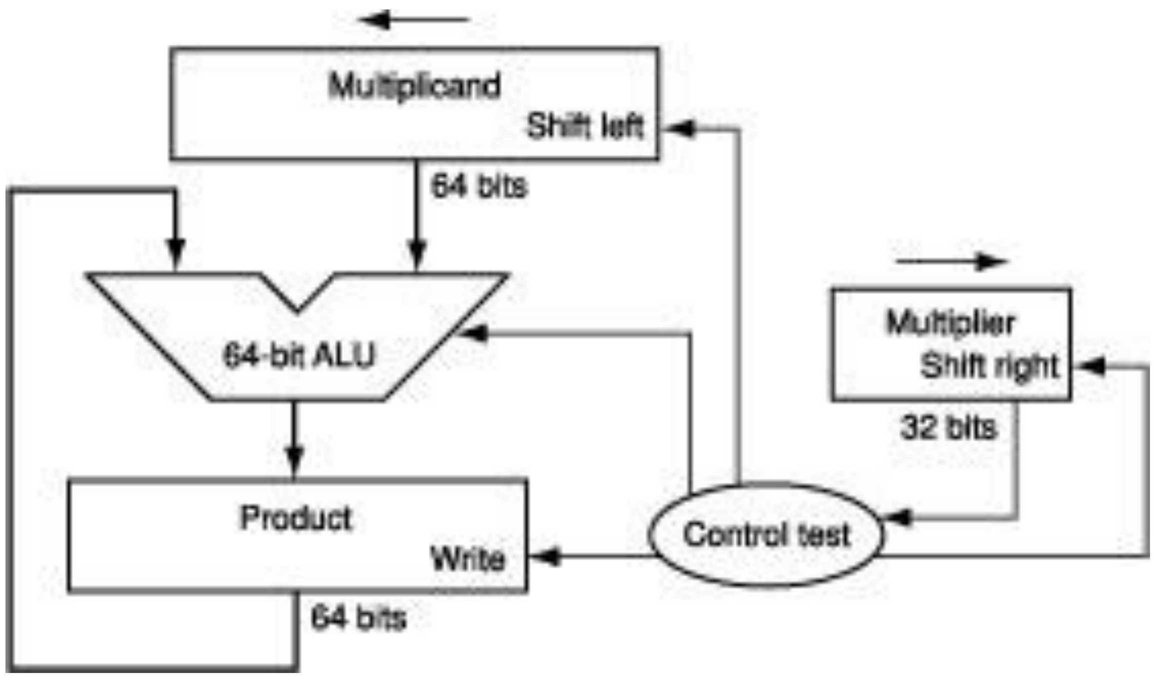
Multiplication Example

```
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

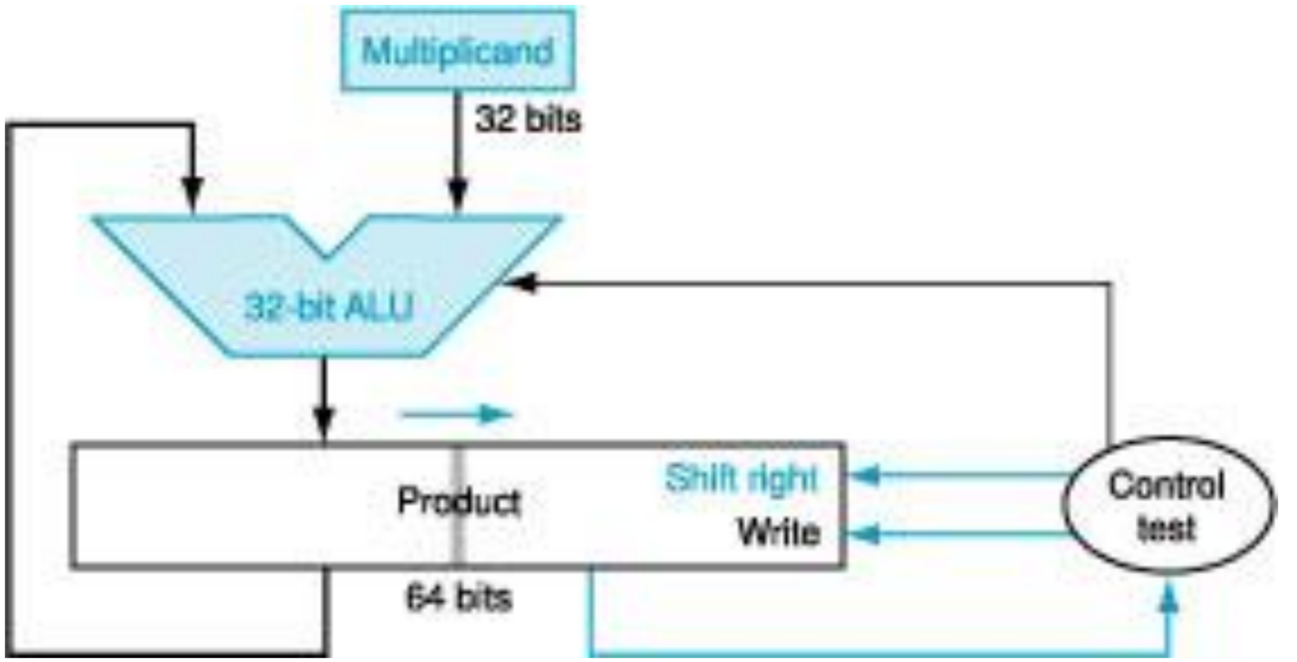
Multiplication Hardware Algorithm 1



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

Multiplication Hardware Algorithm 2



- 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

MIPS Instructions

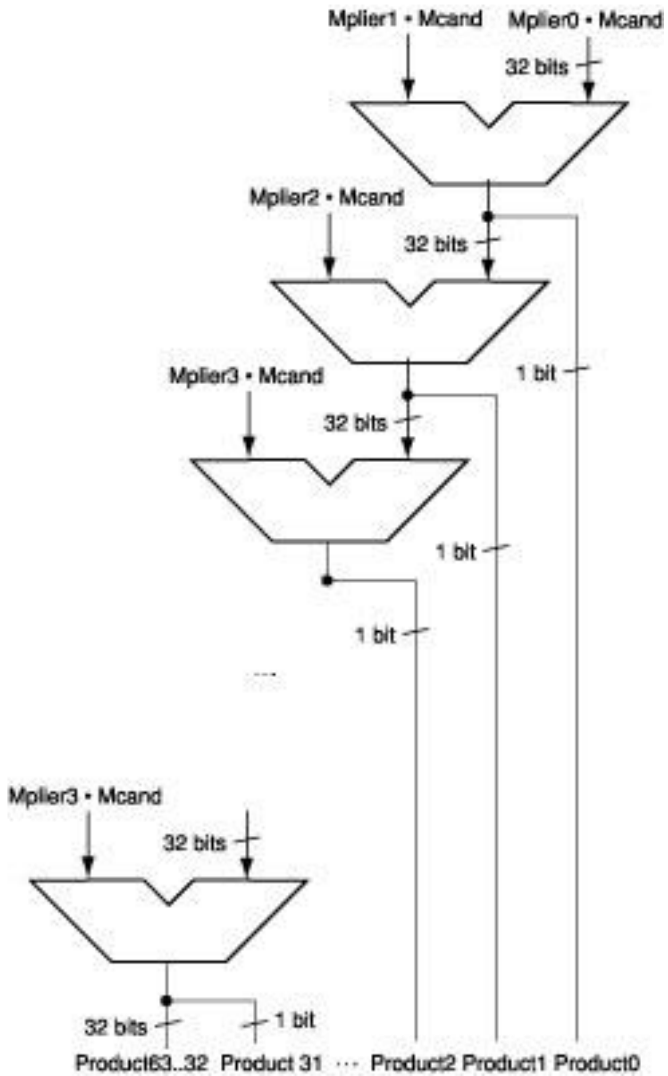
- The product of two 32-bit numbers can be a 64-bit number
  -- hence, in MIPS, the product is saved in two 32-bit registers

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

Multiplication 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

$$
\begin{array}{rrl}
 & \underline{1001_{ten}} & \text{Quotient} \\
\text{Divisor} \quad 1000_{ten} & |\quad 1001010_{ten} & \text{Dividend} \\
 & \underline{-1000} & \\
 & 10 & \\
 & 101 & \\
 & 1010 & \\
 & \underline{-1000} & \\
 & 10_{ten} & \text{Remainder}
\end{array}
$$

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	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_{two}) by 2_{ten} (0010_{two})

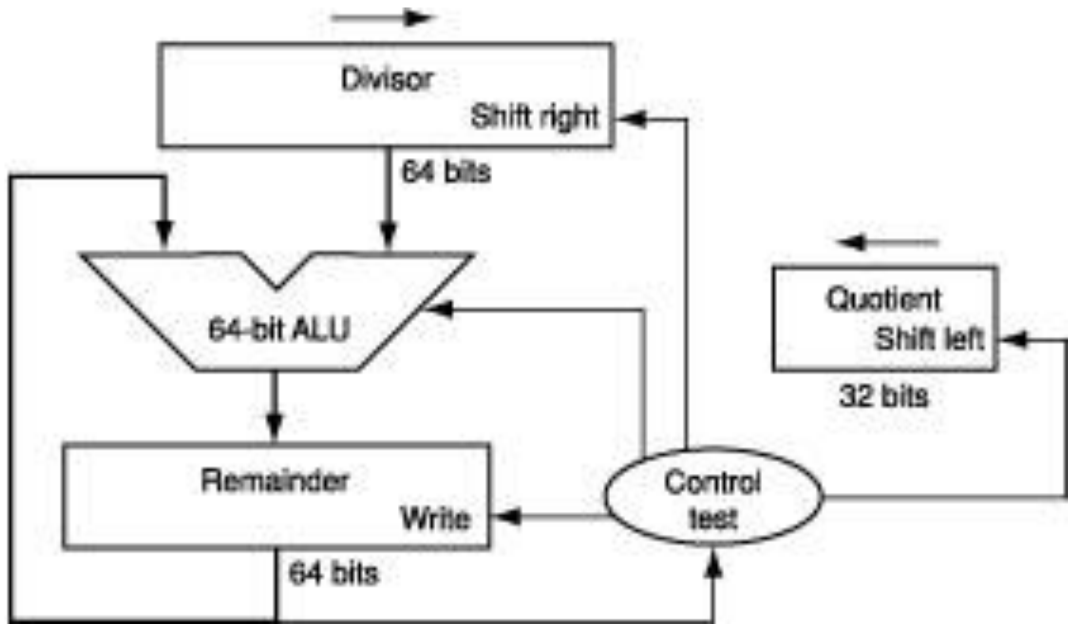
Iter	Step	Quot	Divisor	Remainder
0	Initial values			
1				
2				
3				
4				
5				

Divide Example

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

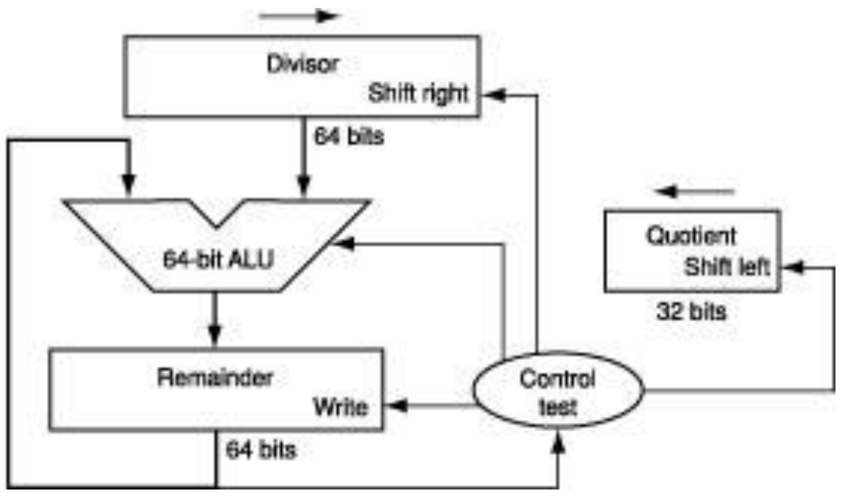
Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 ➔ +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000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

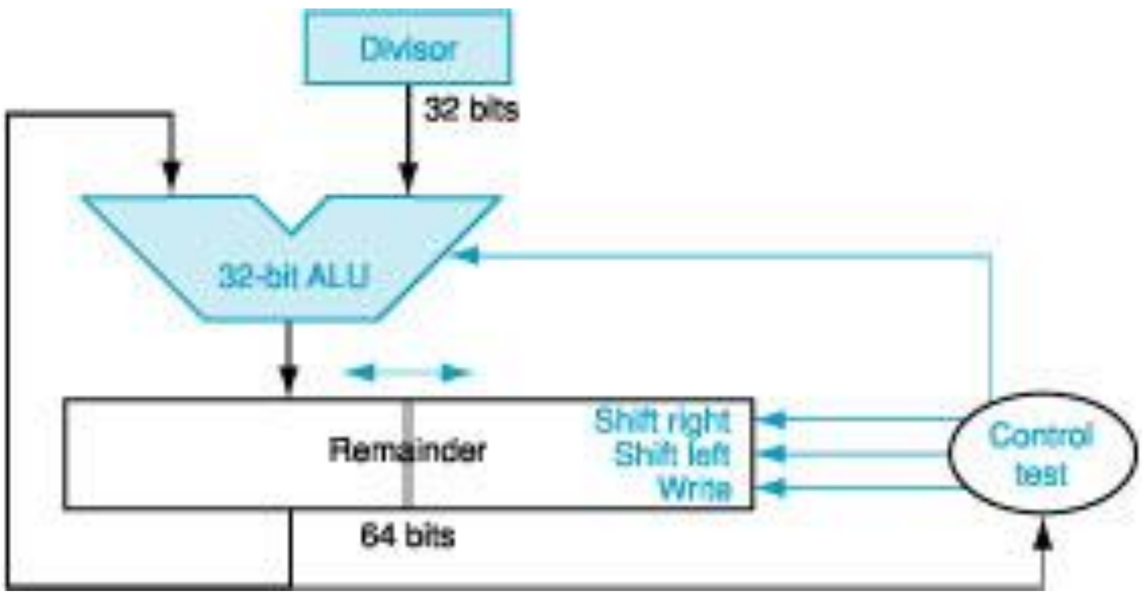
Hardware for Division



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

Efficient Division





Efficient Division Example

Iter	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
1	$(Rem)_{7-4} = (Rem)_{7-4} - Div$ $(Rem)_{7-4} < 0$ ➔ +Div Shift Rem left with 0 as least significant(LSB)	0010	1110 0111 0000 0111 0000 1110
2	Same steps as 1	0010	1110 1110 0000 1110 0001 1100
3	Same steps as 1	0010	1111 1100 0001 1100 0011 1000
4	$(Rem)_{7-4} = (Rem)_{7-4} - Div$ $(Rem)_{7-4} >= 0$ ➔ Shift Rem left with 1 as LSB	0010	0001 1000 0011 0001
5	$(Rem)_{7-4} = (Rem)_{7-4} - Div$ $(Rem)_{7-4} >= 0$ ➔ Shift Rem left with 1 as LSB	0010	0001 0001 0010 0011

After 5 iterations the lower half (bits 3-0) contains the quotient and the upper half (bits 7-4) should be shifted right to get the remainder.

Divisions involving Negatives

- Simplest solution: convert to positive and adjust sign later

- Note that multiple solutions exist for the equation:
  Dividend = Quotient x Divisor + Remainder

+7 div +2	Quo =	Rem =
-7 div +2	Quo =	Rem =
+7 div -2	Quo =	Rem =
-7 div -2	Quo =	Rem =

Divisions involving Negatives

- Simplest solution: convert to positive and adjust sign later
- Note that multiple solutions exist for the equation:
  Dividend = Quotient x Divisor + Remainder

+7 div +2	Quo = +3	Rem = +1
-7 div +2	Quo = -3	Rem = -1
+7 div -2	Quo = -3	Rem = +1
-7 div -2	Quo = +3	Rem = -1

Convention: Dividend and remainder have the same sign
Quotient is negative if signs disagree
These rules fulfil the equation above

Floating Point

- Done in the class

The SPIM Simulator

- Download QtSPIM and try it.