

Lecture 5, Multiply and Divide

Multiply: conditional shift and add


- multiplicand \times multiplier = product

$$2 \times 11 = 22$$

- idea: sum of multiplicand *mc* shifted successively by one bit relative to multiplier *mp*

- $$\begin{array}{r}
 0010 \text{ } mc \\
 \times 1011 \text{ } mp \\
 \hline
 \dots 0010 \leftarrow mc \text{ shifted 0 bit} \times \text{bit 0 of } mp \text{ (1)} \\
 0010 \leftarrow " " 1 \times " 1 " \text{ (1)} \\
 0010 \leftarrow " " 2 \times " 2 " \text{ (0)} \\
 + 0010 \leftarrow " " 3 \times " 3 " \text{ (1)} \\
 \hline
 0010110
 \end{array}$$

bit 0 is the LSB (Least Significant Bit)

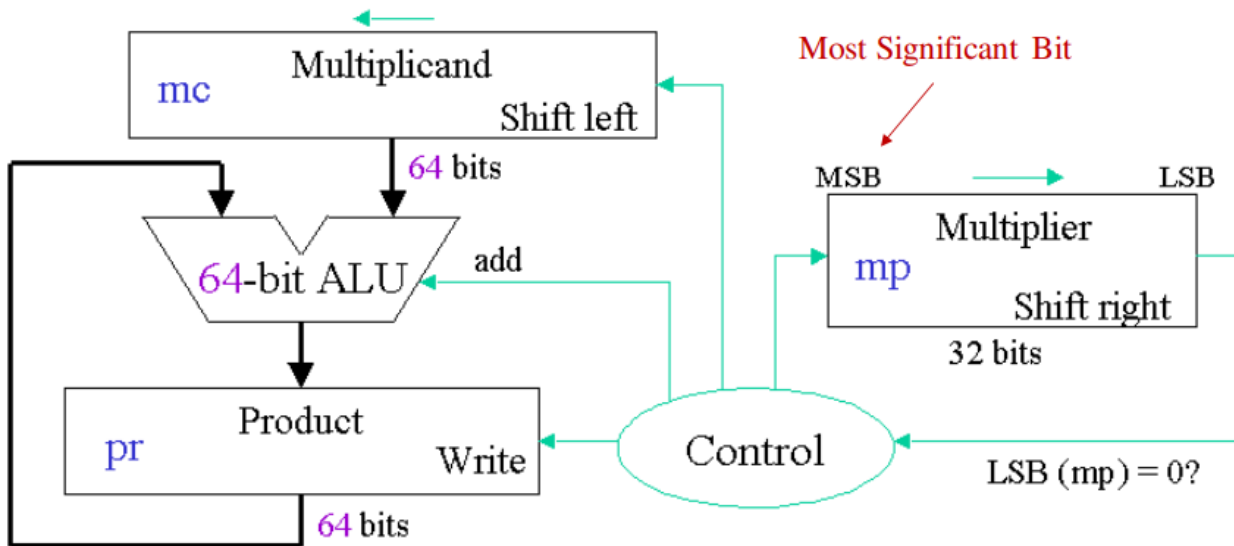


wl 2024 5.4

The algorithm can be summarised as performing **conditional shift and add**.

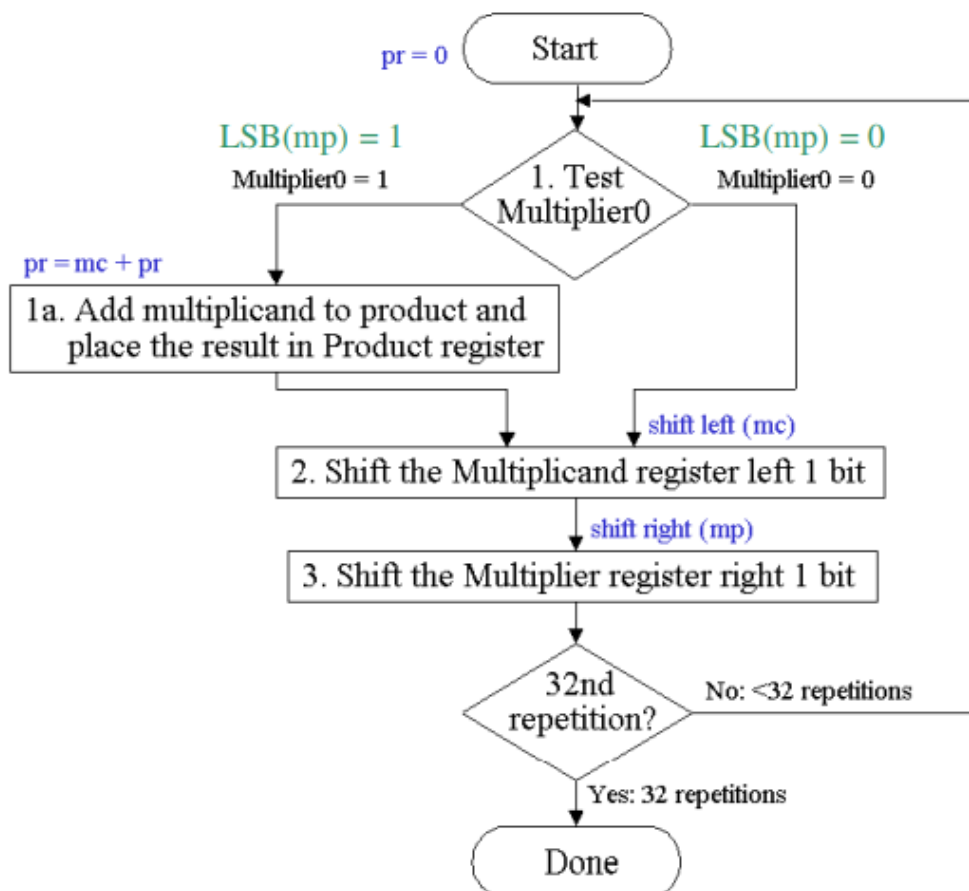
The product of two numbers is produced by adding together the successively shifted versions of the multiplicand *mc*, if the corresponding bit of the multiplier *mp* is one – so the algorithm can be summarised as performing conditional shift and add. Note that here one of the two numbers involved in the multiplication is known as the multiplier, while “multiplier” is also used to denote the hardware block capable of carrying out multiplication, just like “adder”. An example is shown in the diagram – all examples in this lecture will be based on unsigned arithmetic for simplicity.

Multiplication hardware: first version



A schematic showing how the ALU and the relevant registers would work together to implement multiplication. Note that this version involves a 64-bit ALU and two 64-bit registers `mc` and `pr` (each can be made from two 32-bit registers), and a 32-bit register `mp`

$$pr = mp \times mc$$



wl 2024 5.6

A flow chart showing how the multiplication algorithm works based on the schematic. Note the condition of `LMB(mp)=1` in order for the `shift` and `add`

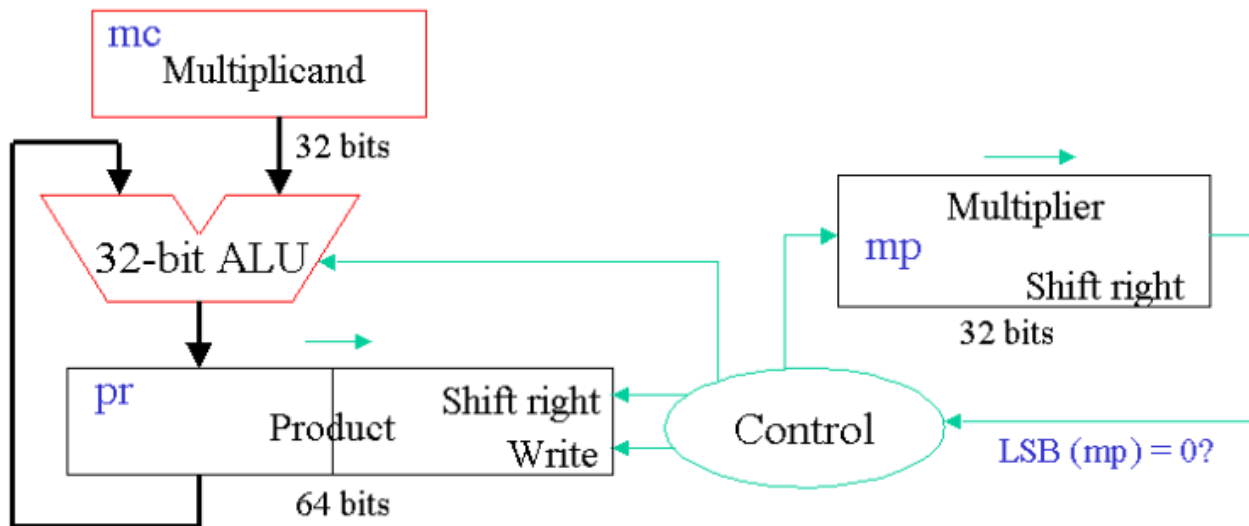
$pr = mc + pr$ to take place. The right shifting of the multiplier mp is to enable testing of its LSB.

Multiply example using first algorithm

		LSB of mp	mp	mc	pr
Iteration	Step		Multiplier	Multiplicand	Product
0	Initial Values		001 1	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		000 1	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		000 0	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 0	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

This is a numerical example showing how the state of the Multiplier register (mp), the Multiplicand register (mc), and the Product register (pr) evolves with the iteration of the algorithm. Note that for illustrative purpose, mc and pr are 8 bits and mp is 4 bits, rather than 64 bits and 32 bits respectively.

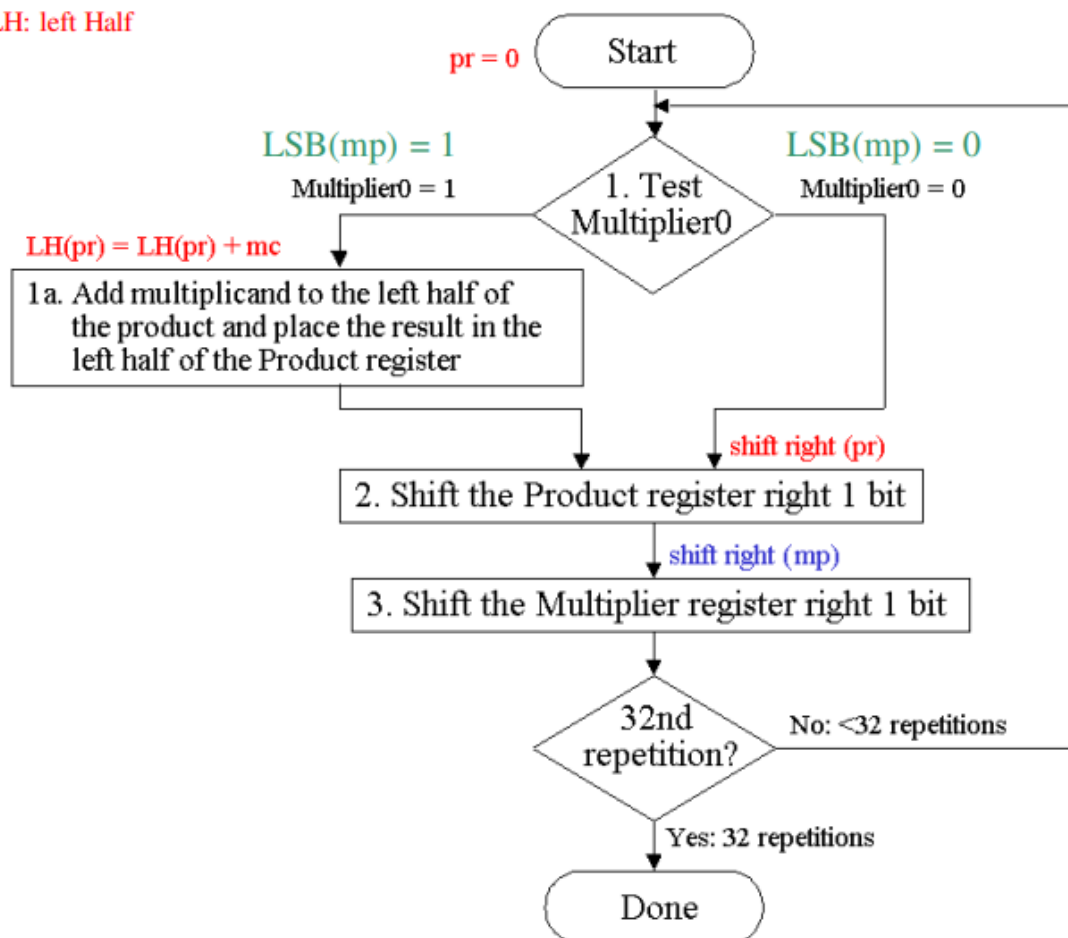
Multiplication hardware: second version



The second version design involves a 32-bit ALU and a 32-bit Multiplicand register. The key insight is that instead of shifting the Multiplicand left, we shift the Product register, which contains the partial results, to the right.

$$pr = mp \times mc$$

LH: left Half



wl 20

Here is a flow chart showing how the second version of the multiplication algorithm works based on the schematic in the last slide. The basic structure is

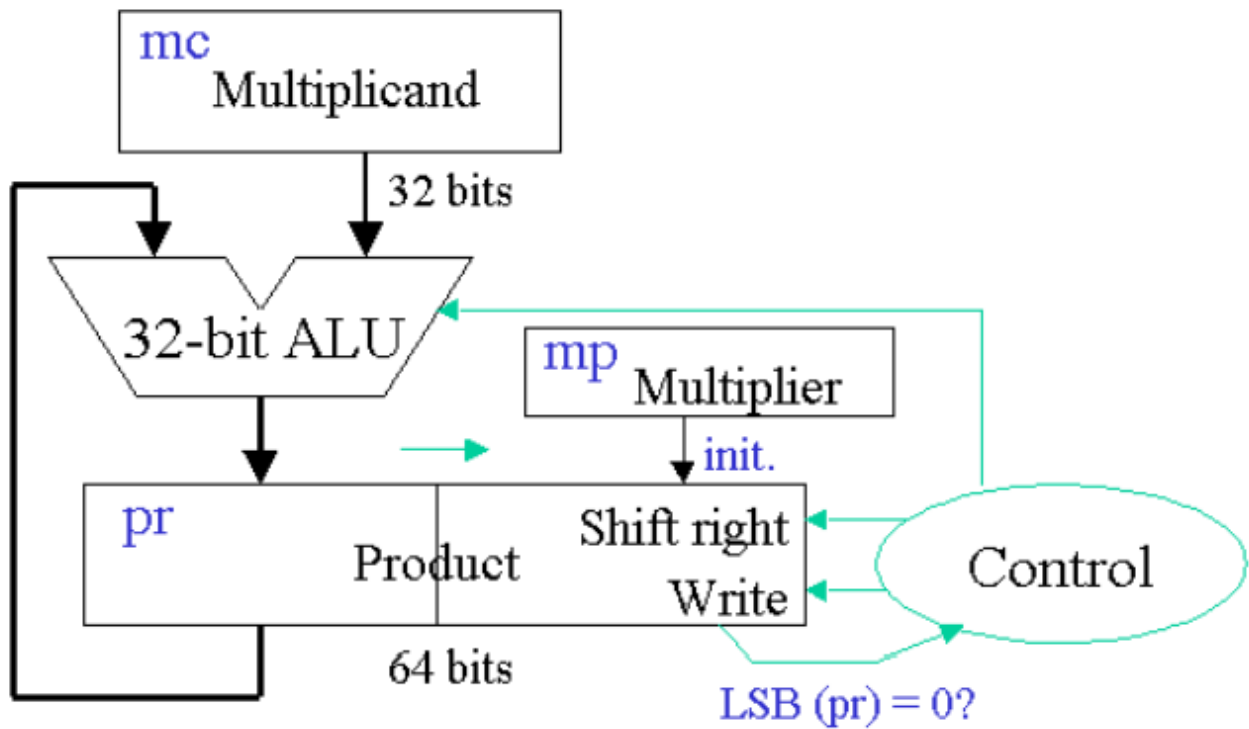
similar to the first version; the main change is the left shift of the Multiplicand `mc` is replaced by the right shift of the Product `pr`.

Multiply example using second algorithm

LSB of mp		mp	mc	pr
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	0011	0010	0000 0000
1	1a: 1 => Prod=Prod+Mcand	0011	0010	0010 0000
	2: Shift right Product	0011	0010	0001 0000
	3: Shift right Multiplier	0001	0010	0001 0000
2	1a: 1 => Prod=Prod+Mcand	0001	0010	0011 0000
	2: Shift right Product	0001	0010	0001 1000
	3: Shift right Multiplier	0000	0010	0001 1000
3	1: 0 => no operation	0000	0010	0001 1000
	2: Shift right Product	0000	0010	0000 1100
	3: Shift right Multiplier	0000	0010	0000 1100
4	1: 0 => no operation	0000	0010	0000 1100
	2: Shift right Product	0000	0010	0000 0110
	3: Shift right Multiplier	0000	0010	0000 0110

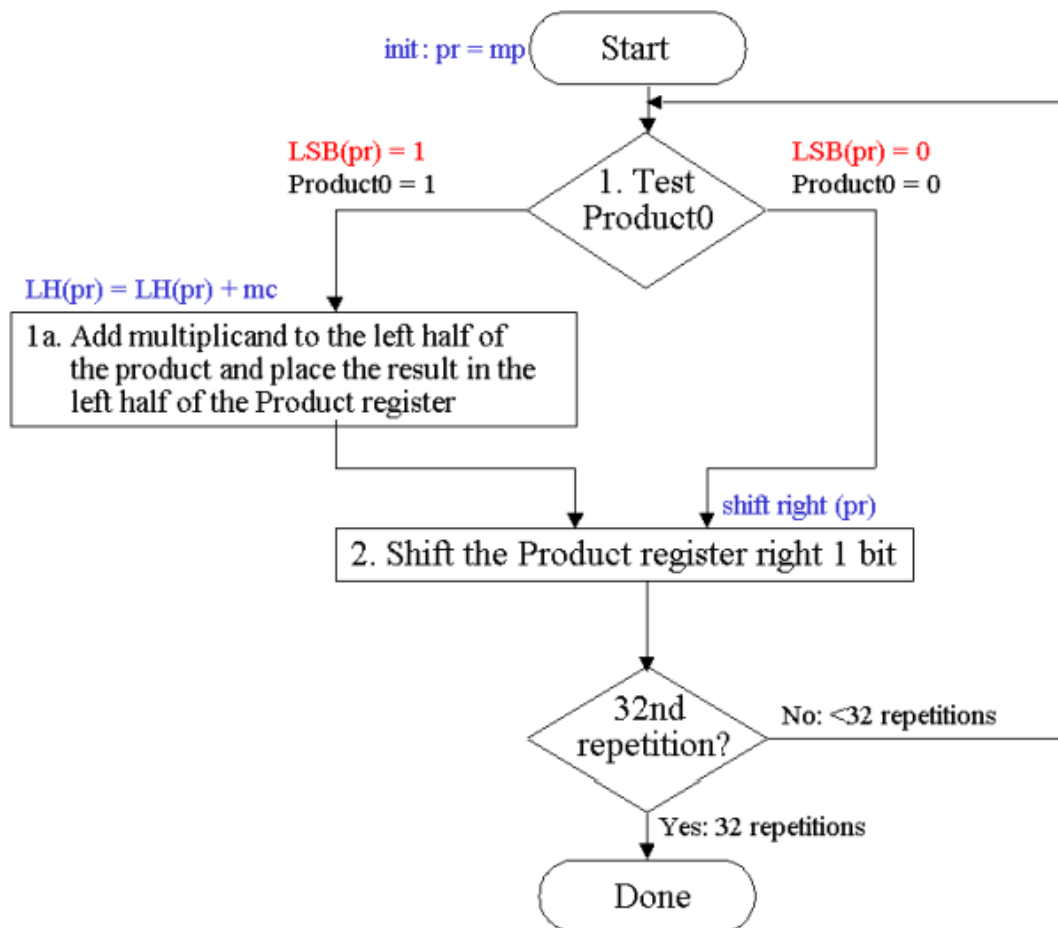
A numerical example for the second multiplication algorithm, showing how the state of the Multiplier register (`mp`), the Multiplicand register (`mc`), and the Product register (`pr`) evolves with the iteration of the algorithm. Note that only `pr` is 8 bits; both `mc` and `mp` are 4 bits

Multiplication hardware: third version



Now that both **mp** and **pr** are both shifting to the right, we do not need a separate register for **mp**. Instead, we would just initialise the right half of the **pr** register by the value of **mp**. So we only need one 32-bit register for **mc**, and a 64-bit register for **pr**.

$$pr = mp \times mc$$



wl 202

So this third algorithm is similar to the second one, except that we now test the LSB of pr rather than LSB of mp since the mp register no longer exists. Similarly no need to shift mp.


Multiply example using third algorithm

LSB of pr mc pr

Iteration	Step	Multiplicand	Product
0	Initial Values	0010	0000 0011
1	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0010 0010	0010 0011 0001 0001
2	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0010 0010	0011 0001 0001 1000
3	1: 0 => no operation 2: Shift right Product	0010 0010	0001 1000 0000 1100
4	1: 0 => no operation 2: Shift right Product	0010 0010	0000 1100 0000 0110

A numerical example for the third multiplication algorithm, showing how the state of the Multiplicand register (mc) and the Product register (pr) evolves with the iteration of the algorithm.

Booth's insight

- from Andrew Booth, Birkbeck College 
- substitute n additions by 1 subtraction, 1 addition
- successive 1s in multiplier mp
⇒ successive addition of shifted multiplicand mc

$$\begin{array}{rcl}
 & & 0010 \text{ mc} \\
 \times & & 0110 \text{ mp} \\
 \hline
 m = 00100 \rightarrow & 0010 & \text{shift left mc since mp1} = 1 \\
 k = 2 & + \quad 0010 & \text{shift left mc since mp2} = 1 \\
 \hline
 & 001100 &
 \end{array}$$

- given number of 1s in mp = k, and initially shifted mc = m, then summing the k terms gives:
 $m + 2m + 2^2m + \dots + 2^{k-1}m$ (geometric series)

Professor Andrew Booth invented an efficient way of doing multiplication. The key insight is to replace successive additions when encountering a string of ones by one addition and one subtraction. This insight is based on the property of a geometric series.

Booth's algorithm

- replace summing k terms $m + 2m + \dots + 2^{k-1}m$ by 1 subtraction and 1 addition: $-m + 2^k m$ (and k shifts to get the 2^k factor)
- proof: let $S = 1+2+\dots+2^{k-1}$, so $2 \times S = 2+4+\dots+2^{k-1}+2^k$
 $S = 2 \times S - S = 2^k + (2^{k-1} - 2^{k-1}) + \dots + (2 - 2) - 1 = 2^k - 1$
- exercise: check that it works for signed numbers
- algorithm detects a string of 1s in mp :

4 cases .. 0 1 1 1 1 .. 1 1 0 0 0

Booth's algorithm exploits the property of a geometric series, which enables replacing k additions by one subtraction and one addition. The proof is simple. There are, however, four cases that need to be considered in the Booth's algorithm.

Comparing the third algorithm and Booth's algorithm for positive numbers

Iteration	Multi- plicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial Values	0000 0110	Initial Values	0000 0110 0
1	0010 0010	1: 0 => no operation 2: Shift right Product	0000 0110 0000 0011	1a: 00 => no operation 2: Shift right Product	0000 0110 0 0000 0011 0
2	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0010 0011 0001 0001	1c: 10 => Prod=Prod-Mcand 2: Shift right Product	1110 0011 0 1111 0001 1
3	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0011 0001 0001 1000	1d: 11 => no operation 2: Shift right Product	1111 0001 1 1111 0000 1
4	0010 0010	1: 0 => no operation 2: Shift right Product	0001 1000 0000 1100	1b: 01 => Prod=Prod+Mcand 2: Shift right Product	0001 1000 1 0000 1100 0

case {
 00: middle of a string of 0s, no action
 01: end of a string of 1s, pr = pr + shifted mc
 10: start of a string of 1s, pr = pr - shifted mc
 11: middle of a string of 1s, no action
 then shift right pr

wl 2024 5.1'

While the third multiplication algorithm considers only one bit at a time, the Booth algorithm considers two bits at a time. The two bits cover the four cases: 00, 01, 10, 11. For simple examples like this one, the Booth algorithm does not show any benefit.

Division

- Dividend = Quotient \times Divisor + Remainder

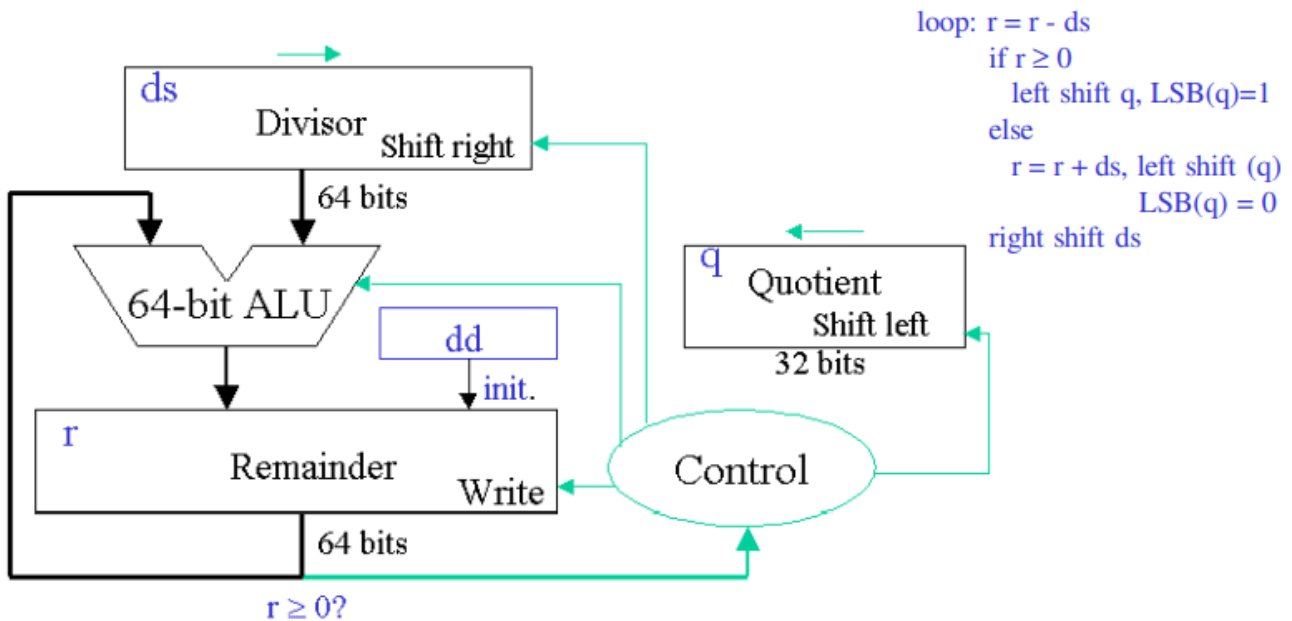
$$74 = 9 \times 8 + 2$$

- $$\begin{array}{r}
 \text{ds } 1000 \overline{) 1001010} \\
 \underline{-1000} \\
 0010 \\
 \underline{0101} \\
 1010 \\
 \underline{-1000} \\
 10
 \end{array}$$

q
 dd (r' : intermediate value)
align MSB(ds) and MSB(dd)
 $r' < ds$: $q = q + \langle 0 \rangle$
 $r' \geq ds$: $q = q + \langle 1 \rangle$
 10 r ($r = r'$ when finished)
- compare r' and ds: calculate $r' = r' - ds$
 - $r' < 0$, $r' = r' + ds$ (restore old value of r')
 - $r' \geq 0$, accept r' for further calculation

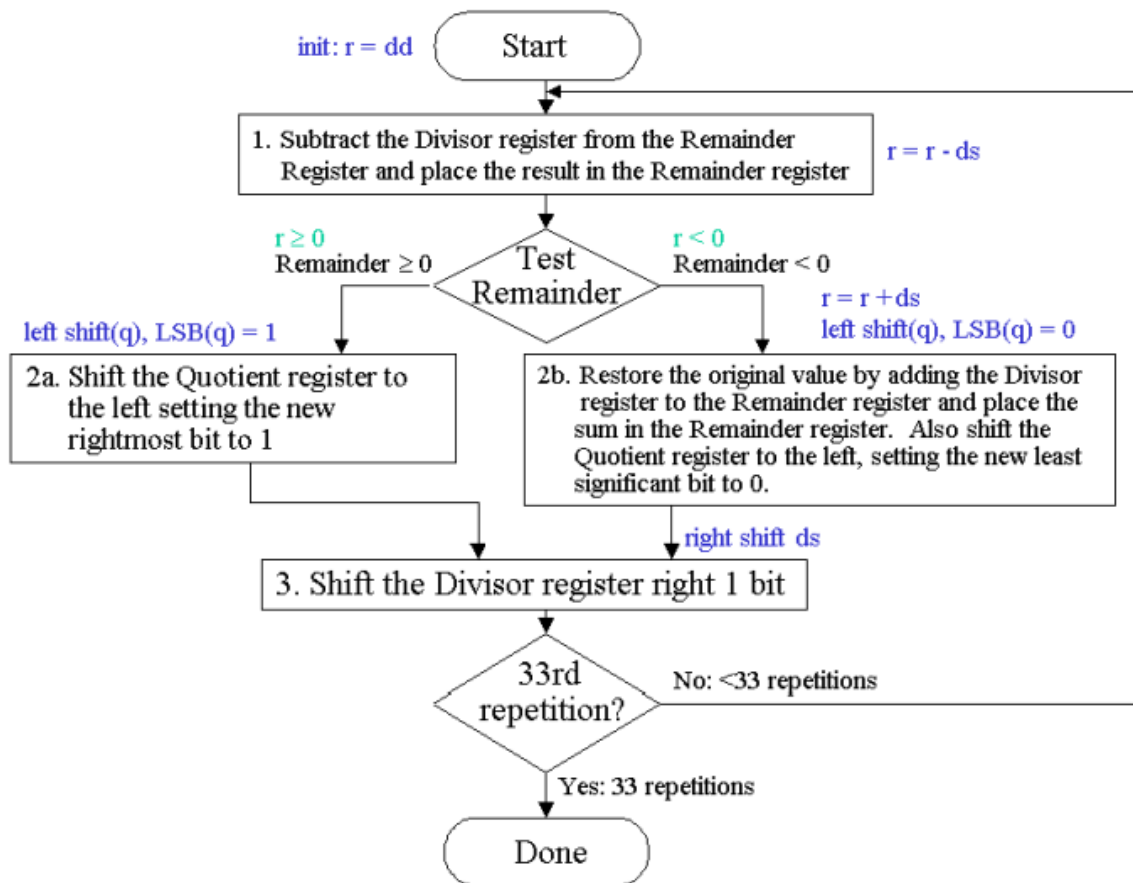
The division algorithm involves conditional shift and subtract. In contrast to the multiplication algorithm which starts from the LSB, the division algorithm starts from the MSB. It is a “restoring” algorithm, since if $(r' = r' - ds) < 0$, then the previous value of r' will be restored by computing $r' = r' + ds$.

First version of the division hardware



This is a schematic showing how the ALU and the relevant registers would work together to implement division. Note that this version involves a 64-bit ALU and two 64-bit registers, ds and r , and a 32-bit register q .

$$dd = (q \times ds) + r$$



wt 2024 5.21

A flow chart showing how the division algorithm works based on the schematic in the last slide. Notice the condition of $r < 0$ would trigger the restoring branch on the right.

Division example using first algorithm

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 \Rightarrow +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 \Rightarrow +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 \Rightarrow +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 \geq 0 \Rightarrow 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 \geq 0 \Rightarrow sll Q, Q0=1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

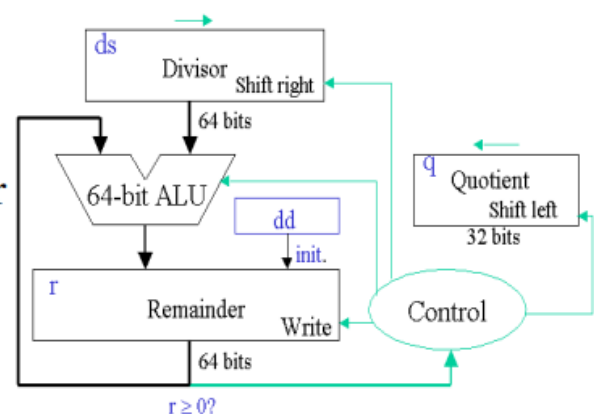
dd=7 q=3
ds=2 r=1

wl 2024 5.22

A numerical example showing how the division algorithm works. Note that restoring takes place in iteration 1, 2 and 3 when the remainder becomes less than zero after the subtraction.

Finishing...

- refining division implementation
 - remainder shift left: **reduce divisor / ALU size**
 - combine quotient and remainder registers
- MIPS instructions
 - multu, divu: unsigned operations
 - result in HI, LO registers
 - mflo: move data from LO register
- exercise: signed numbers for
 - multiplication (P&H: p.187)
 - division (P&H: p.193)



wl 2024 5.23

Just like the first version of the multiplication algorithm which can be optimised twice, similar optimisations can be applied to the division algorithm.

