14:332:231
DIGITAL LOGIC DESIGN

Ivan Marsic, Rutgers University

Electrical & Computer Engineering

Fall 2013

Lecture #3: Addition, Subtraction, Multiplication, and Division

2's-Complement Representation

RECALL FROM THE LAST LECTURE:

n-bit 2's-complement representation of D: $[D]_2 = 2^n - D_2$

How to compute it?

$$[D]_2 = (2^n - 1 - D_2) + 1$$

$2^n - 1$: 1 1 ··· 1 ← n bits

$-D$: $- d_{n-1}d_{n-2} \cdots d_0$

$d'_{n-1}d'_{n-2} \cdots d'_0$

$+ \quad 1$

$[D]_2$

$1 - d_i = d'_i$: $1 - 0 = 1$, $1 - 1 = 0$

1. Complement the bits
2. Add 1 to the Least Significant Bit
3. Discard carry out from Most Significant Bit

Addition with 2's Complement

- Added by ordinary binary addition, ignoring any carries beyond the MSB
- The result *must* be inside the range of the numbers represented by n-bits.

Otherwise overflow occurs, and the result is *not* correct.

Example, number of bits limited to $n = 5$

Then, the range is $-2^{5-1} = \mathbf{-16}$ ··· $2^{5-1} - 1 = \mathbf{+15}$ ~ 32 numbers

	$+5_{10}$	00101		+9	01001		+12	01100
+	$+7_{10}$	00111	+	–8	11000	+	+7	00111
	$+12_{10}$	01100		+1	1|00001		+19	10011

2's complement of +8:
8_{10} = 01000 → 10111
+ 1
11000 = -8_{10}

ignore carries beyond MSB

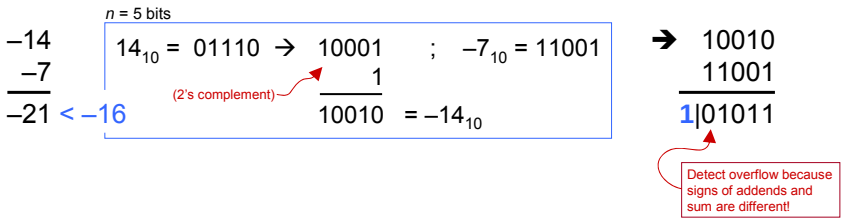
negative number resulted from adding 2 positive numbers → ***overflow***

Carries and Overflow

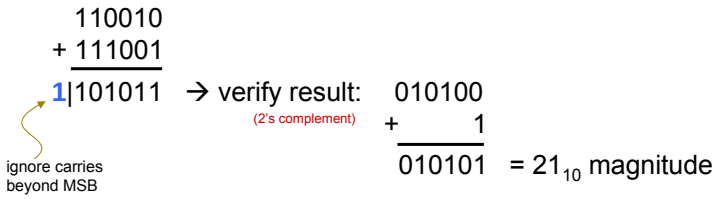
- We ignore carries beyond MSB because we are adding *two's complement* numbers as if they were *unsigned* numbers
- A carry beyond MSB is an artifact of adding the sign bits and does not indicate *overflow*
  - For example, every time we add two negative numbers, a carry beyond MSB occurs, but not necessarily an overflow
  - On the other hand, in a previous-slide example, overflow occurred without a carry beyond MSB

Overflow Detection Rule (1)

Overflow: If the sign of the addends is the same but different from the sign of the result.



If n = 6 bits, no overflow, range of numbers –32 ··· +31 :



5 of 15

Overflow Detection Rule (2)

- Overflow occurs when the value affects the sign bit:
  - adding two positives yields a negative
  - adding two negatives gives a positive
  - subtract a negative from a positive and get a negative
  - subtract a positive from a negative and get a positive
- No overflow when adding a positive and a negative number
- No overflow when subtracting two numbers of same sign
- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ? **cannot occur !**
  - Can overflow occur if A is 0 ? **can occur !**
    (for A – B if B = -2^{n-1})
    [e.g., for n=5: 0 – (–16) = +16]

6 of 15

Subtraction with 2's Complement

- $A - B = A + (-B) = A + [B]_2$

  ➔ Subtraction identical to addition, the sign absorbed by the representation

- Again, the result *must* be inside the range of the numbers represented by n-bits.

Otherwise overflow occurs, and the result is *not* correct.

Example, $n = 5$, the range is $-2^{5-1} = -16 \quad \cdots \quad 2^{5-1} - 1 = 15$

	+5	00101		+9	01001		–8	11000
–	+8	11000	–	+9	10111	–	+9	10111
	–3	11101		0	1|00000 = 0_{10}		–17	1|01111

8_{10} = 01000 → 10111 (+1) = 11000 = -8_{10}

2's complement of +9: 9_{10} = 01001 → 10110 (+1) = 10111 = -9_{10}

ignore carries beyond MSB

positive number → ***overflow***

Multiplication in Decimal

- An example in decimal:

$214_{10} \times 45_{10}$ ➡

214×45

$5 \times 214 \times 10^0$

$+ 4 \times 214 \times 10^1$ ➡

214	multiplicand
× 45	× multiplier
1070	
+ 8560	
= 9630	= product

- We do 214 × 5 = 1070 and then add to it the result of 214 × 4 = 856 *right-shifted by one column*.

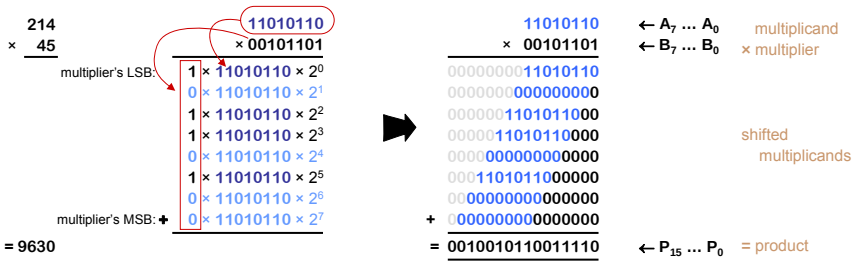
(1) For each digit of Multiplier, multiply Multiplicand by it.
(2) Multiply the product by the order of the digit ($\times 10^i$), i.e., shift it by one to the left:

zzz
× aaaa
bbbb
+ cccc0
+ dddd00
+ eeee000

etc. …

Multiplication in Binary

- Multiplying in binary follows the same form as in decimal:



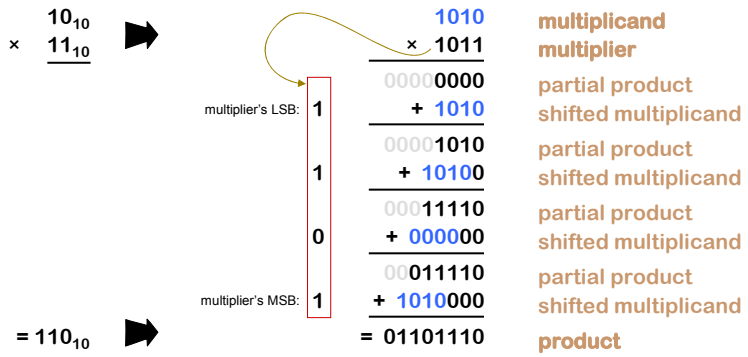
- Product *P* is composed purely of selecting, shifting and adding multiplicand *A*. The i^{th} bit of multiplier *B* indicates whether a shifted version of *A* is to be selected in the i^{th} row of the sum.

Multiplication in Binary

- Because there are only two digits in binary (0 and 1). The multiplication algorithm becomes only:
  1. Shift Multiplicand
  2. Multiply Shifted Multiplicand by 1 or 0
  3. Add the Shifted Multiplicands
- So we can perform multiplication using just full adders and a little logic for selection, in a layout which performs the shifting.

Multiplication with Partial Products

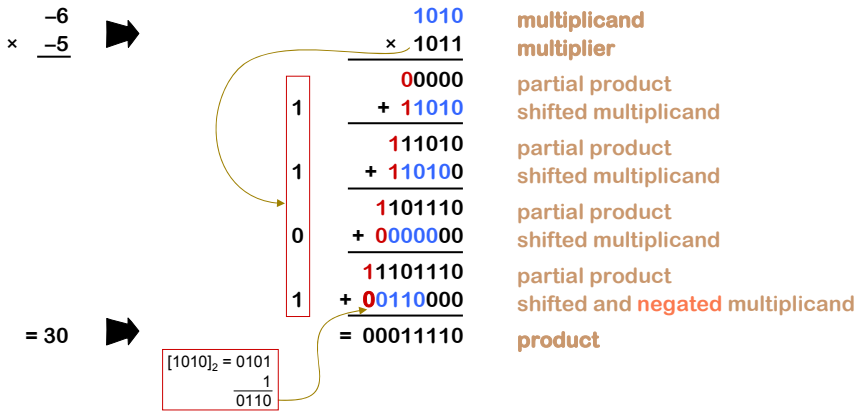
- In digital systems, more convenient to work with ***partial products***, instead of listing all shifted multiplicands and then adding them



Multiplication with 2's Complement (1)

- Two's complement multiplication works the same as unsigned multiplication:

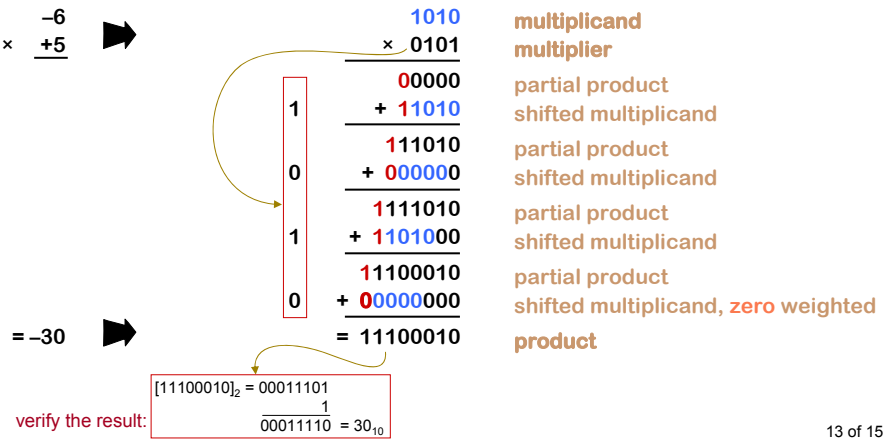
shifted multiplicand is weighted by the multiplier bit, except for the MSB which, when "1" (i.e., negative multiplier), has a **negative** weight



Multiplication with 2's Complement (2)

- Two's complement multiplication works the same as unsigned multiplication:

when multiplier is positive, its MSB has zero weight:



Decimal Division

827_{10} dividend
÷ 21_{10} divisor
= 39_{10} quotient
8_{10} remainder

827 | 21

827 | 21

1. Select most-significant digit from Dividend to compare to Divisor

   8 < 21

2. It's smaller than Divisor; so, consider two digits

   82 > 21

3. Find greatest ***d*** (from 1 to 9) that satisfies:

   82 ≥ 21 × d

82 > 21×1 = 21
82 > 21×2 = 42
82 > 21×3 = 63
82 < 21× 4 = 84
⇒ use **3** = (4–1)

4. Determine ***d*** using:
   - Intuition (guessing) when done by human
   - Algorithm that increases ***d*** until
     - either *d* × 21 > 82; use (*d*–1)
     - or *d* = 9

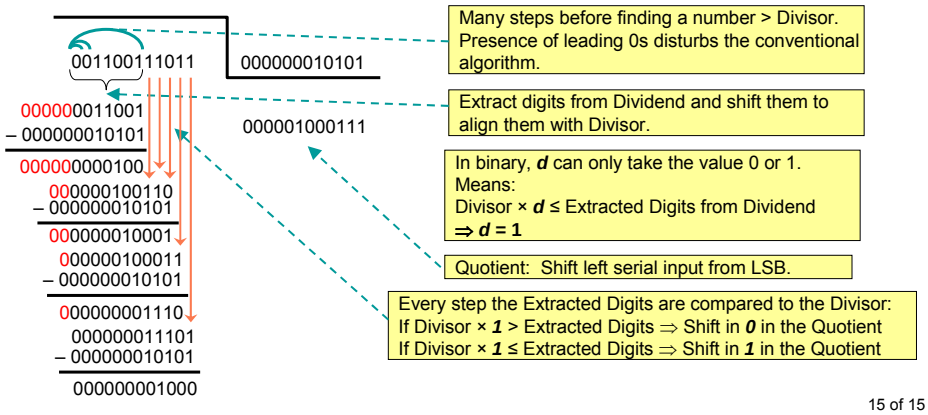
197 > 21×1
...
197 > 21×8
197 > 21×9 = 189
⇒ use **9**

827 | 21
–63
19
197
–189
8 ← remainder

39 ← quotient

Binary Division

	827_{10}	dividend	001100111011_2
÷	21_{10}	divisor	÷ 000000010101_2
=	39_{10}	quotient	= 000001000111_2
	8_{10}	remainder	000000001000_2



15 of 15