Multiplication in binary

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

$$
\begin{array}{rl}
\mathbf{11010110} & A_7 \ldots A_0 \\
\mathbf{x00101101} & B_7 \ldots B_0 \\
\hline
\mathbf{11010110} & \\
\mathbf{000000000} & \\
\mathbf{1101011000} & \\
\mathbf{11010110000} & \\
\mathbf{000000000000} & \\
\mathbf{1101011000000} & \\
\mathbf{00000000000000} & \\
\mathbf{000000000000000} & \\
\hline
\mathbf{0010010110011110} & P_{15} \ldots P_0 \\
\hline
\end{array}
$$

- Note that the product *P* is composed purely of selecting, shifting and adding *A*. The *i*th column of *B* indicates whether or not a shifted version of *A* is to be selected or not in the *i*th row of the sum.

- So we can perform multiplication using just full adders and a little logic for selection, in a layout which performs the shifting.

Top

Notes:

Multiplication in decimal

Starting with an example in decimal:

$$
\begin{array}{r}
214 \\
\times 45 \\
\hline
1070 \\
+8560 \\
\hline
9630 \\
\hline
\end{array}
$$

Note that we do $214 \times 5 = 1070$ and then add to it the result of $214 \times 4 = 856$ *right-shifted by one column.*

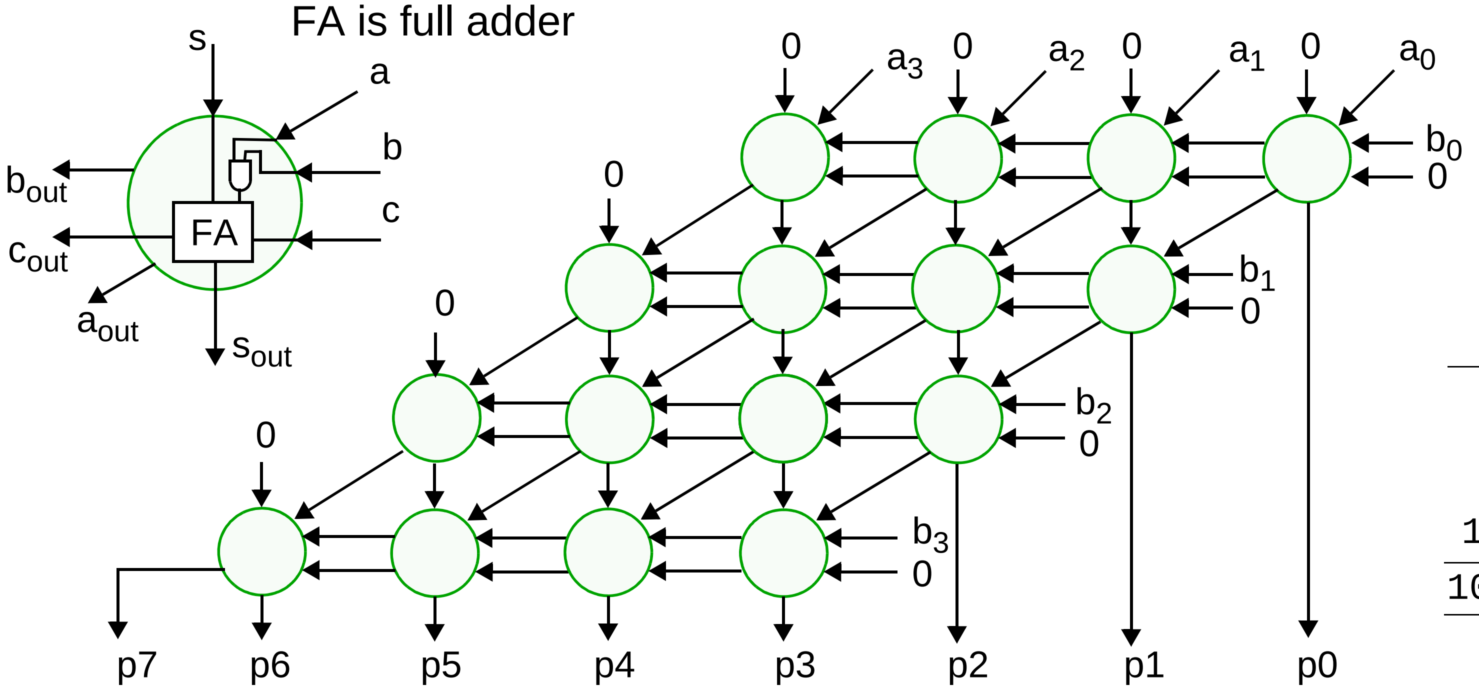
For each additional column in the second operand, we shift the multiplication of that column with the first operand by another place.

$$
\begin{array}{r}
zzz \\
x\mathit{aaaa} \\
\hline
\mathit{bbbb} \\
+\mathit{cccc}0 \\
+\mathit{dddd}00 \\
+\mathit{eeee}000
\end{array}
\qquad \text{etc...}
$$

Developed by: www.steepestascent.com

Structure for multiplication

- This figure shows a four-bit multiplication:



Example:

```
    1101    13
  × 1011    11
  --------
    1101
   1101
  0000
 1101
 --------
10001111   143
```

- The *AND* gate connected to *a* and *b* performs the selection for each bit. The diagonal structure of the multiplier implicitly inserts zeros in the appropriate columns and shifts the *a* operands to the right.

- Note that this structure does not work for signed two’s complement!

Notes:

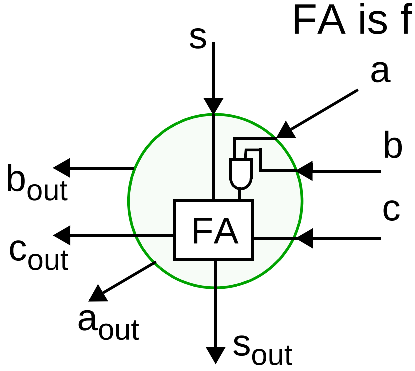
Note the function of the simple AND gate.

The operation of multiplying 1’s and 0’s is the same AND 1’s and 0’s

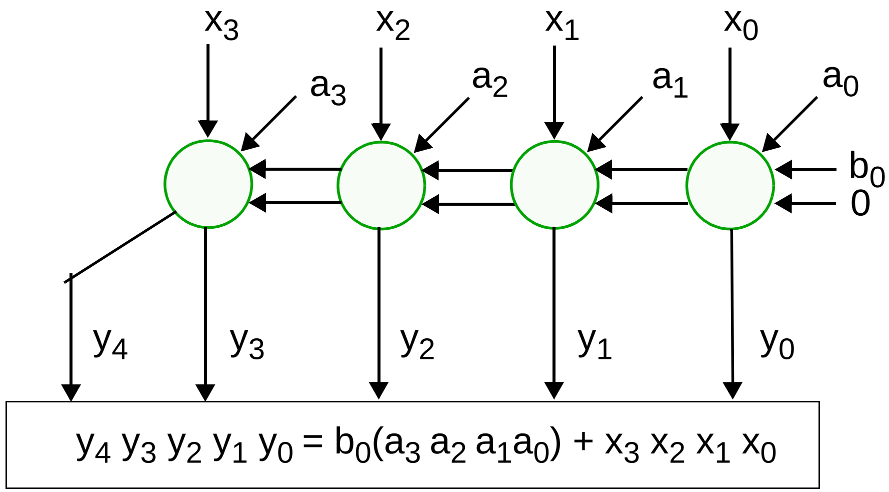
A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

Z = A x B (where x = multiply) or in Boolean algebra Z = A and B = AB

Hence the AND gate is the bit multiplier. The function of one partial product stage of the multiplier is as shown below.

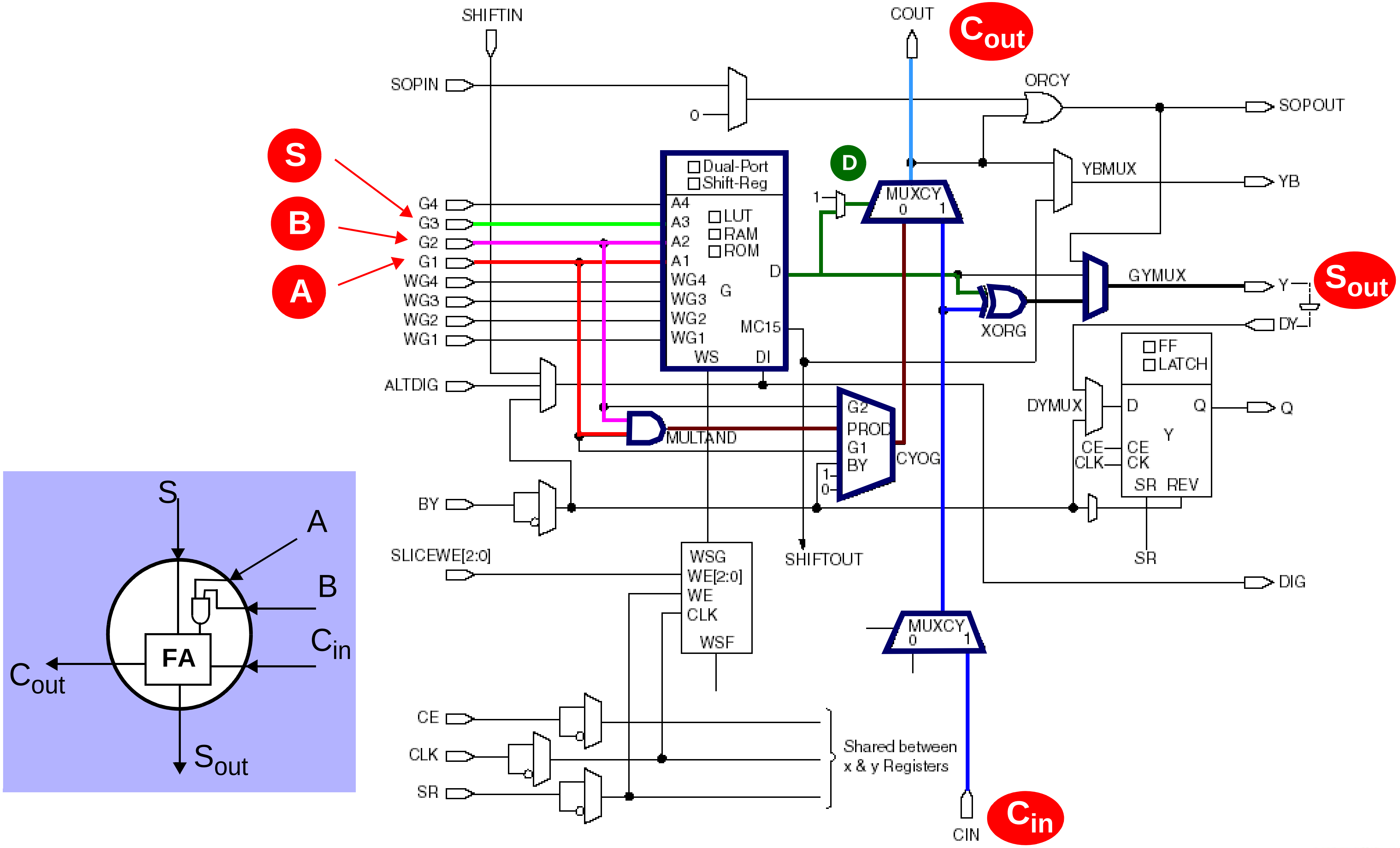


FA is full adder



$y_4\ y_3\ y_2\ y_1\ y_0 = b_0(a_3\ a_2\ a_1 a_0) + x_3\ x_2\ x_1\ x_0$

Xilinx Virtex-II Pro multiplication

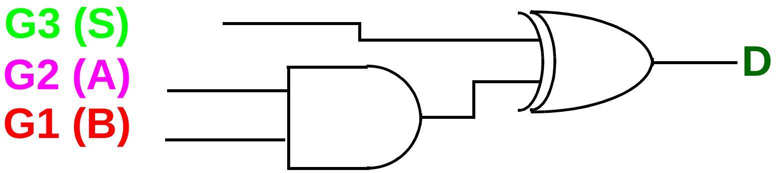


Top

Notes:

Picture of Xilinx-II Pro slice (upper half) taken from “Virtex-II Pro Platform FPGAs: Introduction and Overview”, DS083-1 (v2.4) January 20, 2003. http://www.xilinx.com

LUT implements the XOR of two ANDs:



The dedicated MULTAND unit is required as the intermediate product **G1G2** cannot be obtained from within the LUT, but is required as an input to MUXCY. The two AND gates perform a one-bit multiply each, and the result is added by the XOR plus the external logic (MUXCY, XORG):

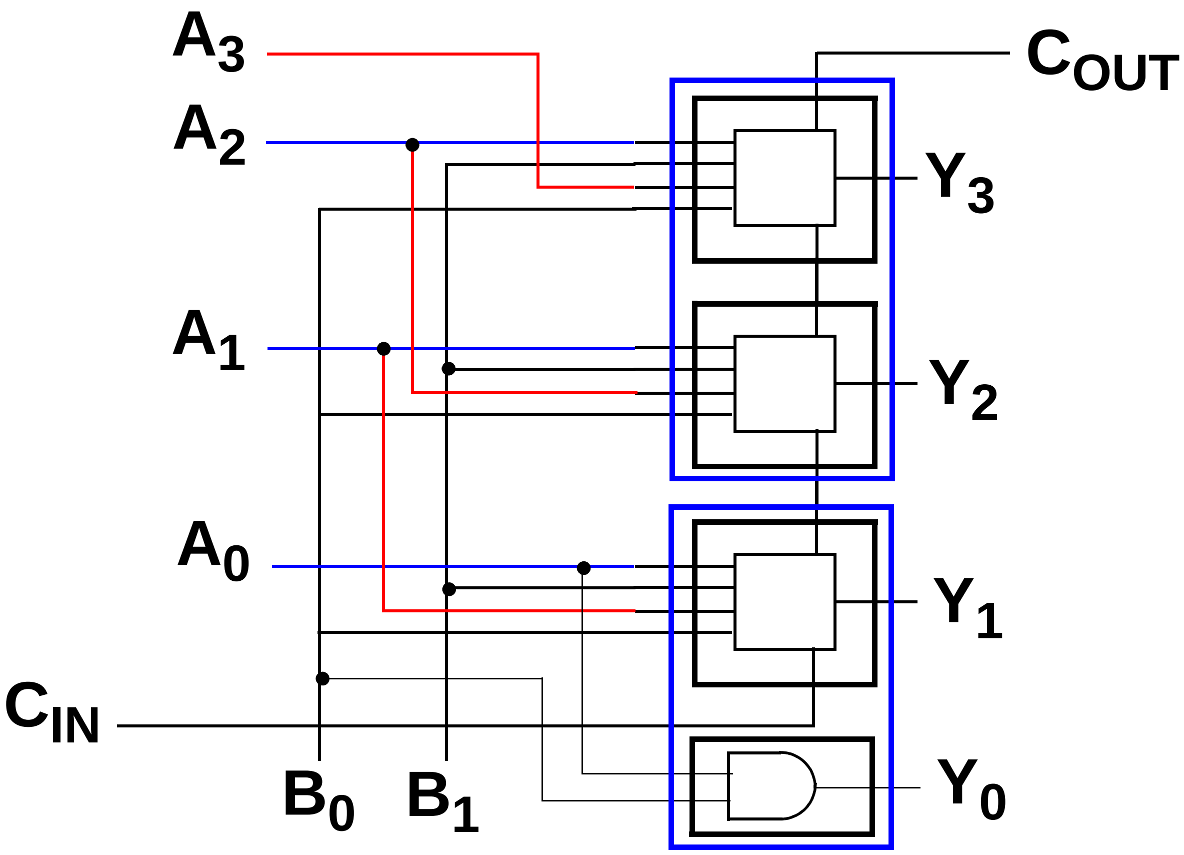
$\mathbf{S_{out}} = \mathbf{C_{IN}}$ xor $\mathbf{D}$, $\mathrm{C_{OUT}} = \mathbf{DAB} + \mathbf{C_{IN}\overline{D}}$

This structure will perform one cell (see below) of the multiplier.

Xilinx Virtex-II Pro multiplication (II)

- Multiplier cells as above can be chained to do bigger multiplies:

$$\begin{array}{r} A_1\,A_0 \\ \times\;B_1\,B_0 \\ C_{OUT}\qquad C_{IN} \\ \hline Y_3Y_2Y_1Y_0 \\ \hline \end{array}$$



Using 2 slices only

Top

Notes:

The first half of the truth table for $\boldsymbol{Y}$ and $\boldsymbol{C_{OUT}}$ (from Slide 3.31):

G1 (B_0)	G2 (A_1)	G3 (B_1)	G4 (A_0)	D	C_{IN}	Y	C_{OUT}
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	1	0	1	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	1	0	1	0
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	1	0	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	0
1	1	1	0	1	0	1	0
1	1	1	1	0	0	0	1

Xilinx Virtex-II Pro multiplication (VI)

- As we can do one bit of a multiply in a slice, we can do an N-bit by 2-bit multiply in N/2 slices. In the example above, we have 4-bit by 2-bit in 2 slices.
- Perhaps the most important thing to note is that this is very complicated!
- Tools are designed to automate the process of connecting the components within a slice in order to perform efficient operations.
- But it is important to note that the tools aren’t infinitely clever, and sometimes we need to bear in mind the structure of the FPGA in order to generate an efficient design.

Top

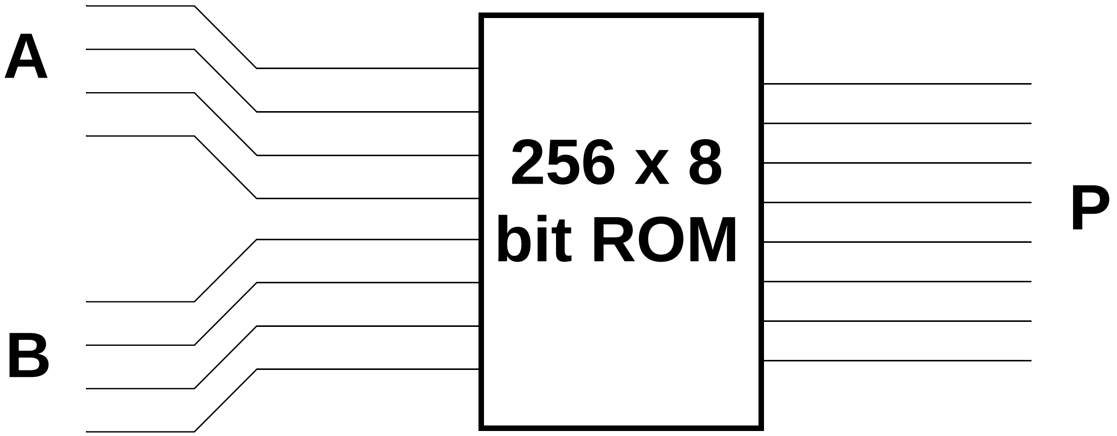
Notes:

The second half of the truth table for *Y* and C_{OUT} (from Slide 3.31):

G1 (B_0)	G2 (A_1)	G3 (B_1)	G4 (A_0)	D	C_{IN}	Y	C_{OUT}
0	0	0	0	0	1	1	0
0	0	0	1	0	1	1	0
0	0	1	0	0	1	1	0
0	0	1	1	1	1	0	1
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	1
1	0	0	0	0	1	1	0
1	0	0	1	0	1	1	0
1	0	1	0	0	1	1	0
1	0	1	1	1	1	0	1
1	1	0	0	1	1	0	1
1	1	0	1	1	1	0	1
1	1	1	0	1	1	0	1
1	1	1	1	0	1	1	1

ROM-based multipliers

- Just as logical functions such as XOR can be stored in a LUT as shown for addition, we can use storage-based methods to do other operations.
- By using a ROM, we can store the result of every possible multiplication of two operands.
- The two operands are concatenated to be used as the address by which to access the ROM.
- The value stored at that address is the multiplication result:



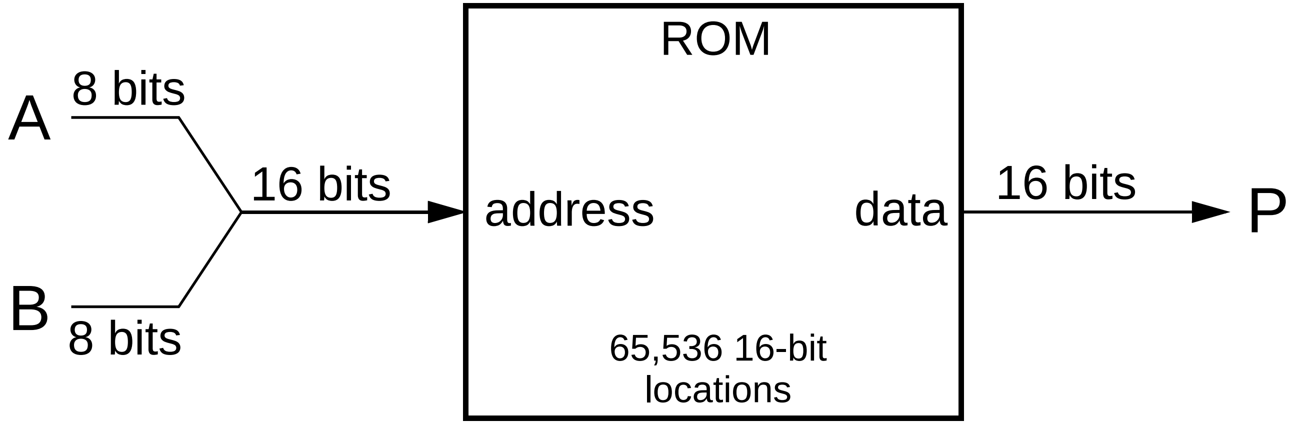
Top

Notes:

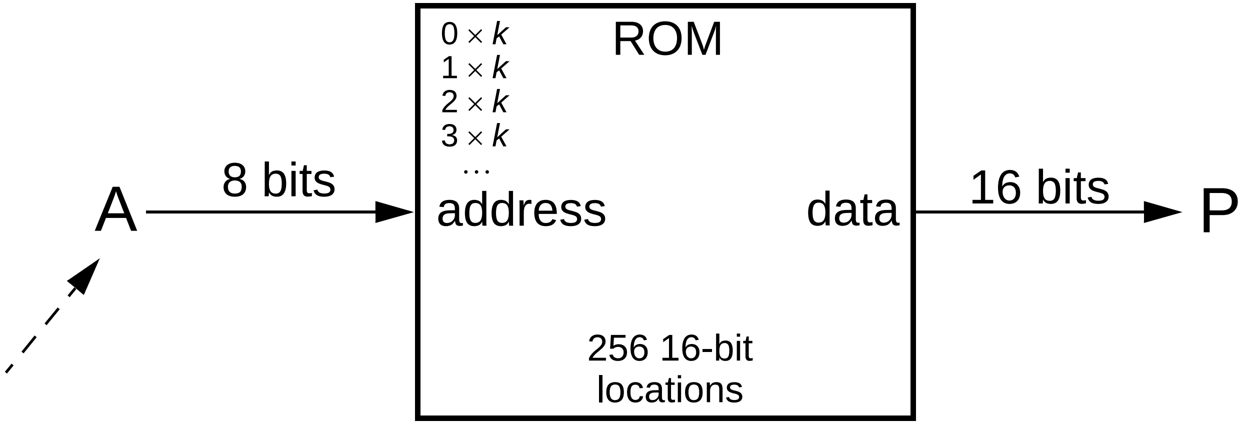
There is one serious problem with this technique: as the operand size grows, the ROM size grows exponentially. For two N bit input operands (therefore an 2N bit output operand) $2N2^{2N}$ bits of storage are required. For example, with 8 bits operands (a fairly reasonable) size, 1Mbit of storage is required - a large quantity. For bigger operands e.g. 16 bits, a huge quantity of storage is required. 16 bit operands require 128Gbits of storage!

Constant ROM-based multipliers

- Consider a ROM multiplier with 8 bit inputs: 65,536 8-bit locations are required



- If input **B** is **constant** and B $= k$ **only 256 locations** are **accessed**



input B removed

- This constitutes a **Constant Coefficient Multiplier (KCM)**

Top

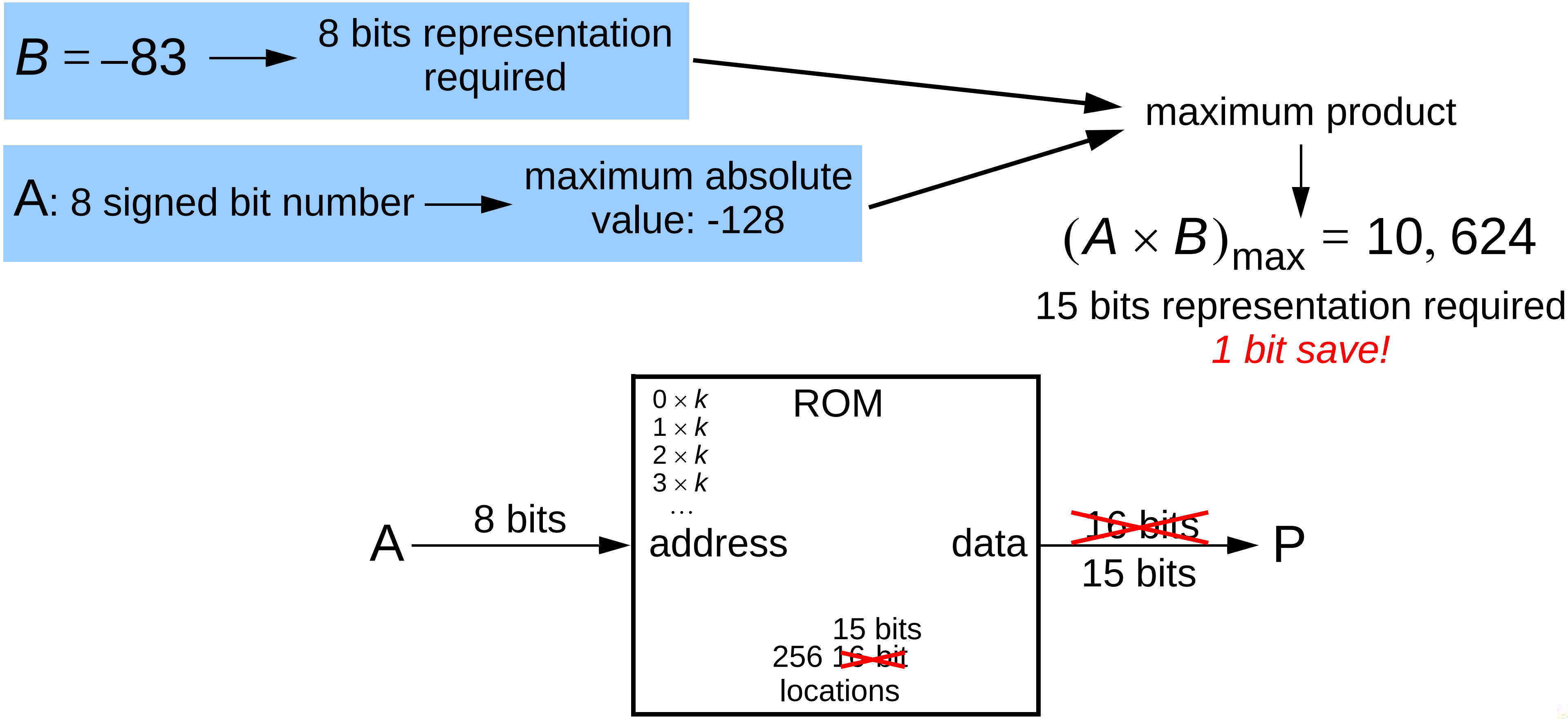
Notes:

In the above example, 8-bit input B is fixed to one value. Which means that in total only 256 out of a total of 65,536 locations are accessed. Therefore, when one of the inputs of the ROM-based multiplier is fixed the size of the required ROM can be reduced.

It is also possible to reduce the memory requirements of this structure if additional knowledge of the constant value is available. For example, if the value of B is 10, the maximum output required for any 8-bit input A will be $-128 \times 10 = -1280$, which can be represented with 12 bits.

Constant Coefficient Multiplier (KCM)

- ROM-based multipliers with a constant input
- This reduces the size of the required ROM
- Further reductions in size requirement can be made if there is knowledge of the constant value



$B = -83$ → 8 bits representation required

A: 8 signed bit number → maximum absolute value: -128

maximum product

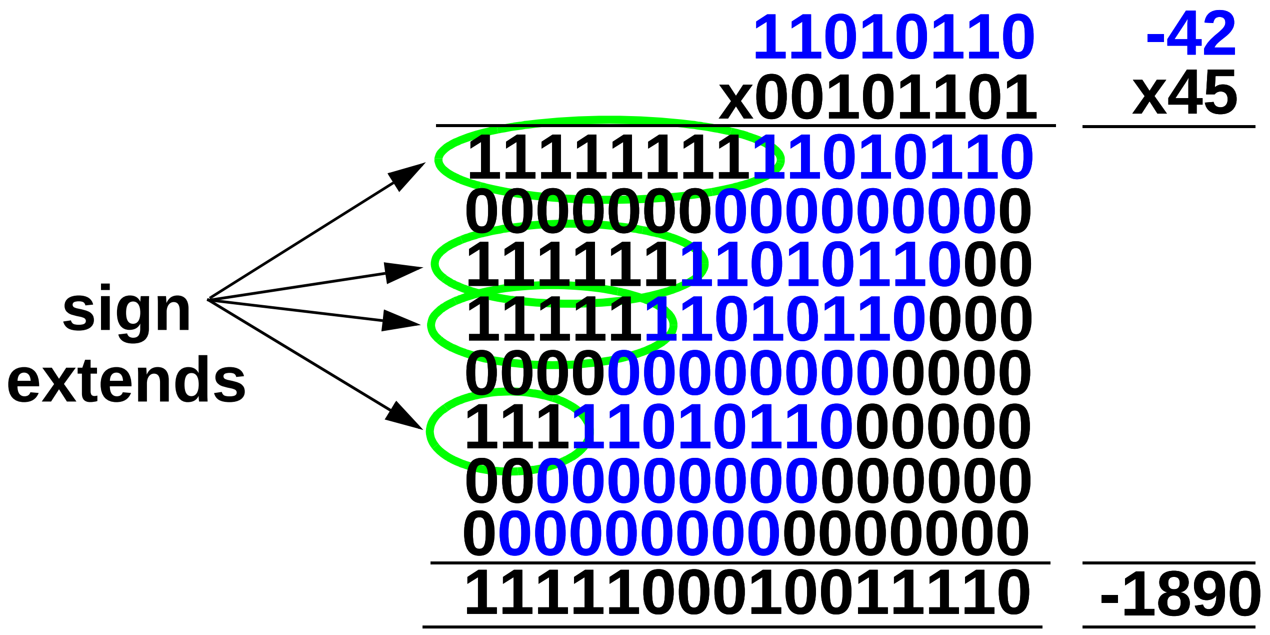
$$(A \times B)_{max} = 10,624$$

15 bits representation required

1 bit save!

2’s complement Multiplication

- For one negative and one positive operand just remember to sign extend the negative operand.



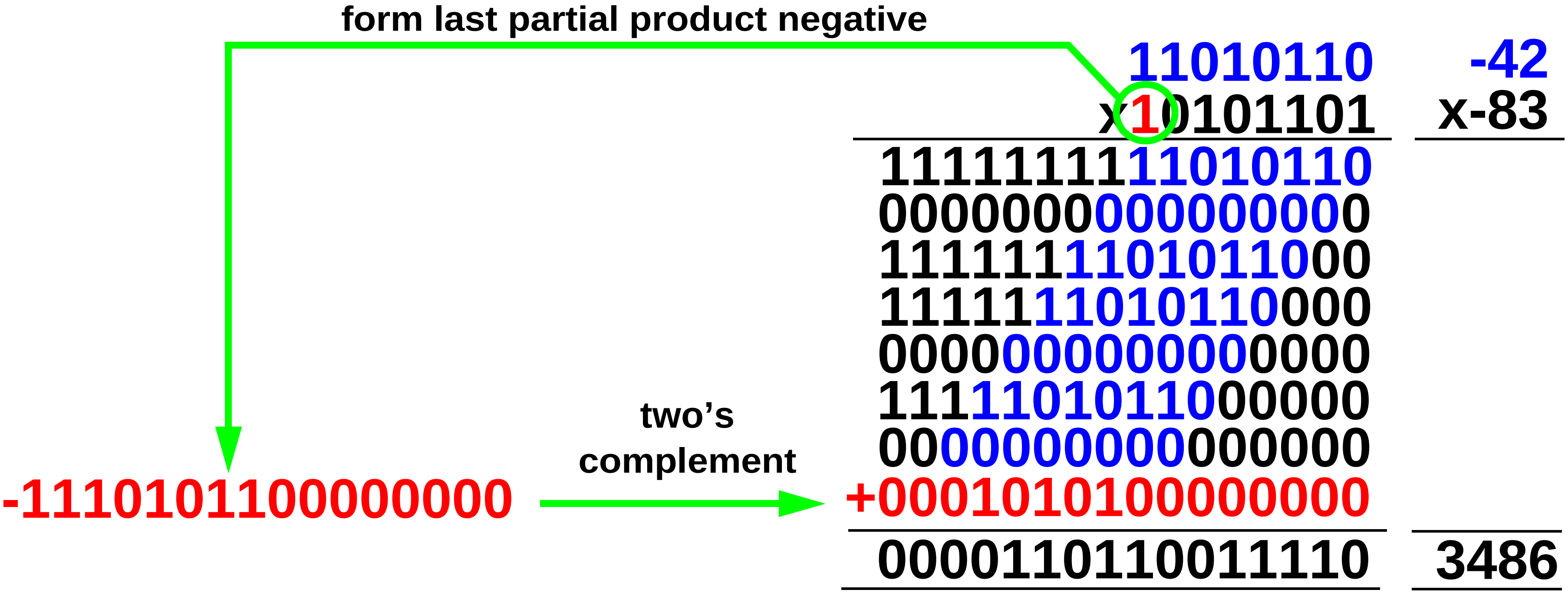
```
        11010110      -42
       x00101101      x45
1111111111010110
0000000000000000
1111111101011000
1111111010110000
0000000000000000
1111101011000000
0000000000000000
0000000000000000
1111100010011110    -1890
```

Notes:

2’s complement multiplication (II)

For both operands negative, subtract the last partial product.

We use the trick of inverting (negating and adding 1) the last partial product and adding it rather than subtracting.



form last partial product negative

```
          11010110     -42
         x10101101     x-83
  1111111111010110
  0000000000000000
  1111111101011000
  1111111010110000
  0000000000000000
  1111101011000000
  0000000000000000
 +0001010100000000
  0000110110011110     3486
```

-1110101100000000

two’s complement

Of course, if both operands are positive, just use the unsigned technique!

The difference between signed and unsigned multiplies results in different hardware being necessary. DSP processors typically have separate unsigned and signed multiply instructions.

Fixed Point multiplication

- Fixed point multiplication is no more awkward than integer multiplication:

```
         11010.110
        x00101.101
  ----------------
         11.010110
        000.000000
       1101.011000
      11010.110000
     000000.000000
    1101011.000000
   00000000.000000
  000000000.000000
  ----------------
 0010010110.011110
 -----------------
```

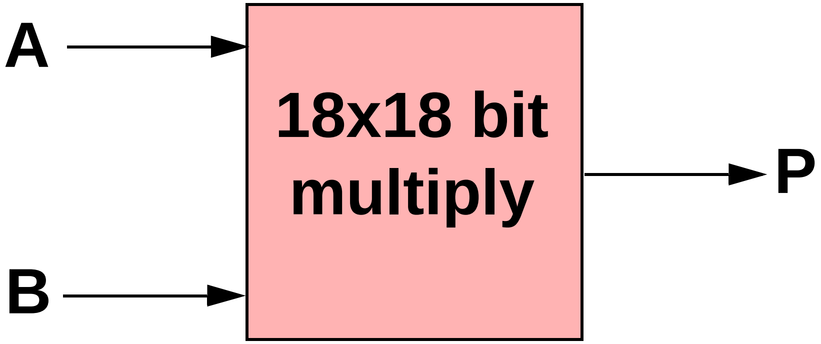
```
      26.750
      x5.625
  ----------
    0.133750
    0.535000
   16.050000
  133.750000
  ----------
  150.468750
  ----------
```

- Again we just need to remember to interpret the position of the binary point correctly.

On-chip multipliers

- The Xilinx Virtex-II Pro FPGA has a set of “on-chip” multipliers.

- These are in hardware on the ASIC, not actually in the user FPGA area, and therefore are permanently available, and they use no slices. They also consume less power than a slice-based equivalent.



- A and B are 18-bit input operands, and P is the 36-bit product $P = A \times B$.

- Depending upon the particular device, between 12 and 512 of these dedicated multipliers are available.