Name: Honnet
Student n°: 0531984

EE2071 Micro electronic workshop: *RTL systolic multiplier*

Example of chip to implement our multiplier

0 Purpose:

This laboratory report is the result of our introduction to the principle of RTL design (Register Transfer Level) in Verilog HDL (Hardware Description Language). The objective is to design a systolic multiplier in Verilog using the Simucad Silos® software. We are going to meet this challenge by firstly designing an HDL description of an architectural version and finally every each module that composes this multiplier from the two 8bits inputs to the 16bits output (without forgetting all the other signal pins). As this multiplier is designed for digital signal processing algorithms, it has to be able to load firstly 2 inputs and then keep 1 input to multiply different values to it, but more details will be given later.

In a first time we are going to look for diverse numbering systems to choose the best design. We are then going to have a look on different multiplier designs to satisfy the requirements. The chosen design will be simply explained and tested bloc by bloc to finally implement and simulate the overall instantiation of all the internal components.

1 Structure of the assignment:

- ✓ Introduction, structure.
- ✓ Numbering systems.
- ✓ Multiplications algorithms.
- ✓ Examples of a few multipliers.
- ✓ Requirement specification.
- ✓ Verilog description of the selected multiplier.
- ✓ Description of the results.
- ✓ Conclusion.

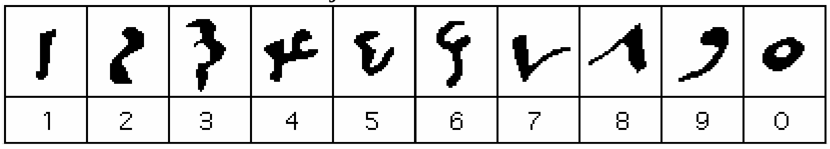
2 Numbering systems:

First of all, let's see have a look on a few number classification. There are two principal notations, the positional and non-positional system (I'm not going to investigate the non positional system). The Babylonians developed the positional system (or place-value system) based essentially on the numerals for 1 and 10. The Egyptians had a system of numerals with distinct hieroglyphs for 1, 10, and all the powers of 10 up to one million. We can see on the following table illustration of the idea of position system:

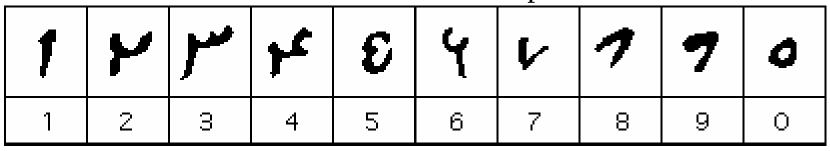
Position	3	2	1	0	-1	-2	...
Weight	b^3	b^2	b^1	b^0	b^{-1}	b^{-2}	...
Digit	a_3	a_2	a_1	a_0	c_1	c_2	...
Decimal example weight	1000	100	10	1	0.1	0.01	...

As everyone knows the numbers commonly used were invented by Arabs but the representation of the ones used nowadays has had an evolution:

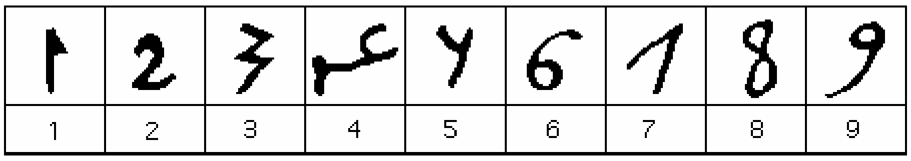
The numerals from al-Sizji's treatise of 969:



The numerals from al-Biruni's treatise copied in 1082:



Al-Banna al-Marrakushi's form of the numerals:



It's known that several numeral bases exist and we don't think about it but almost everybody use 3 of them every day. The most commonly used is obviously the base 10 (called base decimal). The question "why 10?" could be asked and the answer is as simple as the numbers of our fingers.
The two other bases are the base 12 (called base duodecimal) and the base 60 (called base sexagesimal). Those bases are simply used in time system, we have 12 hours before the midday and 12 other before the midnight (we use it also for the 12 months in a year). The base sexagesimal is used for the 60 seconds in a minute and the 60 minutes in an hour (but was already used by the Babylonians).

There is also a few other used bases beginning by the base 1, but before that let's talk about the 0:
The word "zero" came via French *zéro* from Venetian language *zero*, which (together with "cipher") came via Italian *zefiro* from Arabic صفر, *ṣafira* = "it was empty", *ṣifr* = "zero", "nothing", which was used to translate Sanskrit *śūnya*, meaning *void* or *empty*… Ptolemy, influenced by Hipparchus and the Babylonians, was using a symbol for zero (a small circle with a long over bar) within a sexagesimal numeral system otherwise using alphabetic Greek numerals. Because it was used alone, not just as a placeholder, this Hellenistic zero was perhaps the first documented use of a number zero in the Old World.
- The smallest base, the base 1 (also called sticks) is more used than we think, we can find it for example in jail cells where the prisoners count the days on the walls:

- The base 2, 8 and 16 (binary, octal and hexadecimal bases), used in all computers and digital systems.
Its use is common because of the simplicity of the root, the base 2: on/off, true/false, in/out, good/bad…
Almost everything is adaptable to the binary system.

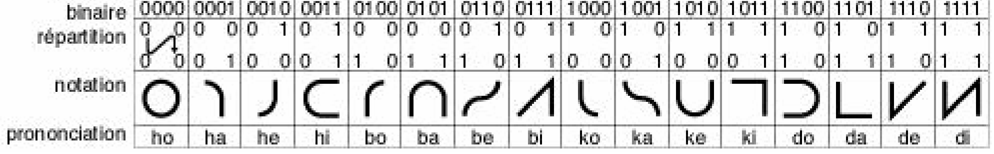
decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
octal	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20
binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000

The base 8 is just less used nowadays but was also practical: it's just a group of 3 binary digits (also called bits) that are finally represented in 1 octal digit (from 0 to 7).
The base 16 is now every where considering that just 1 character can represent a value between 0 and 15, the information density is really better and the simplicity of encoding is the same than in the octal system:

binary	1 0101 1010 1010 1100 1111 0111						
regrouped by 4	1	0101	1010	1010	1100	1111	0111
regrouped in hexadecimal	1	5	A	A	C	F	7
hexadecimal	15AACF7						

It almost exists an infinity of other numeral system but they are not interesting in our domain.

Just for the anecdote, a famous French (funny) singer, Bobby LAPOINTE invented his own numeral system:
The numeration "Bibi":



décimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
binaire	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
répartition	0 0 / 0 0	0 0 / 0 1	0 1 / 0 0	0 1 / 0 1	0 0 / 1 0	0 0 / 1 1	0 1 / 1 0	0 1 / 1 1	1 0 / 0 0	1 0 / 0 1	1 1 / 0 0	1 1 / 0 1	1 0 / 1 0	1 0 / 1 1	1 1 / 1 0	1 1 / 1 1
notation																
prononciation	ho	ha	he	hi	bo	ba	be	bi	ko	ka	ke	ki	do	da	de	di

Why Bibi? Because 16 can be written 2 to the power 2 to the power 2 and as we talk about binary for the base 2, we could use the term« Bi-Binary » for the base 4, and « Bi-Bi-Binary » for the base 16, but it was too long then the artist decided to shorten it in "BiBi". Boby Lapointe invented the notation and pronunciation of the 16 numbers using 4 consonant and 4 vowels:

HO, HA, HE, HI, BO, BA, BE, BI, KO, KA, KE, KI, DO, DA, DE, DI.

To go back in a more serious domain, we are going to design a multiplier that allows taking negative values. We thus need to investigate this domain:

1.1 Sign-and-magnitude

One may first approach this problem of representing a number's sign by allocating one sign bit to represent the sign: set that bit (often the most significant bit) to 0 for a positive number, and set to 1 for a negative number. The remaining bits in the number indicate the magnitude (or absolute value). Hence in a byte with only 7 bits (apart from the sign bit), the magnitude can range from 0000000 (0) to 1111111 (127). Thus you can represent numbers from −12710 to +12710. A consequence of this representation is that there are two ways to represent 0, 00000000 (0) and 10000000 (−0) which is a real waste.

This approach is directly comparable to the common way of showing a sign (placing a "+" or "−" next to the number's magnitude). Some early binary computers (e.g. IBM 7090) used this representation, perhaps because of its natural relation to common usage. (Many decimal computers also used sign-and-magnitude.)

Binary value	**One's complement interpretation**	**Unsigned interpretation**
00000000	0	0
00000001	1	1
...	...	...
01111101	125	125
01111110	126	126
01111111	127	127
10000000	−127	128
10000001	−126	129
10000010	−125	130
...	...	...
11111110	−1	254
11111111	−0	255

The values of an 8-bit integer

Alternatively, a system known as ones' complement can be used to represent negative numbers. The ones' complement form of a negative binary number is the bitwise NOT applied to it — the complement of its positive counterpart. Like sign-and-magnitude representation, ones' complement has two representations of 0: 00000000 (+0) and 11111111 (−0). As an example, the ones' complement form of 00101011 (43) becomes 11010100 (−43). The range of signed numbers using ones' complement in a conventional eight-bit byte is -127_{10} to $+127_{10}$.

To add two numbers represented in this system, one does a conventional binary addition, but it is then necessary to add any resulting carry back into the resulting sum. To see why this is necessary, consider the following example showing the case of the addition of −1 (11111110) to +2 (00000010).

```
     binary    decimal
    11111110     -1
 +  00000010     +2
............    ...
  1 00000000      0   <-- not the correct answer
           1     +1   <-- add carry
............    ...
    00000001      1   <-- correct answer
```

In the previous example, the binary addition alone gives 00000000 => not the correct answer! Only when the carry is added back in does the correct result (00000001) appear.

This numeric representation system was common in older computers; the PDP-1 and UNIVAC 1100/2200 series, among many others, used ones'-complement arithmetic.

Note on terminology: The system is referred to as "ones' complement" because the negation of *x* is formed by subtracting *x* from a long string of ones. Two's complement arithmetic, on the other hand, forms the negation of *x* by subtracting *x* from a single large power of two.

1.2 Two's complement

The problems of multiple representations of 0 and the need for the end-around carry are circumvented by a system called two's complement. In two's complement, negative numbers are represented by the bit pattern which is one greater (in an unsigned sense) than the ones' complement of the positive value. In two's-complement, there is only one zero (00000000), that point is really important.

♣ Negating a number (whether negative or positive) is done by inverting all the bits and then adding 1 to that result. Addition of a pair of two's-complement integers is the same as addition of a pair of unsigned numbers (except for detection of overflow, if that is done). For instance, a two's-complement addition of 127 and −128 gives the same binary bit pattern as an unsigned addition of 127 and 128, as can be seen from the table:

Decimal	**Two's complement**
127	0111 1111
64	0100 0000
1	0000 0001
0	0000 0000
-1	1111 1111
-64	1100 0000
-127	1000 0001
-128	1000 0000

Some 8-bits numbers to note

♣ An easier method to get the two's complement of a number is as follows:

	Example 1	Example 2
1. Starting from the right, find the first '1':	**010100**1	**0101**100
2. Invert all of the bits to the left of that one:	**101011**1	**1010**100

…the underlined bits staying unchanged.

In computer circuitry, this easier method is no faster than the "complement and add one" method; both methods require working sequentially from right to left, propagating logic changes. The method of complementing and adding one can be sped up by a carry look-ahead adder circuit; the alternative method can be sped up by a similar logic transformation.

♣ A more formal definition of two's complement negative number (denoted by N^* in this example) is derived from the equation $N^* = 2^n - N$, where N is the corresponding positive number and n is the number of bits in the representation.

For example, to find the 4 bit representation of -5:

$N = 5_{10}$ therefore $N = 0101_2$
$n = 4$

Hence:

$$N^* = 2^n - N = 2^4 - 5_{10} = 10000_2 - 0101_2 = 1011_2$$

The calculation can be done entirely in base 10, converting to base 2 at the end:

$$N^* = 2^n - N = 2^4 - 5 = 11_{10} = 1011_2$$

1.3 Comparison table

The following table compares the representation of the integers between positive and negative eight (inclusive) using 4 bits.

4-bit Integer Representations

Decimal	Unsigned	Sign and Magnitude	Ones' Complement	Two's Complement	Excess-7 (Biased)
+8	1000	N/A	N/A	N/A	1111
+7	0111	0111	0111	0111	1110
+6	0110	0110	0110	0110	1101
+5	0101	0101	0101	0101	1100
+4	0100	0100	0100	0100	1011
+3	0011	0011	0011	0011	1010
+2	0010	0010	0010	0010	1001
+1	0001	0001	0001	0001	1000
(+)0	0000	0000	0000	0000	0111
(−)0	N/A	1000	1111	N/A	N/A
−1	N/A	1001	1110	1111	0110
−2	N/A	1010	1101	1110	0101
−3	N/A	1011	1100	1101	0100
−4	N/A	1100	1011	1100	0011
−5	N/A	1101	1010	1011	0010
−6	N/A	1110	1001	1010	0001
−7	N/A	1111	1000	1001	0000
−8	N/A	N/A	N/A	1000	N/A

3 ***Multiplications algorithms***

Theory

The product of two *n*-bit numbers can potentially have 2*n* bits. If the precision of the two two's complement operands is doubled before the multiplication, direct multiplication (discarding any excess bits beyond that precision) will provide the correct result. For example, take 5 × −6 = −30. First, the precision is extended from 4 bits to 8. Then the numbers are multiplied, discarding the bits beyond 8 (shown by 'x'):

```
  00000101  (5)
× 11111010  (−6)
 =========
      1010
    1010
   101
  101
 x01
xx1
=========
xx11100010  (−30)
```

This is very inefficient; by doubling the precision ahead of time, all additions must be double-precision and at least twice as many partial products are needed than for the more efficient algorithms actually implemented in computers. Some multiplication algorithms are designed for two's complement, notably Booth's multiplication algorithm. Methods for multiplying sign-magnitude numbers don't work with two's complement numbers without adaptation. There isn't usually a problem when the multiplicand (the one being repeatedly added to form the product) is negative; the issue is setting the initial bits of the product correctly when the multiplier is negative.

Two methods for adapting algorithms to handle two's complement numbers are common:

- First check to see if the multiplier is negative. If so, negate (i.e., take the two's complement of) both operands before multiplying. The multiplier will then be positive so the algorithm will work. And since both operands are negated, the result will still have the correct sign.
- Subtract the partial product resulting from the sign bit instead of adding it like the other partial products.

As an example of the second method, take the common add-and-shift algorithm for multiplication. Instead of shifting partial products to the left as is done with pencil and paper, the accumulated product is shifted right, into a second register that will eventually hold the least significant half of the product. Since the least significant bits are not changed once they are calculated, the additions can be single precision, accumulating in the register that will eventually hold the most significant half of the product. In the following example, again multiplying 5 by −6, the two registers are separated by "|":

```
 0101  (5)
×1010 (−6)
 ====|====
 0000|0000  (first partial product (rightmost bit is 0))
 0000|0000  (shift right)
 0101|0000  (add second partial product (next bit is 1))
 0010|1000  (shift right)
 0010|1000  (add third partial product: 0 so no change)
 0001|0100  (shift right)
 1100|0100  (subtract last partial product since it's from sign bit)
 1110|0010  (shift right, preserving sign bit, giving the final answer, −30)
```

Implementations:

Older multiplier architectures employed a shifter and accumulator to sum each partial product, often one partial product per cycle, trading off speed for die area. Modern multiplier architectures use the Baugh-Wooley algorithm, Wallace trees, or Dadda multipliers to add the partial products together in a single cycle. The performance of the Wallace tree implementation is sometimes improved by Booth encoding one of the two multiplicands, which reduces the number of partial products that must be summed.

♣ Booth's multiplication algorithm Procedure:

If x is the count of bits of the multiplicand, and y is the count of bits of the multiplier :

* Draw a grid of three lines, each with squares for x + y + 1 bits. Label the lines respectively A (add), S (subtract), and P (product).

* In two's complement notation, fill the first x bits of each line with :

- A: the multiplicand
- S: the negative of the multiplicand
- P: zeroes

* Fill the next y bits of each line with :

- A: zeroes
- S: zeroes
- P: the multiplier

* Fill the last bit of each line with a zero.

* Do both of these steps y times :

1. If the last two bits in the product are...
   - 00 or 11: do nothing.
   - 01: P = P + A. Ignore any overflow.
   - 10: P = P + S. Ignore any overflow.
2. Arithmetically shift the product right one position.

* Drop the last bit from the product for the final result.

Example of Booth's multiplication:

Find 3 × -4:

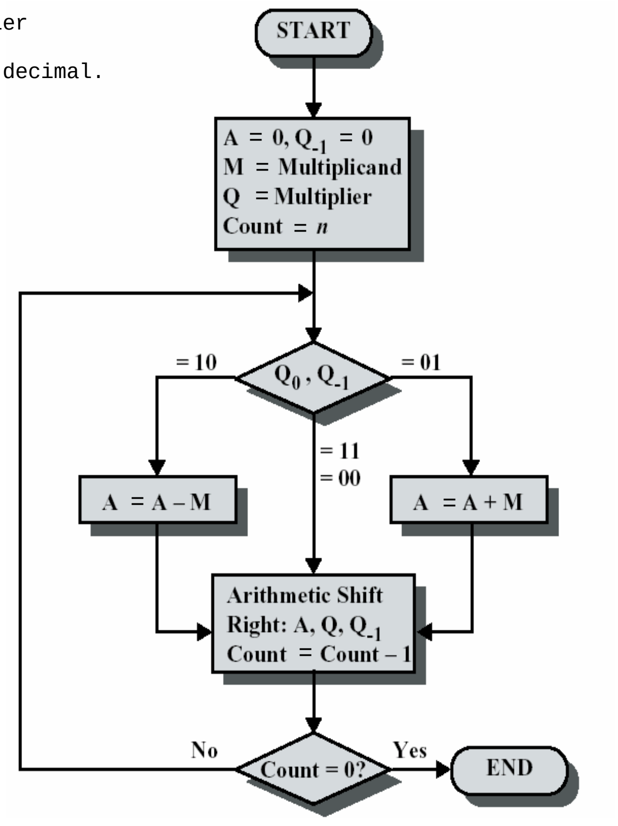
* A = 0011 0000 0
* S = 1101 0000 0
* P = 0000 1100 0

* Perform the loop four times :
  - P = 0000 1100 0. The last two bits are 00.
  - P = 0000 0110 0. A right shift.
  - P = 0000 0110 0. The last two bits are 00.
  - P = 0000 0011 0. A right shift.
  - P = 0000 0011 0. The last two bits are 10.
  - P = 1101 0011 0. P = P + S.
  - P = 1110 1001 1. A right shift.
  - P = 1110 1001 1. The last two bits are 11.
  - P = 1111 0100 1. A right shift.

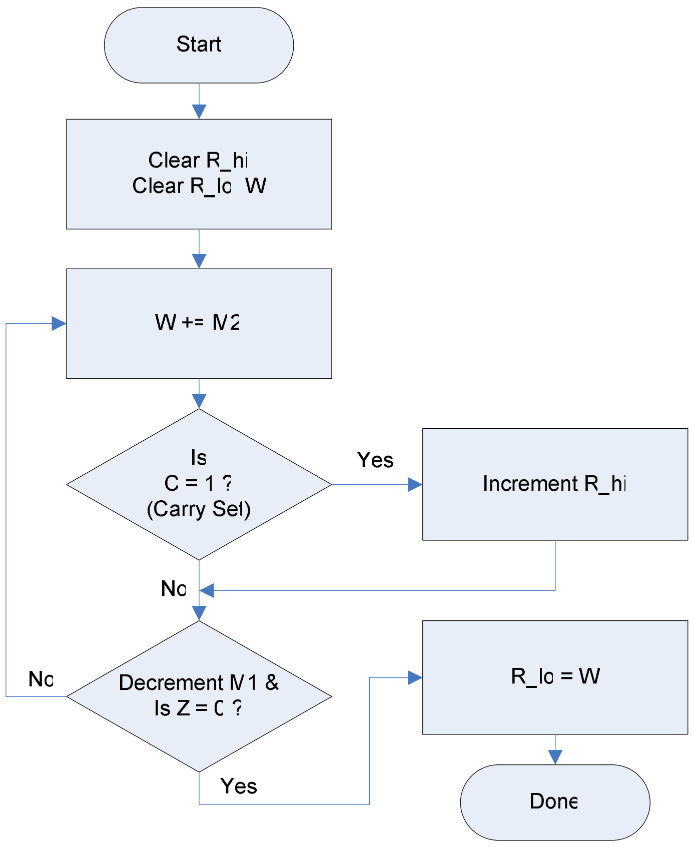
=> The product is 1111 0100, which is -12.

<u>Practical example of implementation in a PIC microcontroller:</u> (I had to use it in a lab for a decimal conversion)

```
; The following codes implement Booth's algorithm for two signed 8 bit numbers.
; It support 8.8 fixed-point format where M is the integer and A is fraction.
; The result will be 16 bits wide.

count        EQU 20
M            EQU 21                    ; Multiplicand
Q            EQU 22                    ; Multiplier and final result
A            EQU 23                    ; Remainder

             ORG         0             ; initialization code
             goto        Main

Main                                   ; 5.5 x 2 = 11 (0B)
             movlw       5             ; load number for Multiplicand, M=5
             movwf       M
             movlw       2             ; load number for Multiplier
             movwf       Q
             movlw       1             ; A=1, this equals 0.5 in decimal.
             movwf       A
             call        Booth_MUL
             sleep

Booth_MUL
             movlw       8             ; number of bits
             movwf       count
             movf        M,W
             xorwf       Q,W           ; store the result sign

bthloop      movf        Q,W
             andlw       0x01
             xorwf       STATUS,F      ; check the pair of bits
             btfss       STATUS,C
             goto        arshft
             movfw       M
             btfsc       Q,0           ; if the Q_0 bit is 1
sub8         subwf       A,F           ; then subtract
             btfss       Q,0
add8         addwf       A,F
arshft       bcf         STATUS,C
             btfsc       A,7
             bsf         STATUS,C
             rrf         A,F
             rrf         Q,F
             decfsz      count,F       ; check if we are done
             goto        bthloop
done         return

             END
```



<u>We have also seen a simpler algorithm:</u> (still in PIC assembler)



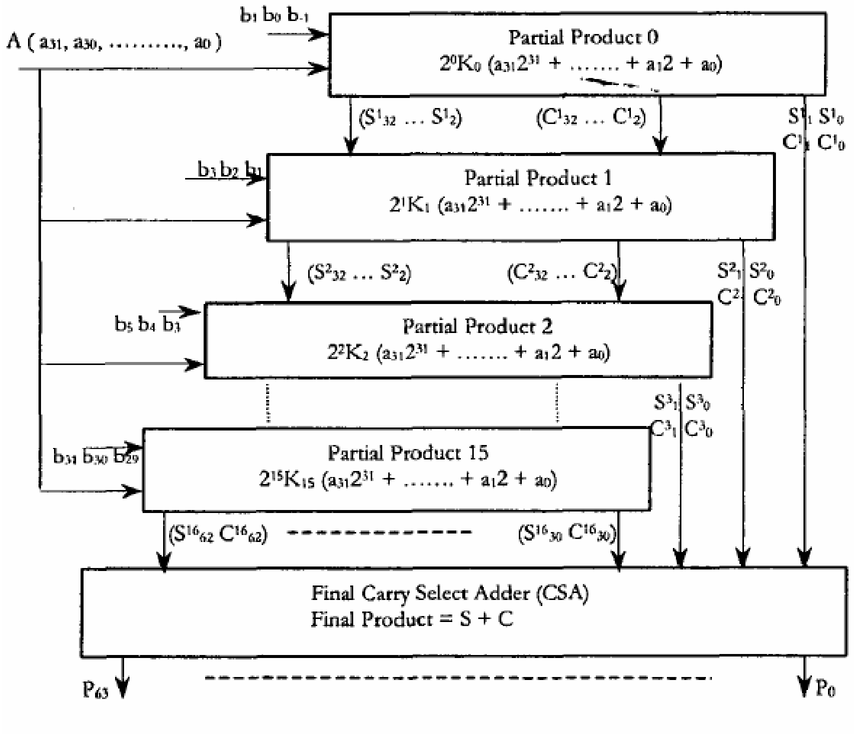
```
; 8 by 8-bit unsigned multiply routine.
; No checks made for M1 or M2 equal to zero
; R_hi, R_lo = M1 * M2

          LIST p=16F877
          #include <p16F877.inc>

M1        equ       20
M2        equ       21
R_lo      equ       22
R_hi      equ       23

Main      movlw h'9c'
          movwf M1
          movlw 4
          movwf M2
          call MUL8by8
          sleep

MUL8by8   clrf      R_hi
          clrf      R_lo
          clrw
loop1     addwf     M2,W       ; add M2 to itself
          btfsc     STATUS,C   ; if carry set
          incf      R_hi       ; increment high byte
          decfsz    M1
          goto      loop1
          movwf     R_lo
          return

          end
```

4 Examples of a few multipliers

As required, I'm going to research different multiplier implementations. I'm going to release my results chronologically, and with more or less details in function of the multiplier found.

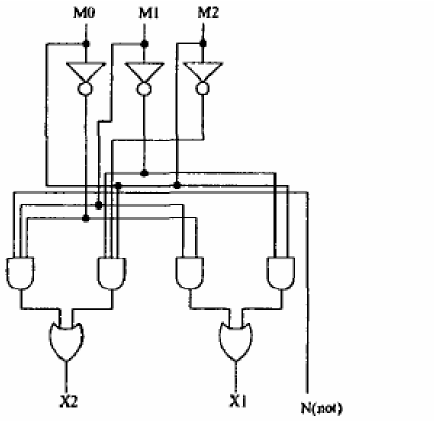
Modified Booth algorithm implementation:

This is one of the most popular techniques to reduce the number of partial products to be added while multiplying two numbers. Reduction in number of partial products depends upon how many bits are recoded. If 3-bit recoding (Radix-4) is used the number of partial products is reduced by half. This is a great saving in terms of silicon area and also speed as number of stages to be added is reduced to half compared to normal add and shift multiplication.

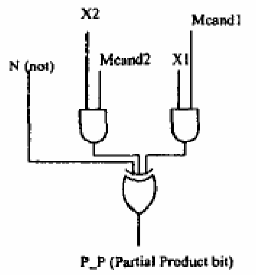


$b_1\, b_0\, b_{-1}$
$A\ (a_{31}, a_{30}, \ldots, a_0)$
Partial Product 0
$2^0K_0\ (a_{31}2^{31} + \ldots + a_12 + a_0)$
$(S^1_{32} \ldots S^1_2)$
$(C^1_{32} \ldots C^1_2)$
$S^1_1\ S^1_0$
$C^1_1\ C^1_0$
$b_3\, b_2\, b_1$
Partial Product 1
$2^1K_1\ (a_{31}2^{31} + \ldots + a_12 + a_0)$
$(S^2_{32} \ldots S^2_2)$
$(C^2_{32} \ldots C^2_2)$
$S^2_1\ S^2_0$
$C^2_1\ C^2_0$
$b_5\, b_4\, b_3$
Partial Product 2
$2^2K_2\ (a_{31}2^{31} + \ldots + a_12 + a_0)$
$S^3_1\ S^3_0$
$C^3_1\ C^3_0$
$b_{31}\, b_{30}\, b_{29}$
Partial Product 15
$2^{15}K_{15}\ (a_{31}2^{31} + \ldots + a_12 + a_0)$
$(S^{16}_{62}\ C^{16}_{62})$
$(S^{16}_{30}\ C^{16}_{30})$
Final Carry Select Adder (CSA)
Final Product = S + C
P_{63}
P_0

Modified Booth algorithm realization



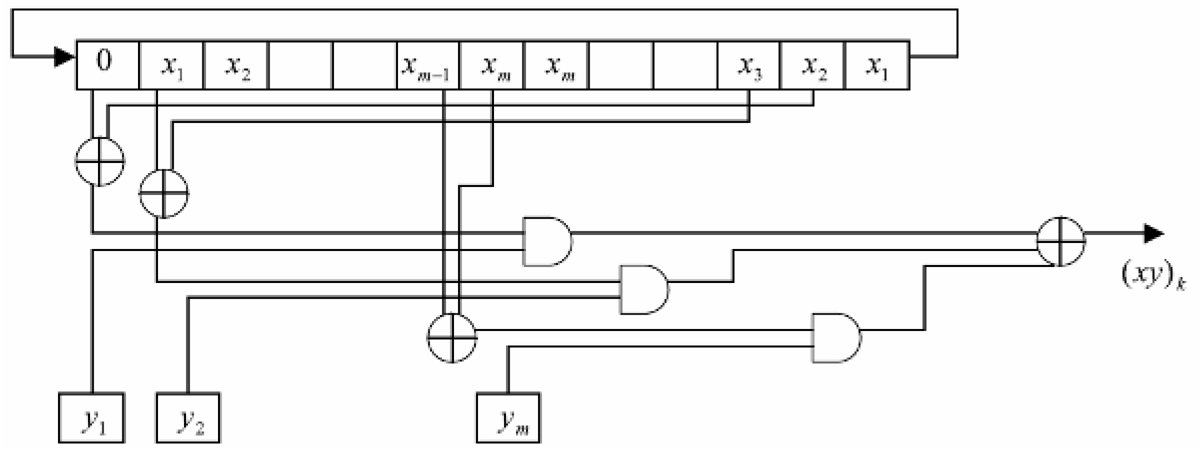
Modified Booth's recoding algorithm module



Partial Product bit generator

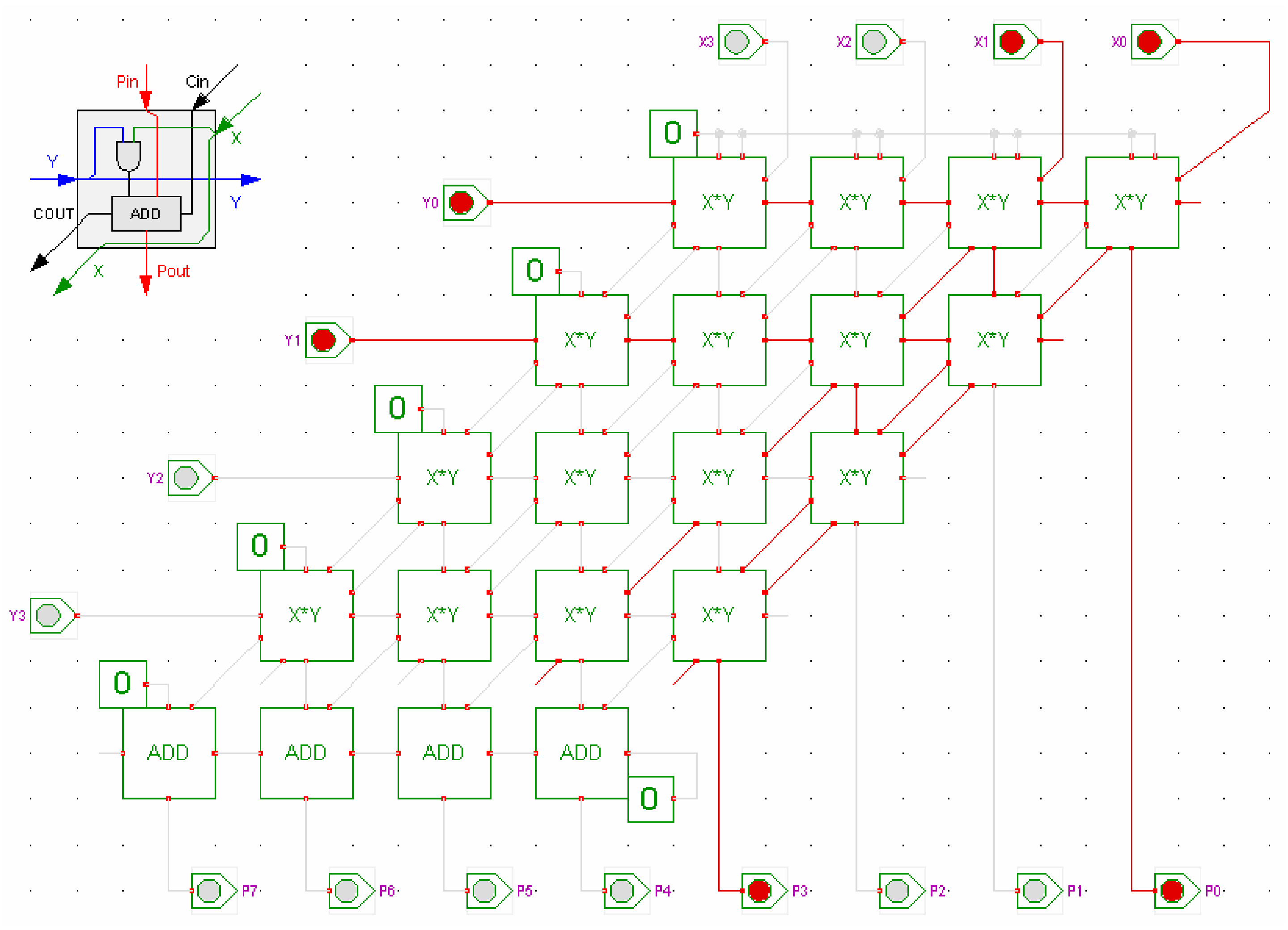
…But the complexity is greatly increased and the requirement specifications are not completely met.

Bit Parallel Systolic Architecture:



This Architecture is one of several versions of the Systolic design, we're going to see later a more accurate and closest description, but the concept represented here is roughly what we are looking for.

Parallel multiplier (4x4 bits):



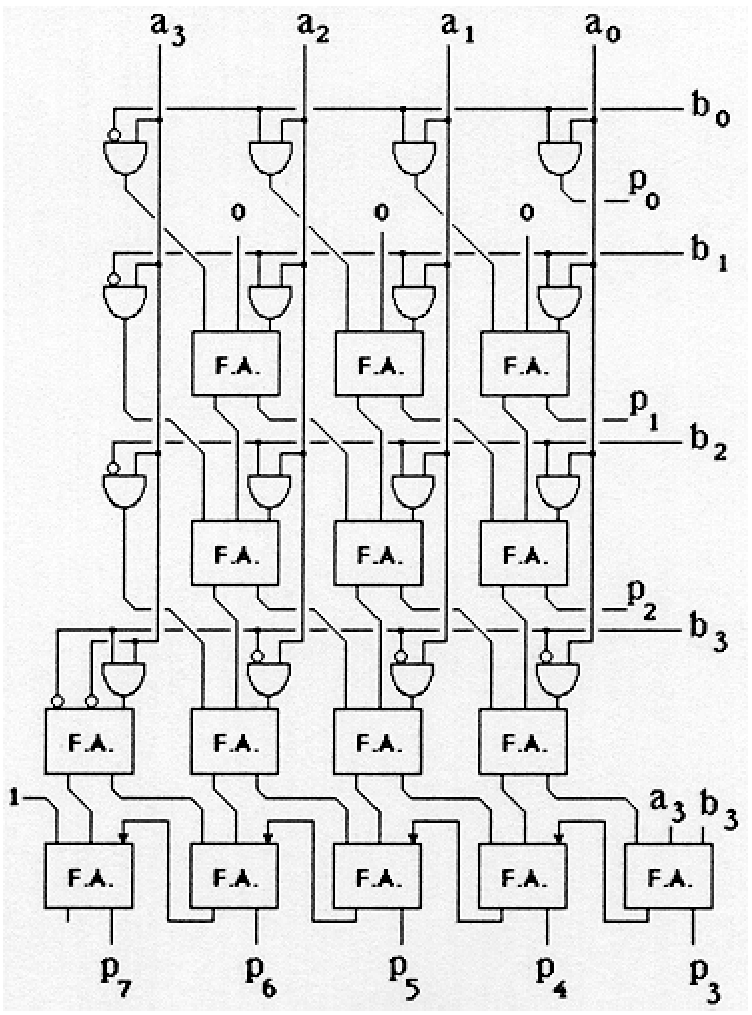
The good point of this multiplier is its easy expandability.

This multiplier takes two 4-bit inputs X and Y and generates the 8-bit product value P. Each multiplier cell uses a standard AND-gate to calculate the 1-bit product of its Xi and Yi inputs, and a standard full adder to sum the partial products.

Naturally, it is more space efficient to use a rectangular orientation of the cells for an actual VLSI implementation. Due to the regularity of the structure, it is feasible to generate the layout of such multipliers automatically for a given integrated circuit technology. While higher speed multipliers are possible, the dense layout of the multiplier array will often compensate any speed advantage of more complex circuits built from standard cells, unless expensive and tedious manual layout is used for the more complex multipliers.

…Here again, the requirement specifications are not completely met, we thus have to continue our researchs.

Serial-Parallel Multiplier



This multiplier is the simplest one, the multiplication is considered as a succession of additions.
If A = (a_n a_{n-1}……a_0)
And B = (b_n b_{n-1}……b_0)

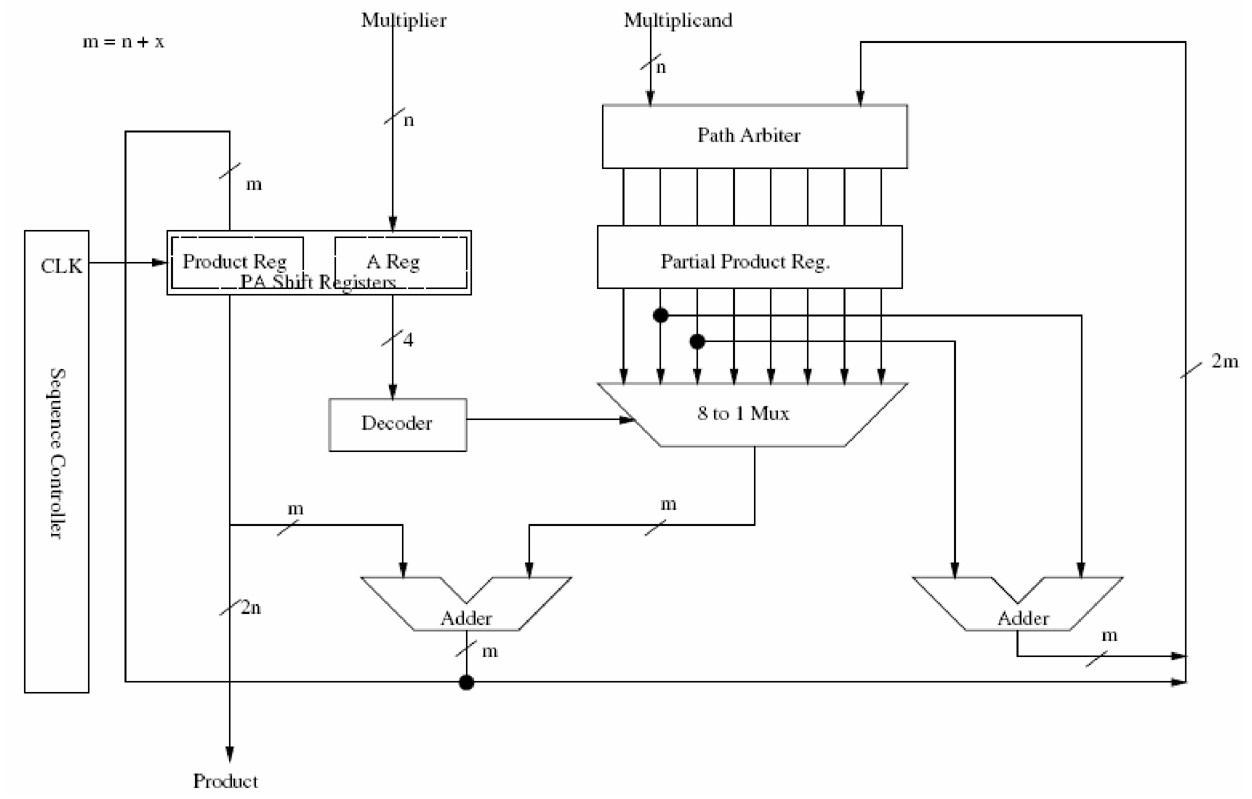
The product A.B is expressed as:
A.B = A.2^n.b_n + A.2^{n-1}.b_{n-1} +…+ A.2_0.b^0

The structure of this multiplier is suited only for positive operands. If the operands are negative and coded in 2's-complements:

1. The most significant bit of B has a negative weight, so a subtraction has to be performed at the last step.
2. Operand A.2^k must be written on 2N bits, so the most significant bit of A must be duplicated. It is easier to shift the content of the accumulator to the right instead of shifting A to the left.

An implementation of sequential multipliers using Booth algorithm (RADIX):

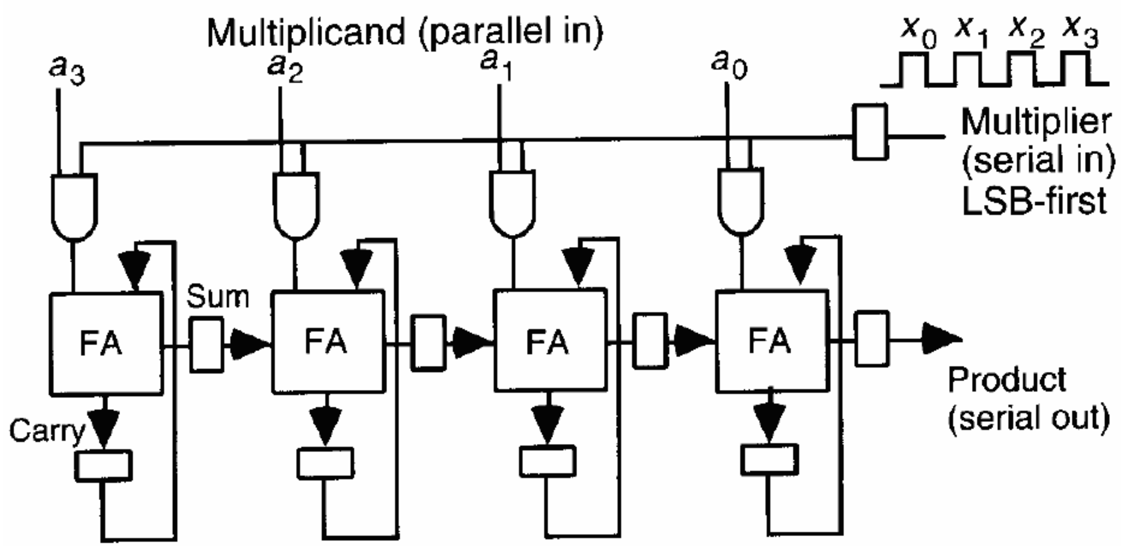
One of the simplest multiplication algorithms is the shift-and-add algorithm but its performance is poor and can be improved through more complicated algorithms, such as the Booth algorithm.



One of the possible implementations using the Booth algorithm(implemented using the radix-8 Booth algorithm).

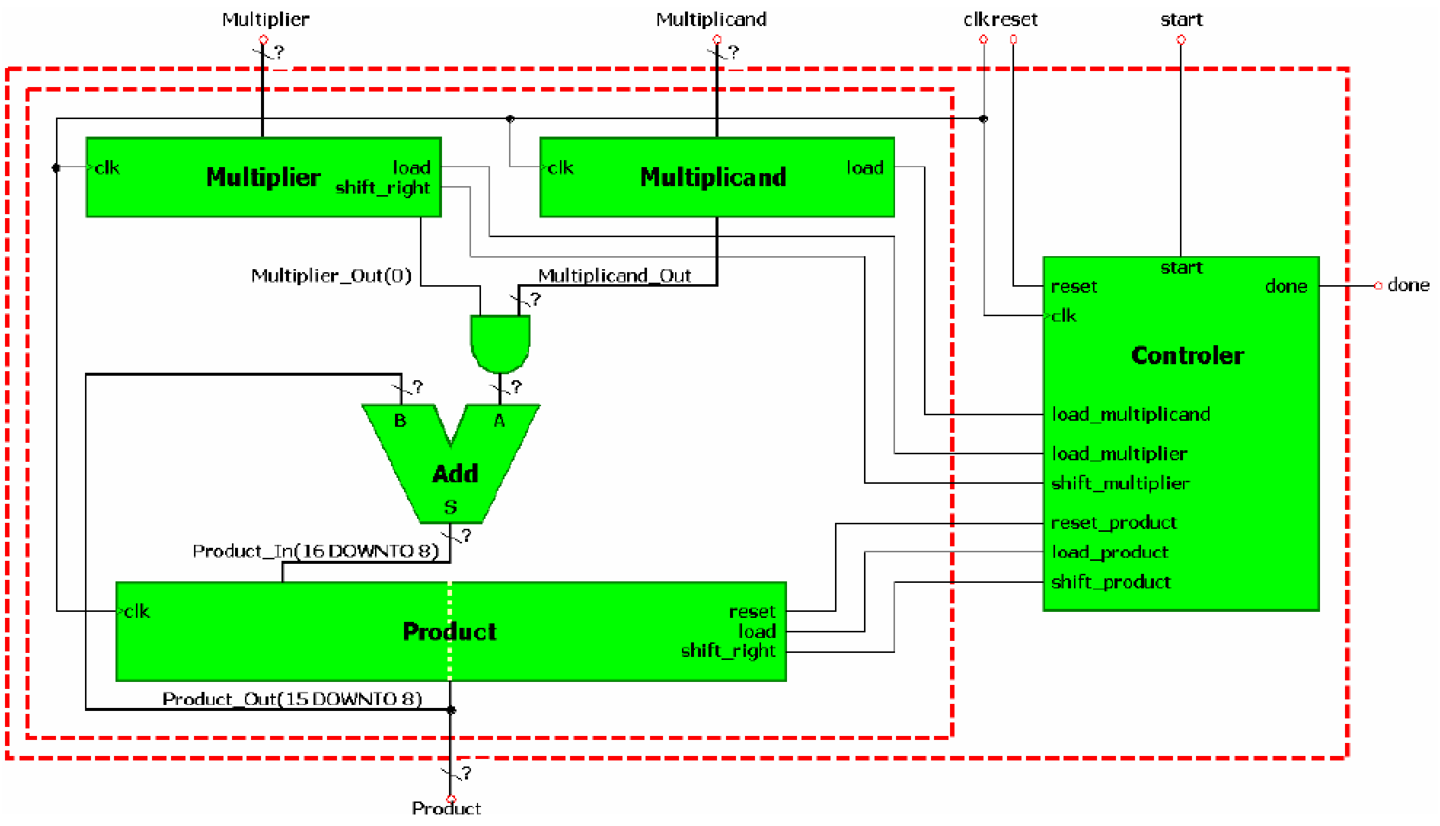
In fact, in order to obtain high performance multipliers, several hybrid multipliers which are implemented through a combination of several algorithms exist. For example, the numbers of partial products are first reduced using the Booth algorithm. Then these partial products are accumulated through other techniques, such as Wallace/Dadda reduction, or carry-save adder compaction. A major drawback of these multipliers is that they require a large amount of silicon area.

<u>Semi-systolic multiplier:</u>



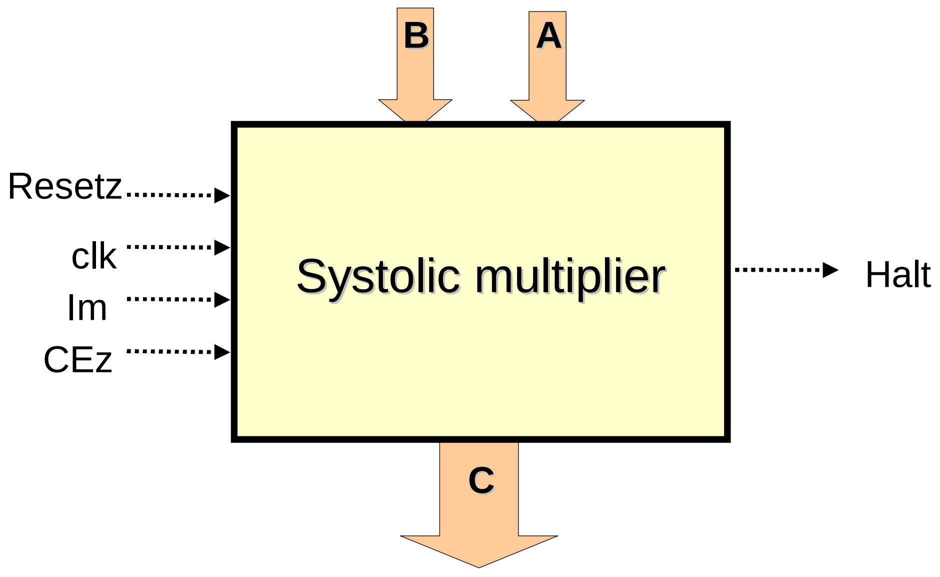
The previous graphical description is not complete but really mean full; the requirements are now almost completely met but we are going to see that the next multiplier corresponds almost exactly to our Requirement specification, and its description is really more accurate:

<u>"Multiplicateur séquentiel":</u>



5 ***Requirement specification:***

For this part, maybe quickly treated, please consider the other explanations given later.
Let's see how our component has to be interfaced to have a mean full picture in head:



Official design specifications:

The parallel/serial multiplier has two 8-bit inputs and four control signals. The output is 16-bit wide and a status signal 'Halt'. It multiplies two words in 2's complement format (7-bit plus a sign). The multiplier is for digital signal processing algorithms which require one of the inputs to be latched inside the multiplier to be considered as a common factor for the multiplication. The multiplier has three phases: initialisation, load both inputs and load only one input. The multiplication based on the extended sign-bit for 2's complement multiplication, i.e., the sign bits for the multiplier and multiplicand are extended indefinitely as shown in the floor plan. The operations and signals of the multiplier are as follows:

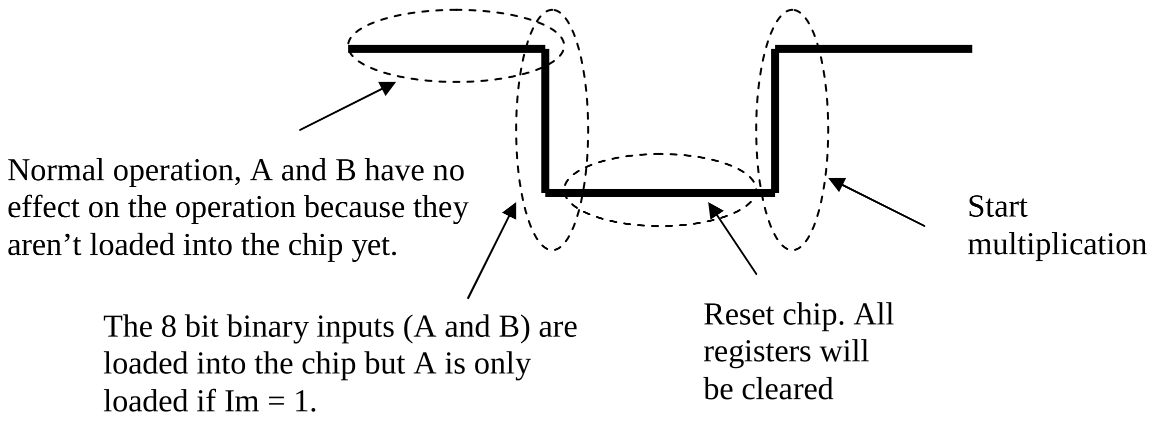
Clock signal:	to synchronise the flow of the operations.
Chip enable:	to enable the chip for operation, and to isolate the output from the global bus, i.e., CEz = 0, the chip is ready for receiving inputs from the input buses and sending the output to the output bus, if CEz = 1, the chip is disabled and latches the previous inputs and the results, while the output register is in the tri-state.
Reset signal:	to reset all the flip-flops to their initial values for a new operation. When RS = 0 all the flip-flops are initialised, and if RS = 1 the multiplier starts normal operation.
Halt signal:	the multiplier generates a halt signal to indicate that the multiplication is completed, and the output can be collected from the output register, and the chip is ready for new input(s).
Input mode:	to load one input or both inputs.
Inputs:	can be loaded in parallel during the initialisation phase.

Details of understanding/Interpretation for the design specifications:

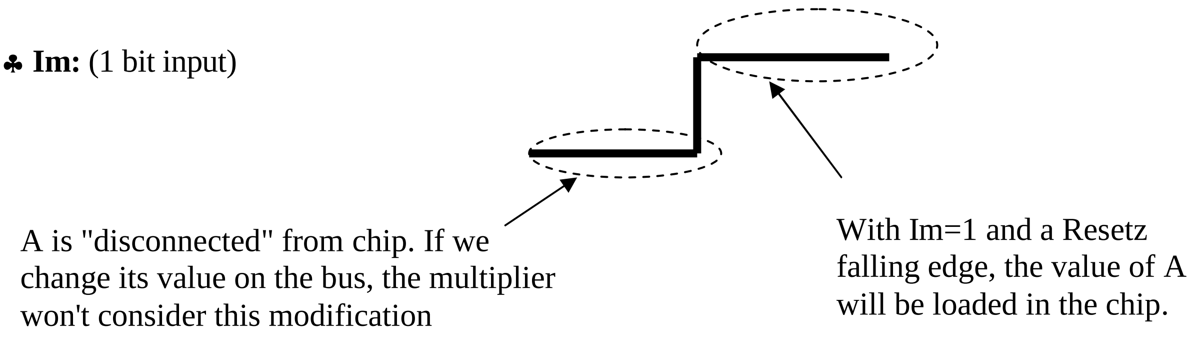
As said previously, **A** and **B** are 8 bits inputs, **C** is a 16 bits output.
The output **C** is in high impedance state when **CEz** = 1. (Disjunction of the chip to the bus for the output C)
The **Resetz** signal has to be sent to (re-)initialize the internal registers state (active low signal).
The signal **Im** (Input mode) allows selecting if we want to load 1 or 2 inputs (**A** is not loaded if **Im** = 0).
The output **Halt** is set @ 1 when the multiplier has completely finished its calculation and is thus ready.

Graphical study of all signals:

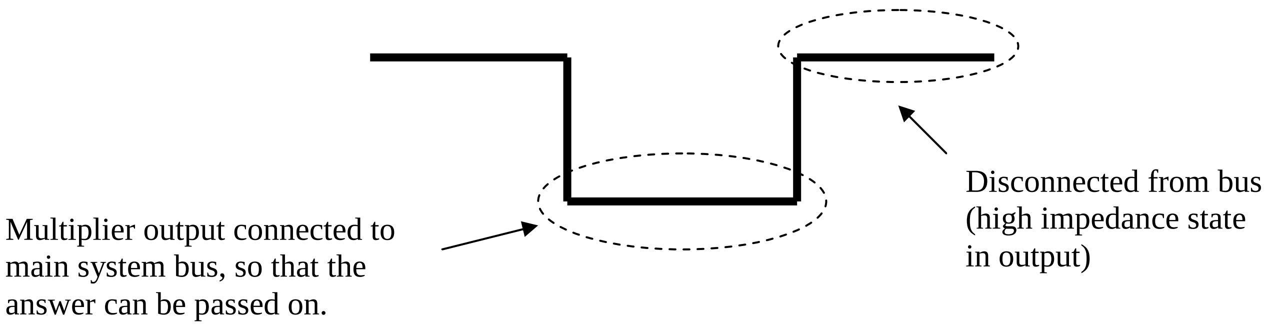
♣ **Resetz:** (1 bit input) Active low.


Normal operation, A and B have no effect on the operation because they aren't loaded into the chip yet.
The 8 bit binary inputs (A and B) are loaded into the chip but A is only loaded if Im = 1.
Reset chip. All registers will be cleared
Start multiplication

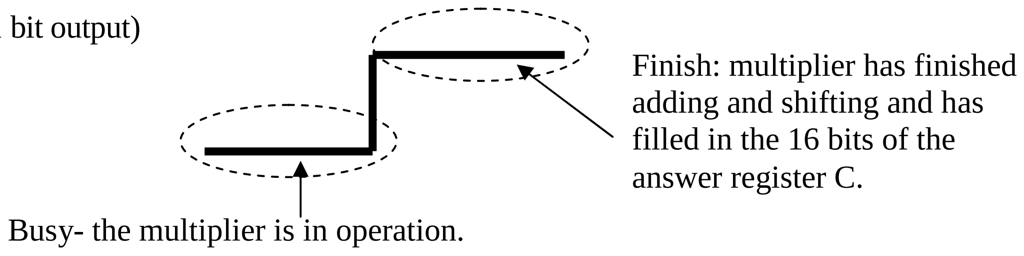
♣ **Im:** (1 bit input)


A is "disconnected" from chip. If we change its value on the bus, the multiplier won't consider this modification
With Im=1 and a Resetz falling edge, the value of A will be loaded in the chip.

♣ **CEz:** (1 bit input) Active low.


Multiplier output connected to main system bus, so that the answer can be passed on.
Disconnected from bus (high impedance state in output)

♣ **Halt:** (1 bit output)


Finish: multiplier has finished adding and shifting and has filled in the 16 bits of the answer register C.
Busy- the multiplier is in operation.

4 *Verilog description of the selected multiplier*

The principle of "shift-add" being chosen, we are thus going to design a first version of our multiplier. This is version is not the Multspec as maybe expected, because this HDL description doesn't change a lot: The difference between the Multspec and architectural model is the process of the multiplication; the depth of the multiplication technique was too simplified in the Multspec version.
This architectural description is almost the final component considering that it's giving what we need, but it's not taking care of the sequential aspect of the systolic multiplication.

Verilog code of architectural description:

```
 1  module multarc(C,halt , A,B,Im,resetz,CE);
 2  //note: as we don't use the clock I took it off.
 3          output [15:0]C;
 4          output halt;
 5          input [7:0] A, B;
 6          input Im, resetz, CE;
 7          reg [15:0] A_reg,B_reg,C_reg,C,mult,sum;
 8          reg halt;
 9          integer i;
10
11  initial begin
12          A_reg = 0;
13          B_reg = 0;
14          C_reg = 0;
15          C = 0;
16          halt = 1;
17          mult = 0;
18          sum = 0;
19  end
20
21  always @ (negedge resetz) begin
22          B_reg = 0;
23          B_reg[7:0] = B;
24          if (B[7]) B_reg[15:8] = 8'hff;
25          if (Im) begin
26                  A_reg = 0;
27                  A_reg[7:0] = A;
28                  if (A[7]) A_reg[15:8] = 8'hff;
29          end
30          C_reg = 0;
31          halt = 0;
32  end
33
34  always @ (posedge resetz) begin
35          for (i=0; i<16; i=i+1) begin
36                  mult = B_reg[0] * A_reg;
37                  sum = sum + mult;
38                  C_reg = C_reg>>1;
39                  C_reg[15] = sum[0];
40                  sum[14:0] = sum[15:1];
41                  B_reg[14:0] = B_reg[15:1];
42          end
43          #0 halt = 1;
44  end
45
46  always @ (CE or C_reg) begin
47          if (!CE) C = C_reg;
48          else C = 16'hzzzz;
49  end
    endmodule
```

♣ The sign bit is extended in line 24 for register B, in line 28 for register A but only if Im is high (only if the two multiplicands are required to be loaded).

♣ The #0 in line 43 is to force the affectation of halt to be the last. The reason that the processor needs to be told to do this is that the processor is modelling a concurrent system where functions are being performed simultaneously.
The processor however, is a sequential processor and can only do one thing at once, so it is required to be told which calculation/function to do first (or last).

♣ Lines 47 and 48 detail the action taken according to the chip enable signal. If the chip enable signal is high then the chip is disabled from the bus, and outputs cannot be taken from an output of the module (high impedance state => disconnection from the bus).

♣ The 'mult' output is added to the sum register, a shift register that is shifted right each time it is added to. The LSB of the SUM register is shifted in to the MSB of the C_reg (virtual output register) which is also shifted right (16 times until finished). Each 'mult' multiplication consists of one bit of the B_reg LSB multiplied by the multiplicand in input register A. When a new multiplication occurs in 'mult', the result is accumulated to the previous contents of sum register, C_reg is then shifted right by 1, and the LSB of sum register is input in to the MSB of the C_reg register; sum register is then shifted right by 1 and so is the input multiplier B_reg (which brings the next significant bit to the LSB to produce the next partial product). Then the loop occurs again, and as before when the loop has finished and the multiplication result is ready in C_reg virtual register, the value is output to register C if CE is low or Z if CE is high.

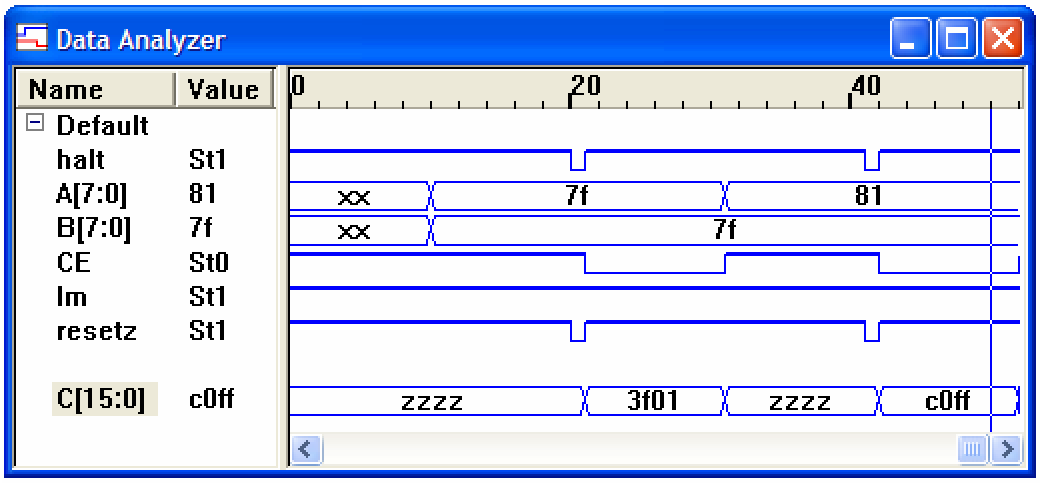
Test of the architectural description:

As this design is just a first overview I just test it simply:

```verilog
      module test;
            reg[7:0] A, B;
            reg Im,resetz,CE;
            wire [15:0] C;
            wire halt;

      multarc TEST(C,halt , A,B,Im,resetz,CE);

      initial begin
            CE = 0;             //output enabled
            resetz = 1;         //disable the reset
            Im = 1;             //load 2 inputs

            #10 A = 127;
            B = 127;
            #10 resetz = 0;     //enable the reset
            #1 resetz = 1;      //disable the reset
            wait(halt);         //wait untill ready
            $display("C = %d",C);

            A = -127;
            B = 127;
            #10 resetz = 0;     //enable the reset
            #1 resetz = 1;      //disable the reset
            wait(halt);         //wait untill ready
            #10 $display("C = %d",C);
      end
      endmodule
```

Chronogram of the test result:

As I've got a few problems to read certain mass of letters I found this solution that allow me to read the result without loose myself in all the lines of the result. I'll do it later anyway, but the least possible, just once.

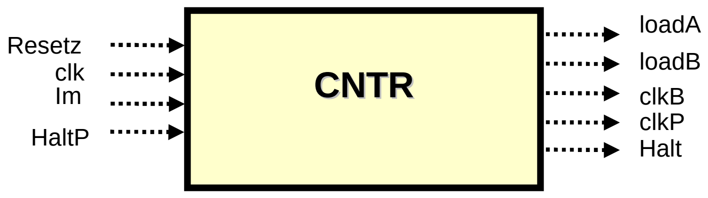


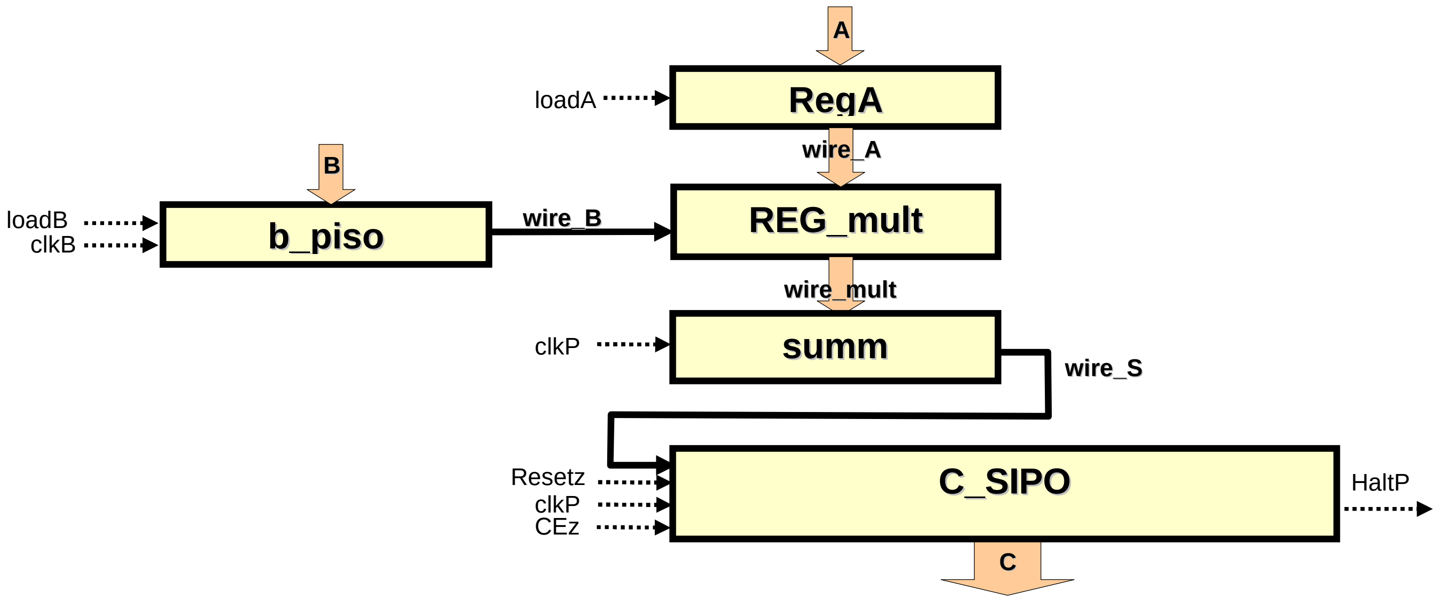
We can see the result obtained for 0x7F × 0x7F (= 127 × 127 = 16129) is 0x3F01 = 16129 as expected.
We can see the result obtained for 0x81 × 0x7F (= -127 × 127 = -16129) is 0xC0FF = -16129 as expected.
I chose 127 because it's the maximum value on 8bit (2's complement representation).
…but the minimum value is obviously -128!

RTL description:

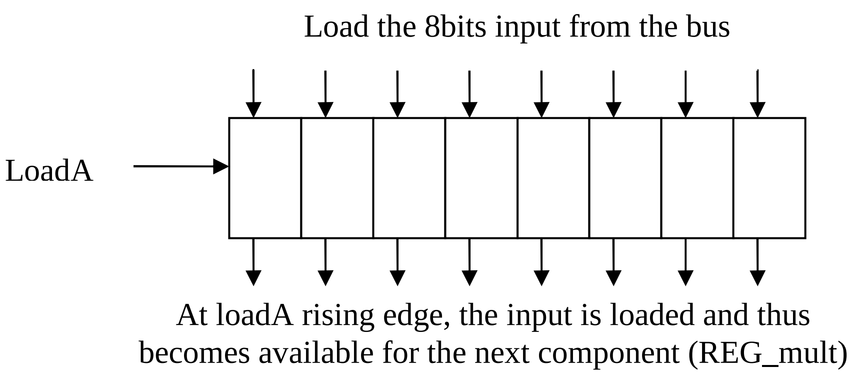
Now we have met a good part of the challenge, we have to complete the requirement list, the sequential aspect being absent in the "Roll call".

Internal components:

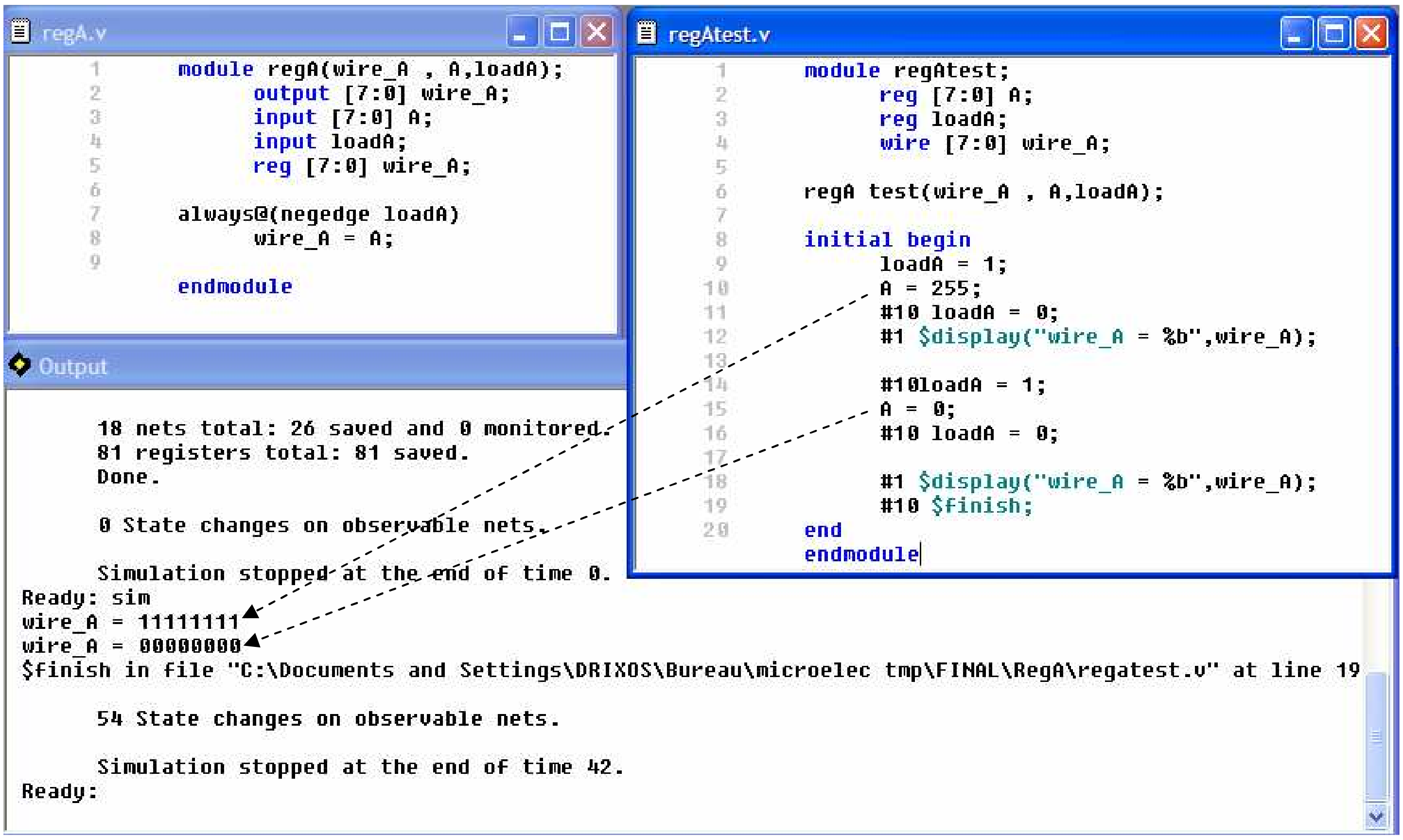




♣ **RegA**: (Parallel In, Parallel Out)


Load the 8bits input from the bus
LoadA
At loadA rising edge, the input is loaded and thus becomes available for the next component (REG_mult)

This component is one of the simplest, its Verilog code and tests are thus really straight forward:



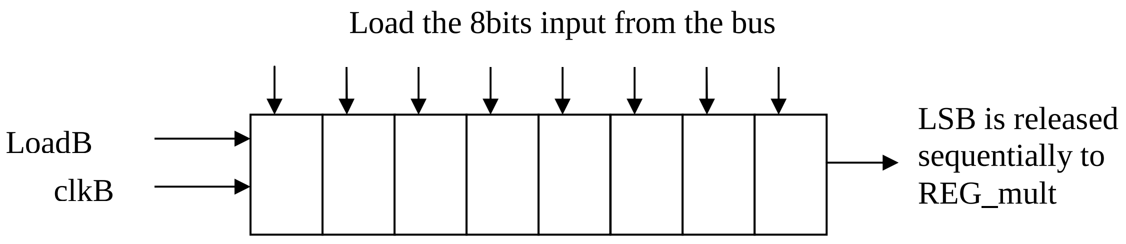
```
module regA(wire_A , A,loadA);
	output [7:0] wire_A;
	input [7:0] A;
	input loadA;
	reg [7:0] wire_A;

always@(negedge loadA)
	wire_A = A;

endmodule
```

```
module regAtest;
	reg [7:0] A;
	reg loadA;
	wire [7:0] wire_A;

regA test(wire_A , A,loadA);

initial begin
	loadA = 1;
	A = 255;
	#10 loadA = 0;
	#1 $display("wire_A = %b",wire_A);

	#10loadA = 1;
	A = 0;
	#10 loadA = 0;

	#1 $display("wire_A = %b",wire_A);
	#10 $finish;
end
endmodule
```

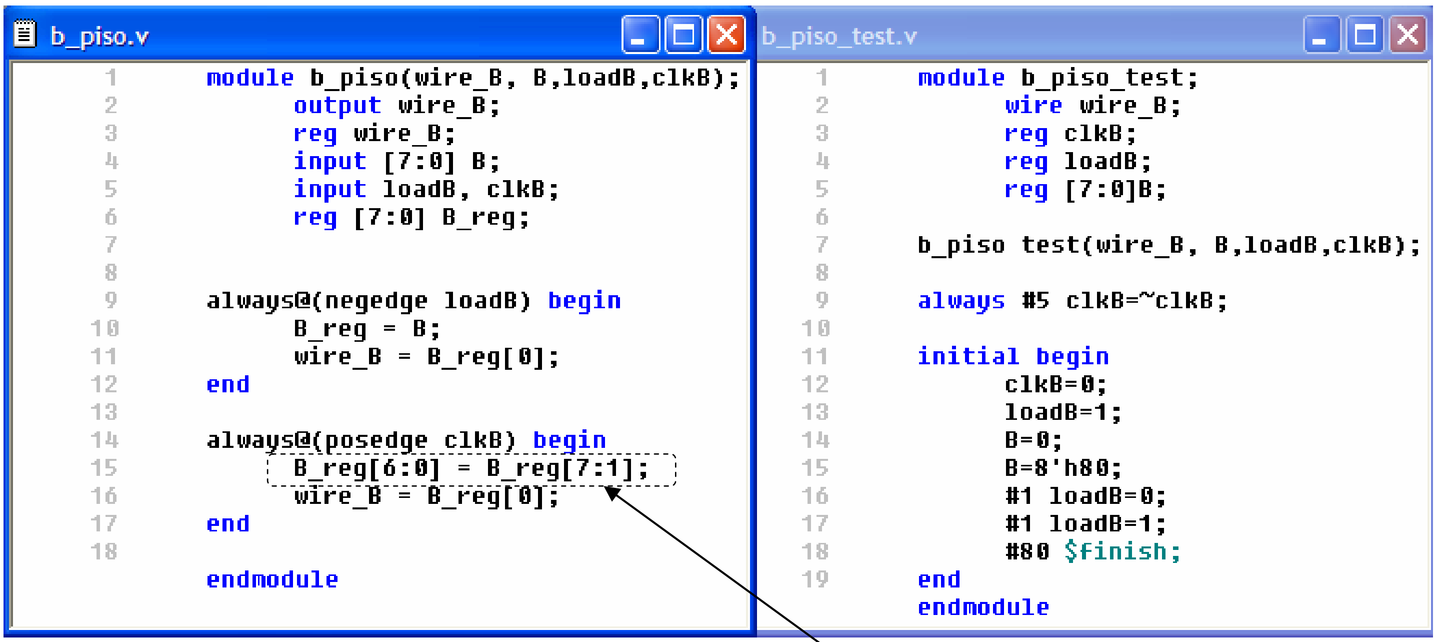
```
	18 nets total: 26 saved and 0 monitored.
	81 registers total: 81 saved.
	Done.

	0 State changes on observable nets.

	Simulation stopped at the end of time 0.
Ready: sim
wire_A = 11111111
wire_A = 00000000
$finish in file "C:\Documents and Settings\DRIXOS\Bureau\microelec tmp\FINAL\RegA\regatest.v" at line 19

	54 State changes on observable nets.

	Simulation stopped at the end of time 42.
Ready:
```

As expected, the value in input is correctly copied in output.

♣ **<u>b_piso</u>**: (Parallel In, Serial Out)


Load the 8bits input from the bus
LoadB
clkB
LSB is released sequentially to REG_mult

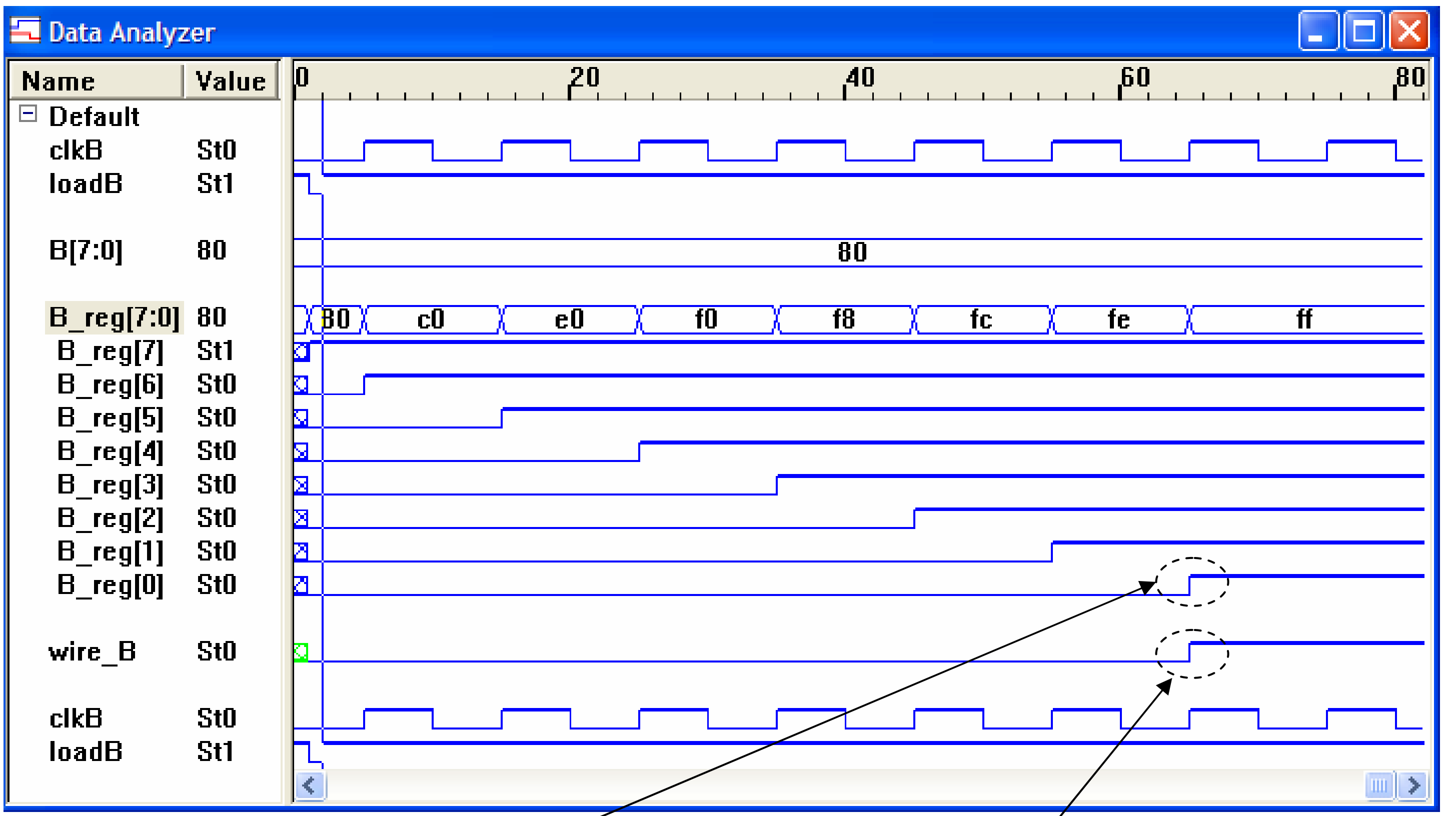
Here the sequence is: 1) input has to be ready
2) send signal LoadB
3) run clkB so that the 8 bit word gets shifted out of the register serially



```verilog
module b_piso(wire_B, B,loadB,clkB);
	output wire_B;
	reg wire_B;
	input [7:0] B;
	input loadB, clkB;
	reg [7:0] B_reg;


always@(negedge loadB) begin
	B_reg = B;
	wire_B = B_reg[0];
end

always@(posedge clkB) begin
	B_reg[6:0] = B_reg[7:1];
	wire_B = B_reg[0];
end

endmodule
```

```verilog
module b_piso_test;
	wire wire_B;
	reg clkB;
	reg loadB;
	reg [7:0]B;

b_piso test(wire_B, B,loadB,clkB);

always #5 clkB=~clkB;

initial begin
	clkB=0;
	loadB=1;
	B=0;
	B=8'h80;
	#1 loadB=0;
	#1 loadB=1;
	#80 $finish;
end
endmodule
```

As we can see, the MSB is not affected by the shift process but it's propagated on the lower significant bits.

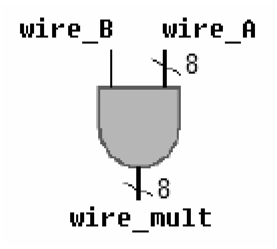


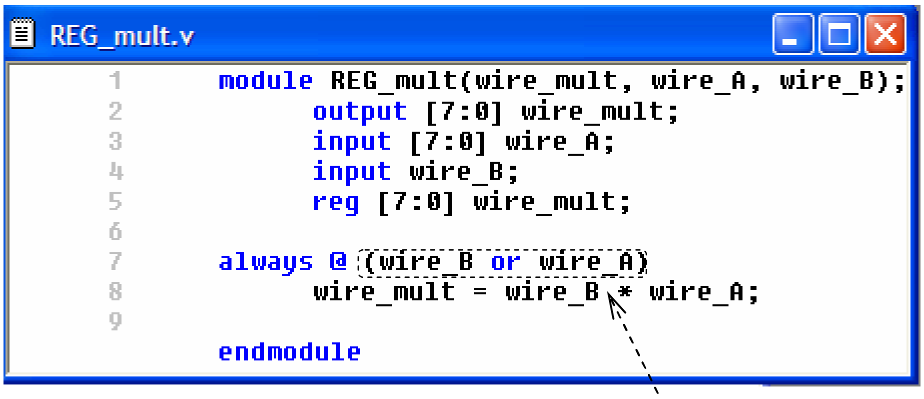
Name	Value
Default	
clkB	St0
loadB	St1
B[7:0]	80
B_reg[7:0]	80
B_reg[7]	St1
B_reg[6]	St0
B_reg[5]	St0
B_reg[4]	St0
B_reg[3]	St0
B_reg[2]	St0
B_reg[1]	St0
B_reg[0]	St0
wire_B	St0
clkB	St0
loadB	St1

As expected, the internal register called B_reg send its MSB to the output (wire_B). We can also see the shift process of the internal register B_reg.
I chose the value 0x80 on purpose: in binary it's 1000 0000 and if we shift it right 8 times, the MSB becomes the LSB then we see the 1 in output whereas we had 0 in the previous states.

♣ **REG_mult**: (combinatorial component)

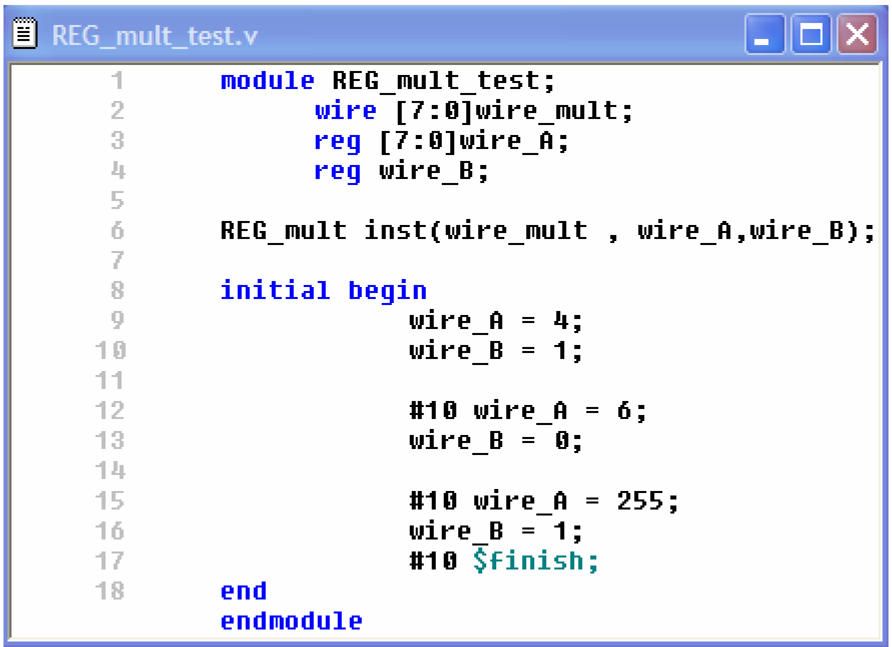
Each bit coming from b_piso is pushed in "wire_B" and is multiplied (logical "and") by each of the 8 bits of wire_A to produce an 8 bit value in the wire_mult output:



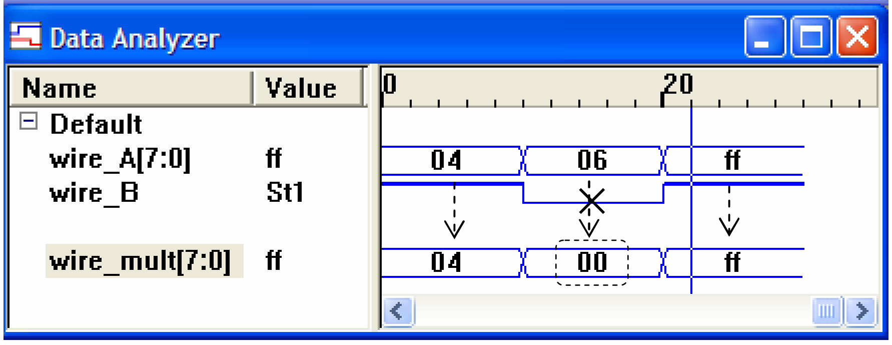


```
module REG_mult(wire_mult, wire_A, wire_B);
        output [7:0] wire_mult;
        input [7:0] wire_A;
        input wire_B;
        reg [7:0] wire_mult;

always @ (wire_B or wire_A)
        wire_mult = wire_B * wire_A;

endmodule
```

Note: we can see that this component is combinatorial because all the inputs are in the sensitivity list.



```
module REG_mult_test;
        wire [7:0]wire_mult;
        reg [7:0]wire_A;
        reg wire_B;

REG_mult inst(wire_mult , wire_A,wire_B);

initial begin
                wire_A = 4;
                wire_B = 1;

                #10 wire_A = 6;
                wire_B = 0;

                #10 wire_A = 255;
                wire_B = 1;
                #10 $finish;
end
endmodule
```



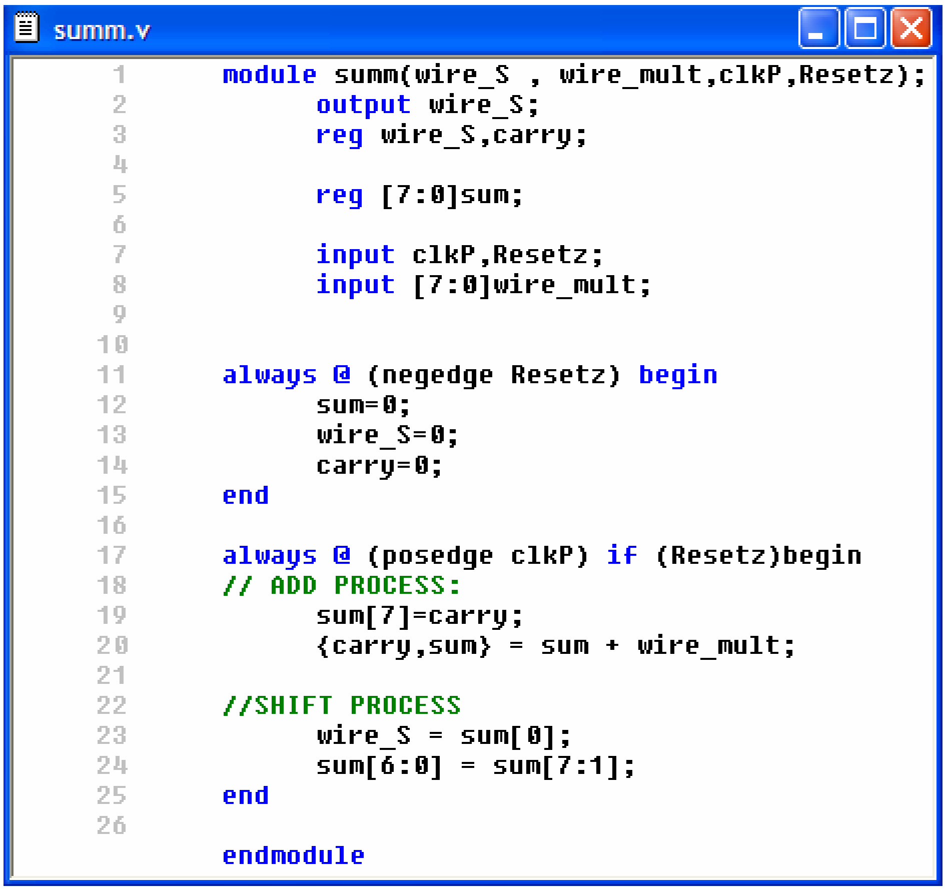
We can see that the 8bits of wire_A are correctly multiplied by the bit of wire_B.

=> wire_mult is equal to 0 if wire_B = 0 and equal to wire_A if wire_B = 1.

Note: The name of this component shouldn't be with REG because it's nothing to do with a register but when I wrote it I didn't really think about it and I never changed it; anyway, it's just a name…

♣ **summ**:

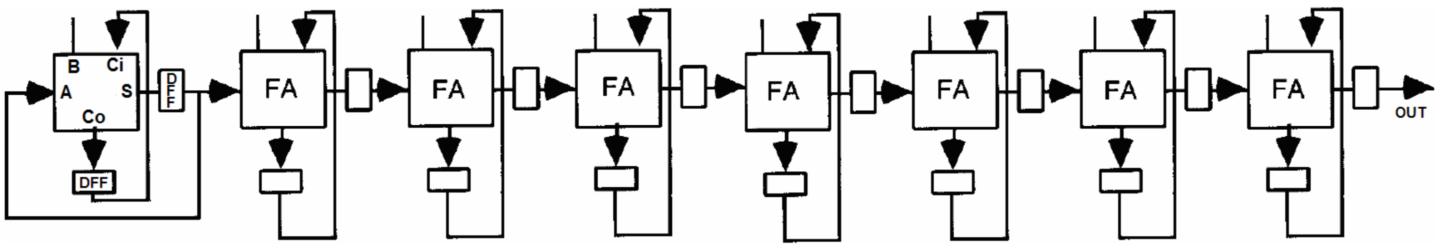
In the beginning, I designed a simple module for the sum that was working alone, but when I have instantiated the final multiplier, it didn't work and I found that the problem was from this component:



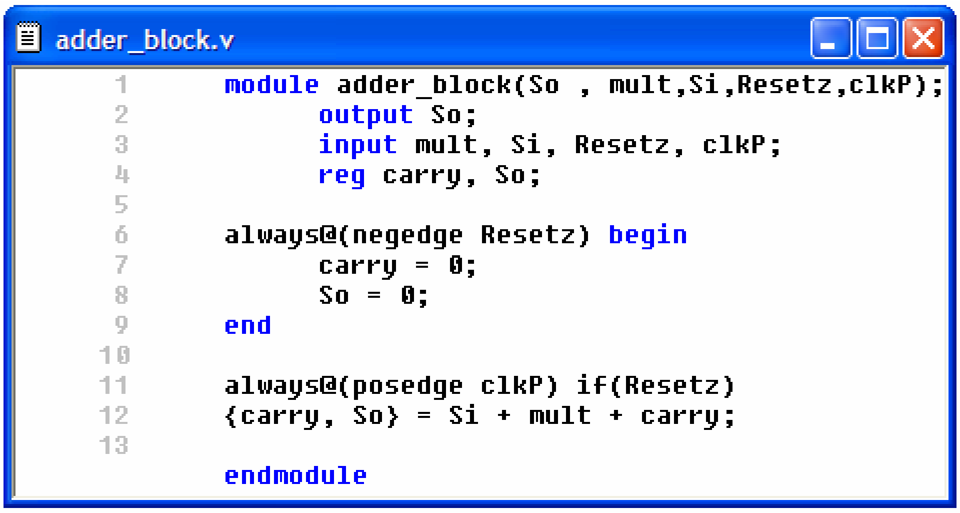
summ.v

```
module summ(wire_S , wire_mult,clkP,Resetz);
        output wire_S;
        reg wire_S,carry;

        reg [7:0]sum;

        input clkP,Resetz;
        input [7:0]wire_mult;


always @ (negedge Resetz) begin
        sum=0;
        wire_S=0;
        carry=0;
end

always @ (posedge clkP) if (Resetz)begin
// ADD PROCESS:
        sum[7]=carry;
        {carry,sum} = sum + wire_mult;

//SHIFT PROCESS
        wire_S = sum[0];
        sum[6:0] = sum[7:1];
end

endmodule
```

I never found the problem then I decided to start again and I used a gate level description that helped me to design a new sum module using this principle:



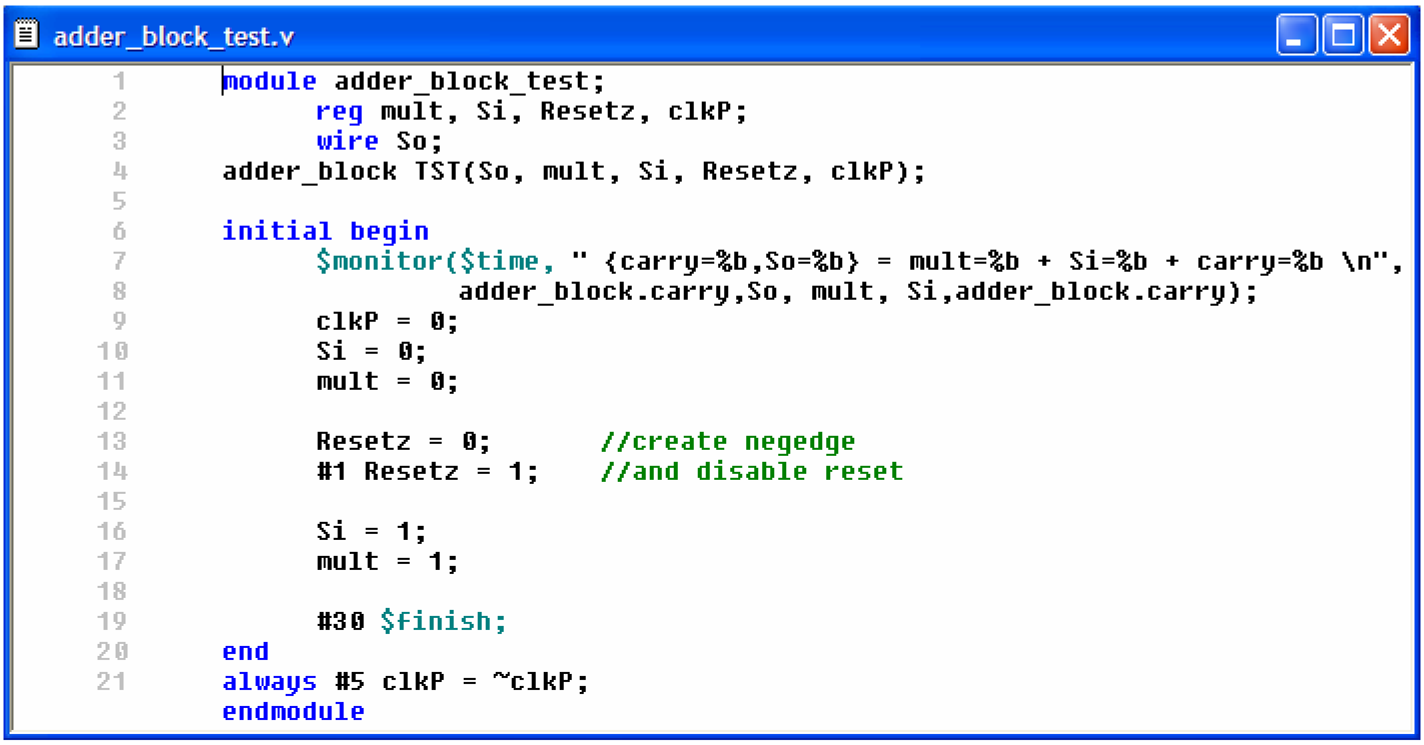
=> I thus implement it with a full adder block (8 cells for the 8 bits) where the carry out is fed back in the carry in at the next clock pulse. This is an implicit way to ripple it quickly and efficiently. Here is the add block cell:



```verilog
module adder_block(So , mult,Si,Resetz,clkP);
	output So;
	input mult, Si, Resetz, clkP;
	reg carry, So;

always@(negedge Resetz) begin
	carry = 0;
	So = 0;
end

always@(posedge clkP) if(Resetz)
{carry, So} = Si + mult + carry;

endmodule
```

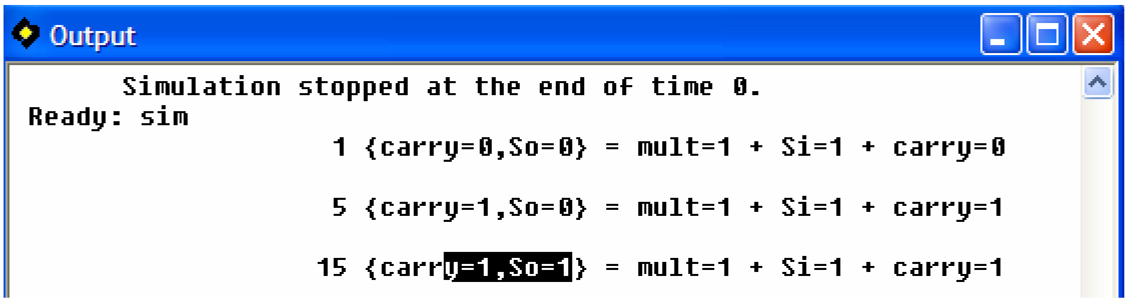
The sum of three 1 bit values can need to be represented on 2 bits.

The curly brackets allow to concatenate the carry and So (making carry as the MSB)



```verilog
module adder_block_test;
	reg mult, Si, Resetz, clkP;
	wire So;
adder_block TST(So, mult, Si, Resetz, clkP);

initial begin
	$monitor($time, " {carry=%b,So=%b} = mult=%b + Si=%b + carry=%b \n",
		adder_block.carry,So, mult, Si,adder_block.carry);
	clkP = 0;
	Si = 0;
	mult = 0;

	Resetz = 0;	//create negedge
	#1 Resetz = 1;	//and disable reset

	Si = 1;
	mult = 1;

	#30 $finish;
end
always #5 clkP = ~clkP;
endmodule
```

As expected, The 2 first lines show that 1+1+0 = 10 (in binary)…

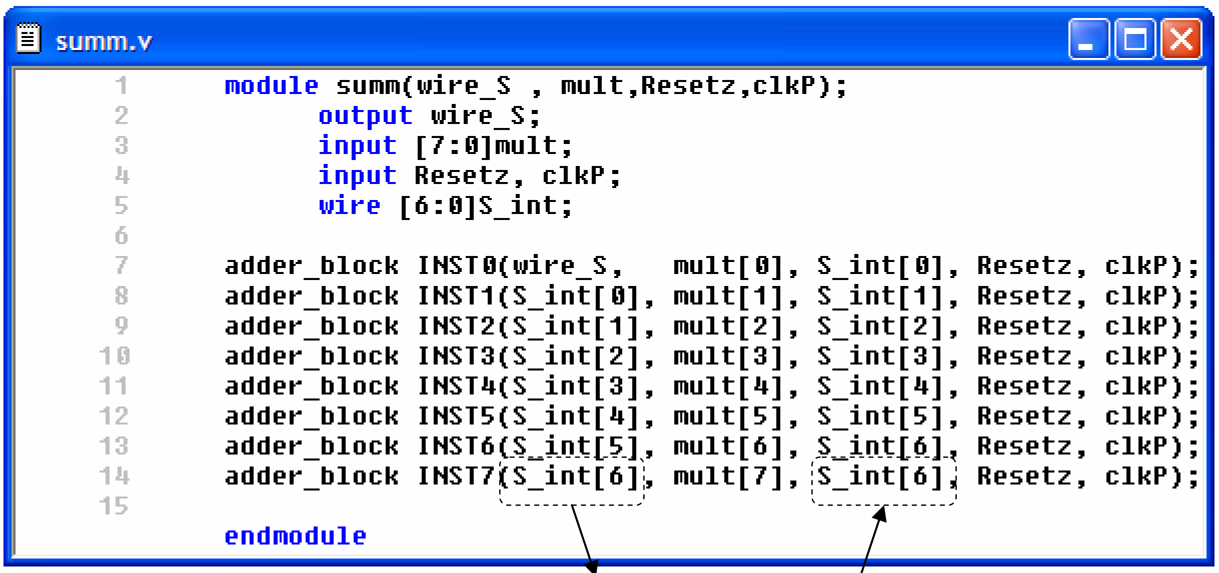


```
      Simulation stopped at the end of time 0.
Ready: sim
                   1 {carry=0,So=0} = mult=1 + Si=1 + carry=0

                   5 {carry=1,So=0} = mult=1 + Si=1 + carry=1

                  15 {carry=1,So=1} = mult=1 + Si=1 + carry=1
```

…and the 2 last lines show that 1+1+1 = 11 (in binary again).

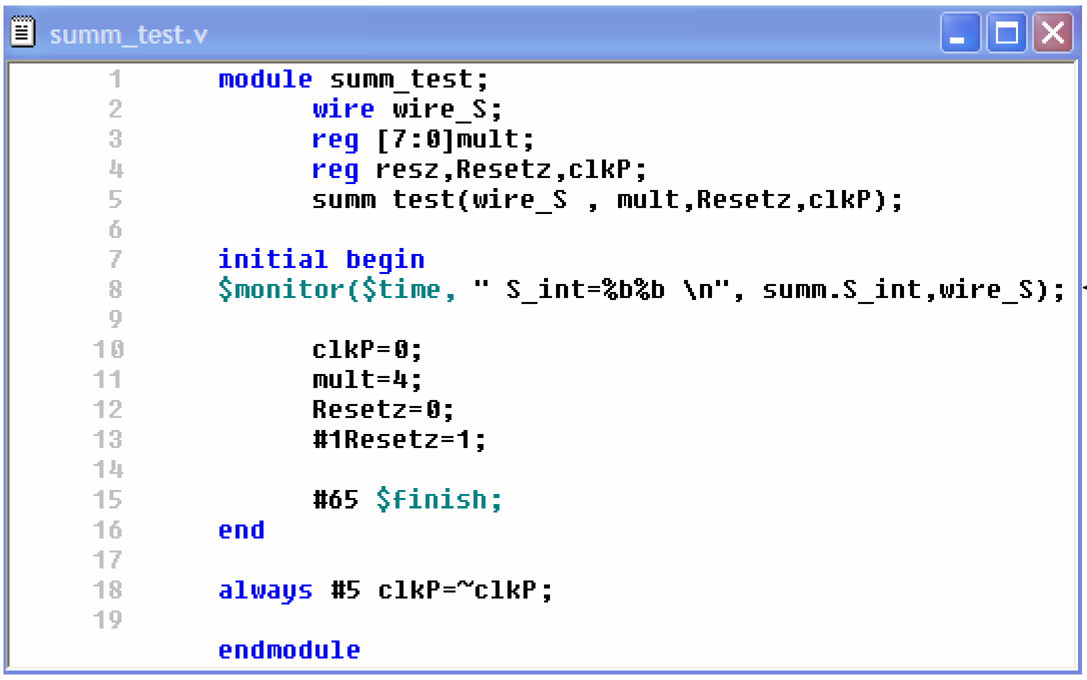
=> Then the functionality is correct.

…and here is the module that instantiate 8 times the full adder cell:



```
module summ(wire_S , mult,Resetz,clkP);
      output wire_S;
      input [7:0]mult;
      input Resetz, clkP;
      wire [6:0]S_int;

adder_block INST0(wire_S,   mult[0], S_int[0], Resetz, clkP);
adder_block INST1(S_int[0], mult[1], S_int[1], Resetz, clkP);
adder_block INST2(S_int[1], mult[2], S_int[2], Resetz, clkP);
adder_block INST3(S_int[2], mult[3], S_int[3], Resetz, clkP);
adder_block INST4(S_int[3], mult[4], S_int[4], Resetz, clkP);
adder_block INST5(S_int[4], mult[5], S_int[5], Resetz, clkP);
adder_block INST6(S_int[5], mult[6], S_int[6], Resetz, clkP);
adder_block INST7(S_int[6], mult[7], S_int[6], Resetz, clkP);

endmodule
```

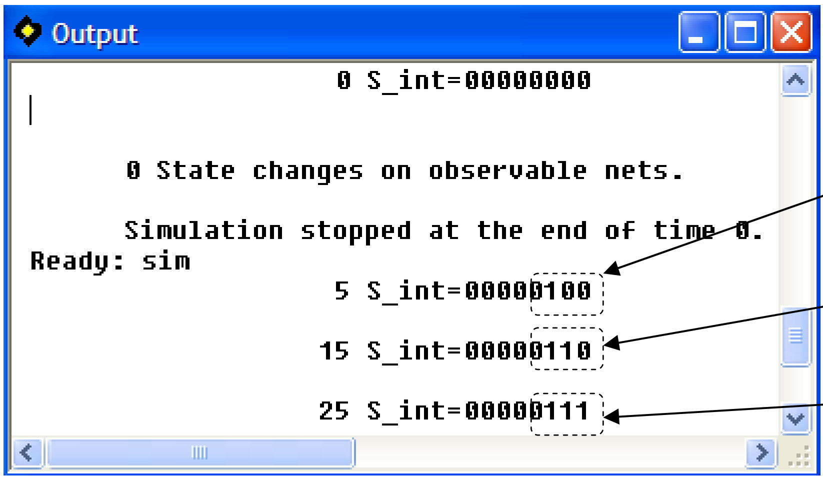
feed back of the sign bit



```
module summ_test;
      wire wire_S;
      reg [7:0]mult;
      reg resz,Resetz,clkP;
      summ test(wire_S , mult,Resetz,clkP);

initial begin
$monitor($time, " S_int=%b%b \n", summ.S_int,wire_S);

      clkP=0;
      mult=4;
      Resetz=0;
      #1Resetz=1;

      #65 $finish;
end

always #5 clkP=~clkP;

endmodule
```

Here we can see a strange way of displaying the result. I've virtually "concatenated" the 7 internal wires called S_int and the output wire_S (being the MSB) to be able to see what is the value of the accumulation result.

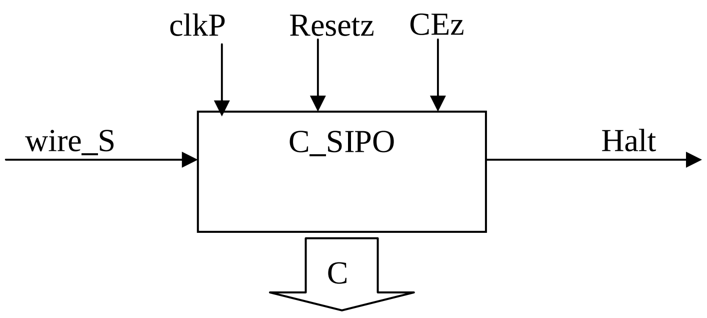
In the internal register called "S_int" we obtain *half* of the sum of the input "mult" and the previous " S_int ".
=> *Half* because of the **shift** action (which gives the entire part of the division by 2 to be more accurate).
We thus obtain as expected:



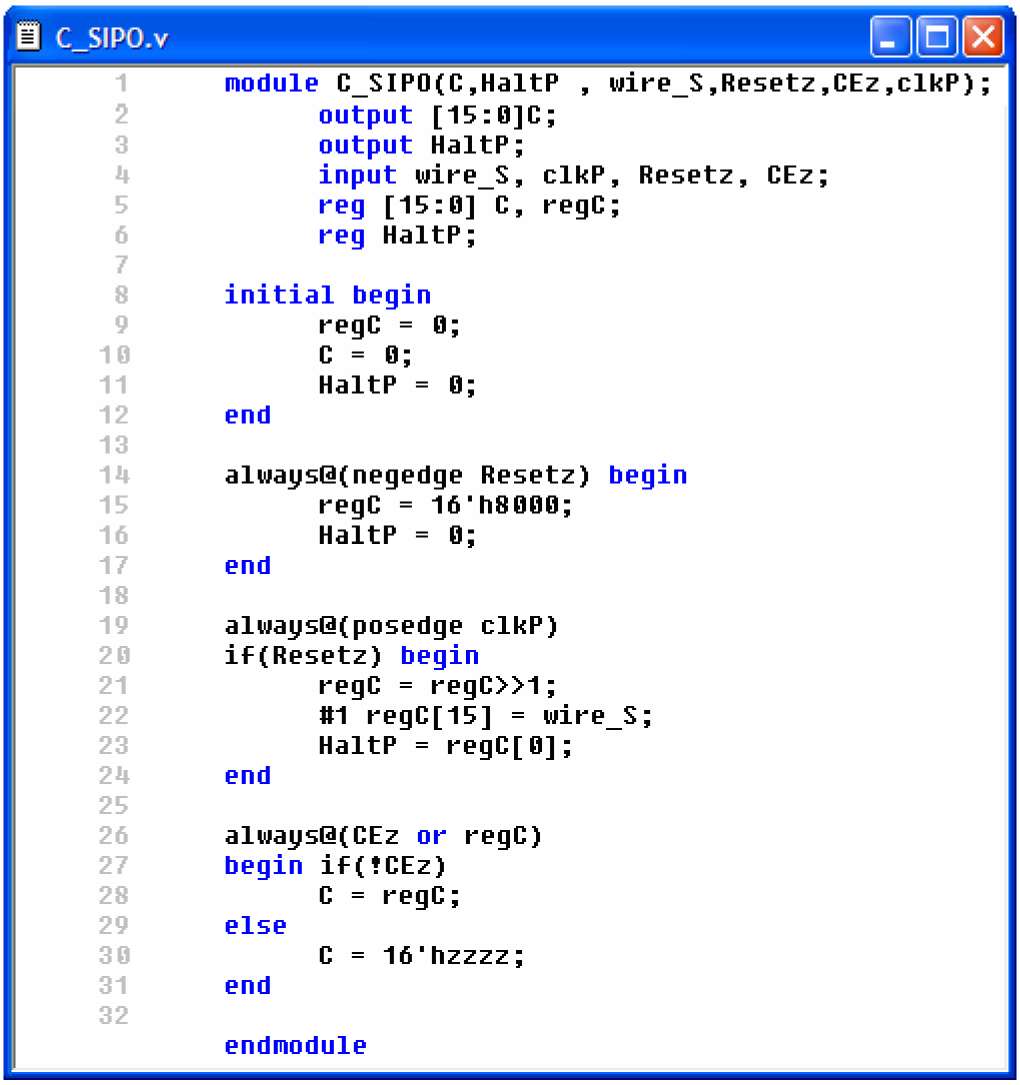
```
                  0 S_int=00000000
|

      0 State changes on observable nets.

      Simulation stopped at the end of time 0.
Ready: sim
                  5 S_int=00000100

                 15 S_int=00000110

                 25 S_int=00000111
```

```
              0                    0
          + 100                  + 4
            100                    4

SHIFT => 010       SHIFT => 4/2 = 2
          +100                   + 4
            110                    6

SHIFT => 011       SHIFT => 6/2 = 3
          +100                   + 4
            111                    7
            …                      …
```

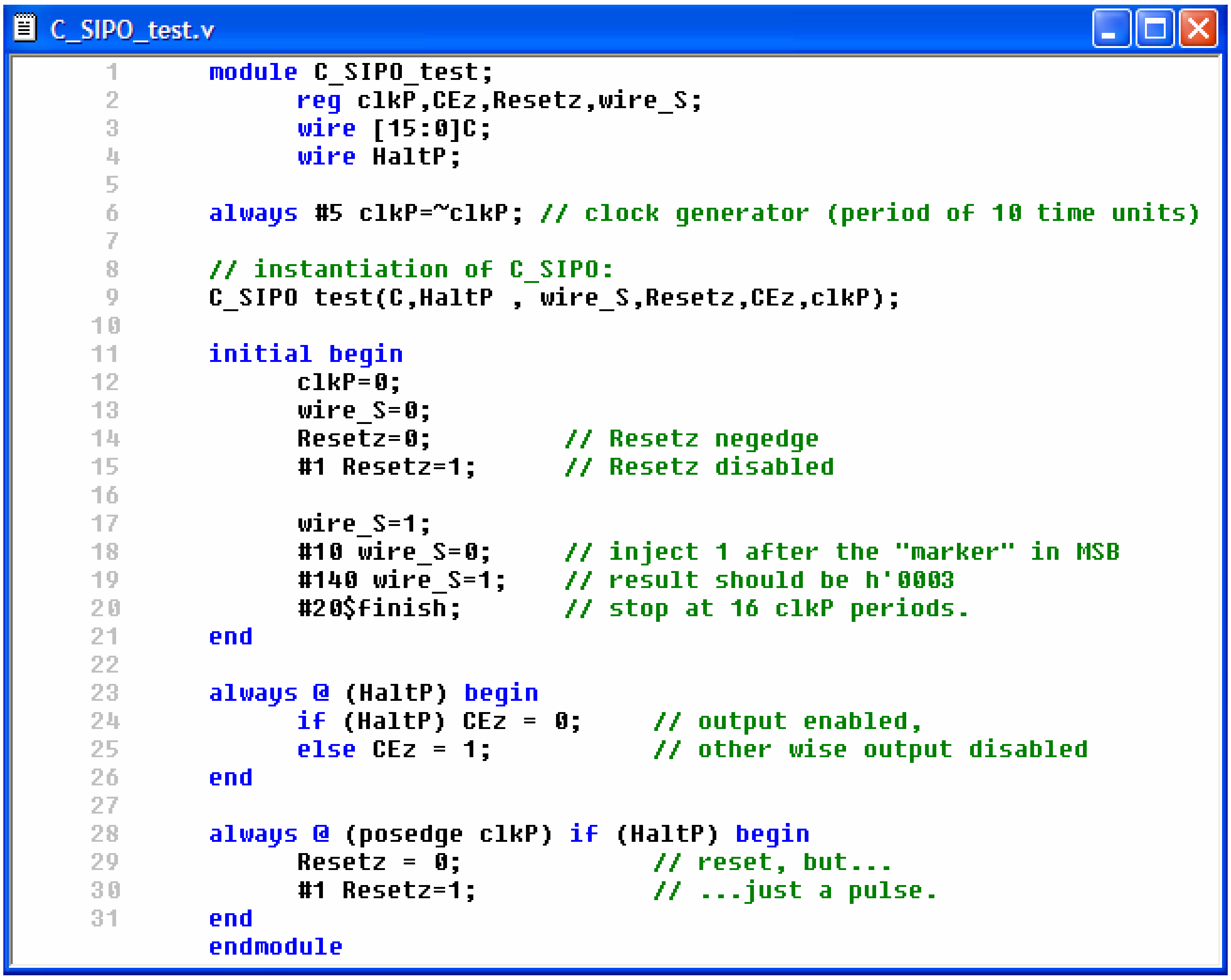
♣ **C_SIPO**: (Serial In, Parallel Out)



This shift register allows implementing the high impedance state when the multiplier is busy, it also allows resetting with the MSB at 1 (it's the marker that will count the 16 clock edges to raise the halt signal when the multiplication is finished) and finally it contains the halt memory cell.



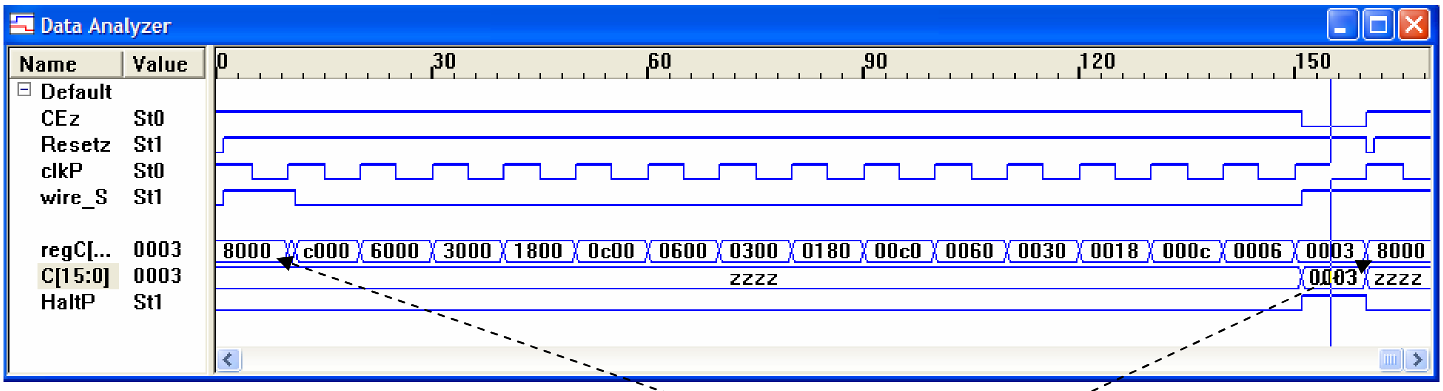
```
module C_SIPO(C,HaltP , wire_S,Resetz,CEz,clkP);
      output [15:0]C;
      output HaltP;
      input wire_S, clkP, Resetz, CEz;
      reg [15:0] C, regC;
      reg HaltP;

initial begin
      regC = 0;
      C = 0;
      HaltP = 0;
end

always@(negedge Resetz) begin
      regC = 16'h8000;
      HaltP = 0;
end

always@(posedge clkP)
if(Resetz) begin
      regC = regC>>1;
      #1 regC[15] = wire_S;
      HaltP = regC[0];
end

always@(CEz or regC)
begin if(!CEz)
      C = regC;
else
      C = 16'hzzzz;
end

endmodule
```


C_SIPO_test.v

```
module C_SIPO_test;
        reg clkP,CEz,Resetz,wire_S;
        wire [15:0]C;
        wire HaltP;

always #5 clkP=~clkP; // clock generator (period of 10 time units)

// instantiation of C_SIPO:
C_SIPO test(C,HaltP , wire_S,Resetz,CEz,clkP);

initial begin
        clkP=0;
        wire_S=0;
        Resetz=0;          // Resetz negedge
        #1 Resetz=1;       // Resetz disabled

        wire_S=1;
        #10 wire_S=0;      // inject 1 after the "marker" in MSB
        #140 wire_S=1;     // result should be h'0003
        #20$finish;        // stop at 16 clkP periods.
end

always @ (HaltP) begin
        if (HaltP) CEz = 0;     // output enabled,
        else CEz = 1;           // other wise output disabled
end

always @ (posedge clkP) if (HaltP) begin
        Resetz = 0;             // reset, but...
        #1 Resetz=1;            // ...just a pulse.
end
endmodule
```

To test if this component shifts correctly, I inject a **1** (just after the marker in the MSB of regC) in its input. The result is simple to expect:

```
1 ->   1000 0000 0000 0000
       1100 0000 0000 0000        AFTER 1 SHIFT
       0110 0000 0000 0000        AFTER 2 SHIFTS
       0011 0000 0000 0000        AFTER 3 SHIFTS…

…      0000 0000 0000 0011        = RESULT EXPECTED (=0x0003)
```



The initialisation is correctly made with the **1** in MSB (the marker) that gives:

0b**1**000 0000 0000 0000 = 0x**8**000

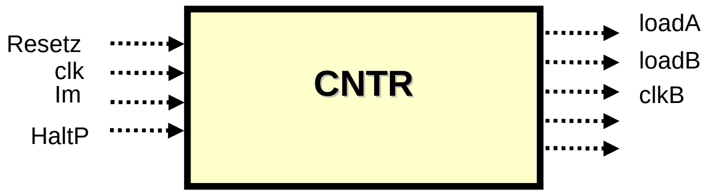
The output is effectively in high impedance state when CEz =1.

And finally, the result expected (0x0003) is obtained.

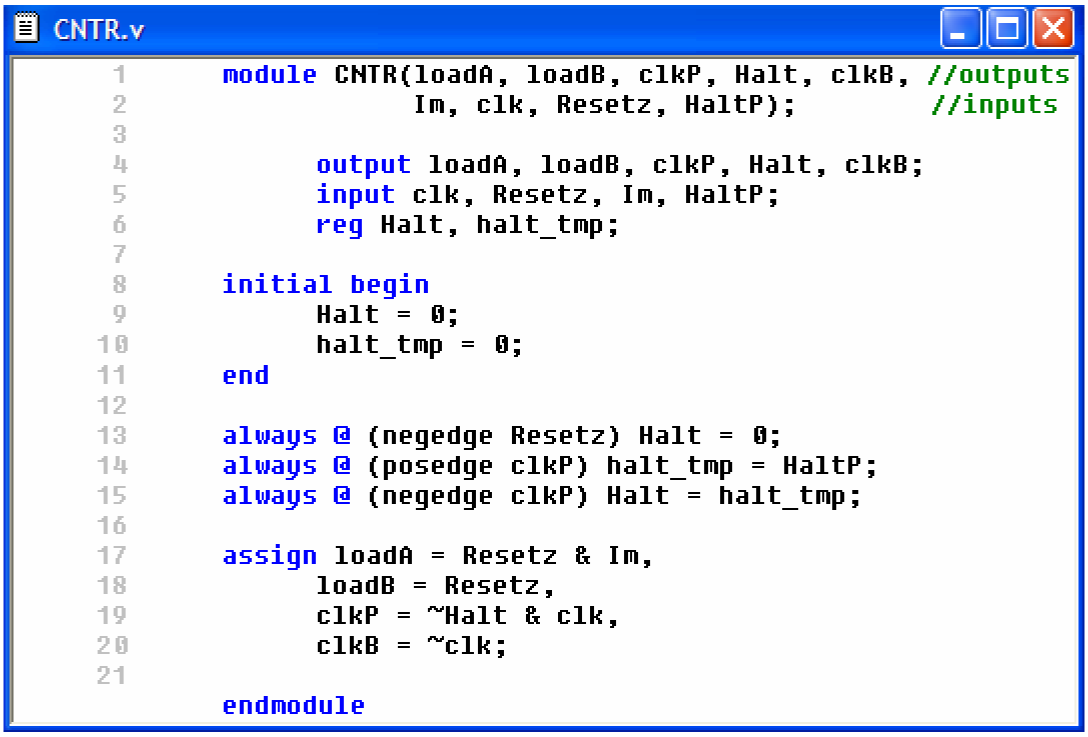
♣ **CNTR**: (control module)

The main purpose of CNTR is to logically generate the signals that are needed to control the modules of the system and maintain synchronicity. This component is thus mainly combinatorial, but we need to delay (synchronously) the HaltP (Provisional) signal then this part is sequential.

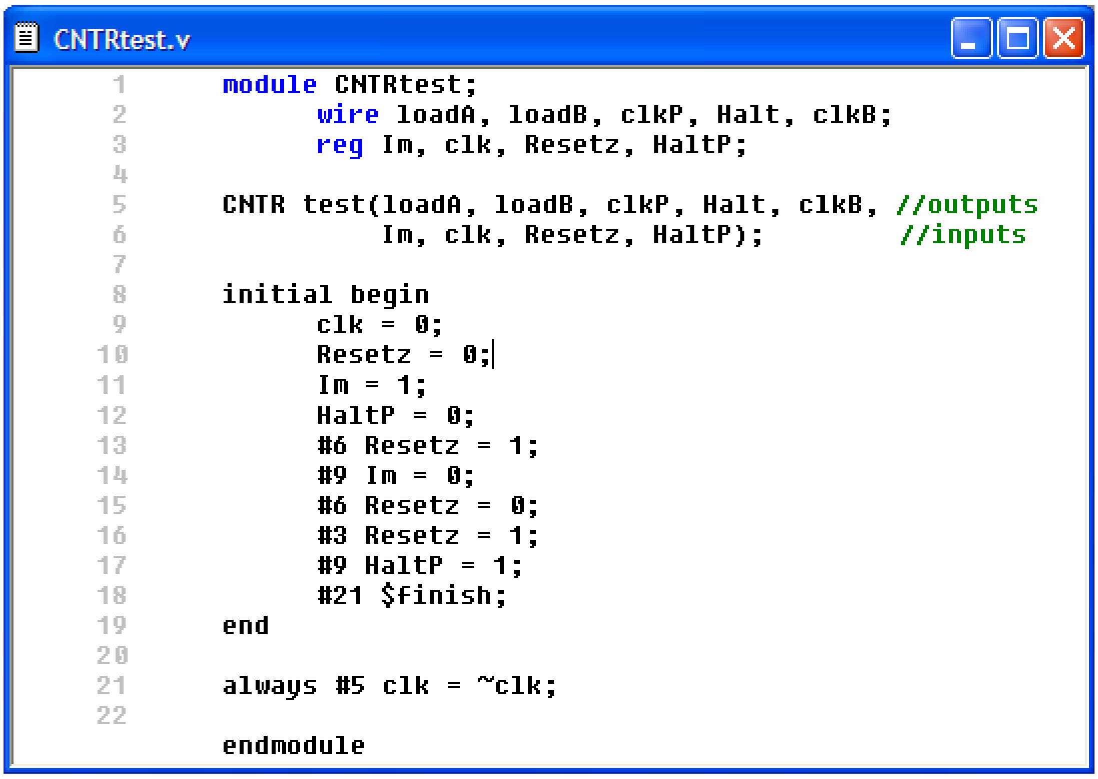
There are two clock signals required in the system to allow for propagation of the multiplication into register C before the next shifting of register B. Therefore clkB is set equal to the negated clock input. The load signals are following Resetz but loadA is only set when Im is also high (when both multiplicands are required).
The Resetz and CEz signals are unchanged then I didn't take care of them (but I need Resetz to load A and B).
The clkP is set so as to be inhibited when Halt is set high (this is the clock that makes the register C and the summ's shift). This is to stop the register C shifting the values out when the multiplication is complete and the full result is in register C. So clkP is assigned by ~Halt & clk.



Halt is reset to zero when the reset signal goes low (system reset) and at the positive edge of the clkP, the Halt signal output from the module is passed to a virtual register halt_tmp, then in the negative edge of PC, the halt_tmp value is passed to the Halt output from the module (which is the real multiplier putput Halt). This creates a delay of one clkP cycle in the propagation of the signal HaltP to Halt. Therefore when the HaltP signal is set high as the LSB of the register C is 1, there is another shift of register C before the halt signal propagates through the CNTR module and the clkP is inhibited. Another reason for inverting clkP is to not loose the first bit of regB, which would be shifted out and lost if the reset signal went low then high before falling edge of the clkP, as the 'mult' module would not be ready to receive it.

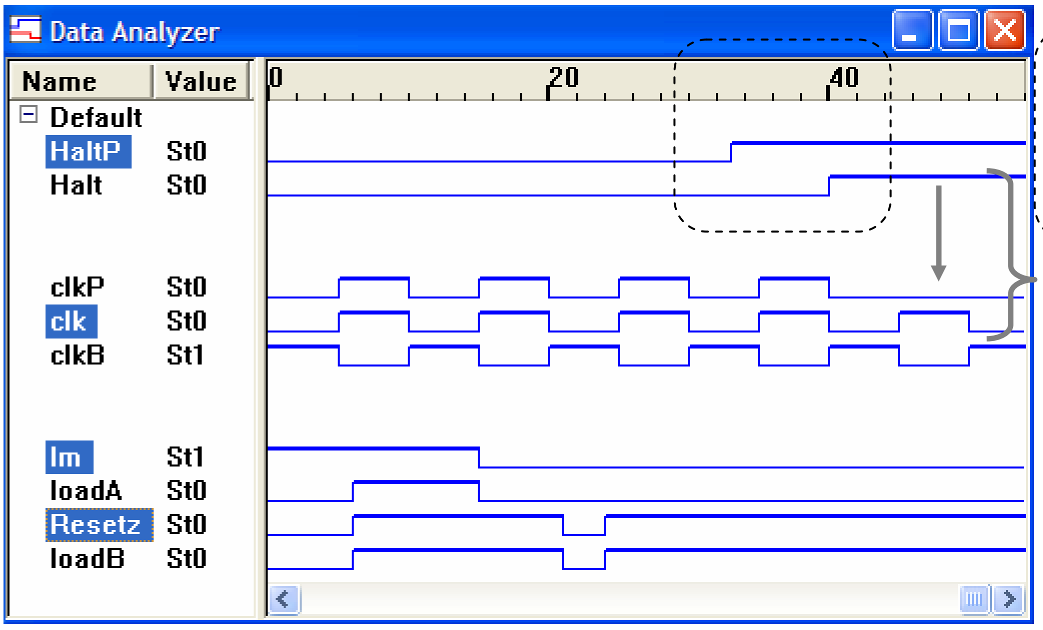


```
module CNTR(loadA, loadB, clkP, Halt, clkB, //outputs
            Im, clk, Resetz, HaltP);        //inputs

      output loadA, loadB, clkP, Halt, clkB;
      input clk, Resetz, Im, HaltP;
      reg Halt, halt_tmp;

initial begin
      Halt = 0;
      halt_tmp = 0;
end

always @ (negedge Resetz) Halt = 0;
always @ (posedge clkP) halt_tmp = HaltP;
always @ (negedge clkP) Halt = halt_tmp;

assign loadA = Resetz & Im,
      loadB = Resetz,
      clkP = ~Halt & clk,
      clkB = ~clk;

endmodule
```



```
module CNTRtest;
      wire loadA, loadB, clkP, Halt, clkB;
      reg Im, clk, Resetz, HaltP;

CNTR test(loadA, loadB, clkP, Halt, clkB, //outputs
          Im, clk, Resetz, HaltP);        //inputs

initial begin
      clk = 0;
      Resetz = 0;
      Im = 1;
      HaltP = 0;
      #6 Resetz = 1;
      #9 Im = 0;
      #6 Resetz = 0;
      #3 Resetz = 1;
      #9 HaltP = 1;
      #21 $finish;
end

always #5 clk = ~clk;

endmodule
```

In the following chronogram, the inputs names are highlighted to be able to distinguish them easily.

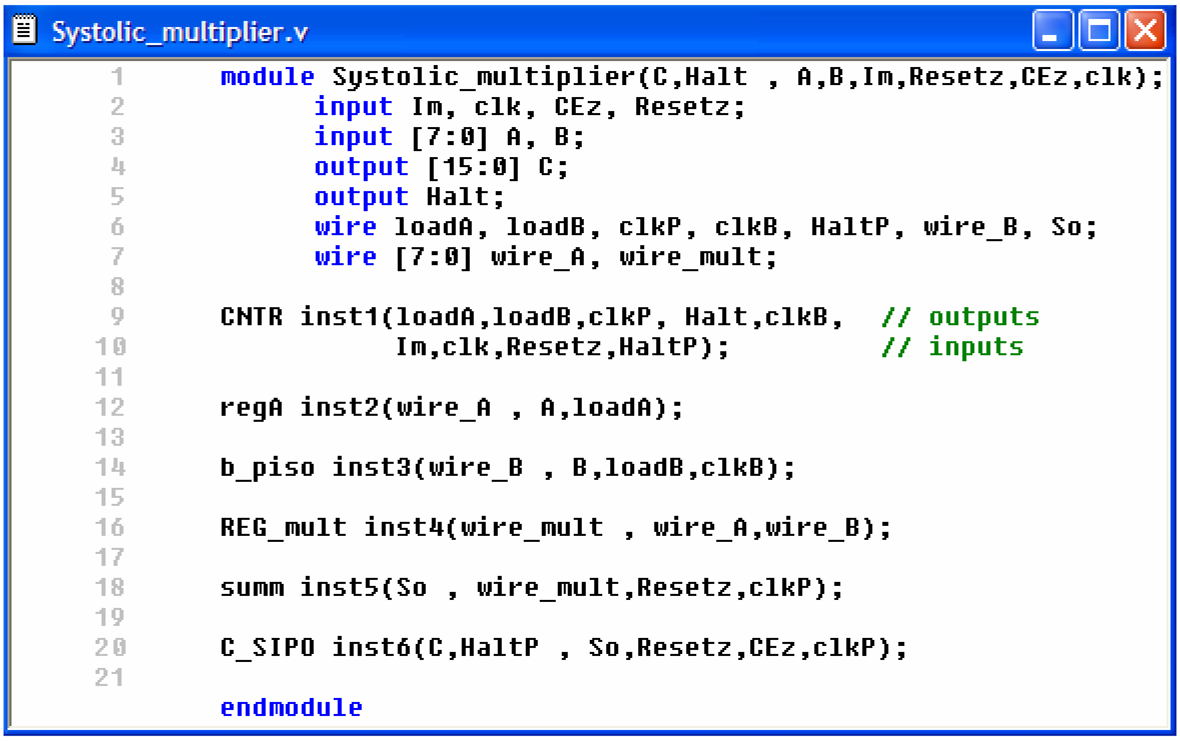


The HaltP input signal is set high at time **33**, and we can see that the Halt signal only goes high at time **40**, the synchronisation is thus good.

The clkP signal is correctly inhibited by Halt and follows the clk as expected; the clkB is ~clk => OK.

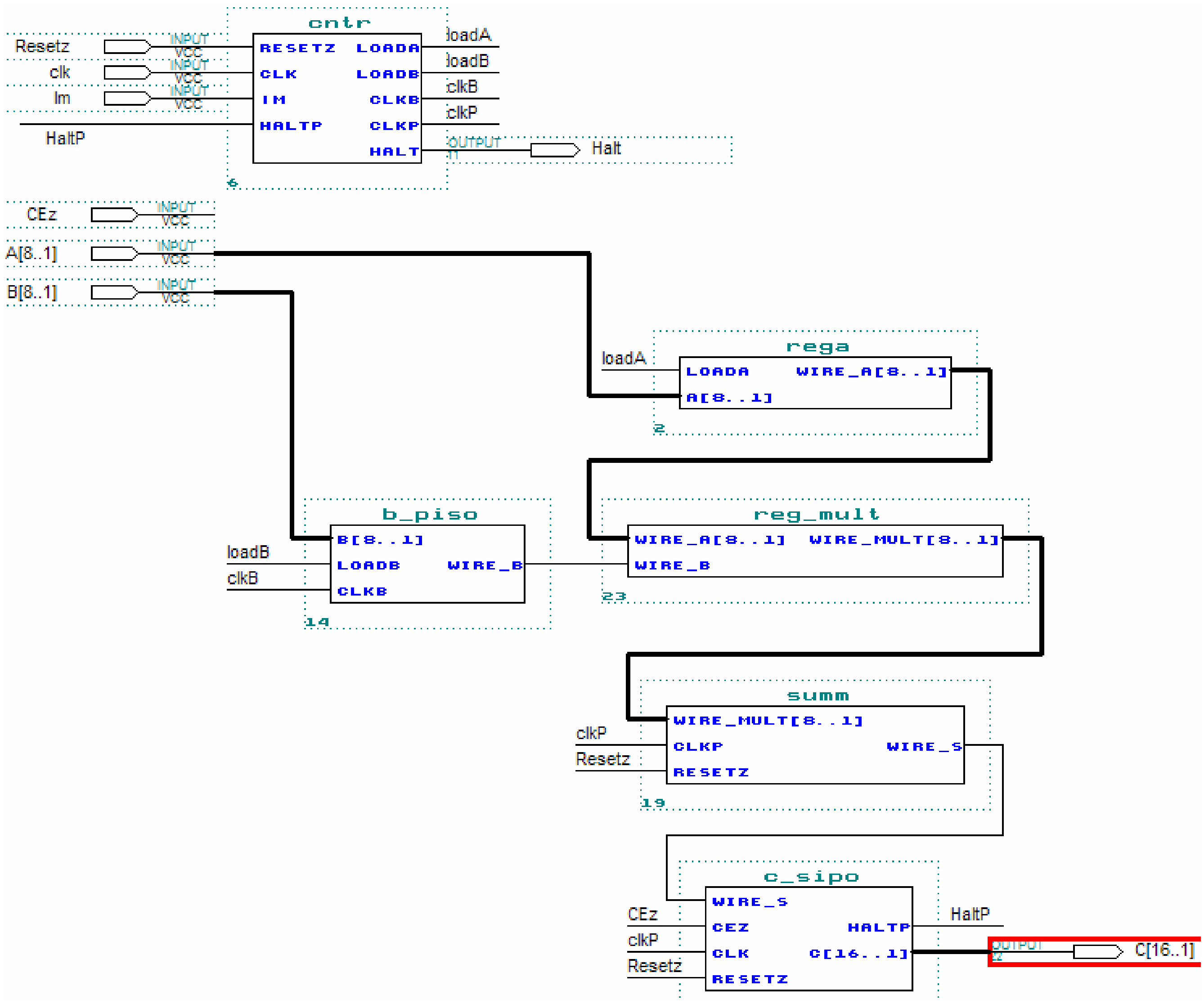
As we can see, the loadA and loadB are risen by Resetz but Im inhibits loadA.

Instantiation of all the components:

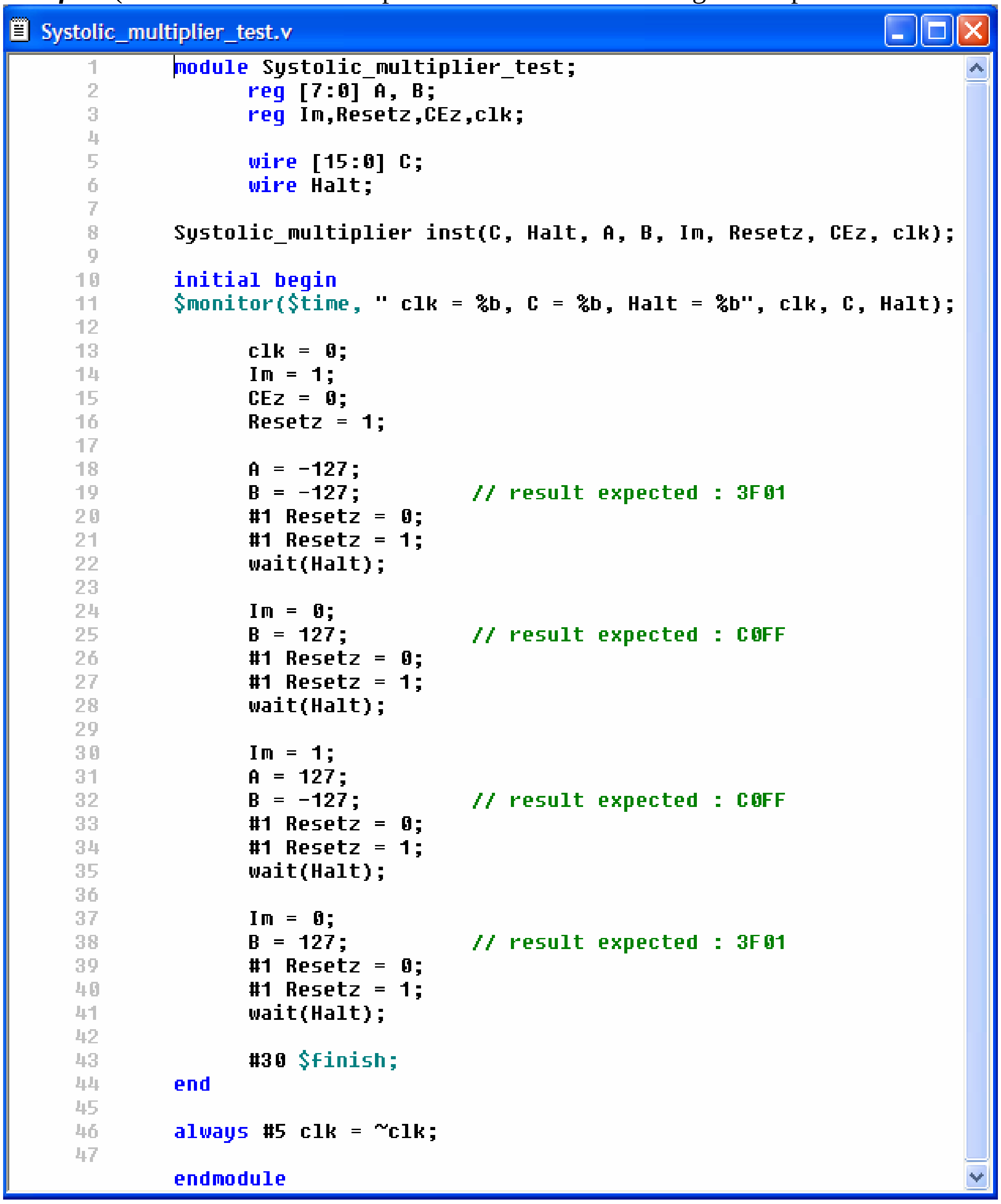


```verilog
module Systolic_multiplier(C,Halt , A,B,Im,Resetz,CEz,clk);
      input Im, clk, CEz, Resetz;
      input [7:0] A, B;
      output [15:0] C;
      output Halt;
      wire loadA, loadB, clkP, clkB, HaltP, wire_B, So;
      wire [7:0] wire_A, wire_mult;

CNTR inst1(loadA,loadB,clkP, Halt,clkB,   // outputs
           Im,clk,Resetz,HaltP);          // inputs

regA inst2(wire_A , A,loadA);

b_piso inst3(wire_B , B,loadB,clkB);

REG_mult inst4(wire_mult , wire_A,wire_B);

summ inst5(So , wire_mult,Resetz,clkP);

C_SIPO inst6(C,HaltP , So,Resetz,CEz,clkP);

endmodule
```

<u>block diagram:</u> (instantiation of all the components using Altera Max+plus®)



Test file: (this first test file is simplified to be able monitoring the output in the result text file, see next page)

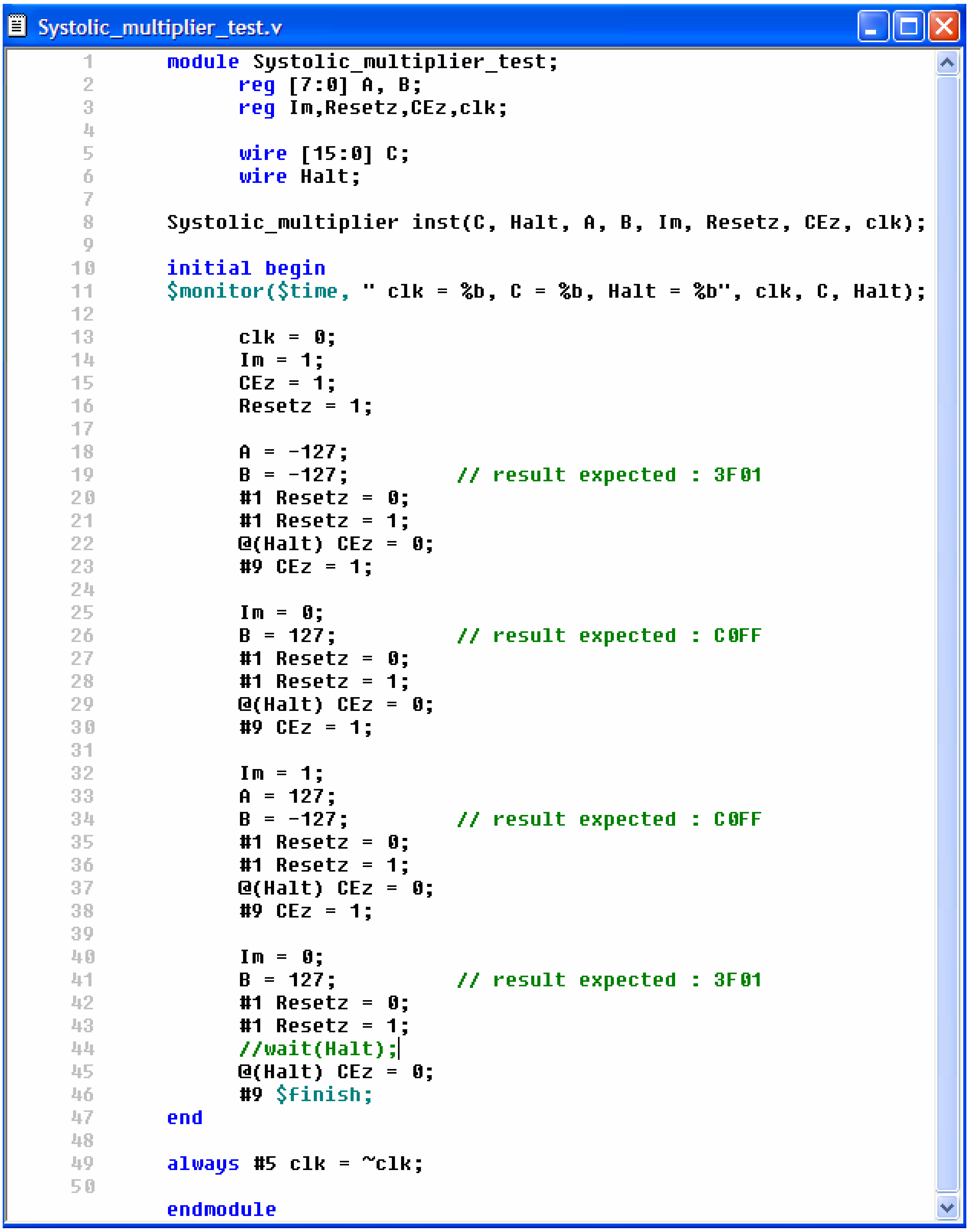


```verilog
module Systolic_multiplier_test;
	reg [7:0] A, B;
	reg Im,Resetz,CEz,clk;

	wire [15:0] C;
	wire Halt;

Systolic_multiplier inst(C, Halt, A, B, Im, Resetz, CEz, clk);

initial begin
$monitor($time, " clk = %b, C = %b, Halt = %b", clk, C, Halt);

	clk = 0;
	Im = 1;
	CEz = 0;
	Resetz = 1;

	A = -127;
	B = -127;		// result expected : 3F01
	#1 Resetz = 0;
	#1 Resetz = 1;
	wait(Halt);

	Im = 0;
	B = 127;		// result expected : C0FF
	#1 Resetz = 0;
	#1 Resetz = 1;
	wait(Halt);

	Im = 1;
	A = 127;
	B = -127;		// result expected : C0FF
	#1 Resetz = 0;
	#1 Resetz = 1;
	wait(Halt);

	Im = 0;
	B = 127;		// result expected : 3F01
	#1 Resetz = 0;
	#1 Resetz = 1;
	wait(Halt);

	#30 $finish;
end

always #5 clk = ~clk;

endmodule
```

7 Description of the results:

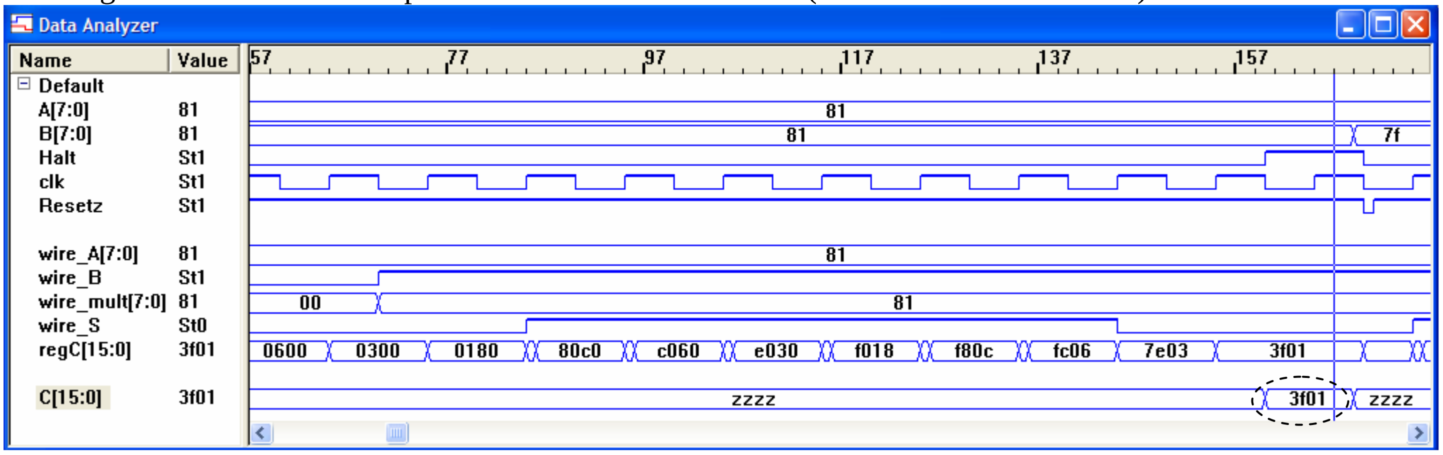
```
  1 clk = 0, C = 1000000000000000, Halt = 0
  5 clk = 1, C = 0100000000000000, Halt = 0
  6 clk = 1, C = 1100000000000000, Halt = 0
 10 clk = 0, C = 1100000000000000, Halt = 0
 15 clk = 1, C = 0110000000000000, Halt = 0
 20 clk = 0, C = 0110000000000000, Halt = 0
 25 clk = 1, C = 0011000000000000, Halt = 0
 30 clk = 0, C = 0011000000000000, Halt = 0
 35 clk = 1, C = 0001100000000000, Halt = 0
 40 clk = 0, C = 0001100000000000, Halt = 0
 45 clk = 1, C = 0000110000000000, Halt = 0
 50 clk = 0, C = 0000110000000000, Halt = 0
 55 clk = 1, C = 0000011000000000, Halt = 0
 60 clk = 0, C = 0000011000000000, Halt = 0
 65 clk = 1, C = 0000001100000000, Halt = 0
 70 clk = 0, C = 0000001100000000, Halt = 0
 75 clk = 1, C = 0000000110000000, Halt = 0
 80 clk = 0, C = 0000000110000000, Halt = 0
 85 clk = 1, C = 0000000011000000, Halt = 0
 86 clk = 1, C = 1000000011000000, Halt = 0
 90 clk = 0, C = 1000000011000000, Halt = 0
 95 clk = 1, C = 0100000001100000, Halt = 0
 96 clk = 1, C = 1100000001100000, Halt = 0
100 clk = 0, C = 1100000001100000, Halt = 0
105 clk = 1, C = 0110000000110000, Halt = 0
106 clk = 1, C = 1110000000110000, Halt = 0
110 clk = 0, C = 1110000000110000, Halt = 0
115 clk = 1, C = 0111000000011000, Halt = 0
116 clk = 1, C = 1111000000011000, Halt = 0
120 clk = 0, C = 1111000000011000, Halt = 0
125 clk = 1, C = 0111100000001100, Halt = 0
126 clk = 1, C = 1111100000001100, Halt = 0
130 clk = 0, C = 1111100000001100, Halt = 0
135 clk = 1, C = 0111110000000110, Halt = 0
136 clk = 1, C = 1111110000000110, Halt = 0
140 clk = 0, C = 1111110000000110, Halt = 0
145 clk = 1, C = 0111111000000011, Halt = 0
150 clk = 0, C = 0111111000000011, Halt = 0
155 clk = 1, C = 0011111100000001, Halt = 0
160 clk = 0, C = 0011111100000001, Halt = 1 => 30F1
161 clk = 0, C = 1000000000000000, Halt = 0
165 clk = 1, C = 0100000000000000, Halt = 0
166 clk = 1, C = 1100000000000000, Halt = 0
170 clk = 0, C = 1100000000000000, Halt = 0
175 clk = 1, C = 0110000000000000, Halt = 0
176 clk = 1, C = 1110000000000000, Halt = 0
180 clk = 0, C = 1110000000000000, Halt = 0
185 clk = 1, C = 0111000000000000, Halt = 0
186 clk = 1, C = 1111000000000000, Halt = 0
190 clk = 0, C = 1111000000000000, Halt = 0
195 clk = 1, C = 0111100000000000, Halt = 0
196 clk = 1, C = 1111100000000000, Halt = 0
200 clk = 0, C = 1111100000000000, Halt = 0
205 clk = 1, C = 0111110000000000, Halt = 0
206 clk = 1, C = 1111110000000000, Halt = 0
210 clk = 0, C = 1111110000000000, Halt = 0
215 clk = 1, C = 0111111000000000, Halt = 0
216 clk = 1, C = 1111111000000000, Halt = 0
220 clk = 0, C = 1111111000000000, Halt = 0
225 clk = 1, C = 0111111100000000, Halt = 0
226 clk = 1, C = 1111111100000000, Halt = 0
230 clk = 0, C = 1111111100000000, Halt = 0
235 clk = 1, C = 0111111110000000, Halt = 0
236 clk = 1, C = 1111111110000000, Halt = 0
240 clk = 0, C = 1111111110000000, Halt = 0
245 clk = 1, C = 0111111111000000, Halt = 0
250 clk = 0, C = 0111111111000000, Halt = 0
255 clk = 1, C = 0011111111100000, Halt = 0
260 clk = 0, C = 0011111111100000, Halt = 0
265 clk = 1, C = 0001111111110000, Halt = 0
270 clk = 0, C = 0001111111110000, Halt = 0
275 clk = 1, C = 0000111111111000, Halt = 0
280 clk = 0, C = 0000111111111000, Halt = 0
285 clk = 1, C = 0000011111111100, Halt = 0
290 clk = 0, C = 0000011111111100, Halt = 0
295 clk = 1, C = 0000001111111110, Halt = 0
300 clk = 0, C = 0000001111111110, Halt = 0
305 clk = 1, C = 0000000111111111, Halt = 0
306 clk = 1, C = 1000000111111111, Halt = 0
310 clk = 0, C = 1000000111111111, Halt = 0
315 clk = 1, C = 0100000011111111, Halt = 0
316 clk = 1, C = 1100000011111111, Halt = 0
320 clk = 0, C = 1100000011111111, Halt = 1 => C0FF
321 clk = 0, C = 1000000000000000, Halt = 0
325 clk = 1, C = 0100000000000000, Halt = 0
326 clk = 1, C = 1100000000000000, Halt = 0
330 clk = 0, C = 1100000000000000, Halt = 0
335 clk = 1, C = 0110000000000000, Halt = 0
336 clk = 1, C = 1110000000000000, Halt = 0
340 clk = 0, C = 1110000000000000, Halt = 0
345 clk = 1, C = 0111000000000000, Halt = 0
346 clk = 1, C = 1111000000000000, Halt = 0
350 clk = 0, C = 1111000000000000, Halt = 0
355 clk = 1, C = 0111100000000000, Halt = 0
356 clk = 1, C = 1111100000000000, Halt = 0
360 clk = 0, C = 1111100000000000, Halt = 0
365 clk = 1, C = 0111110000000000, Halt = 0
366 clk = 1, C = 1111110000000000, Halt = 0
370 clk = 0, C = 1111110000000000, Halt = 0
375 clk = 1, C = 0111111000000000, Halt = 0
376 clk = 1, C = 1111111000000000, Halt = 0
380 clk = 0, C = 1111111000000000, Halt = 0
385 clk = 1, C = 0111111100000000, Halt = 0
386 clk = 1, C = 1111111100000000, Halt = 0
390 clk = 0, C = 1111111100000000, Halt = 0
395 clk = 1, C = 0111111110000000, Halt = 0
396 clk = 1, C = 1111111110000000, Halt = 0
400 clk = 0, C = 1111111110000000, Halt = 0
405 clk = 1, C = 0111111111000000, Halt = 0
410 clk = 0, C = 0111111111000000, Halt = 0
415 clk = 1, C = 0011111111100000, Halt = 0
420 clk = 0, C = 0011111111100000, Halt = 0
425 clk = 1, C = 0001111111110000, Halt = 0
430 clk = 0, C = 0001111111110000, Halt = 0
435 clk = 1, C = 0000111111111000, Halt = 0
440 clk = 0, C = 0000111111111000, Halt = 0
445 clk = 1, C = 0000011111111100, Halt = 0
450 clk = 0, C = 0000011111111100, Halt = 0
455 clk = 1, C = 0000001111111110, Halt = 0
460 clk = 0, C = 0000001111111110, Halt = 0
465 clk = 1, C = 0000000111111111, Halt = 0
466 clk = 1, C = 1000000111111111, Halt = 0
470 clk = 0, C = 1000000111111111, Halt = 0
475 clk = 1, C = 0100000011111111, Halt = 0
476 clk = 1, C = 1100000011111111, Halt = 0
480 clk = 0, C = 1100000011111111, Halt = 1 => C0FF
481 clk = 0, C = 1000000000000000, Halt = 0
485 clk = 1, C = 0100000000000000, Halt = 0
486 clk = 1, C = 1100000000000000, Halt = 0
490 clk = 0, C = 1100000000000000, Halt = 0
495 clk = 1, C = 0110000000000000, Halt = 0
500 clk = 0, C = 0110000000000000, Halt = 0
505 clk = 1, C = 0011000000000000, Halt = 0
510 clk = 0, C = 0011000000000000, Halt = 0
515 clk = 1, C = 0001100000000000, Halt = 0
520 clk = 0, C = 0001100000000000, Halt = 0
525 clk = 1, C = 0000110000000000, Halt = 0
530 clk = 0, C = 0000110000000000, Halt = 0
535 clk = 1, C = 0000011000000000, Halt = 0
540 clk = 0, C = 0000011000000000, Halt = 0
545 clk = 1, C = 0000001100000000, Halt = 0
550 clk = 0, C = 0000001100000000, Halt = 0
555 clk = 1, C = 0000000110000000, Halt = 0
560 clk = 0, C = 0000000110000000, Halt = 0
565 clk = 1, C = 0000000011000000, Halt = 0
566 clk = 1, C = 1000000011000000, Halt = 0
570 clk = 0, C = 1000000011000000, Halt = 0
575 clk = 1, C = 0100000001100000, Halt = 0
576 clk = 1, C = 1100000001100000, Halt = 0
580 clk = 0, C = 1100000001100000, Halt = 0
585 clk = 1, C = 0110000000110000, Halt = 0
586 clk = 1, C = 1110000000110000, Halt = 0
590 clk = 0, C = 1110000000110000, Halt = 0
595 clk = 1, C = 0111000000011000, Halt = 0
596 clk = 1, C = 1111000000011000, Halt = 0
600 clk = 0, C = 1111000000011000, Halt = 0
605 clk = 1, C = 0111100000001100, Halt = 0
606 clk = 1, C = 1111100000001100, Halt = 0
610 clk = 0, C = 1111100000001100, Halt = 0
615 clk = 1, C = 0111110000000110, Halt = 0
616 clk = 1, C = 1111110000000110, Halt = 0
620 clk = 0, C = 1111110000000110, Halt = 0
625 clk = 1, C = 0111111000000011, Halt = 0
630 clk = 0, C = 0111111000000011, Halt = 0
635 clk = 1, C = 0011111100000001, Halt = 0 => 30F1
                         ...
```

For the previous result text file I thus fully enabled the chip output (no state Z) to be able to see its evolution. But as the multiplier is supposed to disable its output when the result is not ready, I changed a little the test file to implement this functionality (by setting CEz at 1 when the device is busy, which places C in a high impedance state).



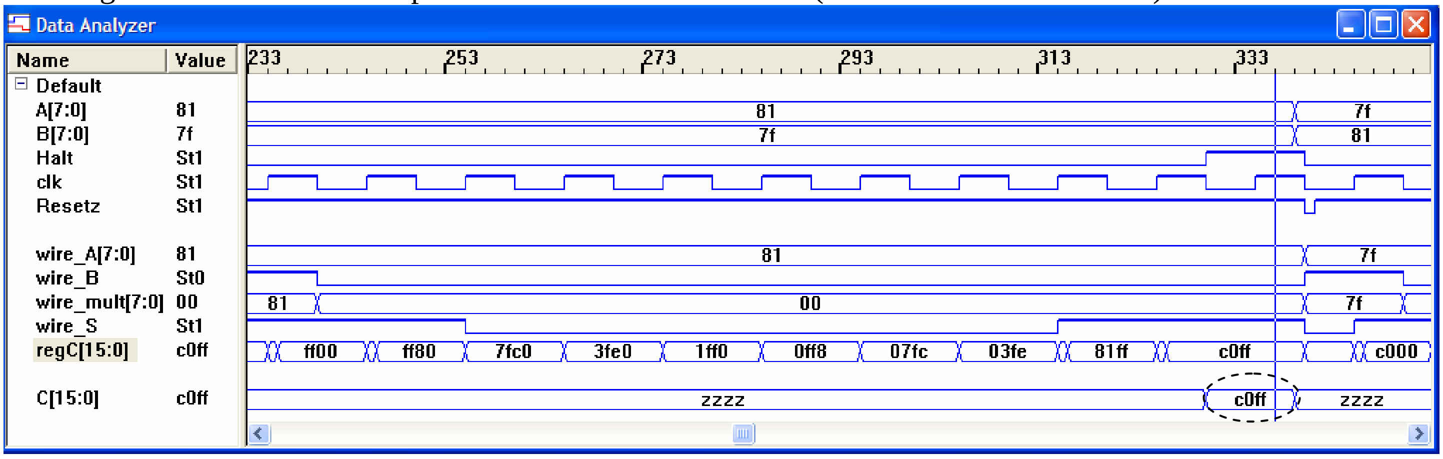
```verilog
module Systolic_multiplier_test;
      reg [7:0] A, B;
      reg Im,Resetz,CEz,clk;

      wire [15:0] C;
      wire Halt;

Systolic_multiplier inst(C, Halt, A, B, Im, Resetz, CEz, clk);

initial begin
$monitor($time, " clk = %b, C = %b, Halt = %b", clk, C, Halt);

      clk = 0;
      Im = 1;
      CEz = 1;
      Resetz = 1;

      A = -127;
      B = -127;         // result expected : 3F01
      #1 Resetz = 0;
      #1 Resetz = 1;
      @(Halt) CEz = 0;
      #9 CEz = 1;

      Im = 0;
      B = 127;          // result expected : C0FF
      #1 Resetz = 0;
      #1 Resetz = 1;
      @(Halt) CEz = 0;
      #9 CEz = 1;

      Im = 1;
      A = 127;
      B = -127;         // result expected : C0FF
      #1 Resetz = 0;
      #1 Resetz = 1;
      @(Halt) CEz = 0;
      #9 CEz = 1;

      Im = 0;
      B = 127;          // result expected : 3F01
      #1 Resetz = 0;
      #1 Resetz = 1;
      //wait(Halt);
      @(Halt) CEz = 0;
      #9 $finish;
end

always #5 clk = ~clk;

endmodule
```

Chronogram result for 1st multiplication: -127 × -127 = 16129 (= 0x81 × 0x81 = 0x3F01)



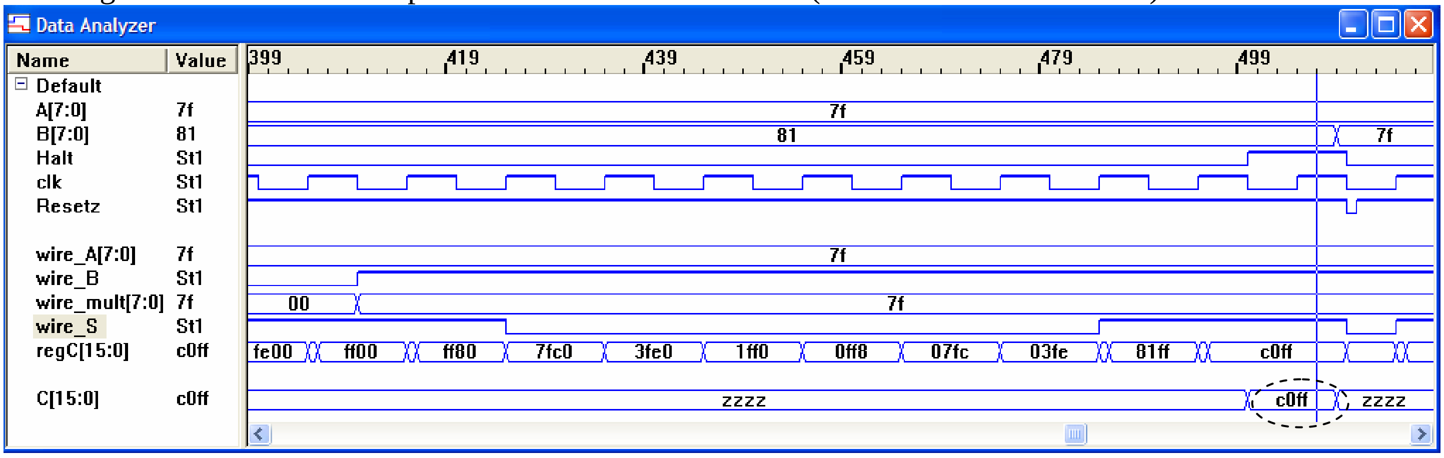
Name	Value
Default	
A[7:0]	81
B[7:0]	81
Halt	St1
clk	St1
Resetz	St1
wire_A[7:0]	81
wire_B	St1
wire_mult[7:0]	81
wire_S	St0
regC[15:0]	3f01
C[15:0]	3f01

Chronogram result for 2nd multiplication: -127 × 127 = -16129 (= 0x81 × 0x7F = 0xC0FF)



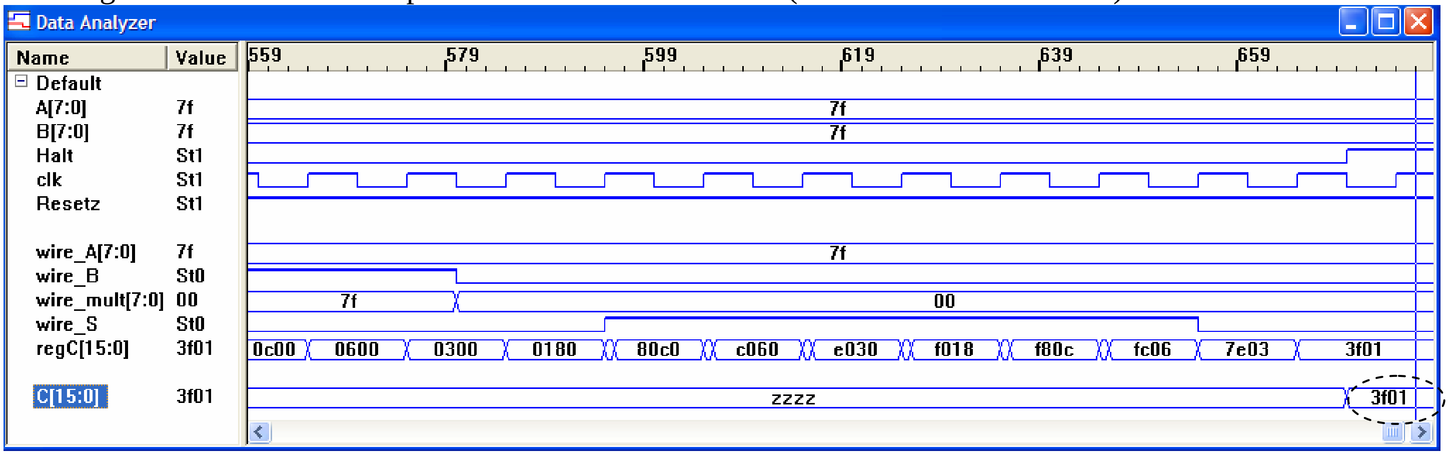
Name	Value
Default	
A[7:0]	81
B[7:0]	7f
Halt	St1
clk	St1
Resetz	St1
wire_A[7:0]	81
wire_B	St0
wire_mult[7:0]	00
wire_S	St1
regC[15:0]	c0ff
C[15:0]	c0ff

Chronogram result for 3rd multiplication: 127 × -127 = -16129 (= 0x7F × 0x81 = 0xC0FF)



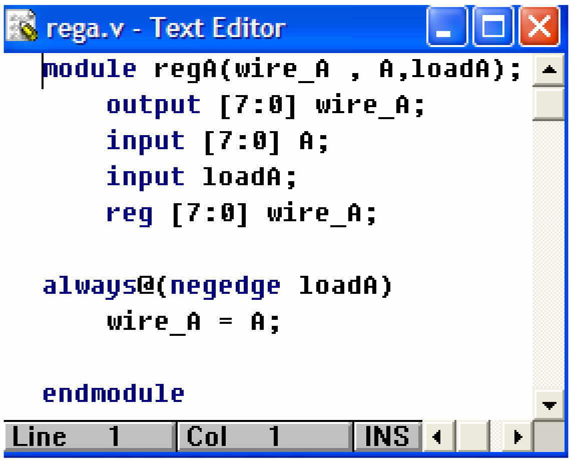
Name	Value
Default	
A[7:0]	7f
B[7:0]	81
Halt	St1
clk	St1
Resetz	St1
wire_A[7:0]	7f
wire_B	St1
wire_mult[7:0]	7f
wire_S	St1
regC[15:0]	c0ff
C[15:0]	c0ff

Chronogram result for 4th multiplication: 127 × 127 = 16129 (=0x7F × 0x7F = 0x3F01)

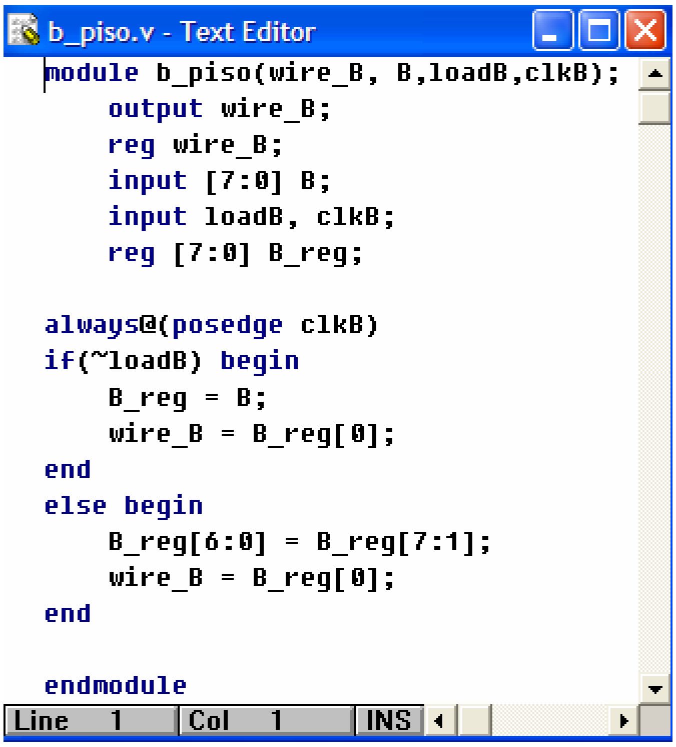


Name	Value
Default	
A[7:0]	7f
B[7:0]	7f
Halt	St1
clk	St1
Resetz	St1
wire_A[7:0]	7f
wire_B	St0
wire_mult[7:0]	00
wire_S	St0
regC[15:0]	3f01
C[15:0]	3f01

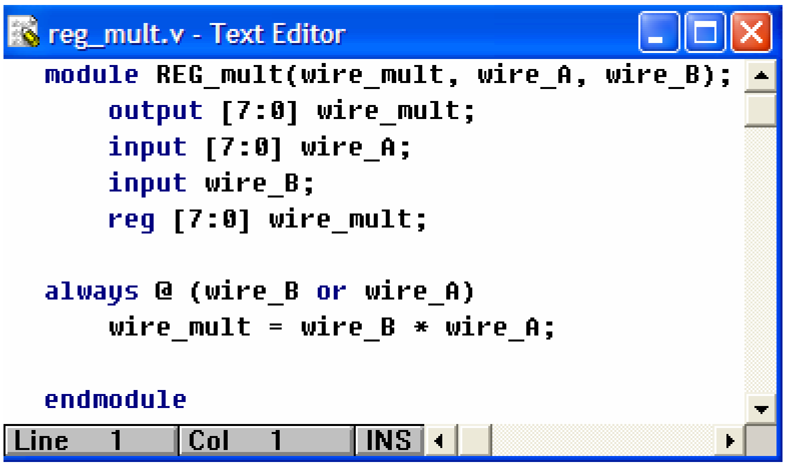
<u>BONUS :</u> I wanted to test the multiplier a little more, then I've done a simulation of the Verilog code in Max+plus. As this software is not working exactly like Silos, I changed some of the codes:
(the main difference is that the "initial" function is not really appreciated by Max+plus)



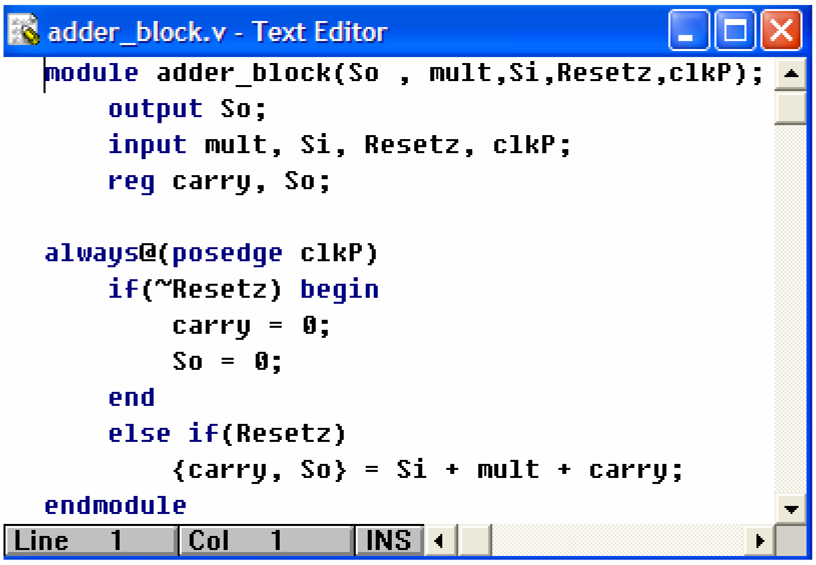
```
module regA(wire_A , A,loadA);
    output [7:0] wire_A;
    input [7:0] A;
    input loadA;
    reg [7:0] wire_A;

always@(negedge loadA)
    wire_A = A;

endmodule
```



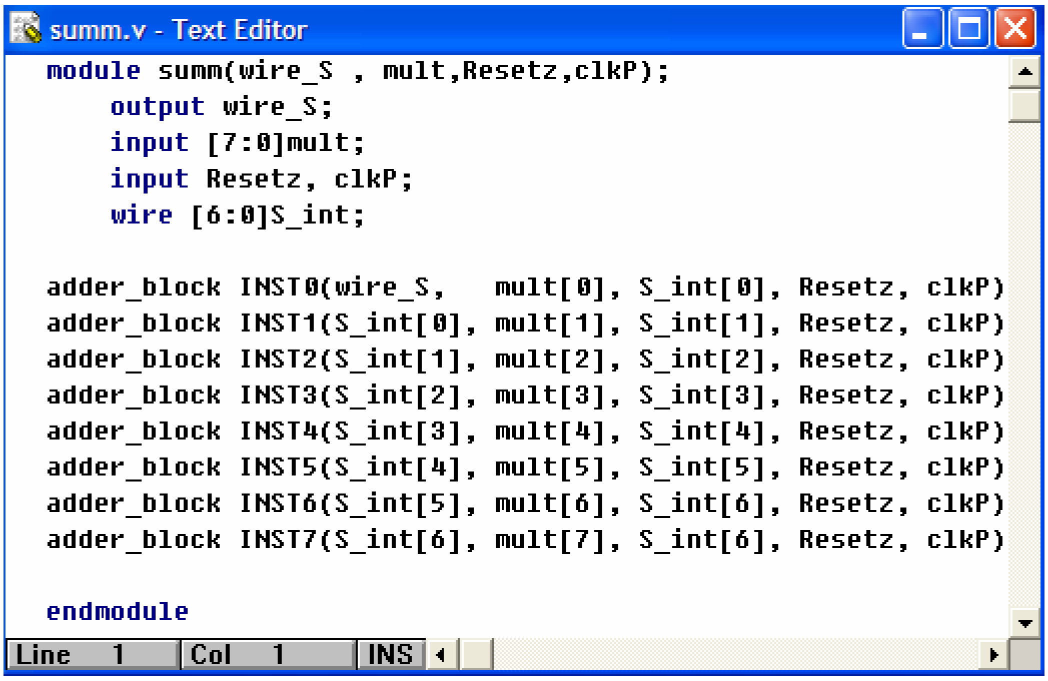
```
module b_piso(wire_B, B,loadB,clkB);
    output wire_B;
    reg wire_B;
    input [7:0] B;
    input loadB, clkB;
    reg [7:0] B_reg;

always@(posedge clkB)
if(~loadB) begin
    B_reg = B;
    wire_B = B_reg[0];
end
else begin
    B_reg[6:0] = B_reg[7:1];
    wire_B = B_reg[0];
end

endmodule
```



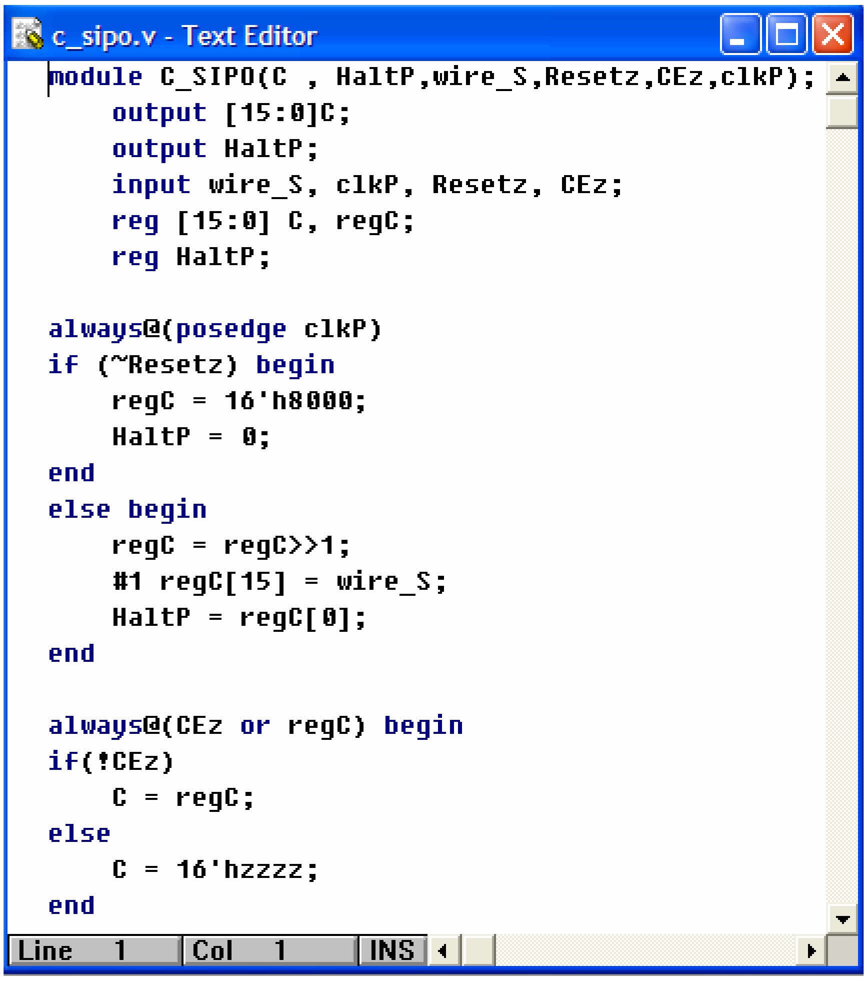
```
module REG_mult(wire_mult, wire_A, wire_B);
    output [7:0] wire_mult;
    input [7:0] wire_A;
    input wire_B;
    reg [7:0] wire_mult;

always @ (wire_B or wire_A)
    wire_mult = wire_B * wire_A;

endmodule
```



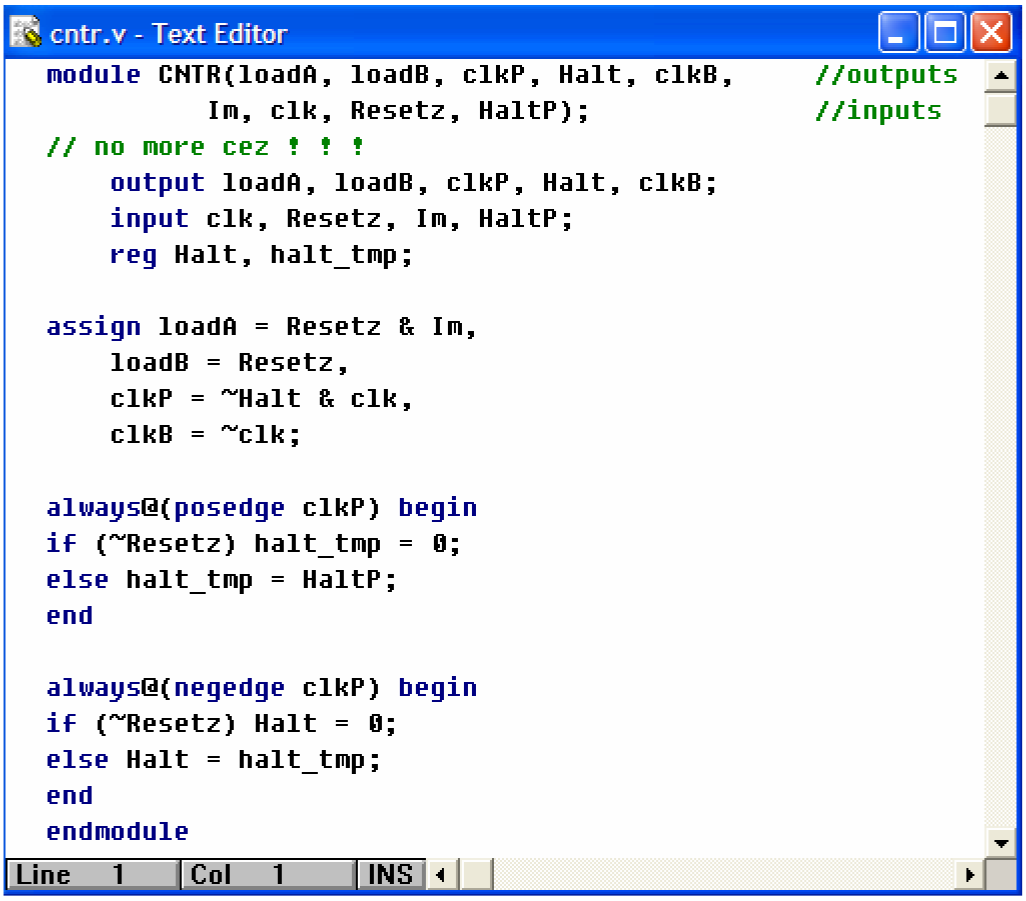
```
module adder_block(So , mult,Si,Resetz,clkP);
    output So;
    input mult, Si, Resetz, clkP;
    reg carry, So;

always@(posedge clkP)
    if(~Resetz) begin
        carry = 0;
        So = 0;
    end
    else if(Resetz)
        {carry, So} = Si + mult + carry;
endmodule
```



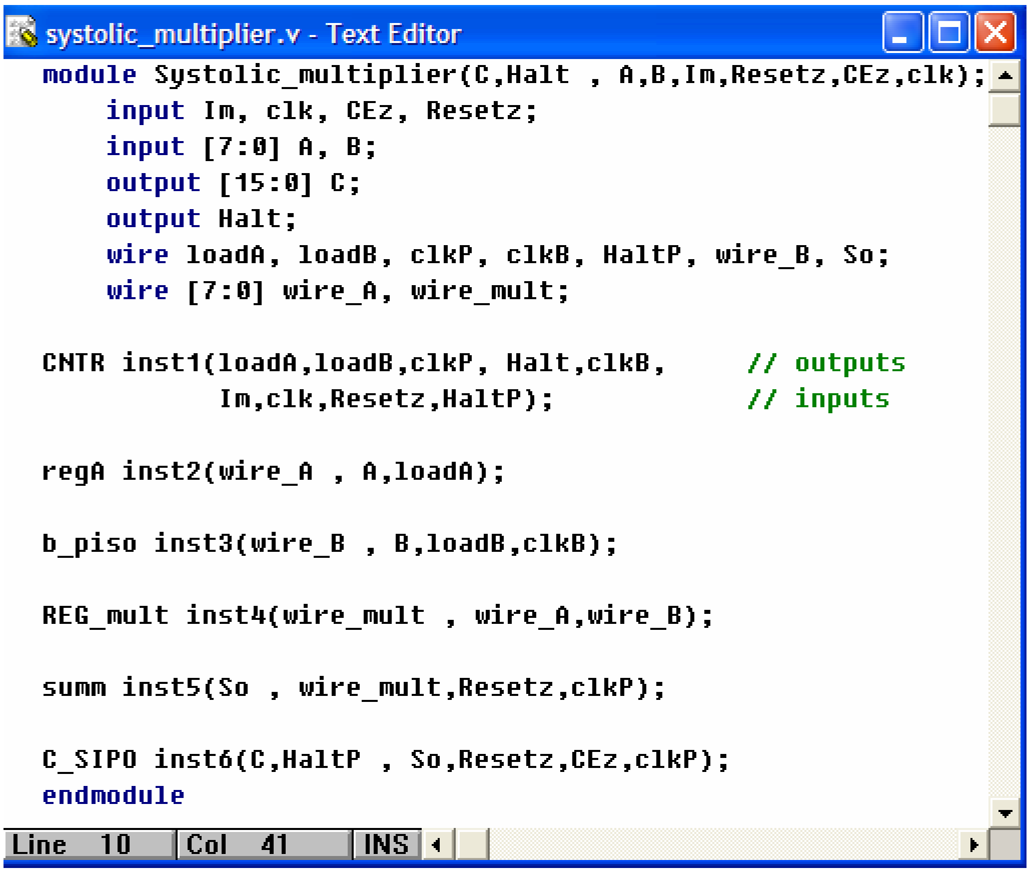
```
module summ(wire_S , mult,Resetz,clkP);
    output wire_S;
    input [7:0]mult;
    input Resetz, clkP;
    wire [6:0]S_int;

adder_block INST0(wire_S,   mult[0], S_int[0], Resetz, clkP)
adder_block INST1(S_int[0], mult[1], S_int[1], Resetz, clkP)
adder_block INST2(S_int[1], mult[2], S_int[2], Resetz, clkP)
adder_block INST3(S_int[2], mult[3], S_int[3], Resetz, clkP)
adder_block INST4(S_int[3], mult[4], S_int[4], Resetz, clkP)
adder_block INST5(S_int[4], mult[5], S_int[5], Resetz, clkP)
adder_block INST6(S_int[5], mult[6], S_int[6], Resetz, clkP)
adder_block INST7(S_int[6], mult[7], S_int[6], Resetz, clkP)

endmodule
```



```
module C_SIPO(C , HaltP,wire_S,Resetz,CEz,clkP);
    output [15:0]C;
    output HaltP;
    input wire_S, clkP, Resetz, CEz;
    reg [15:0] C, regC;
    reg HaltP;

always@(posedge clkP)
if (~Resetz) begin
    regC = 16'h8000;
    HaltP = 0;
end
else begin
    regC = regC>>1;
    #1 regC[15] = wire_S;
    HaltP = regC[0];
end

always@(CEz or regC) begin
if(!CEz)
    C = regC;
else
    C = 16'hzzzz;
end
```



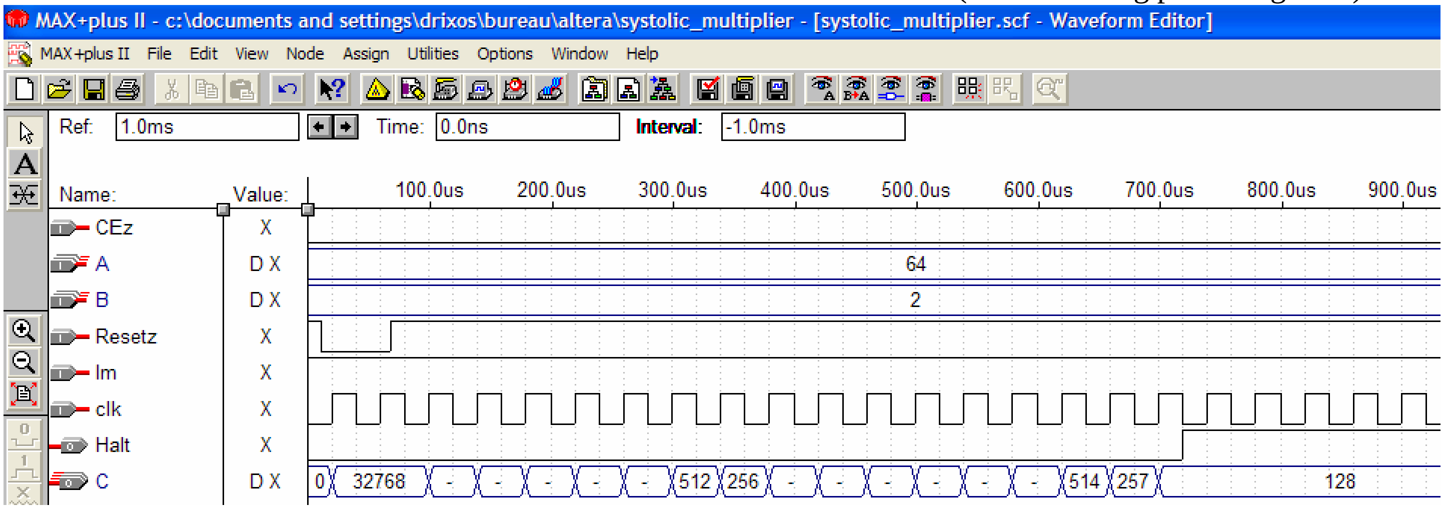
```
module CNTR(loadA, loadB, clkP, Halt, clkB,      //outputs
          Im, clk, Resetz, HaltP);               //inputs
// no more cez ! ! !
    output loadA, loadB, clkP, Halt, clkB;
    input clk, Resetz, Im, HaltP;
    reg Halt, halt_tmp;

assign loadA = Resetz & Im,
    loadB = Resetz,
    clkP = ~Halt & clk,
    clkB = ~clk;

always@(posedge clkP) begin
if (~Resetz) halt_tmp = 0;
else halt_tmp = HaltP;
end

always@(negedge clkP) begin
if (~Resetz) Halt = 0;
else Halt = halt_tmp;
end
endmodule
```


systolic_multiplier.v - Text Editor

```
module Systolic_multiplier(C,Halt , A,B,Im,Resetz,CEz,clk);
    input Im, clk, CEz, Resetz;
    input [7:0] A, B;
    output [15:0] C;
    output Halt;
    wire loadA, loadB, clkP, clkB, HaltP, wire_B, So;
    wire [7:0] wire_A, wire_mult;

CNTR inst1(loadA,loadB,clkP, Halt,clkB,     // outputs
           Im,clk,Resetz,HaltP);            // inputs

regA inst2(wire_A , A,loadA);

b_piso inst3(wire_B , B,loadB,clkB);

REG_mult inst4(wire_mult , wire_A,wire_B);

summ inst5(So , wire_mult,Resetz,clkP);

C_SIPO inst6(C,HaltP , So,Resetz,CEz,clkP);
endmodule
```

Line 10 | Col 41 | INS

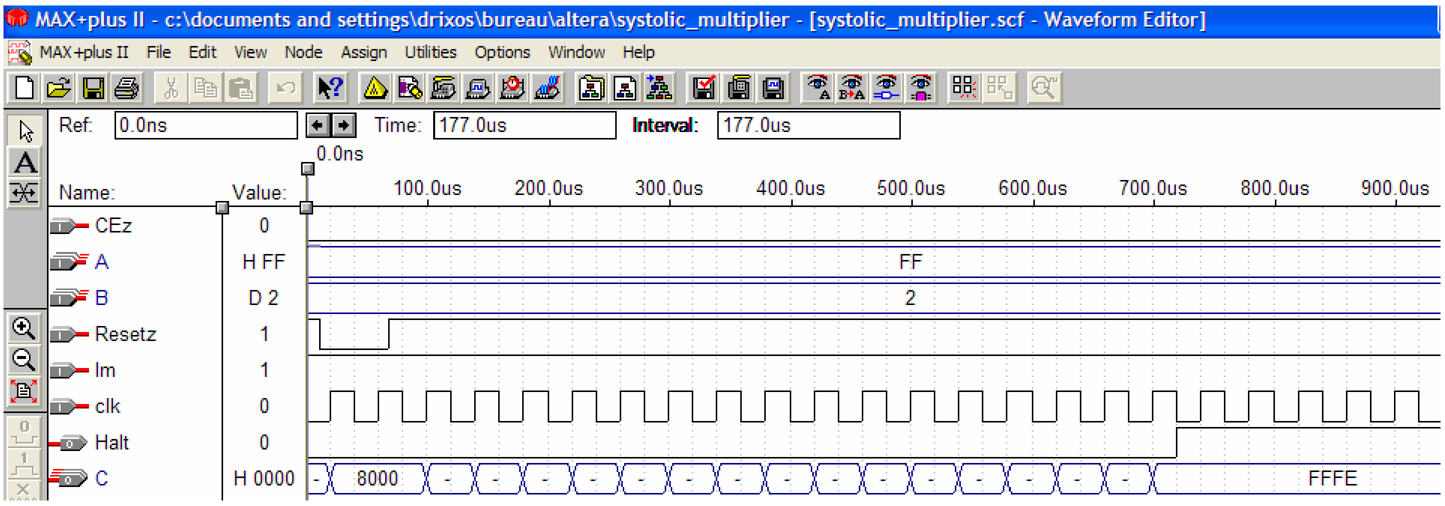
SIMPLE EXAMPLE VALUES FOR THE SIMULATION: (the interesting part being after)


MAX+plus II - c:\documents and settings\drixos\bureau\altera\systolic_multiplier - [systolic_multiplier.scf - Waveform Editor]

Ref: 1.0ms Time: 0.0ns Interval: -1.0ms

```
Name:    Value:  100.0us 200.0us 300.0us 400.0us 500.0us 600.0us 700.0us 800.0us 900.0us
CEz      X
A        D X     64
B        D X     2
Resetz   X
Im       X
clk      X
Halt     X
C        D X     0 32768 - - - - - 512 256 - - - - - - 514 257 128
```

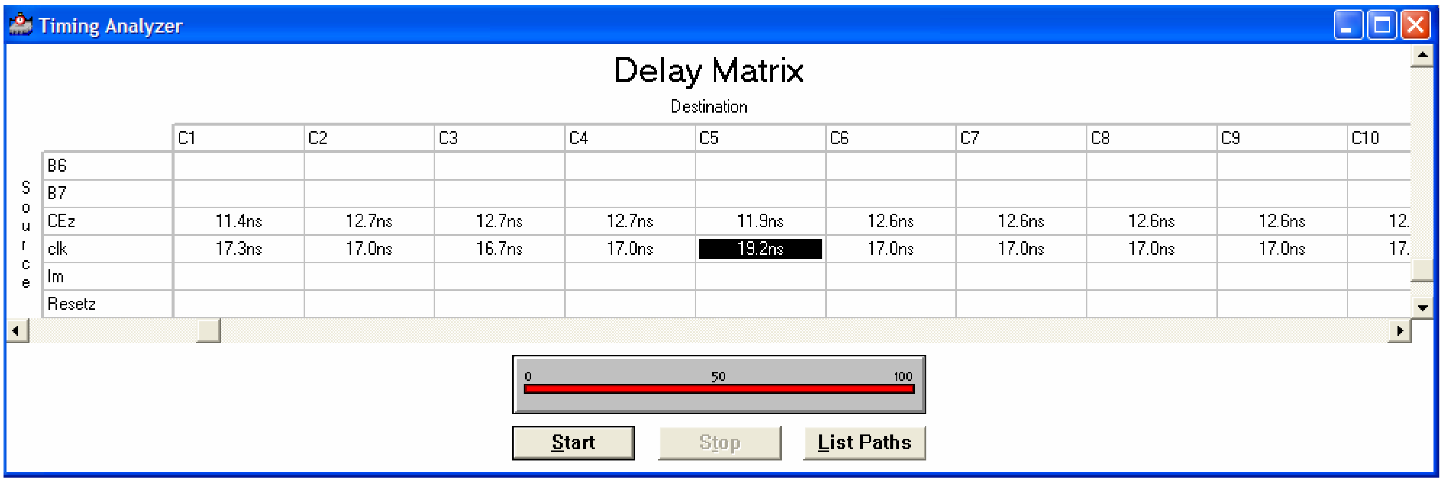
We can se that positive and negative values are working (FF = -1 => 2×FF = -2 and FFFE = -2).


MAX+plus II - c:\documents and settings\drixos\bureau\altera\systolic_multiplier - [systolic_multiplier.scf - Waveform Editor]

Ref: 0.0ns Time: 177.0us Interval: 177.0us

```
Name:    Value:  0.0ns 100.0us 200.0us 300.0us 400.0us 500.0us 600.0us 700.0us 800.0us 900.0us
CEz      0
A        H FF    FF
B        D 2     2
Resetz   1
Im       1
clk      0
Halt     0
C        H 0000  - 8000 - - - - - - - - - - - - - - - FFFE
```

Time analysis: This part is interesting because we can estimate the maximum speed of our component to compare it with the architectural description.



Delay Matrix

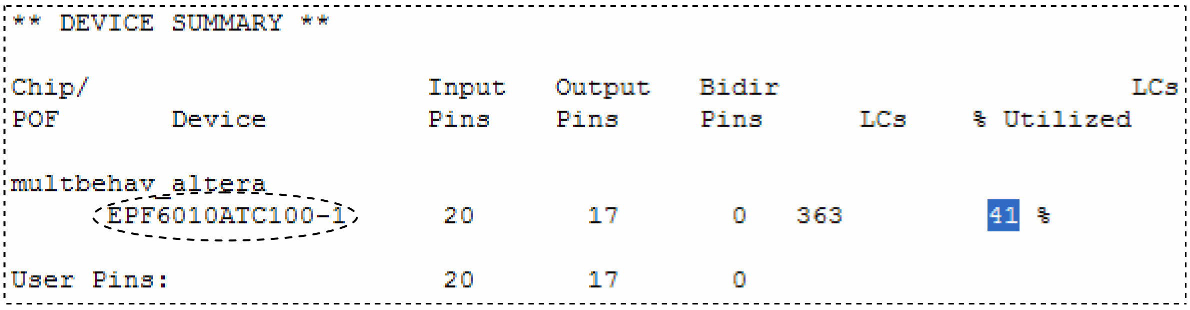
Destination

Source	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
B6										
B7										
CEz	11.4ns	12.7ns	12.7ns	12.7ns	11.9ns	12.6ns	12.6ns	12.6ns	12.6ns	12.
clk	17.3ns	17.0ns	16.7ns	17.0ns	19.2ns	17.0ns	17.0ns	17.0ns	17.0ns	17.
Im										
Resetz										

This analysis gives a maximum time of 19.2ns in hot conditions.
The maximum speed is thus around 1/19.2ns ≈ 52 MHz (but obviously, this value is just an estimation)

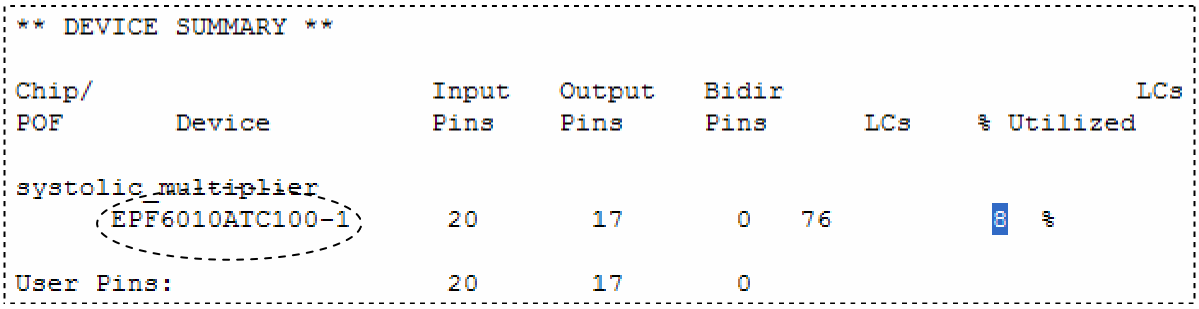
In the report files (*.rpt) we can see, among other things, the chip selected by Max+plus

For the behavioural multiplier, the MAX7000 family didn't fit then I chose a FLEX6000 (`EPF6010ATC100`):



```
** DEVICE SUMMARY **

Chip/                      Input   Output   Bidir                LCs
POF        Device          Pins    Pins     Pins      LCs    % Utilized

multbehav_altera
      EPF6010ATC100-1       20      17       0    363          41 %

User Pins:                  20      17       0
```

for the RTL multiplier (I chose also chose the FLEX600 to compare the occupation ratio):

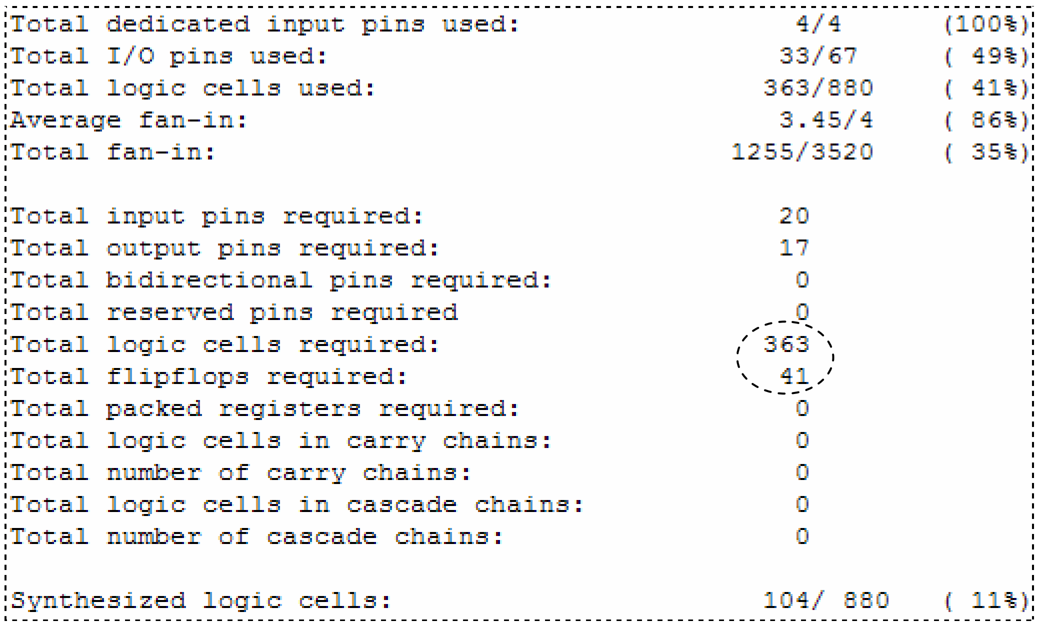


```
** DEVICE SUMMARY **

Chip/                      Input   Output   Bidir                LCs
POF        Device          Pins    Pins     Pins      LCs    % Utilized

systolic_multiplier
      EPF6010ATC100-1       20      17       0    76           8  %

User Pins:                  20      17       0
```

As we can see the occupation ration is greatly better in the RTL design: 41 / 8 = more than 5 times better!

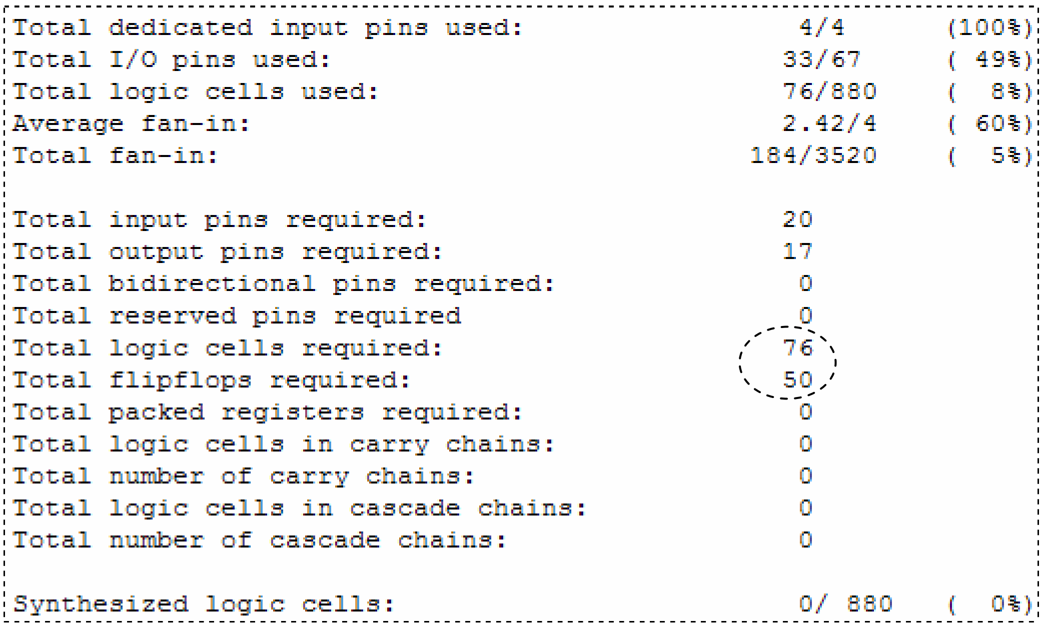
Note: I tried to compile the architectural but it made my computer freeze every time…

For the behavioural multiplier (part2):



```
Total dedicated input pins used:              4/4       (100%)
Total I/O pins used:                         33/67      ( 49%)
Total logic cells used:                     363/880     ( 41%)
Average fan-in:                             3.45/4      ( 86%)
Total fan-in:                            1255/3520      ( 35%)

Total input pins required:                  20
Total output pins required:                 17
Total bidirectional pins required:           0
Total reserved pins required                 0
Total logic cells required:                363
Total flipflops required:                   41
Total packed registers required:             0
Total logic cells in carry chains:           0
Total number of carry chains:                0
Total logic cells in cascade chains:         0
Total number of cascade chains:              0

Synthesized logic cells:                   104/ 880     ( 11%)
```

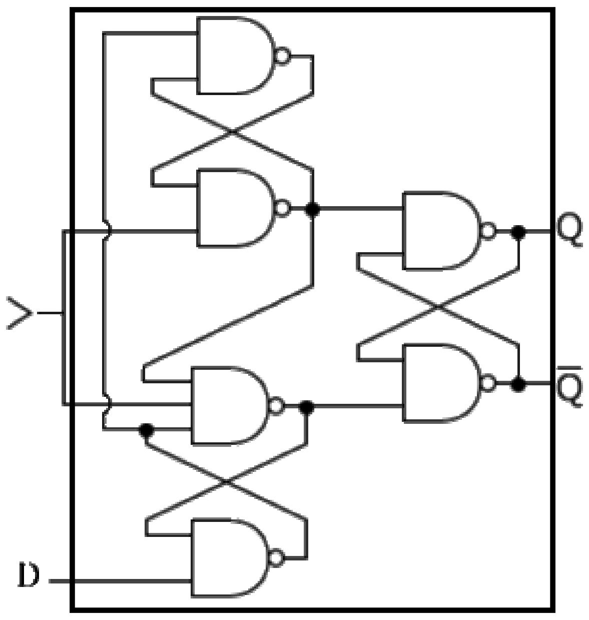
for the RTL multiplier(part2):



```
Total dedicated input pins used:              4/4       (100%)
Total I/O pins used:                         33/67      ( 49%)
Total logic cells used:                      76/880     (  8%)
Average fan-in:                             2.42/4      ( 60%)
Total fan-in:                             184/3520      (  5%)

Total input pins required:                  20
Total output pins required:                 17
Total bidirectional pins required:           0
Total reserved pins required                 0
Total logic cells required:                 76
Total flipflops required:                   50
Total packed registers required:             0
Total logic cells in carry chains:           0
Total number of carry chains:                0
Total logic cells in cascade chains:         0
Total number of cascade chains:              0

Synthesized logic cells:                     0/ 880     (  0%)
```

=> My RTL multiplier takes a few DFF more but if we compare the quantity of logic cells used there is 287 more in the behavioural than in the RTL.

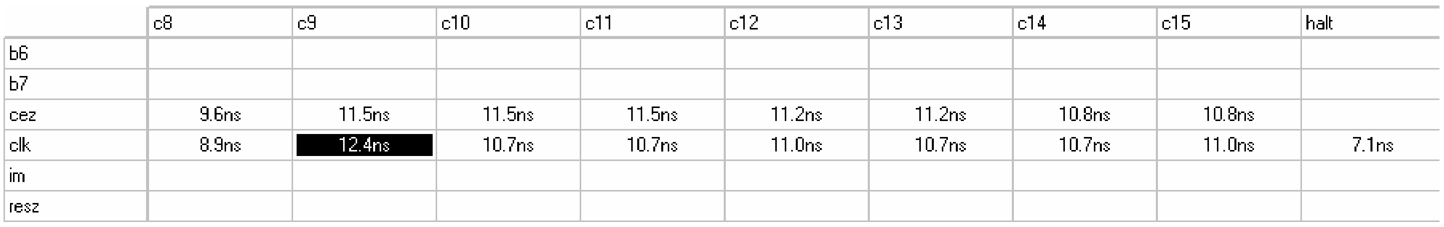
Even if we assume that it's because of a bad choice and we rebuild DFF with 6 NAND gates we have 287 / 6 = more than 47 DFF in excess.



<u>A positive-edge-triggered D flip-flop</u> => => =>

This diagram shows that, as I sais before, a DFF is composed by 6 NAND gates

Let's look at the time analysis of the automatic synthesis of the **behavioural multiplier** :



	c8	c9	c10	c11	c12	c13	c14	c15	halt
b6									
b7									
cez	9.6ns	11.5ns	11.5ns	11.5ns	11.2ns	11.2ns	10.8ns	10.8ns	
clk	8.9ns	12.4ns	10.7ns	10.7ns	11.0ns	10.7ns	10.7ns	11.0ns	7.1ns
im									
resz									

This analysis seems to give a maximum time of 12.4ns. The maximum speed is thus around 1/12.4ns ≈ 80MHz (but obviously, this value is just an estimation)

Compared to the 52 MHz of the RTL description it's not what we would expect but we have to remember that the electronic is a perpetual compromise, the speed is better but it uses a lot more of silicon area...

8 ***<u>Conclusion</u>***

I'm really happy to have taken the time to compare the RTL synthesis and the behavioural synthesis with Max+plus. This is a good finalisation of the comparison of automatic and semi-manual synthesis. This report has been a really good approach to the synthesis fight. This assignment made me discover a lot of tricks with Max+plus simulator, Silos simulator and also Verilog HDL (definitively confusing considering that I've seen VHSIC HDL previous year).

However, I have discovered a good overview of these complex uses, but I’m obviously still very far of the full potentials. I also have discovered what is to investigate for real: as much for the numbering systems as for the multiplications algorithms and multipliers, I took a lot of time but I also learned a lot. And obviously I studied different levels of abstraction, where I've particularly struggled at the end.

My programs are definitively not the only means to reach the aim and obviously, improvements exist but anyway, I'm really proud to have discovered new design techniques and a new HDL, I know it also exists A$_{ltera}$HDL, SystemC, and more than ten or so but I still have the time…

4 ***<u>References:</u>***

<u>BOOKS</u>: *Fundamentals of DIGITAL LOGIC with Verilog design (Brown Vanesic - Mc Graw Hill)*
DIGITAL FUNDAMENTALS 8th ed. (Thomas FLOYD - Pearson Education International)
The Verilog Hardware Description Language 5th ed. (Thomas & Moorby's - Kluwer Academic)
Verilog HDL: A Guide to Digital Design and Synthesis. (Samir Palnitkar - Prentice Hall)

<u>WEBSITES</u>: http://en.wikipedia.org
http://www.cours.polymtl.ca/ele2300/acetates.htm
http://ieeexplore.ieee.org
http://tams-www.informatik.uni-hamburg.de
http://www.dec.usc.es
http://www-history.mcs.st-andrews.ac.uk
http://lapwww.epfl.ch

Index:

Here is the Verilog code of the **behavioural multiplier** description used to compare to the RTL synthesis:

```
module multbehav_altera(halt,c, //outputs
          clk,a,b,resz,cez,im); // inputs

    output halt;
    output [15:0] c;
    reg [15:0] ar, br, cr, c;
    reg halt;
    input [7:0] a,b;
    input clk,resz,cez,im;

always@(cez or cr) begin
    if(!cez) c = cr;
    else c = 16'hzzzz;
end

always@(posedge clk) begin
    if(!resz) begin
        br=0;
        br[7:0] = b;
        if (b[7])
            br[15:8] = 8'hff;
            if(im) begin
                ar=0;
                ar[7:0] = a;
                if (a[7]) ar[15:8] = 8'hff;
            end
        cr=0;
        halt=0;
    end
    else begin
        cr = ar*br;
        halt = 1;
    end
end
endmodule
```