

Multiplication & Division

Ref: Chapter 4

Binary Multiplication *Pencil & Paper Algorithm*

$$\begin{array}{r} 1000 \leftarrow \text{multiplicand} \\ \times 1001 \leftarrow \text{multiplier} \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 1001000 \leftarrow \text{product} \end{array}$$

Multiplication Algorithm

- The algorithm is different than the “repeated addition” algorithm we used in our MIPS multiply subroutine.
- We take advantage of *positional representation*.
 - fewer additions (many fewer)

Comparing Algorithms

- Repeated Addition:
 - could have $O(2^{32})$ additions
- Pencil & Paper Algorithm:
 - always have 32 additions.
 - also need to do some shifting at each step.

Binary Multiplication

- At each step we multiply the multiplicand by a single digit from the multiplier.
 - In binary, we multiply by either 1 or 0 (much simpler than decimal).
- Keep a running sum instead of storing and adding all the partial products at the end.

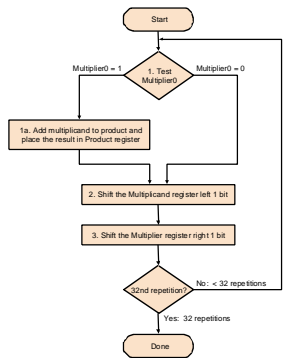


Figure 4.26

Exercise

- 4 bit multiplication of 2x3 (0010 x 0011)
- 4 steps required.
- 8 bit result

Trace for 0010 x 0011

Step	Multiplier	Multiplicand	Product
<i>initial</i>	0011	00000010	00000000
1a			00000010
2		00000100	
3	0001		
1a			00000110
2		00001000	
3	0000		
1			00000110
2		00010000	
3	0000		
1			00000110
2		00010000	
3	0000		

Step numbers as labeled in Fig 4.26 (slide 5)

32 bit multiplication

- 32 iterations
- At each iteration i
 - conditionally add multiplicand $\ll i$ to running sum.
- Result will be 64 bits long!
 - last step: multiplicand $\ll 31$

Implementing Multiplication

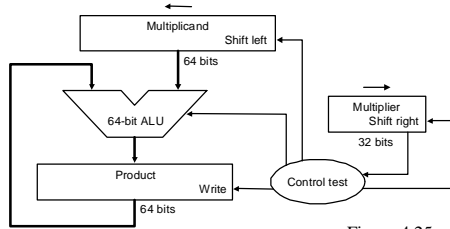


Figure 4.25

Adjustments to Algorithm

- One big problem with the algorithm/implementation shown:
 - need a 64 bit ALU (adder).
- We can fix this by:
 - Leaving the multiplicand alone (don't shift).
 - shift the running sum right at each iteration.
 - Add multiplicand to *left half* of running sum.

Adjusted Algorithm

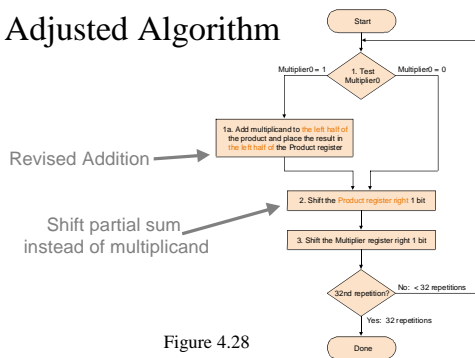
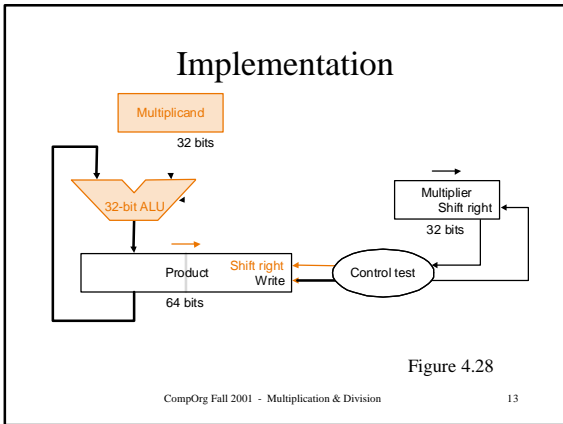


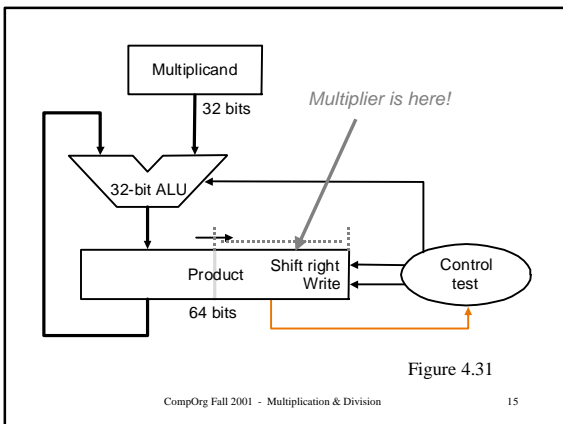
Figure 4.28



Wasted Space

- In the implementation of the adjusted algorithm it is possible to save space by putting the multiplier in the rightmost 32 bits of the product *register*.
- The algorithm is the same, but steps 2 and 3 are done at the same time.

CompOrg Fall 2001 - Multiplication & Division 14



0010 x 0011

Step	Multiplicand	Product
	0010	00000010
1a	0010	00100010
2,3		00010000
1a	0010	00110000
2,3		00011000
1	0010	00011000
2,3		00001100
1	0010	00001100
2,3		00000110

multiplier is rightmost 4 bits

1st partial product in left 4 bits

Shift Right

signed vs. unsigned

- Previous algorithms work for unsigned.
- We could:
 - convert both multiplier and multiplicand to positive before starting, but remember the signs.
 - adjust the product to have the right sign (might need to negate the product).

Supporting Signed Integers

- We can adjust the previous algorithm to work with signed integers
- make sure each shift is an *arithmetic* shift
 - right shift – extend the sign bit (keep the MS bit the same instead of shifting in a 0).
- Since addition works for signed numbers, the multiply algorithm will now work for signed numbers.

1110 x 0011 (-2 x 3)

Step	Multiplicand	Product
	1110	00000111
1a	1110	11100011
2,3		11110001
1a	1110	11010001
2,3		11101000
1	1110	11101000
2,3		11110100
1	1110	11110100
2,3		11111010

multiplier is rightmost 4 bits
 1st partial product in left 4 bits
 Shift Right
 Result is 2s complement

CompOrg Fall 2001 - Multiplication & Division 19

Booth's Algorithm

- Requires that we can do an addition or a subtraction each iteration (not always an addition).
- Uses the following property
 - the value of any consecutive string of 1s in a binary number can be computed with one subtraction.

CompOrg Fall 2001 - Multiplication & Division 20

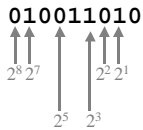
A different way to compute integer values

$0110 \quad 2^3 - 2^1 \quad (8 - 2 = 6)$
 $\uparrow \quad \uparrow$
 $2^3 \quad 2^1$

$0011111000 \quad 2^8 - 2^3 \quad (256 - 8 = 248)$
 $\uparrow \quad \uparrow$
 $2^8 \quad 2^3$

CompOrg Fall 2001 - Multiplication & Division 21

A little more complex example



$$2^8 \cdot 2^7 + 2^5 \cdot 2^3 + 2^2 \cdot 2^1$$

$$256 \cdot 128 + 32 \cdot 8 + 4 \cdot 2$$

An overview of Booth's Algorithm

- When doing multiplication, strings of 0s in the multiplier require only shifting (no addition).
- When doing multiplication, strings of 1s in the multiplier require an operation only at each end.
- We need to add or subtract only at positions in the multiplier where there is a transition from a 0 to a 1, or from a 1 to a 0.

New First Step of Multiplication

- Look at rightmost 2 bits of multiplier:
 - 00 or 11: do nothing
 - 01: Marks the end of a string of 1s
 - add multiplicand to partial product (running sum)
 - 10: Marks the beginning of a string of 1s
 - subtract multiplicand from partial product.
- Initially *pretend* there is a 0 past the rightmost bit of the multiplier.

Shifting

- The second step of the algorithm is the same as before: shift the product/multiplier right one bit.
 - Arithmetic shift needed for signed arithmetic.
 - The bit shifted off the right is saved and used by the next step 1 (which looks at 2 bits).

Trace of Booth's Alg. for 2x3

Step	Multiplicand	Product	Description
	0010	00000011 0	Initial Values
1	0010	11100011 0	10: subtract
2		11110001 1	shift right
1	0010	11110001 1	11: nothing
2		11111000 1	shift right
1	0010	00011000 1	01: add
2		00001100 0	shift right
1	0010	00001100 0	00: nothing
2		00000110 0	shift right

These are the 2 bits used in step 1 →

2 x -3

Step	Multiplicand	Product	Description
	0010	00001101 0	Initial Values
1	0010	11101101 0	10: subtract
2		11110110 1	shift right
1	0010	00010110 1	01: add
2		00001011 0	shift right
1	0010	11101011 0	10: subtract
2		11110101 1	shift right
1	0010	11110101 1	11: nothing
2		11111010 1	shift right

Booth's Algorithm

- The algorithm implemented in most processors.
- The book includes an algebraic proof that Booth's algorithm works (pg 263).
- What happens if the multiplier looks like this: **0101010101010101...** ?

CompOrg Fall 2001 - Multiplication & Division

28

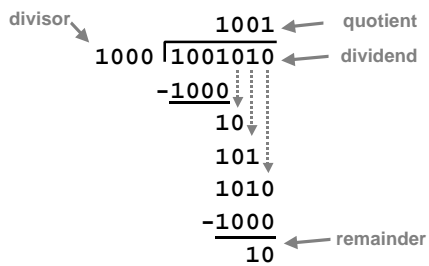
Integer Division

- In general, we divide a $2n$ bit number by an n bit number.
 - divide 64 bit dividend by a 32 bit divisor.
- Uses same approach as multiplication
 - make use of positional representation of binary integers to avoid simple repeated subtraction algorithm.

CompOrg Fall 2001 - Multiplication & Division

29

Binary Long Division



CompOrg Fall 2001 - Multiplication & Division

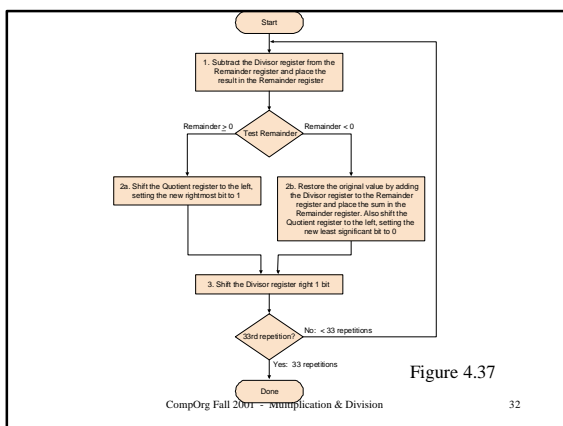
30

Unsigned Division Algorithm

- 1 iteration per bit (actually, we need 1 extra step!)
- Each iteration:
 - subtract divisor from a *partial remainder* and test to see if less than 0.
 - less than zero: divisor didn't fit once, so left shift in a 0 in quotient and add divisor back to partial remainder.
 - \geq zero: it fit, left shift a 1 in quotient.
 - shift divisor register right.

CompOrg Fall 2001 - Multiplication & Division

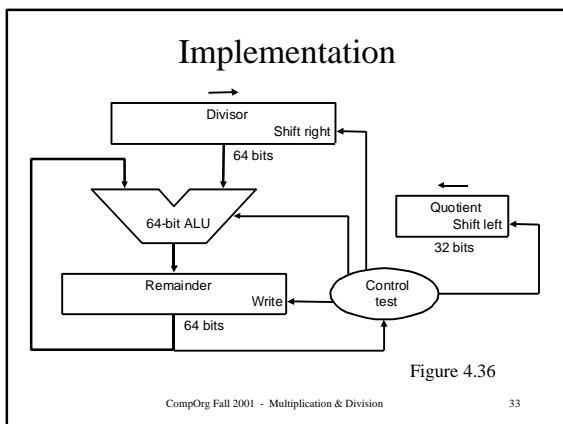
31



CompOrg Fall 2001 - Multiplication & Division

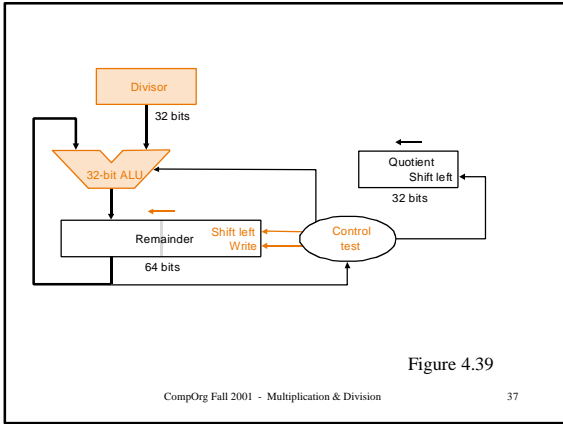
32

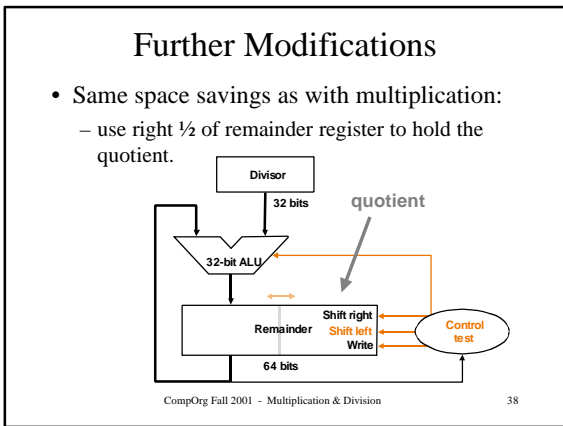
Implementation

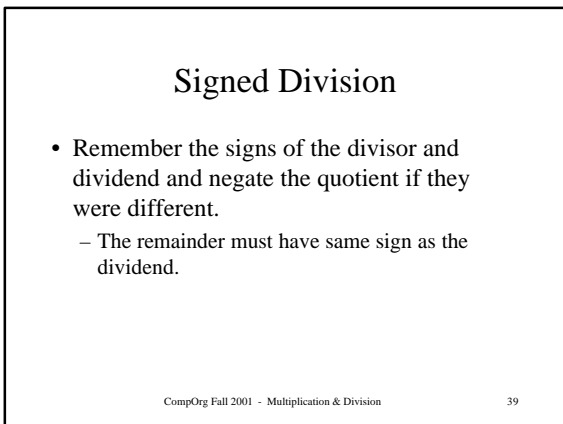


CompOrg Fall 2001 - Multiplication & Division

33







MIPS Multiplication

- Special register pair used to hold 64 bit product
 - two 32 bit registers: named “hi” and “lo”.
 - after multiplication the product is spread over the 2 registers.

CompOrg Fall 2001 - Multiplication & Division

40

mult and **multu** instructions

```
mult reg1, reg2
```

```
multu reg1, reg2
```

Result is always in **hi** and **lo** registers

CompOrg Fall 2001 - Multiplication & Division

41

Reading **hi** and **lo**

```
mflo reg move from lo
```

```
mfhi reg move from hi
```

move from hi/lo to any register.

CompOrg Fall 2001 - Multiplication & Division

42

MIPS Division

`div reg1, reg2`

`divu reg1, reg2`

divide reg1/reg2

After a division instruction the result is in
hi/lo:

- **hi** holds the 32 bit remainder

- **lo** holds the 32 bit quotient

CompOrg Fall 2001 - Multiplication & Division

43
