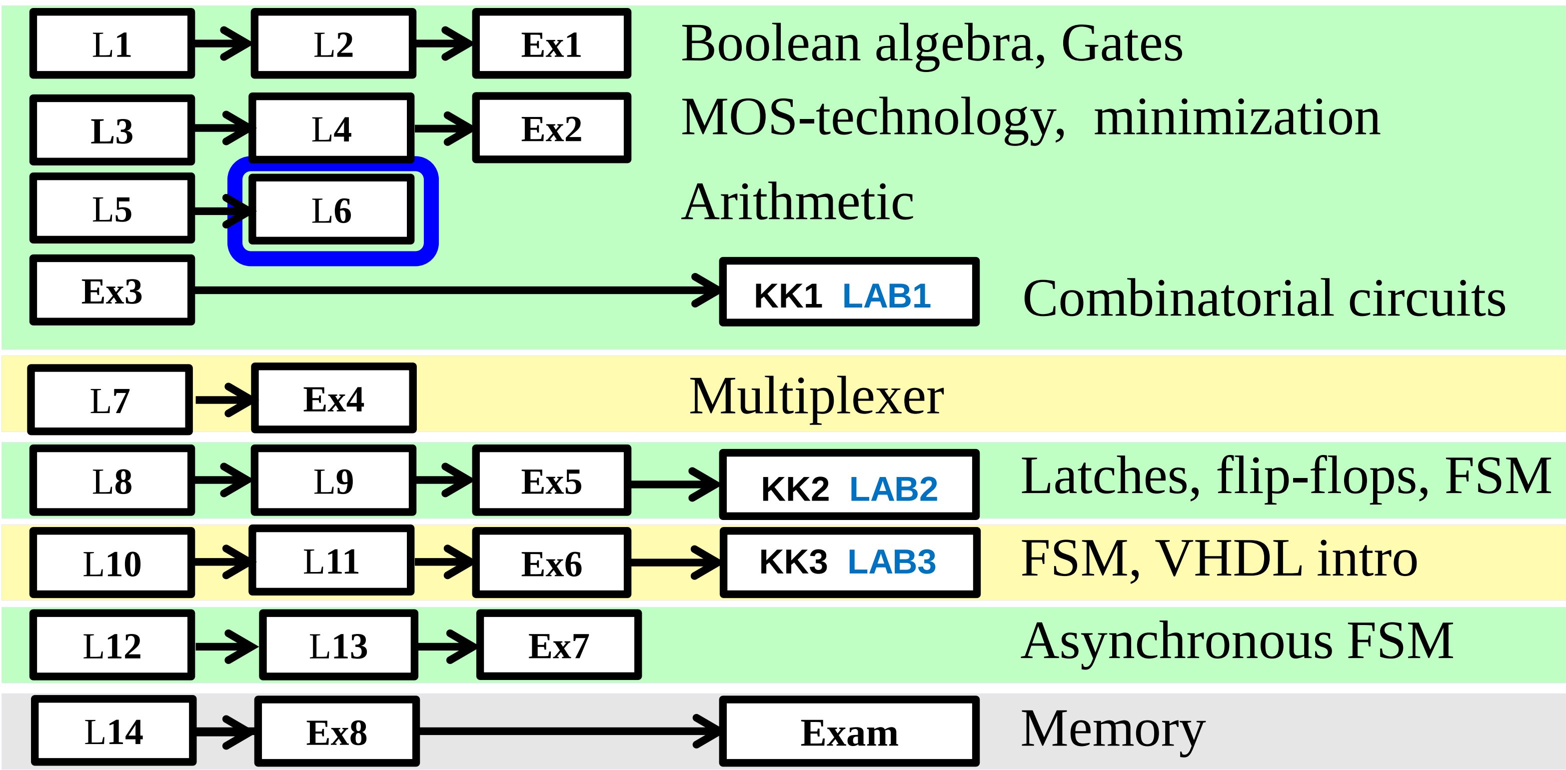
KTH Informations- och kommunikationsteknik

IE1204 Digital Design

L6: Digital Arithmetic II

Masoumeh (Azin) Ebrahimi
KTH/ICT
mebr@kth.se

IE1204 Digital Design



This lecture covers …

- BV pp.291-302, 311, 319, 371, 692-693

Number representations

- A number can be represented in binary in many ways
- The most common types of numbers are:
  - Unsigned integers = can only be positive
  - Signed integers = can be positive or negative
    - 1's complement, 2's complement, sign-and-magnitude
  - Fixed-point numbers
  - Floating-point numbers

Integers

Positive (unsigned) integers:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	1	1	0	1

$= 1*2^6 + 1*2^5 + 1*2^3 + 1*2^2 + 1*2^0 = 109$

Negative integers:

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	1	1	0	1

$= -1*2^7 + 1*2^6 + 1*2^5 + 1*2^3 + 1*2^2 + 1*2^0 = -19$

2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	0	1

$= -1*2^5 + 1*2^3 + 1*2^2 + 1*2^0 = -19$

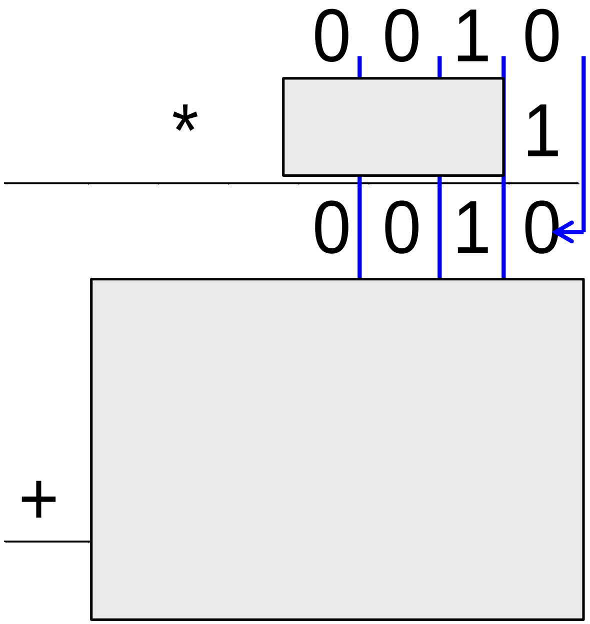
Two important points to remember about 2’s complement :

- The sign bit has a negative weight.
- Sign bit should be extended to fit the larger bits.

Multiplication of two (positive) integers

				0	0	1	0	2	Multiplicand
		*		1	0	1	1	11	Multiplier
				0	0	1	0		
			0	0	1	0			
		0	0	0	0				
+	0	0	1	0					
	0	0	1	0	1	1	0	22	Product

Multiplication of two (positive) integers

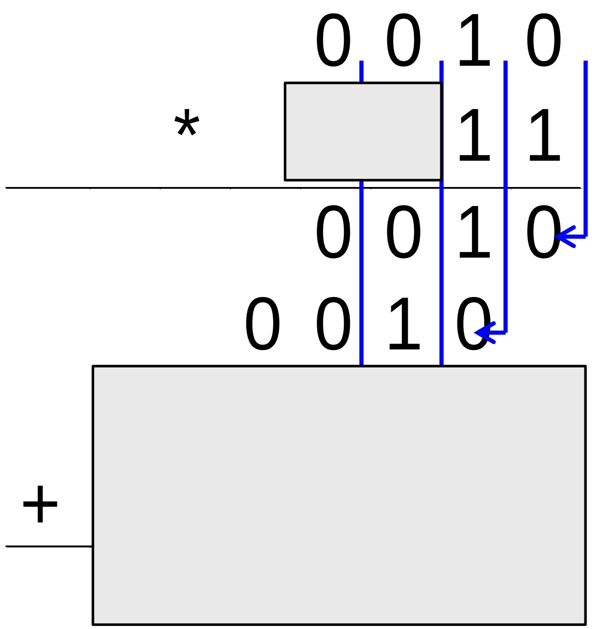


2 Multiplicand

11 Multiplier

22 Product

Multiplication of two (positive) integers

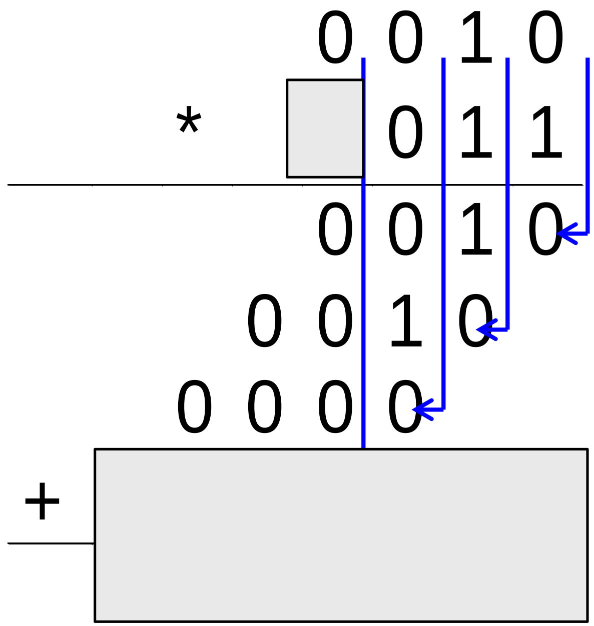


2 Multiplicand

11 Multiplier

22 Product

Multiplication of two (positive) integers



2 Multiplicand

11 Multiplier

22 Product

Multiplication of two (positive) integers

				0	0	1	0	2	Multiplicand
*				1	0	1	1	11	Multiplier
				0	0	1	0		
			0	0	1	0			
		0	0	0	0				
+	0	0	1	0					
	0	0	1	0	1	1	0	22	Product

Multiplication with a sign

					0	0	1	0		**2**	Multiplicand
	*				1	0	1	1		**-5**	Multiplier
					0	0	1	0			
				0	0	1	0				
			0	0	0	0					
+	1	1	1	0							
	1	1	1	0	1	1	0			**-10**	Product

The sign bit has negative weight!
=> 2's complement!

The sign bit in the other way

					1	0	1	1	**-5**	Multiplicand
	*				0	0	1	0	**2**	Multiplier
					0	0	0	0		
Sign extension →		1	1	1	0	1	1			
			0	0	0	0				
	+	0	0	0	0					
		1	1	1	0	1	1	0	**-10**	Product

The multiplication of two negative numbers

```
              1 0 1 1        -5
        *     1 0 1 1        -5
      ---------------------------
          1 1 1 1 0 1 1
          1 1 1 0 1 1
            0 0 0 0
        + 0 1 0 1
      ---------------------------
     (1)  0 0 1 1 0 0 1      25
```

Sign extension

The sign bit has negative weight!
=> 2's complement!

Carry-bit is ignored

To make it easy for us ...

- Use only positive numbers in the multiplication
  - Convert to positive numbers
  - Keep track of the result's sign

    (+ +) => +; (+, -) => -; (-, +) => -; (-, -) => +

  - Use 2’s complement for negative numbers

To make it easy for us ...

- Multiply -5 by +2 by first converting -5 to the positive number and then keeping track of the result's sign.

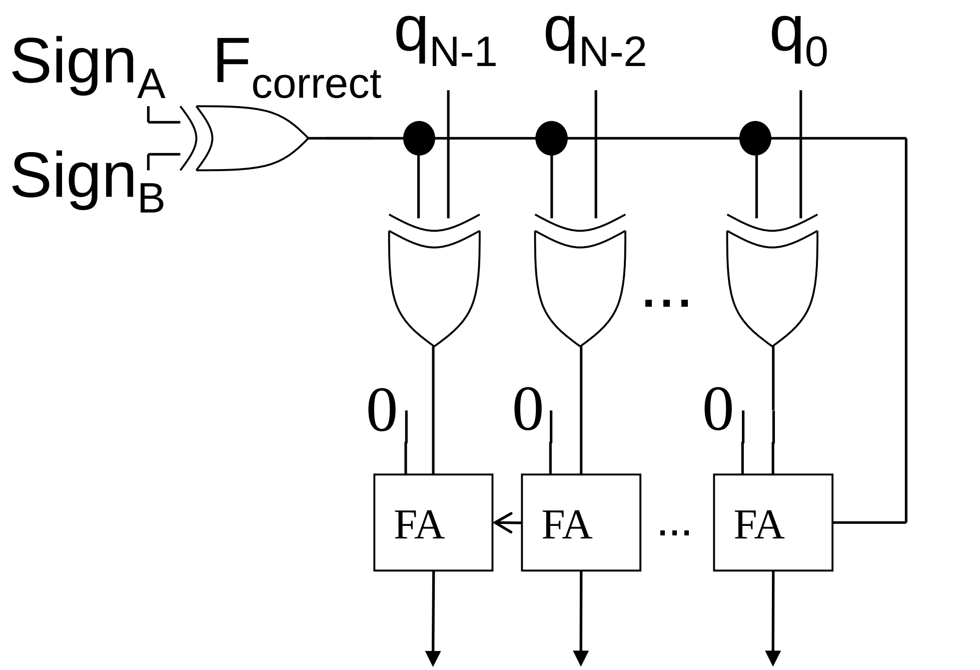
```
              0 1 0 1        -5
      *       0 0 1 0         2
  ---------------------------------
              0 0 0 0
            0 1 0 1
          0 0 0 0
  +     0 0 0 0
  ---------------------------------
        0 0 0 1 0 1 0       +10
2's complement
      1 1 1 0 1 1 0         -10
```

A simple solution (cont'd.)

$Sign_B$ \ $Sign_A$	0	1
0	0	1
1	1	0

$F_{correct} = Sign_A \text{ xor } Sign_B$

The correction is done by inverting the bits, and then add 1



2’s complement of product (when correction is needed)

Multiplication

$$(a_3a_2a_1a_0)\cdot(b_3b_2b_1b_0)=(p_7p_6p_5p_4p_3p_2p_1p_0)$$

Multiplicand   Multiplier

$$
\begin{array}{cccccccccl}
 & & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \text{AND} \\
 & & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & \text{AND} \\
+ & & & c_{40} & c_{30} & c_{20} & c_{10} & & & \text{FA, carry} \\
\hline
 & & & & r_{14} & r_{13} & r_{12} & q_1 & q_0 & \\
 & & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & \text{AND} \\
+ & & c_{41} & c_{31} & c_{21} & c_{11} & & & & \text{FA, carry} \\
\hline
 & & & r_{25} & r_{24} & r_{23} & q_2 & & & \\
 & & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & \text{AND} \\
+ & c_{42} & c_{32} & c_{22} & c_{12} & & & & & \text{FA, carry} \\
\hline
 & q_7 & q_6 & q_5 & q_4 & q_3 & & & &
\end{array}
$$

Multiplication (two positive integers)

$$(a_3a_2a_1a_0)\cdot(b_3b_2b_1\boxed{b_0}) = (q_7q_6q_5q_4q_3q_2q_1q_0)$$

Multiplicand Multiplier

$a_3\boxed{b_0}$ $a_2\boxed{b_0}$ $a_1\boxed{b_0}$ $a_0\boxed{b_0}$ AND

Multiplication (two positive integers)

$$(a_3a_2a_1a_0)\cdot(b_3b_2\boxed{b_1}b_0)=(q_7q_6q_5q_4q_3q_2q_1q_0)$$

$$\begin{array}{rcccccl}
 & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \text{AND} \\
 & a_3\boxed{b_1} & a_2\boxed{b_1} & a_1\boxed{b_1} & a_0\boxed{b_1} & & \text{AND} \\
+\quad c_{40} & c_{30} & c_{20} & c_{10} & & & \text{FA, carry} \\
\hline
 & r_{14} & r_{13} & r_{12} & q_1 & q_0 &
\end{array}$$

Multiplication (two positive integers)

$$(a_3a_2a_1a_0)\cdot(b_3\boxed{b_2}b_1b_0)=(q_7q_6q_5q_4q_3q_2q_1q_0)$$

$$
\begin{array}{cccccccccl}
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & & \text{AND} \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & & \text{AND} \\
+ & & c_{40} & c_{30} & c_{20} & c_{10} & & & & \text{FA, carry} \\
\hline
 & & & r_{14} & r_{13} & r_{12} & q_1 & q_0 & & \\
 & & a_3\boxed{b_2} & a_2\boxed{b_2} & a_1\boxed{b_2} & a_0\boxed{b_2} & & & & \text{AND} \\
+ & c_{41} & c_{31} & c_{21} & c_{11} & & & & & \text{FA, carry} \\
\hline
 & & r_{25} & r_{24} & r_{23} & q_2 & & & &
\end{array}
$$

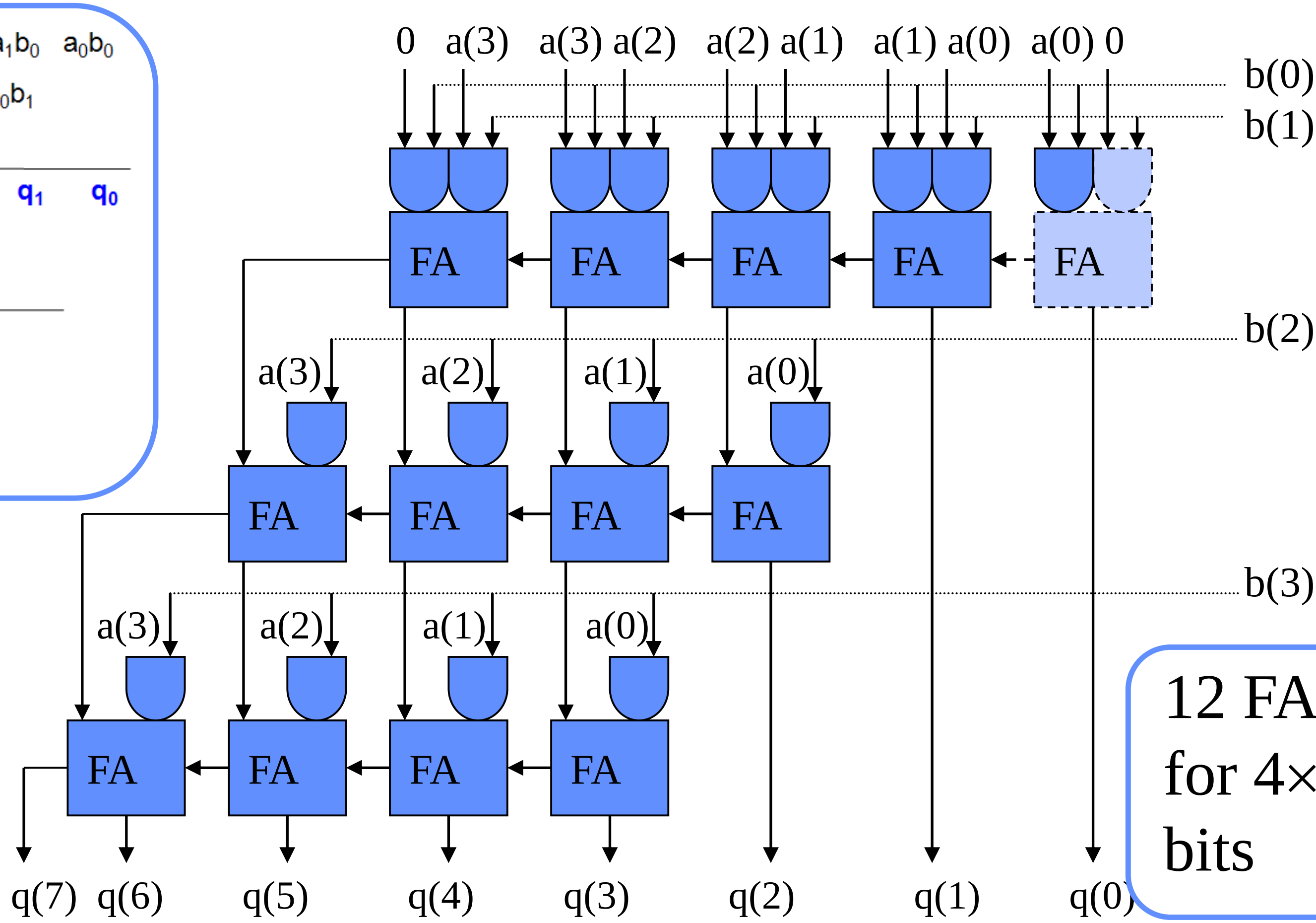
Multiplication (two positive integers)

$$(a_3a_2a_1a_0)\cdot(\boxed{b_3}b_2b_1b_0)=(q_7q_6q_5q_4q_3q_2q_1q_0)$$

					a_3b_0	a_2b_0	a_1b_0	a_0b_0	AND
				a_3b_1	a_2b_1	a_1b_1	a_0b_1		AND
+			c_{40}	c_{30}	c_{20}	c_{10}			FA, carry
				r_{14}	r_{13}	r_{12}	q_1	q_0	
			a_3b_2	a_2b_2	a_1b_2	a_0b_2			AND
+		c_{41}	c_{31}	c_{21}	c_{11}				FA, carry
			r_{25}	r_{24}	r_{23}	q_2			
		$a_3\boxed{b_3}$	$a_2\boxed{b_3}$	$a_1\boxed{b_3}$	$a_0\boxed{b_3}$				AND
+	c_{42}	c_{32}	c_{22}	c_{12}					FA, carry
	q_7	q_6	q_5	q_4	q_3				

The multiplier (two 4-bit positive numbers)

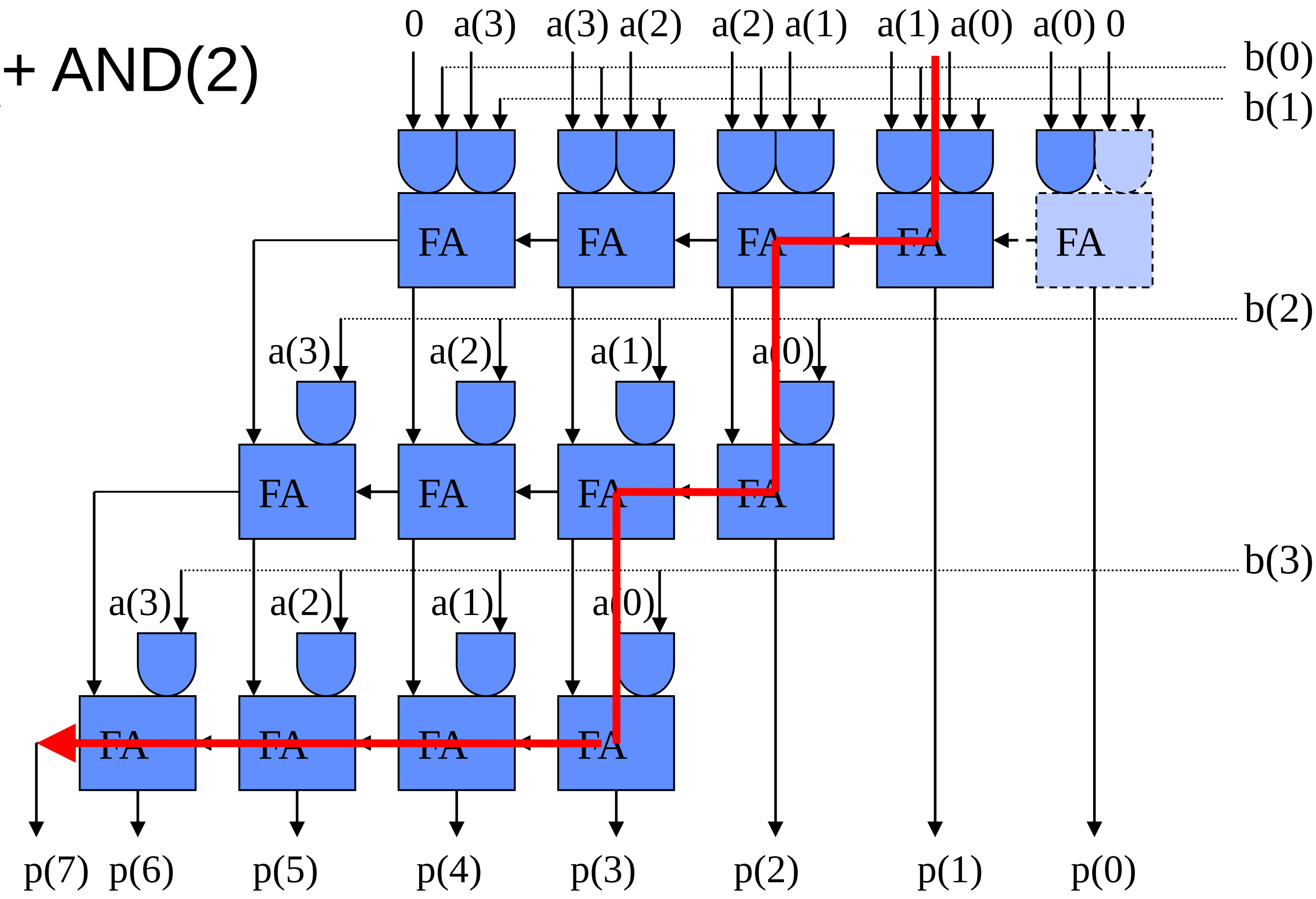
					a_3b_0	a_2b_0	a_1b_0	a_0b_0
				a_3b_1	a_2b_1	a_1b_1	a_0b_1	
+				c_{40}	c_{30}	c_{20}	c_{10}	
					r_{14}	r_{13}	r_{12}	q_1	q_0
			a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+			c_{41}	c_{31}	c_{21}	c_{11}		
			r_{25}	r_{24}	r_{23}	q_2		
		a_3b_3	a_2b_3	a_1b_3	a_0b_3			
+		c_{42}	c_{32}	c_{22}	c_{12}			
	q_7	q_6	q_5	q_4	q_3			



12 FA for 4×4 bits

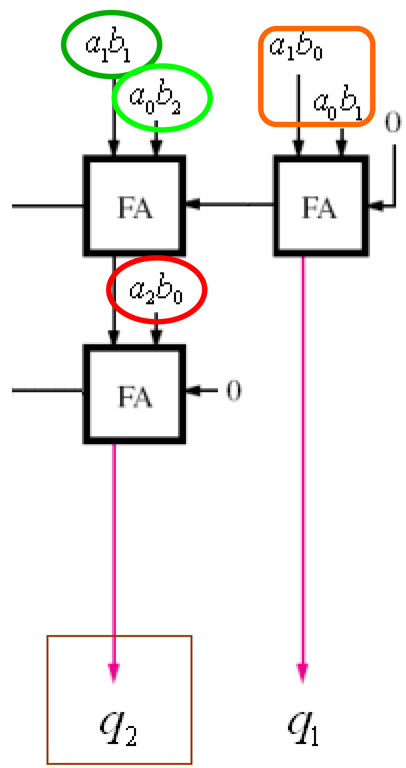
The multiplier (two 4-bit positive numbers)

$T_{MUL} = 8 * T_{FA} + AND(2)$



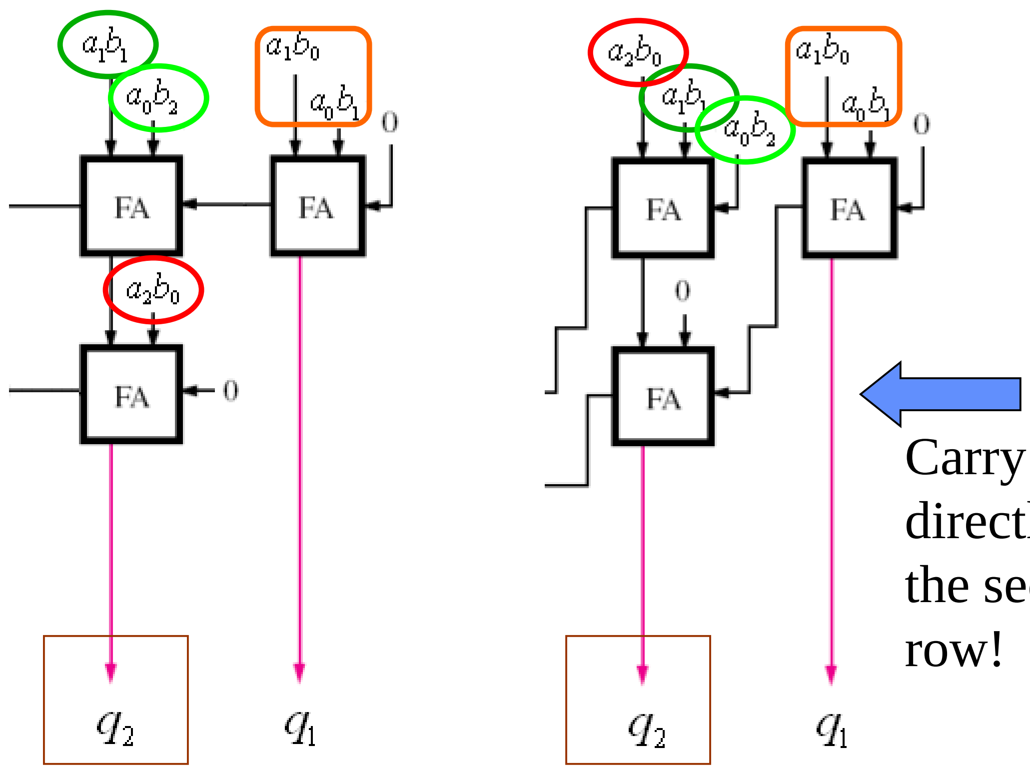
Can we change the calculation order?

					a_3b_0	a_2b_0	a_1b_0	a_0b_0
				a_3b_1	a_2b_1	a_1b_1	a_0b_1	
+			c_{40}	c_{30}	c_{20}	c_{10}		
				r_{14}	r_{13}	r_{12}	q_1	q_0
			a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+		c_{41}	c_{31}	c_{21}	c_{11}			
			r_{25}	r_{24}	r_{23}	q_2		
		a_3b_3	a_2b_3	a_1b_3	a_0b_3			
+	c_{42}	c_{32}	c_{22}	c_{12}				
	q_7	q_6	q_5	q_4	q_3			



Can we change the calculation order?

In this way q_2 will get the same value but with a different bit order!



Carry is fed directly to the second row!

A quicker solution - Carry-Save Multiplier

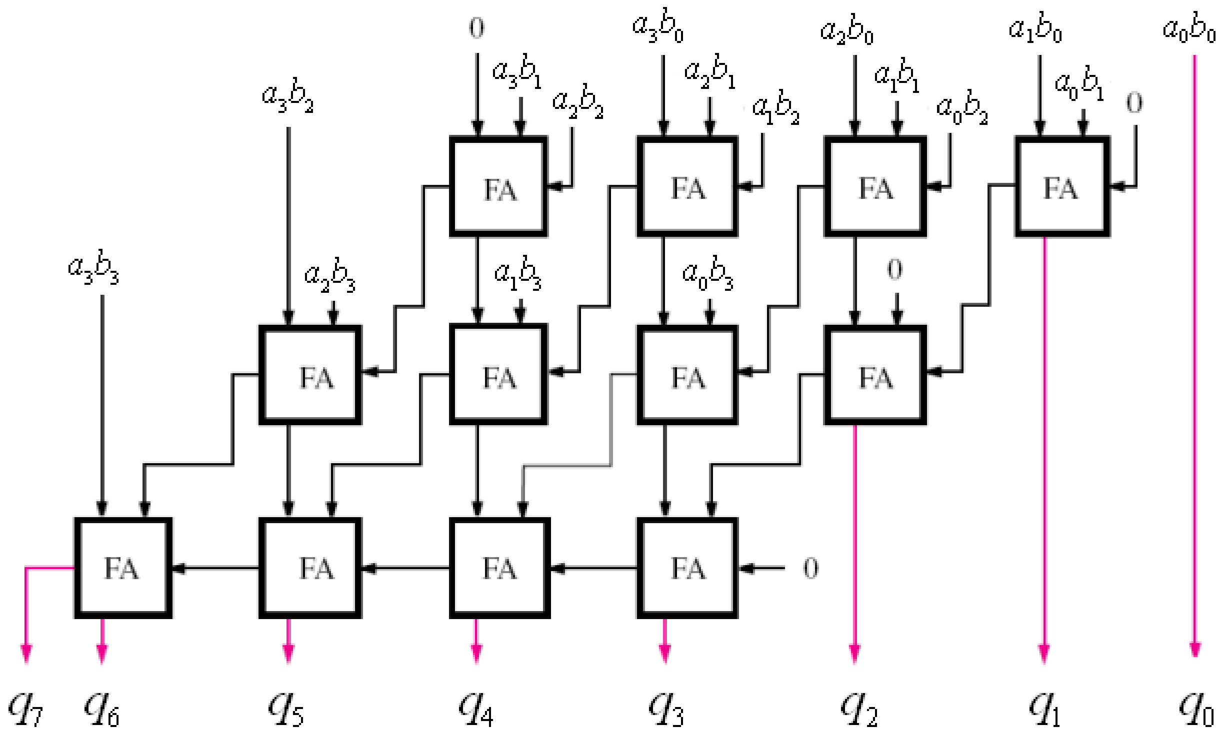


Figure 5.45 Multiplier carry-save array.

More details: (BV: page 311)

A quicker solution - Carry-Save Multiplier

$T_{MUL} \sim (2*N-2)*T_{FA}$

Reduces the delay because the carry is fed directly to the next row!

Carry is delayed through 6 stages!

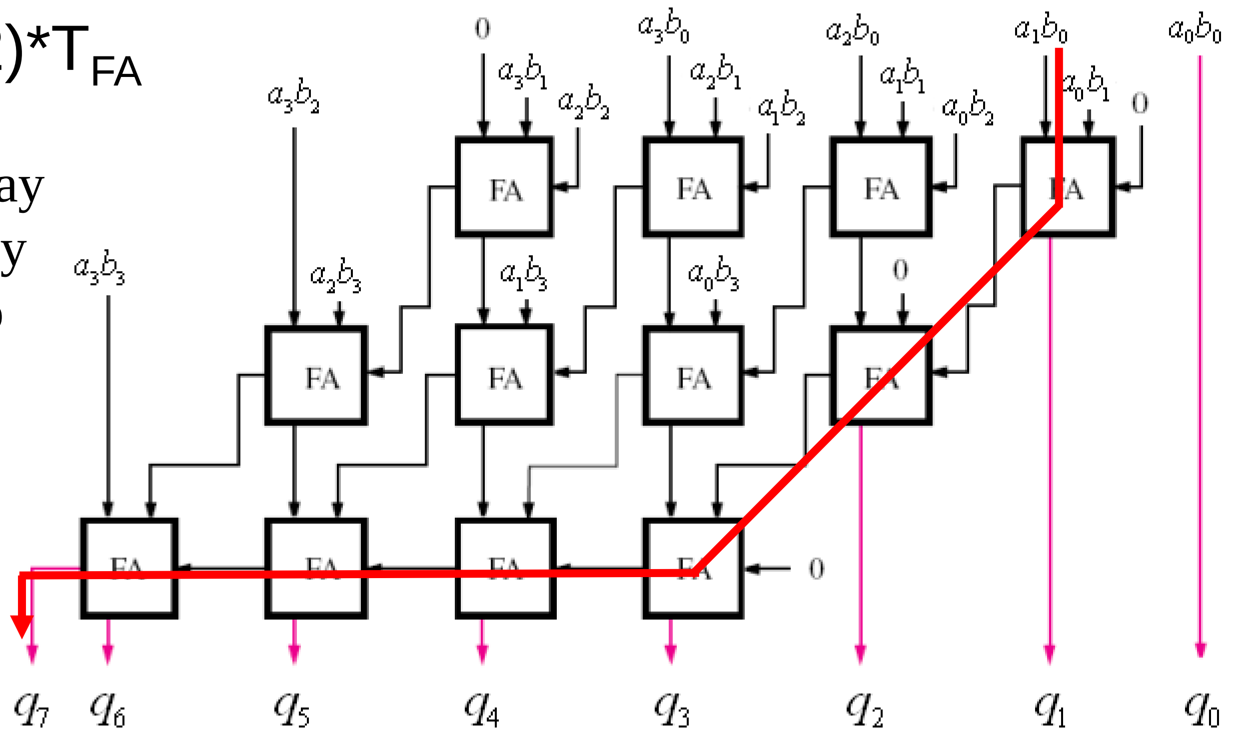


Figure 5.45 Multiplier carry-save array.

More details: (BV: page 311)

Multiplication by 2

0101 * 2 = 1010 (5 * 2 = 10)

1010 * 2 = 10100 (-6 * 2 = -12)

01010101 * 2 = 010101010 (85 * 2 = 190)

10010101 * 2 = 100101010 (-107 * 2 = -214)

Compare with multiplication by 10 in base 10:
63 * 10 = 630, -63 * 10 = -630 etc.

Multiplication by two is equivalent to shifting by 1 bit position left

Multiplication by 2^n

0101 * 2 = 1010 (5 * 2 = 10)
0101 * 2^2= 10100 (5 * 4 = 20)
0101 * 2^3= 101000 (5 * 8 = 40)
0101 * 2^4= 1010000 (5 * 16 = 80)

Compare with multiplication by 10 in base 10:

6 * 10 = 60, 6 * 100 = 600, 6 * 1000 = 6000, etc.

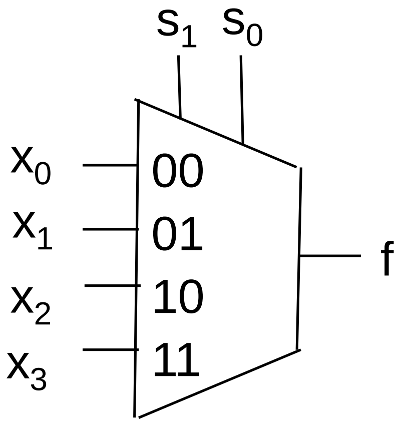
Multiplication by 2^n is equivalent to shifting by
n bit positions left

Multiplication by 2^n

- A multiplication by 2^n can be done by shifting all bits n positions to the left and filling in with zeros
- Calculate 13 * 8:
- 13 * 8 can be calculated by shifting (01011) three bits to the left
  - Result: 01011000 corresponds to $(104)_{10}$
  - Note that you need more bits to represent the result!

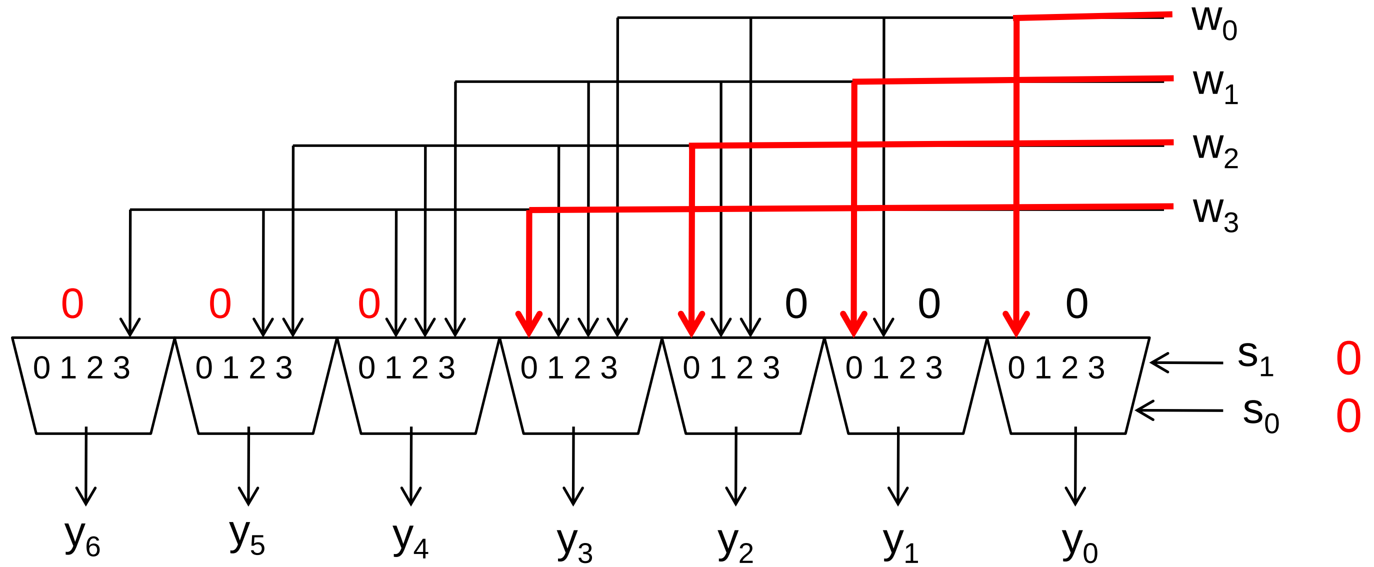
Implementation of left shift by barrel shifter

- The shifter circuit shown on the next slide shifts the bits of a 4-bit input vector to the left by 1, 2 or 3 bits (i.e. multiplies by $2^1, 2^2, 2^3$)
- It fills the vacated bits on the right side with 0
- 4-to-1 multiplexers are used in the circuit



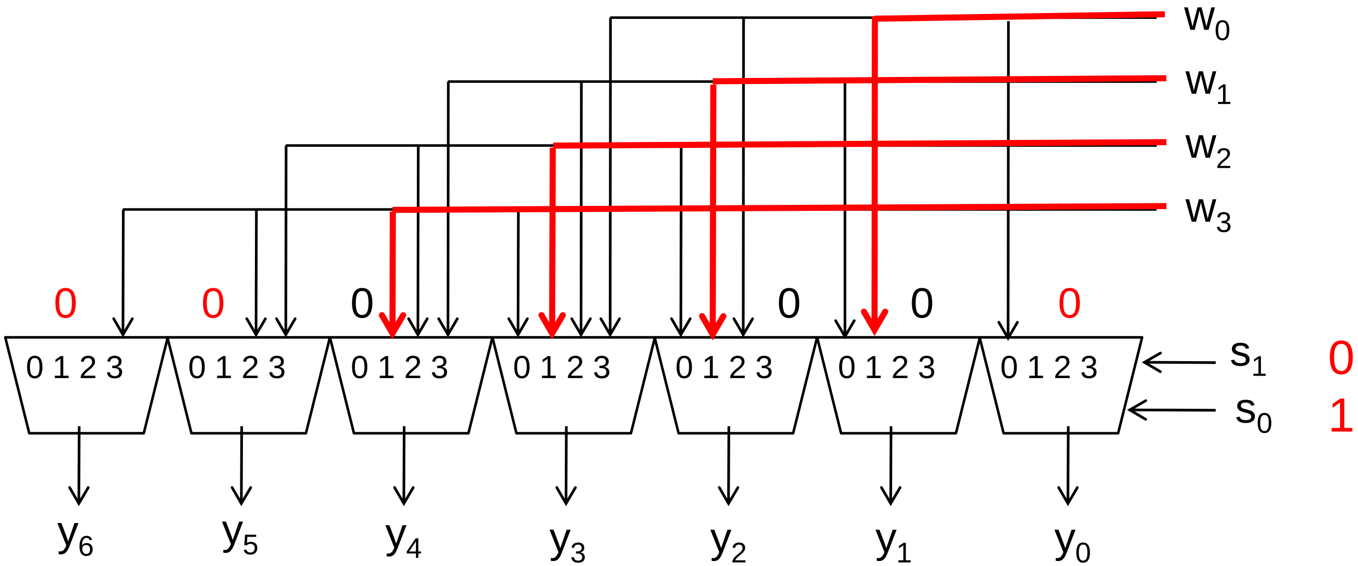
s_1	s_0	f
0	0	x_0
0	1	x_1
1	0	x_2
1	1	x_3

Shifter to the left



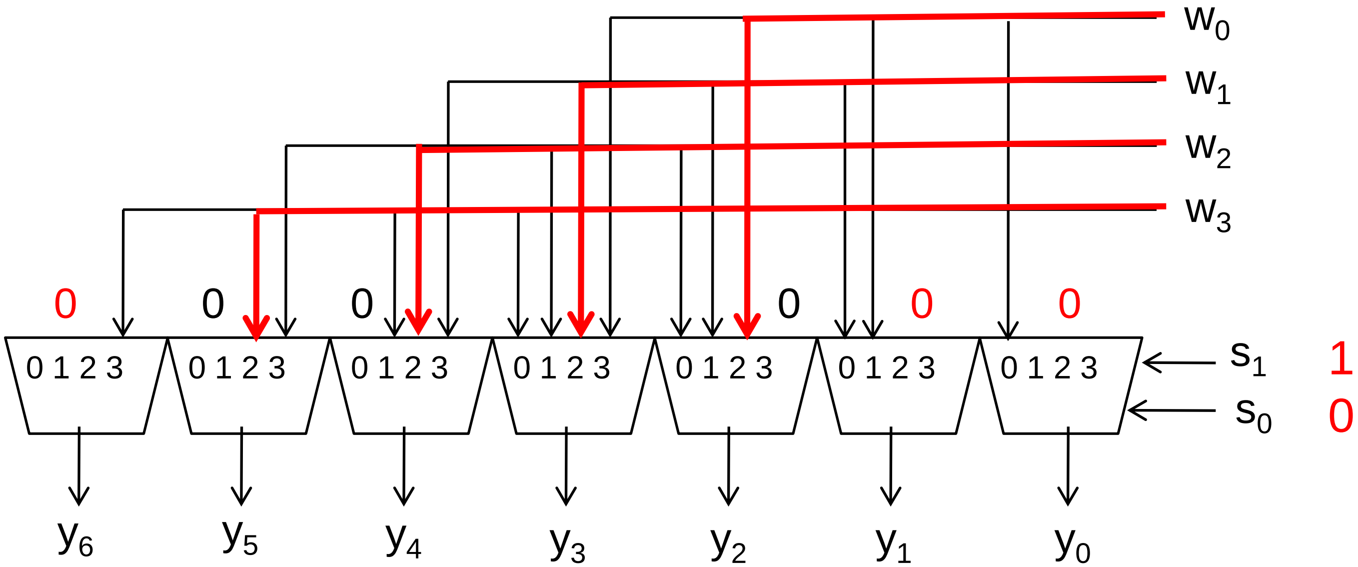
s_1	s_0	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	w_3	w_2	w_1	w_0
0	1	0	0	w_3	w_2	w_1	w_0	0
1	0	0	w_3	w_2	w_1	w_0	0	0
1	1	w_3	w_2	w_1	w_0	0	0	0

Shifter to the left



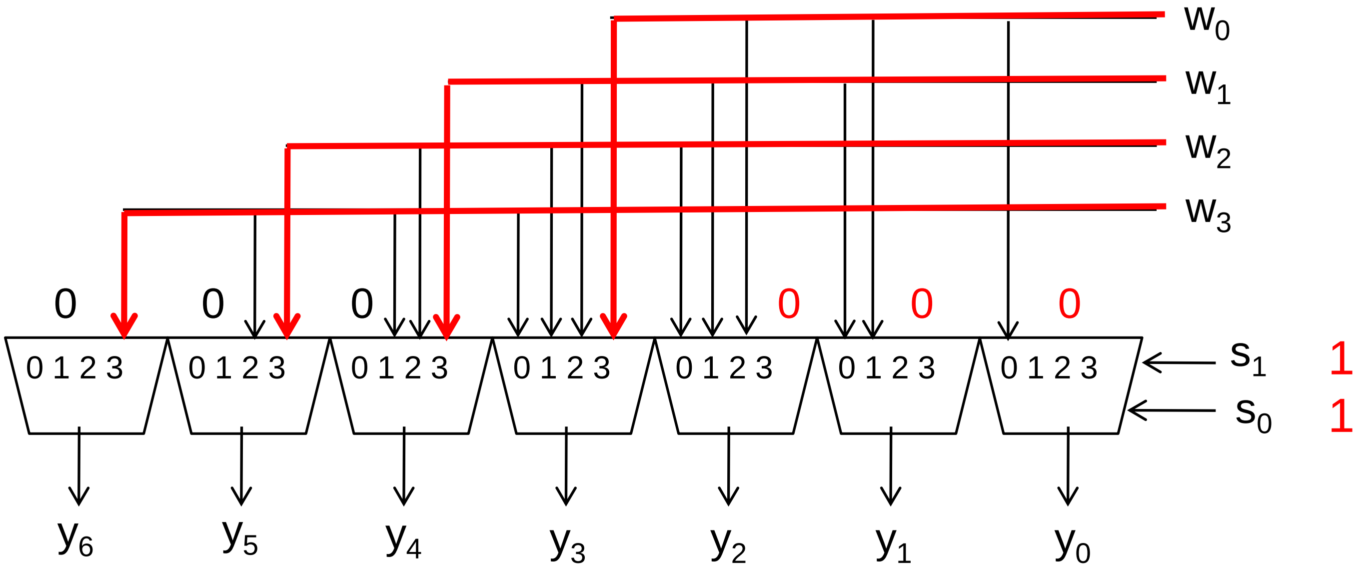
s_1 s_0	y_6 y_5 y_4 y_3 y_2 y_1 y_0
0 0	0 0 0 w_3 w_2 w_1 w_0
0 1	0 0 w_3 w_2 w_1 w_0 0
1 0	0 w_3 w_2 w_1 w_0 0 0
1 1	w_3 w_2 w_1 w_0 0 0 0

Shifter to the left



s_1	s_0	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	w_3	w_2	w_1	w_0
0	1	0	0	w_3	w_2	w_1	w_0	0
1	0	0	w_3	w_2	w_1	w_0	0	0
1	1	w_3	w_2	w_1	w_0	0	0	0

Shifter to the left



s_1 s_0	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0 0	0	0	0	w_3	w_2	w_1	w_0
0 1	0	0	w_3	w_2	w_1	w_0	0
1 0	0	w_3	w_2	w_1	w_0	0	0
1 1	w_3	w_2	w_1	w_0	0	0	0

Barrel shifter

- A barrel shifter places the shifted bits into the vacated position
- See VR p. 371, Fig 6.56

Division between two (positive) integer (BV p. 693 - Fig 10.21)

```
      15
   ______
9 ) 140
     9
    ___
     50
     45
    ___
      5
```

Using decimal numbers

```
                 00001111          ← Q
              ____________
B → 1001 )  100 01100              ← A
              1001
             _____
              10 001
                10 01
               ______
                10000
                  1001
                 _____
                   1110
                   1001
                  _____
                    101            ← R
```

Using binary numbers

A is dividend, B is divisor, Q = A/B is quotient, R is remainder

Example of division

Quotient

0 1 0 1

Divisor 1 0 | 1 0 1 1 Dividend

Example of division

```
              Quotient
               0 1 0 1       ←  11/2 = 5
Divisor   1 0 |1 0 1 1  Dividend
             - 1 0
               0 0 1
                - 0 0
                 0 1 1
                  - 1 0
                      1      ←  Reminder = 1
```

$$\frac{a}{b} = q + \frac{r}{b}$$

Example of division, cont.

```
          0 1 0 1
        _________
  1 0 | 0 1 0 1 1
    -   0 0
      ___________
          1 0
      -   1 0
        _________
          0 0 1
           -0 0
        _________
            0 1 1
             -1 0
            ______
                1
```

11/2 = 5

Reminder = 1

Example of division, cont.

$$\frac{a}{b} = q + \frac{r}{b} \qquad \frac{11}{2} = 5 + \frac{1}{2}$$

a:	0 1 0 1 1	r:	
b: 1 0		q:	

Example of division, cont.

$$\frac{a}{b} = q + \frac{r}{b} \qquad \frac{11}{2} = 5 + \frac{1}{2}$$

Dividend	a: 0 1 0 1 1	r:
Divisor	b: -1 0	q: 0

Example of division, cont.

$$\frac{a}{b} = q + \frac{r}{b} \qquad \frac{11}{2} = 5 + \frac{1}{2}$$

a: 0 1 0 1 1 r:

b: -1 0 q: 0 0

Example of division, cont.

$$\frac{a}{b} = q + \frac{r}{b} \qquad \frac{11}{2} = 5 + \frac{1}{2}$$

a:	0 1 0 1 1	r:	
b:	-1 0	q:	0 0 1

Example of division, cont.

$$\frac{a}{b} = q + \frac{r}{b} \qquad \frac{11}{2} = 5 + \frac{1}{2}$$

a: 0 0 0 1 1 r:

b: -1 0 q: 0 0 1 0

Example of division, cont.

$$\frac{a}{b} = q + \frac{r}{b} \qquad \frac{11}{2} = 5 + \frac{1}{2}$$

a:	0 0 0 1 1	r:	
b:	-1 0	q:	0 0 1 0 1

Example of division, cont.

$$\frac{a}{b} = q + \frac{r}{b} \qquad \frac{11}{2} = 5 + \frac{1}{2}$$

0 0 0 0 1 $\quad r$: 1

q: 0 0 1 0 1

Example of division, cont.

- Dividend $1011_2 = 11_{10}$
- Divisor $10_2 = 2_{10}$

Subtract/Restore method

- Quotient $0101_2 = 5_{10}$

$$\frac{a}{b} = q + \frac{r}{b} \qquad \frac{11}{2} = 5 + \frac{1}{2}$$

```
        0 0 1 0 1 1
      - 1 0
0 ← [-]     1
        0 1 0
      -   1 0
1 ← [+]   0 0
          0 0 1
        -   1 0
0 ← [-]       1
              0 1 1
            -   1 0
1 ← [+]           1
```

subtract

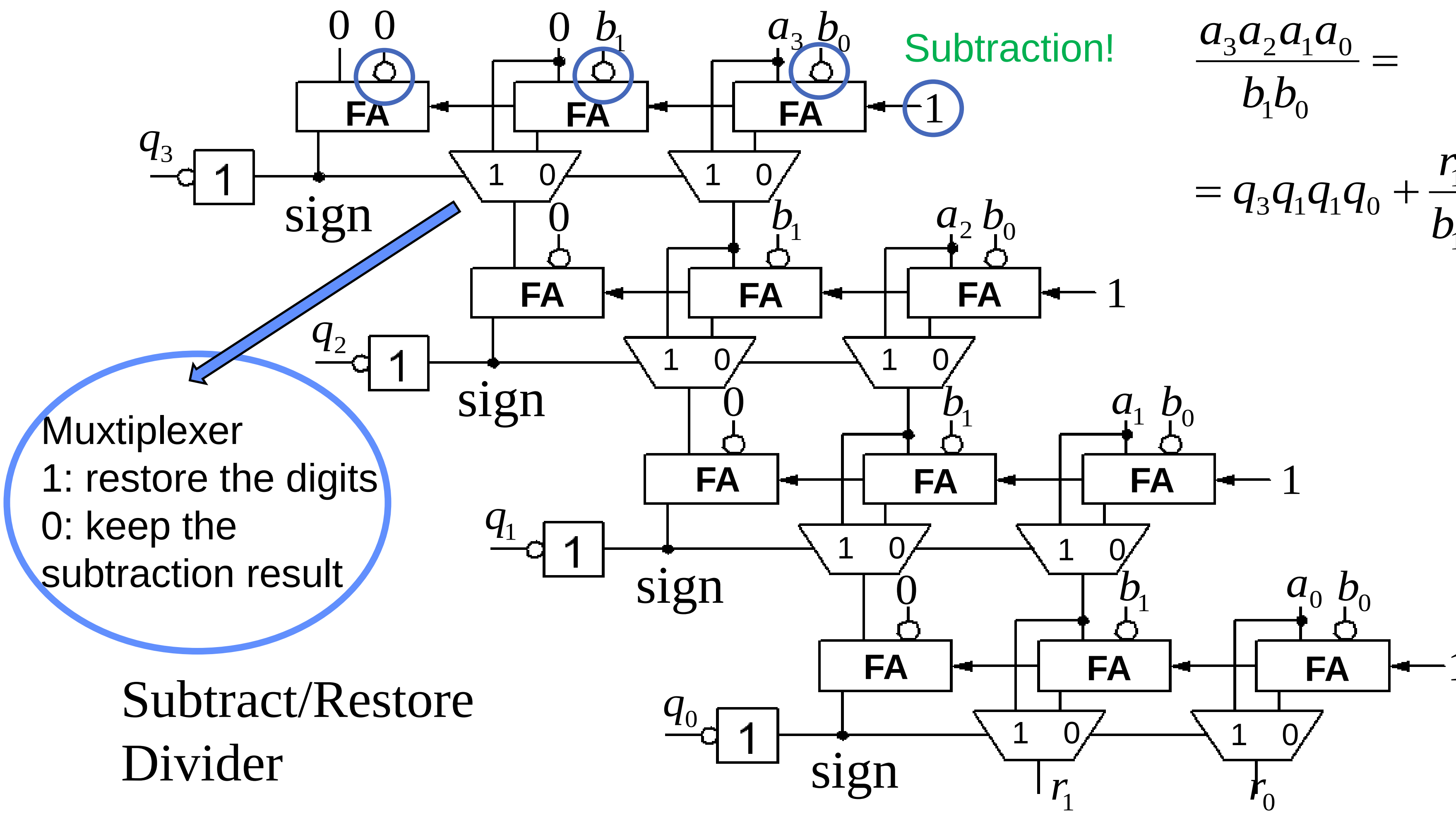
- → restore
subtract

+ → keep
subtract

- → restore
subtract

- Remainder 1

The divider circuit



$$\frac{a_3a_2a_1a_0}{b_1b_0} =$$

$$= q_3q_1q_1q_0 + \frac{r_1r_0}{b_1b_0}$$

Subtract/Restore Divider

Division with negative integers

- Division of negative numbers is quite tricky
- One way of performing the division is
  - Convert to positive numbers
  - Keep track of the result's sign

    (+ +) => +, (+, -) => -, (-, +) => -, (-, -) => +

  - Use 2's complement for negative numbers if necessary

Division by 2

01010/2 = 00101 (10/2 = 5)

10100/2 = 11010 (-12 / 2 = -6)

Compare with division by 10 in base 10:

630/10 = 63, -630 / 10 = -63, etc.

Logical vs Arithmetic right shifts

- There is a distinction between logical and arithmetic shifts
  - Logical right shift simply shift right. The bits should not be interpreted as a number. Therefore, the left side is just filled with 0's
  - Arithmetic right shift treats the bits as a number. The sign bit is retained. Therefore, the left side should be filled with the sign bit

Division by 2^n

$1010/2 = 101$ $(10/2 = 5)$
$10100/2^2 = 101$ $(20/4 = 5)$
$101000/2^3 = 101$ $(40/8 = 5)$
$1010000/2^4 = 101$ $(80/16 = 5)$

compare with division by 10 in base 10:

$60/10 = 6$, $600/100 = 6$, $6000/1000 = 6$, etc.

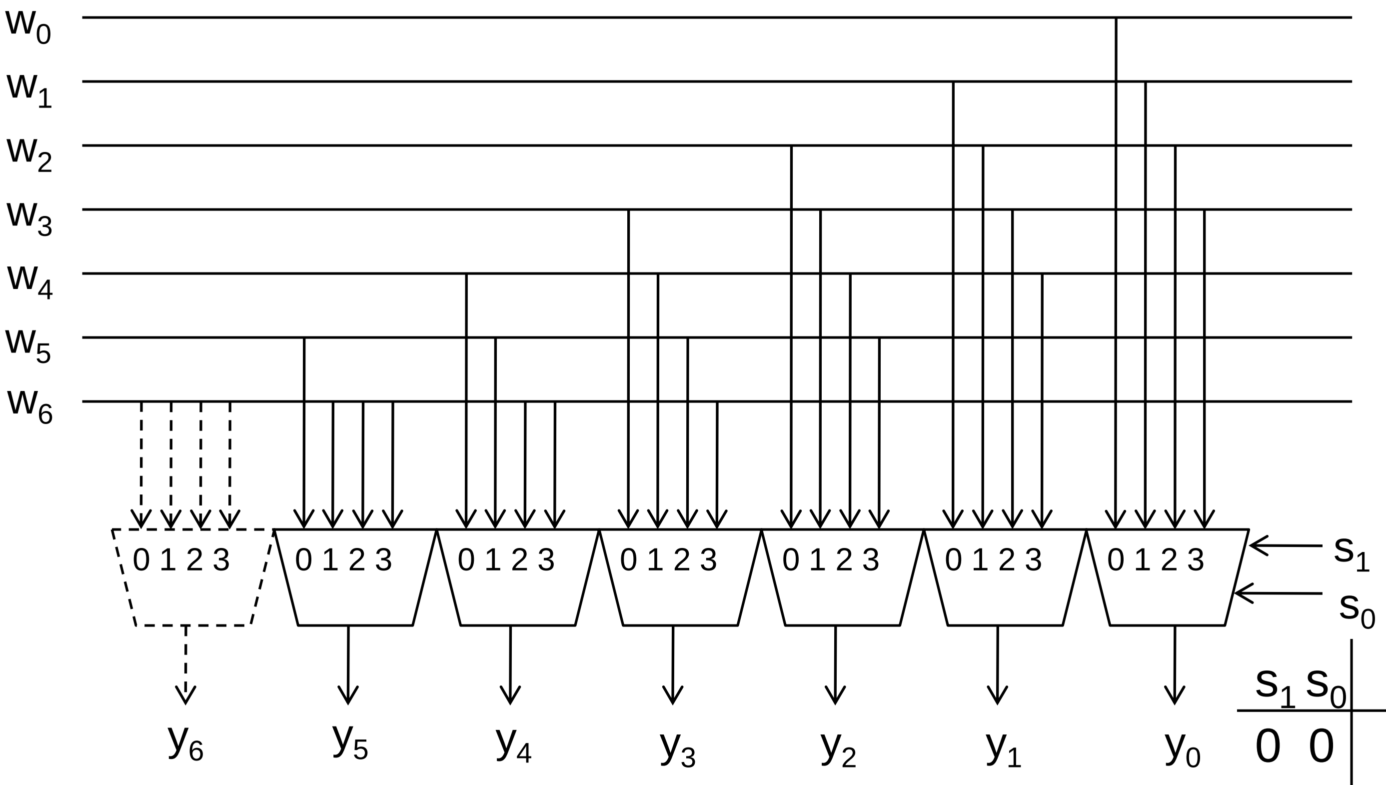
Division by 2^n

- A division by 2^n can be done by shifting all bits *n* steps to the right

- Note that the result may not be correct, as it really needs the bits "after the comma"

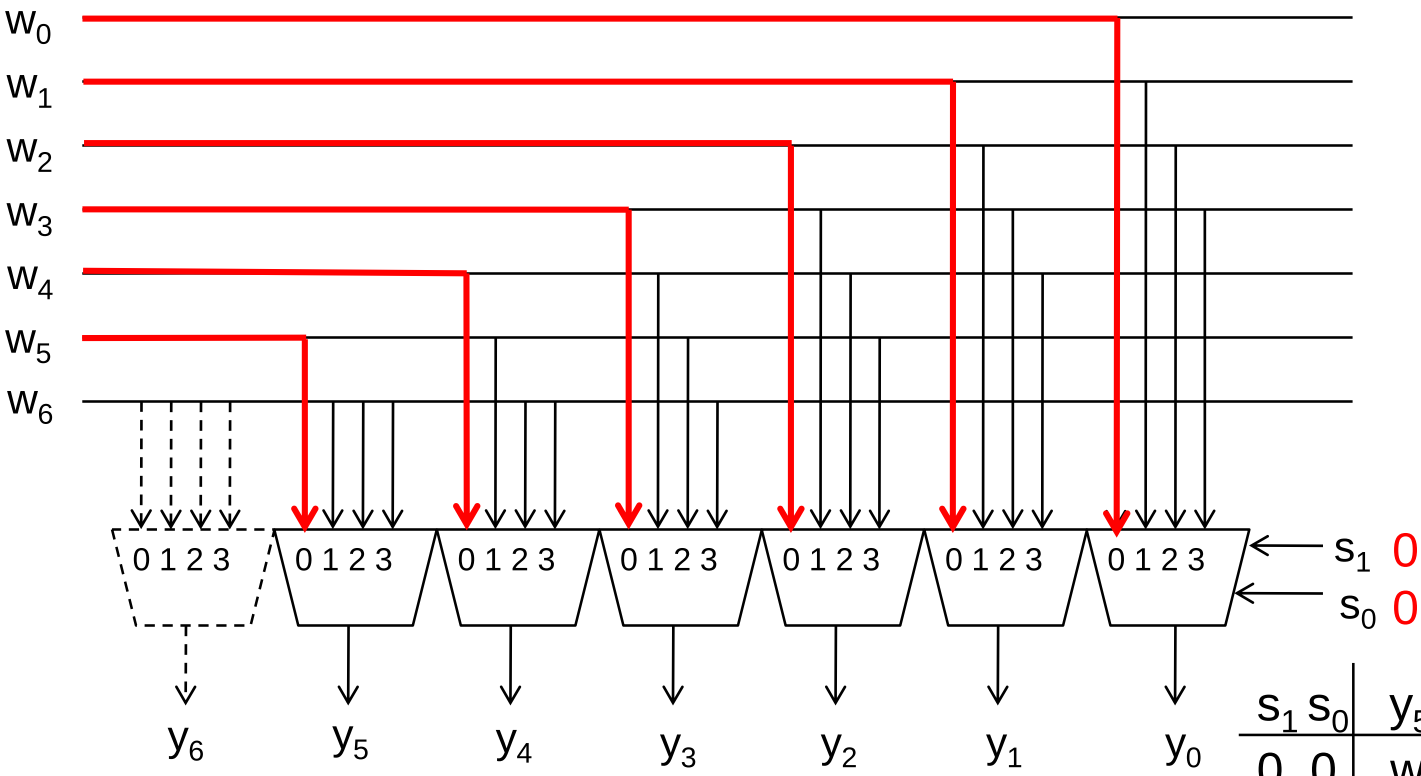
- 17/4 corresponds to shifting 010001 two bits to the right
  - Result: 000100 = $(4)_{10}$
  - Because $(0.25)_{10}$ can not be represented, the result is not accurate!

Shifter to the right



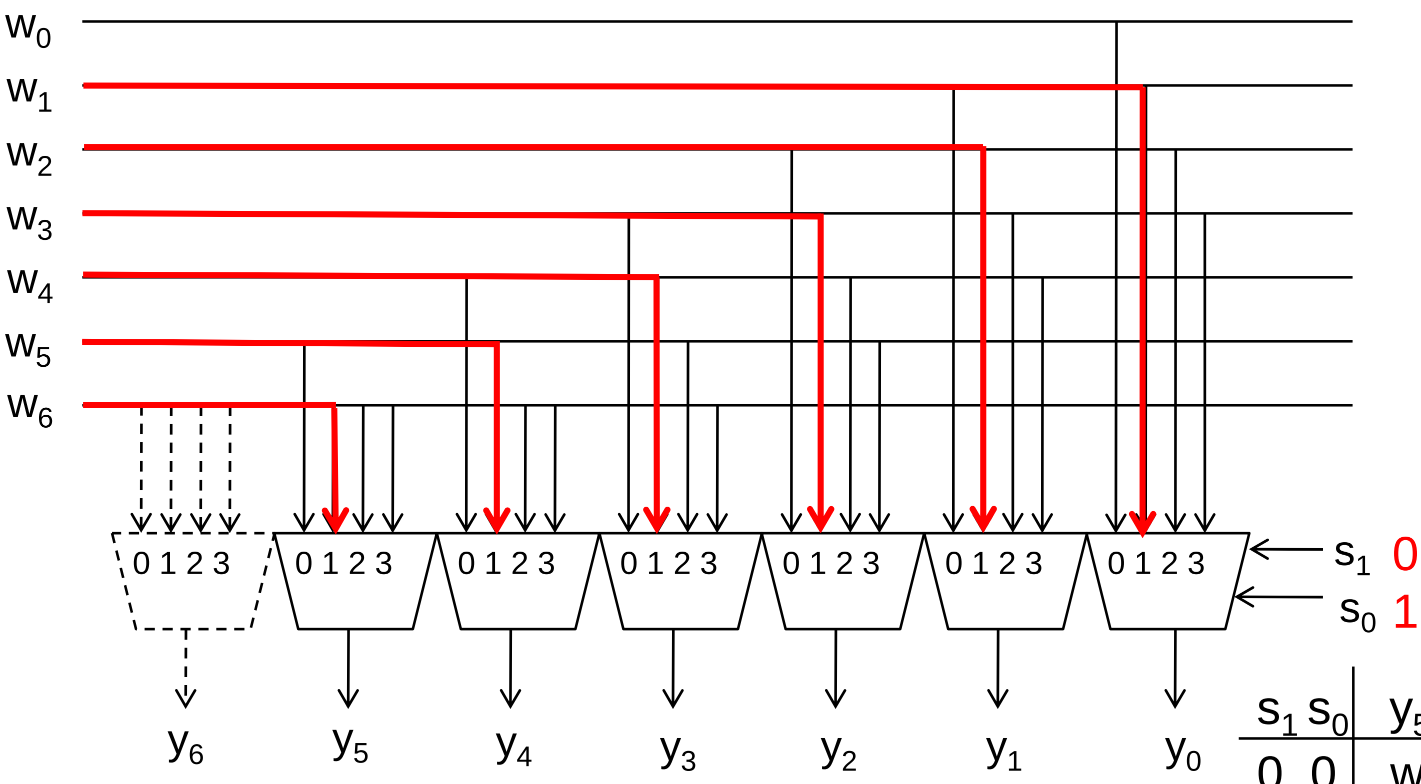
s_1 s_0	y_5 y_4 y_3 y_2 y_1 y_0
0 0	$w_5 w_4 w_3 w_2 w_1 w_0$
0 1	$w_6 w_5 w_4 w_3 w_2 w_1$
1 0	$w_6 w_6 w_5 w_4 w_3 w_2$
1 1	$w_6 w_6 w_6 w_5 w_4 w_3$

Shifter to the right



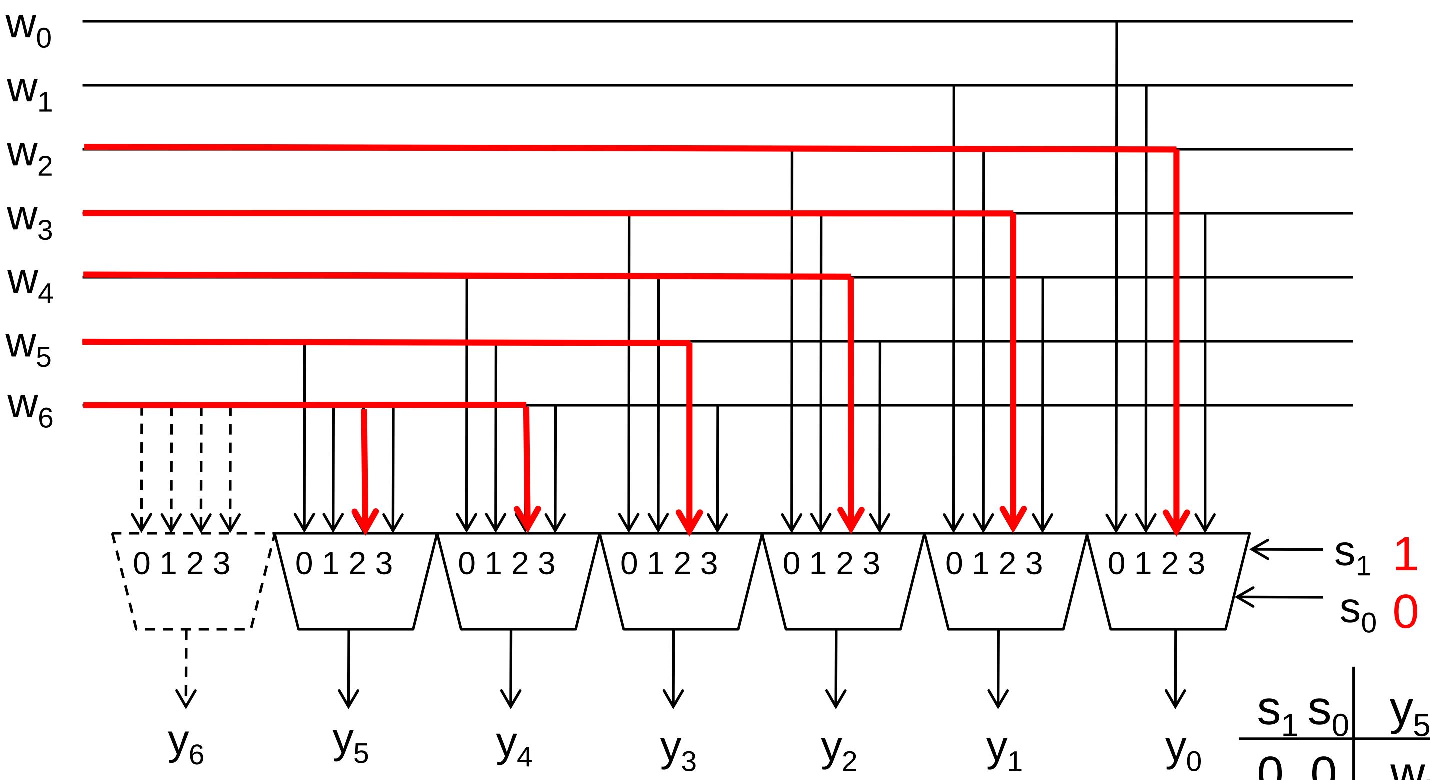
s_1 s_0	y_5 y_4 y_3 y_2 y_1 y_0
0 0	$w_5 w_4 w_3 w_2 w_1 w_0$
0 1	$w_6 w_5 w_4 w_3 w_2 w_1$
1 0	$w_6 w_6 w_5 w_4 w_3 w_2$
1 1	$w_6 w_6 w_6 w_5 w_4 w_3$

Shifter to the right



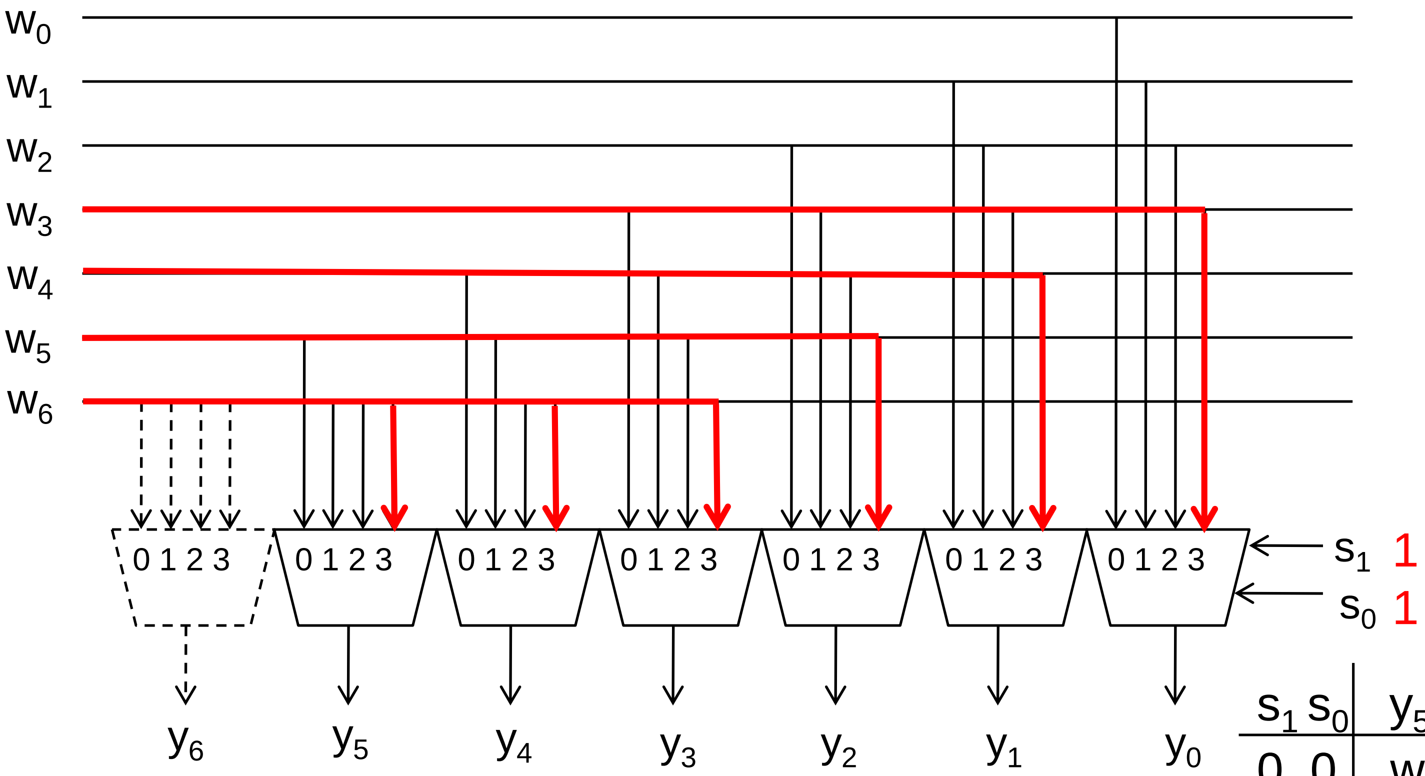
s_1 s_0	$y_5\,y_4\,y_3\,y_2\,y_1\,y_0$
0 0	$w_5w_4w_3w_2w_1w_0$
0 1	$w_6w_5w_4w_3w_2w_1$
1 0	$w_6w_6w_5w_4w_3w_2$
1 1	$w_6w_6w_6w_5w_4w_3$

Shifter to the right



s_1 s_0	y_5 y_4 y_3 y_2 y_1 y_0
0 0	$w_5w_4w_3w_2w_1w_0$
0 1	$w_6w_5w_4w_3w_2w_1$
1 0	$w_6w_6w_5w_4w_3w_2$
1 1	$w_6w_6w_6w_5w_4w_3$

Shifter to the right

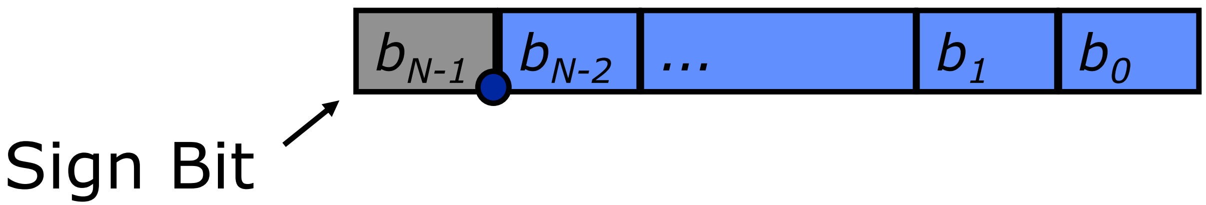


s_1 s_0	y_5 y_4 y_3 y_2 y_1 y_0
0 0	$w_5 w_4 w_3 w_2 w_1 w_0$
0 1	$w_6 w_5 w_4 w_3 w_2 w_1$
1 0	$w_6 w_6 w_5 w_4 w_3 w_2$
1 1	$w_6 w_6 w_6 w_5 w_4 w_3$

Fixed-point numbers

- A fixed-point number consists of integer and fraction parts (e.g. 0.11)
- It can be written as:

$$B = b_{N-1}\,.\,b_{N-2} \ldots b_1\, b_0 \quad \text{where } b_i \in \{0,1\}$$



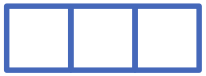
Decimal: $D = -b_{N-1}\,2^{0} + b_{N-2}\,2^{-1} + \ldots + b_1\,2^{-(N-2)} + b_0\,2^{-(N-1)}$

This format is also named Q_{N-1}*-format* or *fractional representation*

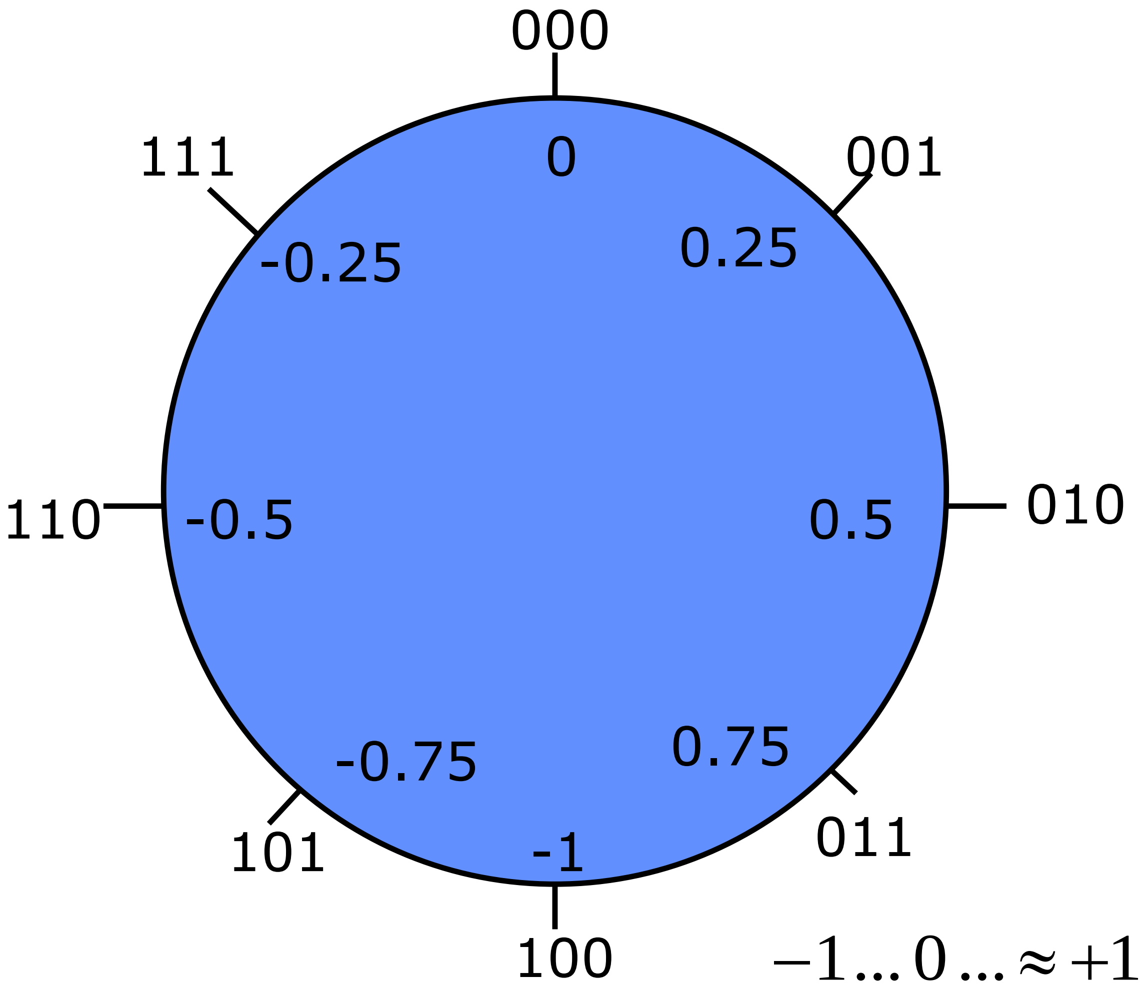
Operations on fixed-point numbers

- Logic circuits which deal with fixed-point numbers are essentially the same as those used for integers
- Fixed-point numbers have a range that is limited by the significant digits used to represent the number
  - i.e. if 8 bits and a sign bit are used to represent a fraction, then the range is 0.00000001 to $\pm$0.99999999
- In scientific computations it is often necessary to deal with very large numbers

Fixed-Point Representation



0.11	0,75
0.10	0,5
0.01	0,25
0.00	0
1.11	$-0,25$
1.10	$-0,5$
1.01	$-0,75$
1.00	-1.0



$$-1 \ldots 0 \ldots \approx +1$$

Floating-Point Numbers

- A floating-point number is represented by a *mantissa* (significant bits) and *an exponent of radix R*

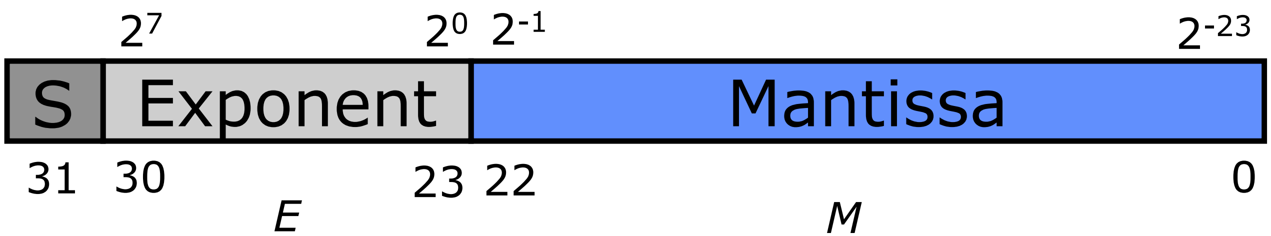
$$\mathit{Mantissa} \times R^{\mathit{Exponent}}$$

S	Exponent	Mantissa
Sign-Bit	*E*	*M*

- Numbers are often *normalized*, i.e. radix point is placed to the right of the 1st none-zero digit (5.213×10^{43})

IEEE-754

- Floating Point Standard IEEE-754 defines a 32-bit floating-number as:



- The value is calculated as:

  $V(B) = (-1)^s * (1.M) * 2^{E-(127)}$

  Example: $(-1)^1 * (1.011) * 2^2 = -101.1$

- Special bit patterns are reserved for representing zero and infinity

Single-precision floating point format IEEE

- Because it is necessary to represent both very large and very small numbers, the exponent can be either positive or negative
- Instead of simply using 8-bit signed numbers as an exponent (in the range -128 to 127), the IEEE standards specifies the exponent in the excess-127 format, where 127 is added to the value of the actual exponent

$$\text{Exponent} + 127 = E$$

- In this way E becomes a positive integer in the range 0 to 255

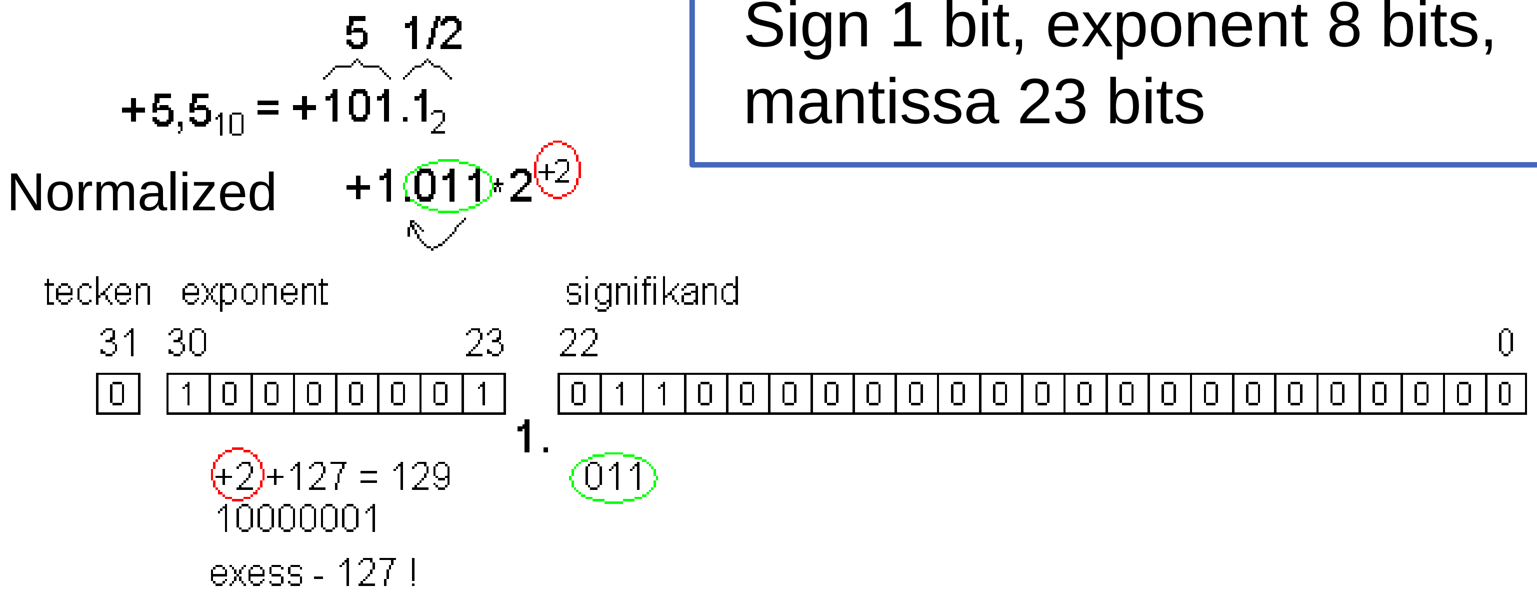
Single-precision floating point format IEEE, cont.

- The mantissa is represented using 23 bits
- The IEEE standard requires a normalized mantissa, which means that the MSB is always equal to 1

  => it is not necessary to include this bit in the mantissa field
- If M is the bit vector in the mantissa field, the actual value of the mantissa is 1.M, which gives a 24-bit mantissa
- This size of *<u>mantissa</u>* filed allows the representation of numbers that have the precision of about 7 decimal digits
- The *<u>exponent</u>* field range 2^{-126} to 2^{127} corresponds to about $10^{\pm 38}$

IEEE – 32 bit floating point format

Sign 1 bit, exponent 8 bits, mantissa 23 bits



Will be treated again in the course Computer Technology

Decimal addition example

$$\begin{array}{lll} & \text{normalized} & \text{aligned} \\ a = 123456.7 & = 1.234567 \cdot 10^5 & \\ b = 101.7654 & = 1.017654 \cdot 10^2 & = 0.001017654 \cdot 10^5 \end{array}$$

The **number which is smaller** (here **b**) is shifted (aligned) so that both numbers will have the ***same*** exponent

$$c = a + b$$

$$\begin{array}{r} 1.234567 \cdot 10^5 \\ +\ 0.001017654 \cdot 10^5 \\ \hline 1.235584654 \cdot 10^5 \end{array}$$

The result have to be normalized (shifted)

Floating point operations are very demanding, if you do not have specialized hardware - addition can be more demanding than multiplication!

Addition of floating-point numbers

- Given two floating-point numbers:

$$a = a_{frac} \cdot 2^{a_{\exp}}$$

$$b = b_{frac} \cdot 2^{b_{\exp}}$$

- The sum of these numbers is:

The number which is smaller is shifted

$$c = a + b$$

$$= \begin{cases} (a_{frac} + (b_{frac} \cdot 2^{-(a_{\exp} - b_{\exp})})) * 2^{a_{\exp}}, \text{if } a_{\exp} \geq b_{\exp} \\ (b_{frac} + (a_{frac} \cdot 2^{-(b_{\exp} - a_{\exp})})) * 2^{b_{\exp}}, \text{if } b_{\exp} \geq a_{\exp} \end{cases}$$

Subtraction of floating-point numbers

- Given two floating-point numbers:

$$a = a_{frac} \cdot 2^{a_{\exp}}$$

$$b = b_{frac} \cdot 2^{b_{\exp}}$$

- The difference between these numbers is:

$$c = a - b$$

The number which is smaller is shifted

$$= \begin{cases} (a_{frac} - (b_{frac} \cdot 2^{-(a_{\exp} - b_{\exp})})) * 2^{a_{\exp}} & \text{, if } a_{\exp} \geq b_{\exp} \\ (b_{frac} - (a_{frac} \cdot 2^{-(b_{\exp} - a_{\exp})})) * 2^{b_{\exp}} & \text{, if } b_{\exp} \geq a_{\exp} \end{cases}$$

Decimal multiplication example

The result has too many digits – round off

$$c = a \cdot b$$

$$a = 4{,}734612 \cdot 10^{3} \quad b = 5{,}417242 \cdot 10^{5}$$

$$c = \boxed{4{,}734612 \cdot 5{,}417242} \cdot 10^{3+5} = \boxed{25{,}6485}38980104 \cdot 10^{8}$$

$$c = 2{,}564854 \cdot 10^{9}$$

normalize

Multiplication involves:

- Addition of exponents
- Multiplication of fractional parts
- Normalization of the answer (shifting)

Multiplication of floating-point numbers

- Given two floating-point numbers:

$$a = a_{frac} \cdot 2^{a_{\exp}}$$

$$b = b_{frac} \cdot 2^{b_{\exp}}$$

- The product of these numbers is:

Much more simple!

$$c = a * b$$

$$= \left(a_{frac} * b_{frac} \cdot 2^{a_{\exp} + b_{\exp}}\right)$$

Division of floating-point numbers

- Given two floating-point numbers:

$$a = a_{frac} \cdot 2^{a_{\exp}}$$

$$b = b_{frac} \cdot 2^{b_{\exp}}$$

- The ratio of these numbers is:

$$\begin{aligned} c &= a / b \\ &= \left(a_{frac} / b_{frac} \cdot 2^{a_{\exp} - b_{\exp}} \right) \end{aligned}$$

Cleaning up after the floating point operations...

- When a floating-point operation is complete, it must be normalized
  – Mantissa is shifted until its first bit is 1
  – For each shift step, exponent is increased or decreased.
  – Mantissa's bits to the right of the first 1 are saved

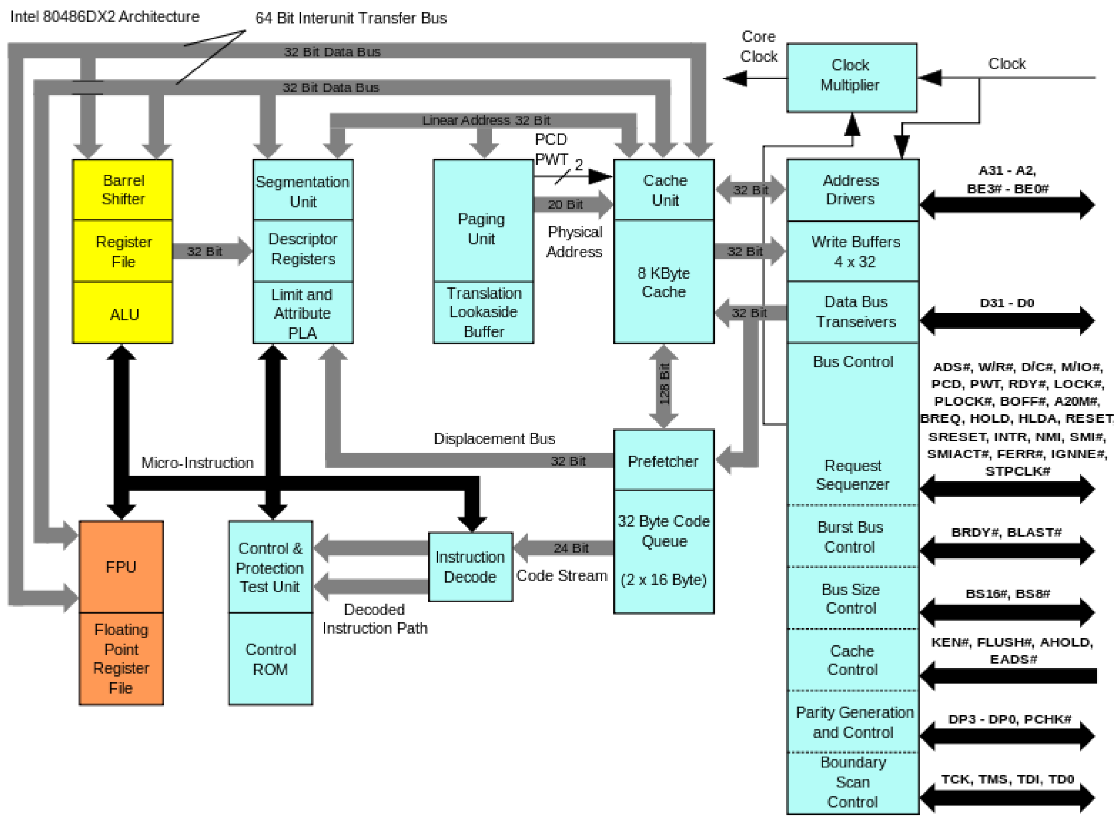
$$V(B) = (-1)^s * (1.M) * 2^{E-(127)}$$

  – If the exponent is zero, the first mantissa's bit is 0

$$V(B) = (-1)^s * (0.M) * 2^{-(126)}$$

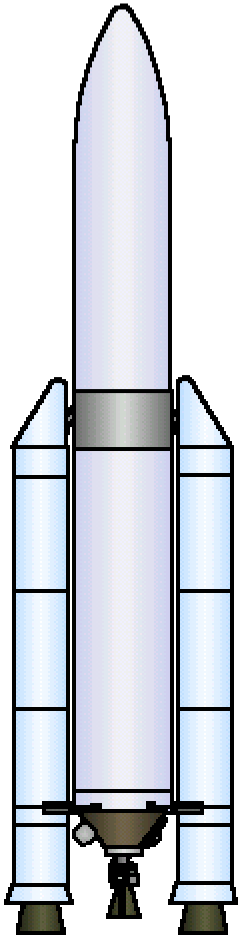
Floating Point Unit

It takes a lot of code and computation to perform floating point operations with a computer with no hardware support for this. PC computers have embedded floating point units since the processor 486 (1989).



Expensive software bug?

Ariane-5 rocket crashes at launch - 1996



`double` (64-bit float)

The translation was wrong – out of range!

`signed short int` (16-bit 2-complement)

Used in the system for horizontal bias

Fixed-Point vs. Floating-Point

- Fixed-Point operations work the same way as integer operations, and therefore fast
- The cost of hardware is significantly greater for floating point processors

Summary

- **Multiplication and division of integers**
  - Convert negative numbers to their positive counterparts
  - Perform multiplication or division
  - Keep track of the sign of the result
  - Convert positive result to its negative counterpart if the result should be negative
- **Multiplication by powers of 2 (by 2^k)**
  - Implemented as a shift to the left by k steps
- **Division by powers of 2 (div 2^k)**
  - Implemented as an (arithmetic) shift to the right by k steps. The sign bit is copied to the left.
- Next lecture: BV 318-339, 60-65, 280-291,341-365