7 Arithmetics

This chapter introduces basic concepts concerning the design of arithmetic gates. Firstly, we illustrate data formats. Secondly, the adder circuit is presented, with its corresponding layout created manually and automatically. Then the comparator, multiplier and the arithmetic and logic unit are also discussed. This chapter also includes details on a student project concerning the design of binary-to-decimal addition and display.

1. Data formats

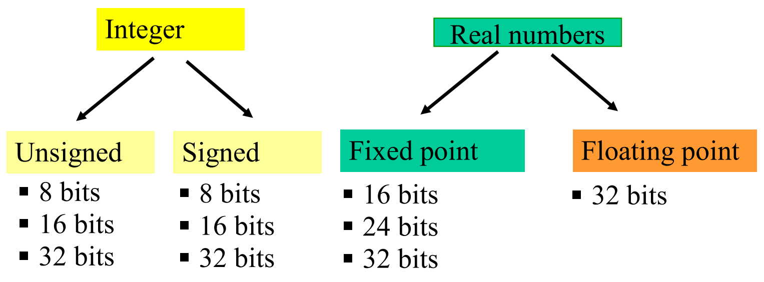


Figure 7-1: Most common data formats used in ASIC designs

The two classes of data formats are the integer and real numbers (Figure 7-1). The integer type is separated into two formats: unsigned format and signed format. The real numbers are also sub-divided into fixed point and floating point descriptions. Each data is coded in 8,16 or 32 bits. In particular cases, other formats are used, such as the exotic 24 bit in some optimized applications, such as in application-specific digital signal processors.

Integer Format

Type	**Size (bit)**	**Usual name**	**Range**
Unsigned integer	8	Byte	0..255
	16	Word	0..65535
	32	Long word	0..4294967295
Signed integer	8	Short integer	-128..+127
	16	Integer	-32768..+ 32767
	32	Long integer	-2147483648..+ 2147483647

Table 7-1: Size and range of usual integer formats

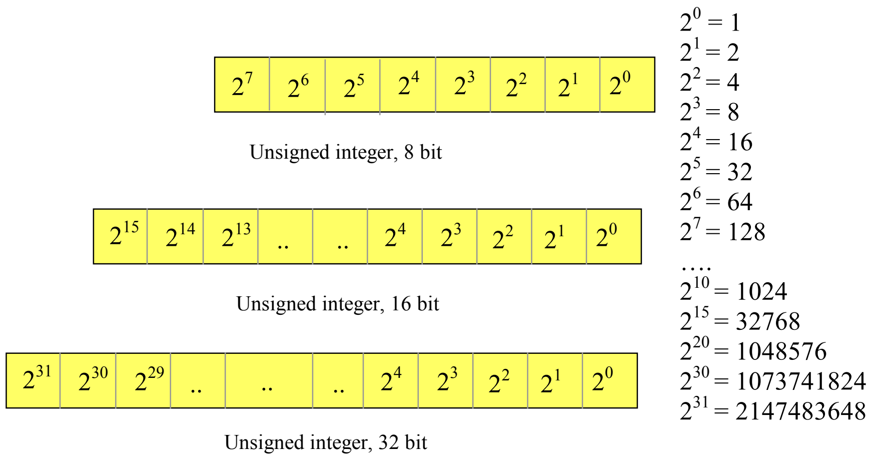


Figure 7-2: Unsigned integer format

A summary of integer formats is reported in table 7-1. The signification of each bit is given in figure 7-2. The unsigned integer format is simply the series of power of 2. As an example, the number 01101011 corresponds to 107, as detailed in equation 7-1.

$$01101011 = 2^6 + 2^5 + 2^3 + 2^1 + 2^0 = 64 + 32 + 8 + 2 + 1 = 107 \qquad \text{(Equ. 7-1)}$$

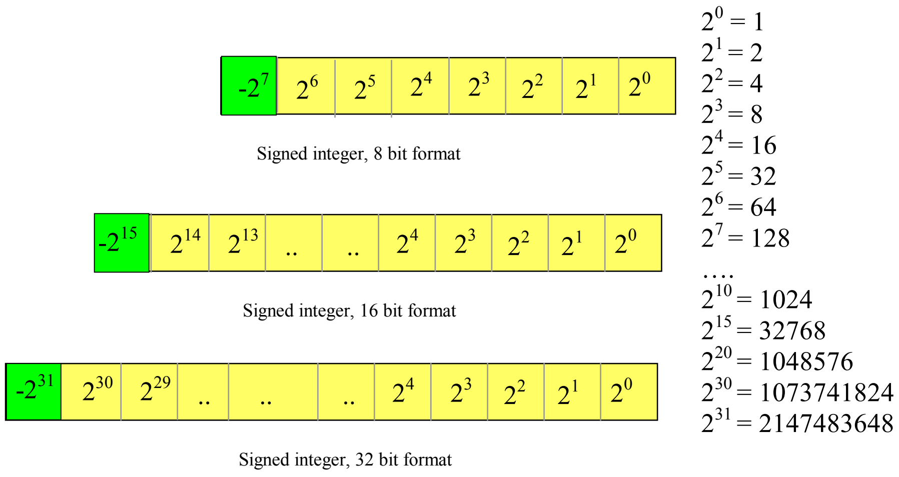


Figure 7-2: Signed integer format

The signed integer format uses the left-most bit for the sign. The coding of the data works as for the unsigned integer, except that the left-most bit accounts for a negative number. In 16 bit format, a 1 in the sign bit equals to -32768. As As an example, the 8-bit signed number 11101011 is detailed in equation 7-2. The sign bit appears in the sum as -128, all the other bit remaining positive.

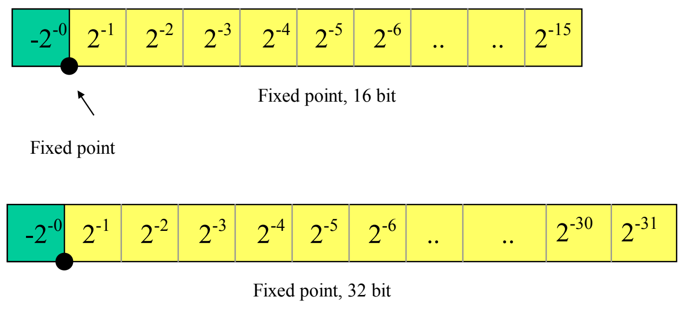
$$11101011 = -2^7 + 2^6 + 2^5 + 2^3 + 2^1 + 2^0 = -128 + 64 + 32 + 8 + 2 + 1 = -21 \qquad \text{(Equ. 7-2)}$$

Real Format

A second important class of numbers is the real format. In digital signal processing, real numbers are often implemented as fixed point numbers. The key idea is to restrict the real numbers within the range [-1.0..+1.0] and to use arithmetic hardware that is compatible with integer hardware. More general real numbers are coded in a 32 bit format. A summary of real formats is reported in table 7-2.

Type	Size (bit)	Usual name	Range
Fixed point	16	Fixed	-1.0..+1.0 (Minimum 0.00003)
	32	Double fixed	-1.0..+1.0 (Minimum 0.00000000046)
Floating point	32	Real	$-3.4^{e}38..3.4^{e}38$ (Minimum 5.8^{e}-39)

Table 7-2: Size and range of usual real formats



$2^{-0} = 1.0$
$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$
$2^{-4} = 0.0625$
$2^{-5} = 0.03125$
$2^{-6} = 0.015625$
....
$2^{-15} = 0.000030517578125$
$2^{-31} = 0.0000000004656$

Figure 7-3: Fixed point numbers in 16 and 32 bit format

In the case of fixed point arithmetic, we read bits as fractions in negative power of 2 (Example of equation 7-3). When the left-most bit is set to 1, it accounts for -1.0 in the addition (Equation 7-4). The main limitation of this format is its limited range from -1.0 to 1.0. Its main advantage is a hardware compatibility with integer circuits, leading to low power computing, a particularly attractive feature for embedded electronics. As an example, most digital signal processing of mobile phones work in fixed point arithmetic.

$$01100100 = 2^{-1} + 2^{-2} + 2^{-5} = 0.5 + 0.25 + 0.03125 = 0.78125 \quad \text{(Equ. 7-3)}$$
$$11100100 = -2^{0} + 2^{-1} + 2^{-2} + 2^{-5} = -1.0 + 0.5 + 0.25 + 0.03125 = \text{-}0.21875 \quad \text{(Equ. 7-4)}$$

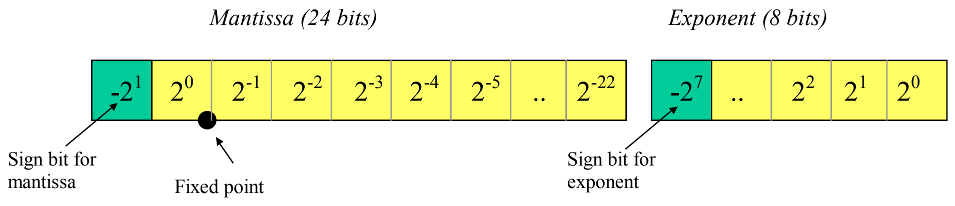


Figure 7-4: Floating point arithmetic format

Finally, floating point data is coded using a mantissa multiplied by an exponent (Figure 7-4). The general formulation of the real number format is as follows:

$$data = mantissa.2^{exponent} \qquad \text{(Equ. 7-5)}$$

The illustration of this format is given in equation 7-6. Numbers may range from -3.4x10^{38} to 3.4x10^{38}.

$$\begin{aligned}
&(0110100...)(0..101) = (2^0 + 2^{-1} + 2^{-3})\text{x}(2^2 + 2^0) \\
&= (1.0 + 0.5 + 0.125)\text{x}(4+1) \\
&= 1.625\text{x}2^5 \\
&= 52.0
\end{aligned} \qquad \text{(Equ. 7-6)}$$

2. The adder circuit

Let's consider two 8-bit unsigned integers A and B. These numbers range from 0 to 255. The arithmetic addition of the two numbers is given in equation 7-7.

$$S = A + B \qquad \text{(Equ. 7-7)}$$

To obtain the arithmetic addition of these numbers, we build elementary circuits which realize the bit-level arithmetic addition, as described below. Remember that if *a0*=1 and *b0*=1, a carry bit is generated in the next column. We need to take into account the carry bit *ci* for each column, by creating a circuit with three logic inputs *ai,bi* and *ci-1* to produce *si* and *ci*. This circuit is called full adder. The adder in column 0 is a more simple circuit, which realizes the addition of two logic data *a0* and *b0*, which is called the half adder.

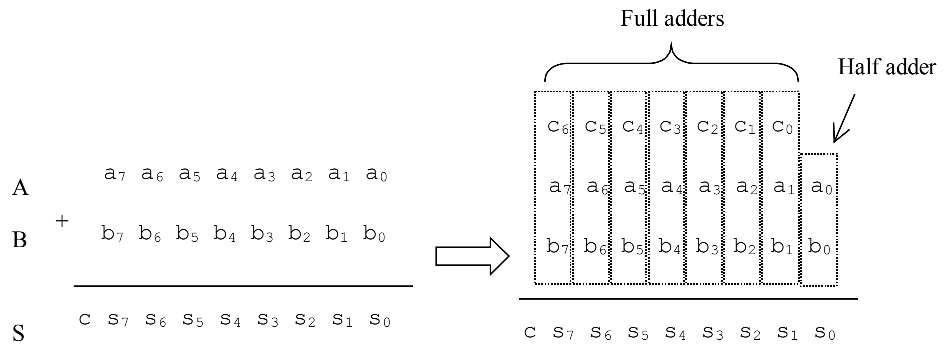


Figure 7-5: 8-bit unsigned integer addition principles

From Logic Design to Layout

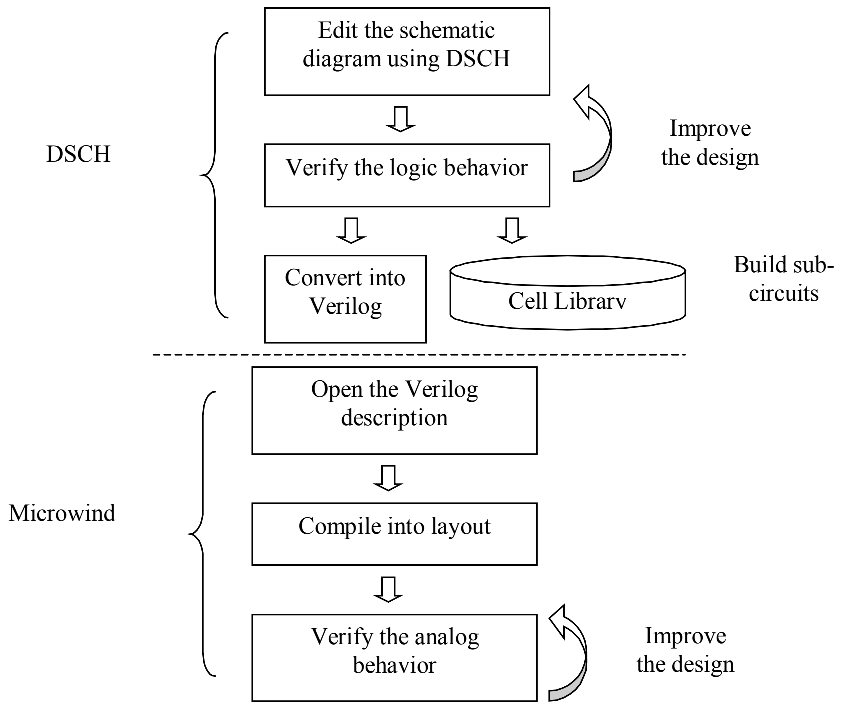


Figure 7-6: Design flow from logic design to layout implementation

When the design starts to be complex, the manual layout is very difficult to conduct, and an automatic approach is preferred, at the price of a less compact design and more rigid implementation methodology. The steps from logic design to layout validation are reported in figure 7-6. The schematic diagram constructed using DSCH is validated first at logic level. At that level, timing analysis is available as well as power consumption estimation. The accuracy of these predictions is quite fair as accurate layout information is still missing. The designer iterates on his circuit until the specified performances are achieved. At that point, it is of key importance to build a sub-circuit and feed a cell library. These sub-circuits may be reused in other designs, if the user provide sufficient information on the performances of the cells.

A specific command in DSCH creates the Verilog description of the logic design, including the list of primitives and some stimulation information. This Verilog text file is understood by Microwind to construct the corresponding layout, with respect to the desired design rules. This means that the layout result will significantly change whether we use 0.8µm or 0.12µm design rules. The supply properties and most stimulation information are added to the layout automatically, according to the logic simulation. Finally, the analog simulation permits to validate the initial design and verify its switching and power consumption performances.

3. Adder Cell Design

In this section, we use the design method presented previously to build an 8-bit adder.

Half Adder

The Half-Adder gate truth-table and schematic diagram are shown in Figure 7-7. The SUM function is made with an XOR gate, the Carry function is a simple AND gate. When both A and B are at 1 (In black in figure 7-7), the *carry* output is asserted, and the *sum* is at zero. The combination of carry and sum is equal to (10), which is equivalent to 2 in binary format.

A	B	Carry	Sum	Result
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	2

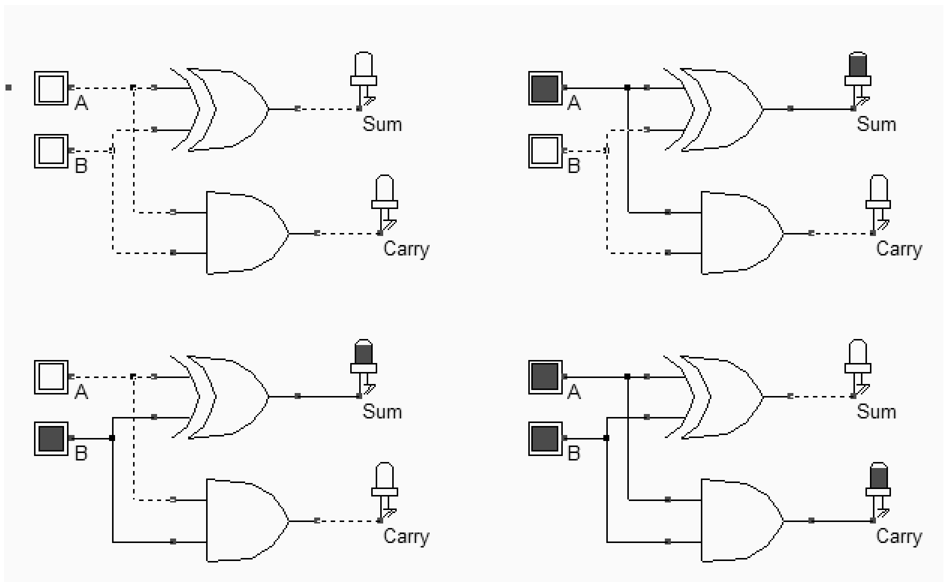
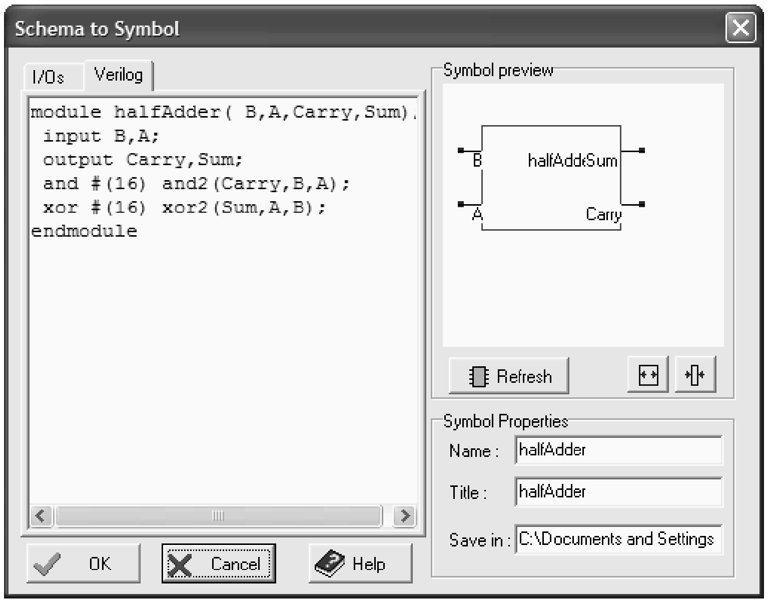


Fig. 7-7. Truth table and schematic diagram of the half-adder gate (halfAdderTest.SCH).

The correct behavior of the adder is proven by the values of the display, which show a result in accordance with the truth table. The command **File → Schema to New Symbol** in DSCH enables to create a sub-circuit. This technique is useful to construct hierarchical designs, by embedding a complete circuit into a single component. The command menu is reported in figure 7-8. By a click on the OK button, a symbol **HaldAdder.SYM** is created. This symbol includes the VERILOG description of the circuit, that is the XOR and AND gate, as seen in the left part of the window. Once created, the symbol may be instantiated in any logic design.



```
module halfAdder( B,A,Carry,Sum);
 input B,A;
 output Carry,Sum;
 and #(16) and2(Carry,B,A);
 xor #(16) xor2(Sum,A,B);
endmodule
```

Name : halfAdder

Title : halfAdder

Save in : C:\Documents and Settings

Fig. 7-8. Creating a half-adder symbol (HalfAdder.SYM)

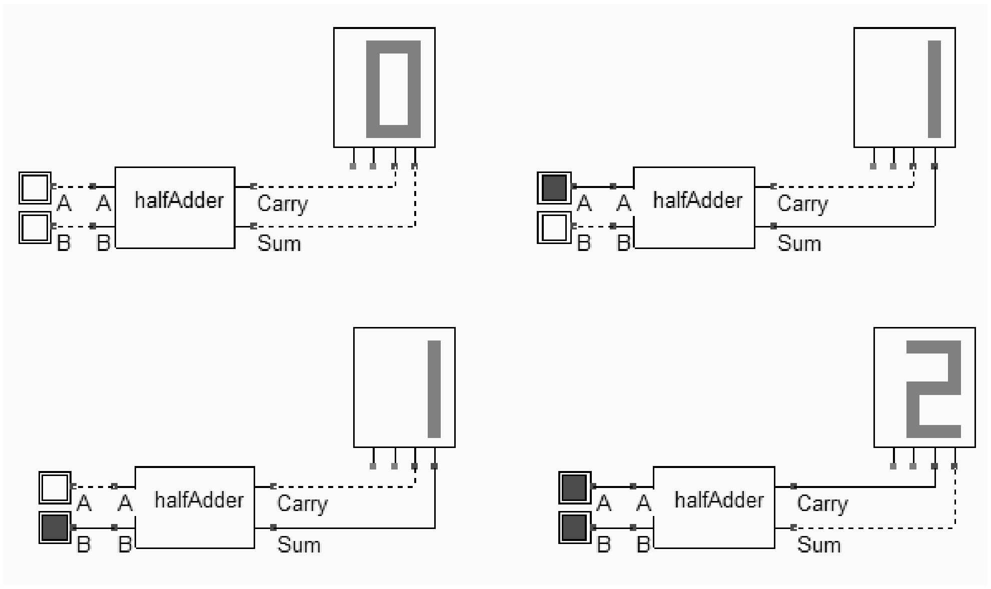
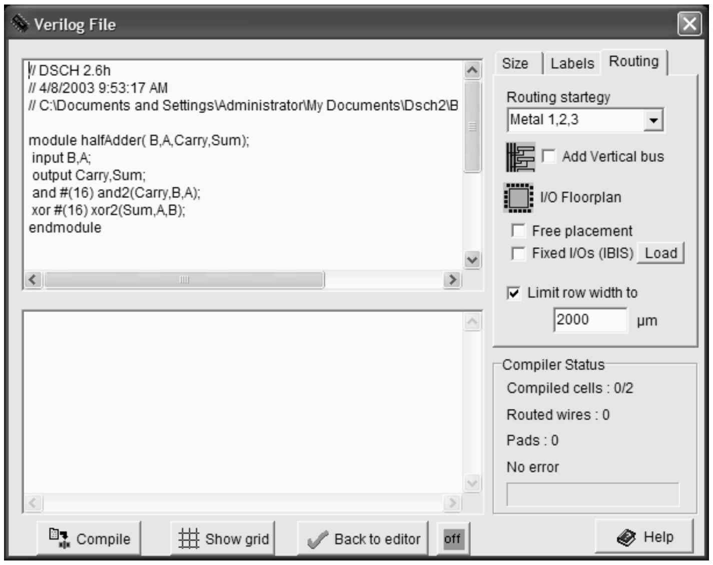


Fig. 7-9. Logic validation of the half-adder symbol (halfAdderTest.SCH).

The logic validation of the half adder is provided in the simulation of figure 7-9. When the symbol was created, a Verilog text was also generated, namely **HalfAdder.TXT**, In Microwind2, the same Verilog text file is used to construct the corresponding layout automatically. Just invoke the command **Compile → Compile Verilog File,** select the corresponding text file and click **Compile**.


```
// DSCH 2.6h
// 4/8/2003 9:53:17 AM
// C:\Documents and Settings\Administrator\My Documents\Dsch2\B

module halfAdder( B,A,Carry,Sum);
 input B,A;
 output Carry,Sum;
 and #(16) and2(Carry,B,A);
 xor #(16) xor2(Sum,A,B);
endmodule
```

Routing startegy: Metal 1,2,3
Add Vertical bus: ☐
I/O Floorplan
Free placement: ☐
Fixed I/Os (IBIS): ☐
Limit row width to: ✓ 2000 µm

Compiler Status
Compiled cells : 0/2
Routed wires : 0
Pads : 0
No error

Fig. 7-10 Using the half adder VERILOG description to pilot the layout compiling in MICROWIND (halfAdder.TXT)

When the compiling is complete, the resulting layout appears shown below. The XOR gate is routed on the left and the AND gate is routed on the right. Click on **Simulate → Start Simulation**. The timing diagrams of figure 7-11 appear where the truth table of the half-adder can be verified.

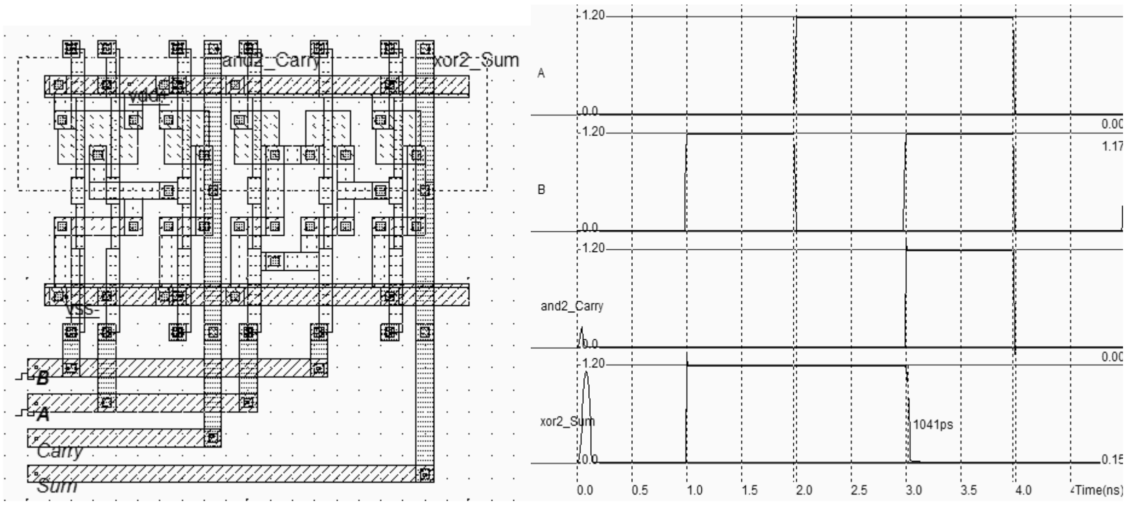


Fig. 7-11. Compiling and simulation of the half-adder gate (Hadd.MSK)

Full-Adder Gate

The truth table and schematic diagram for the full-adder are shown in Figure 7-12. The most straightforward implementation of the CARRY cell is to use a combination of AND, OR gates. Using negative logic equivalence, the AND,OR combination becomes a series of NAND gates, as suggested by equation 7-8.Concerning the sum, one common approach consists in using a three-input XOR gate. This 3-input XOR is usually constructed from two stages of 2-input XOR.

A	B	C	Carry	Sum	Result
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	2
1	0	0	0	1	1
1	0	1	1	0	2
1	1	0	1	0	2
1	1	1	1	1	3

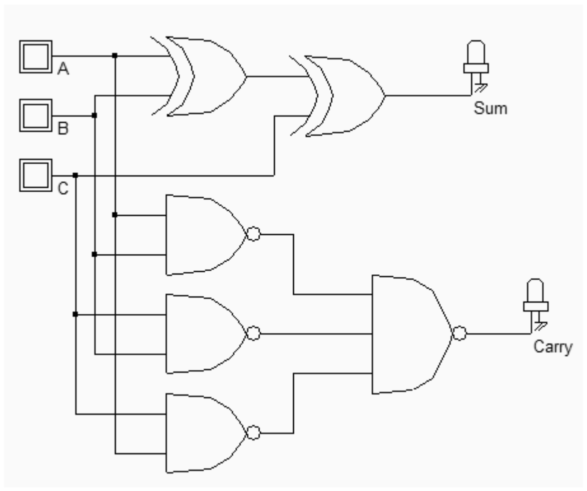


Fig. 7-12. The truth table and schematic diagram of a full-adder(FADD.SCH)

$$sum = A \wedge B \wedge C$$

$$carry = (A \& B) \mid (A \& C) \mid (B \& C) \qquad \text{(Equ 7-8)}$$

$$carry = \overline{\overline{(A \& B)} \& \overline{(A \& C)} \& \overline{(B \& C)}}$$

Full-Adder using Complex gate

A more efficient circuit consists in the use of complex gates. We apply this technique for the *carry* cell: rather than assembling NAND gates (Total 3x4+6=18 transistors), we assemble MOS devices in AND/OR combinations. The carry function built using MOS in parallel or in series is presented in figure 7-13. A tiny re-arrangement of the Boolean expression leads to a 12 transistor complex gate (Equation 7-9) rather than a 14 transistor one, if we include the 2 MOS devices of the final stage inverter. The carry circuit shown in figure 7-13 is strictly equivalent to the carry circuit of figure 7-12. We avoid the needs for cascaded NAND gates, and so the number of transistors is lower.

$$carry = (A \& B) \mid (C \& (A \mid B)) \qquad \text{(Equ 7-9)}$$

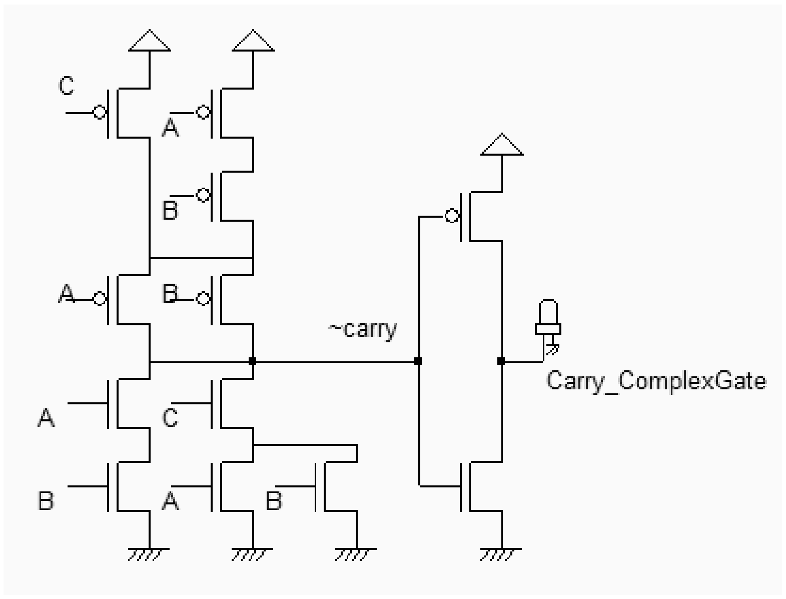


Fig. 7-13. The carry cell based on a complex gate requires less transistors (Fadd.SCH)

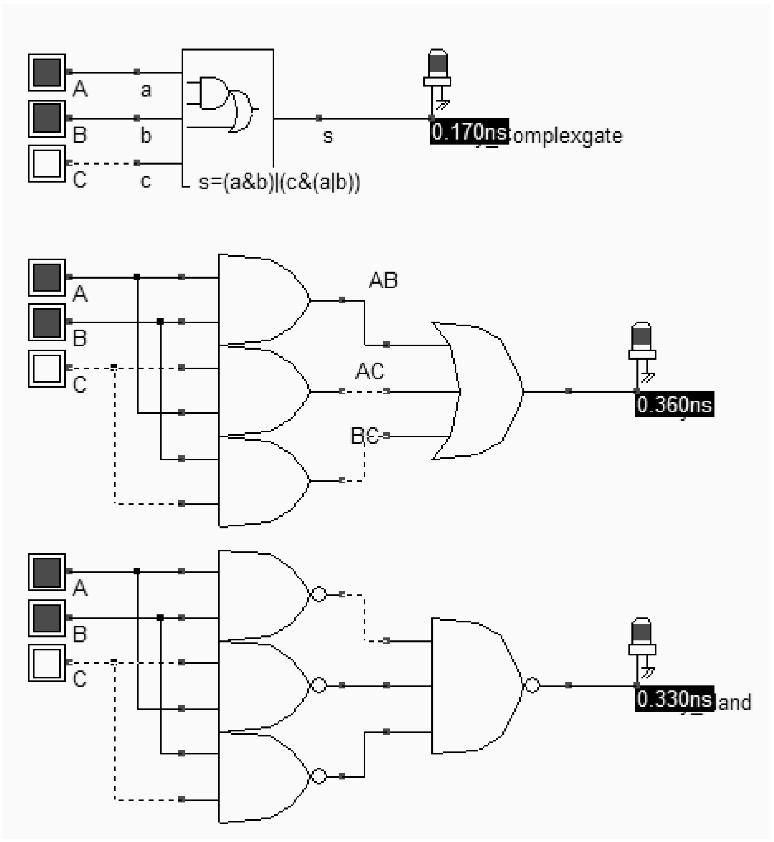


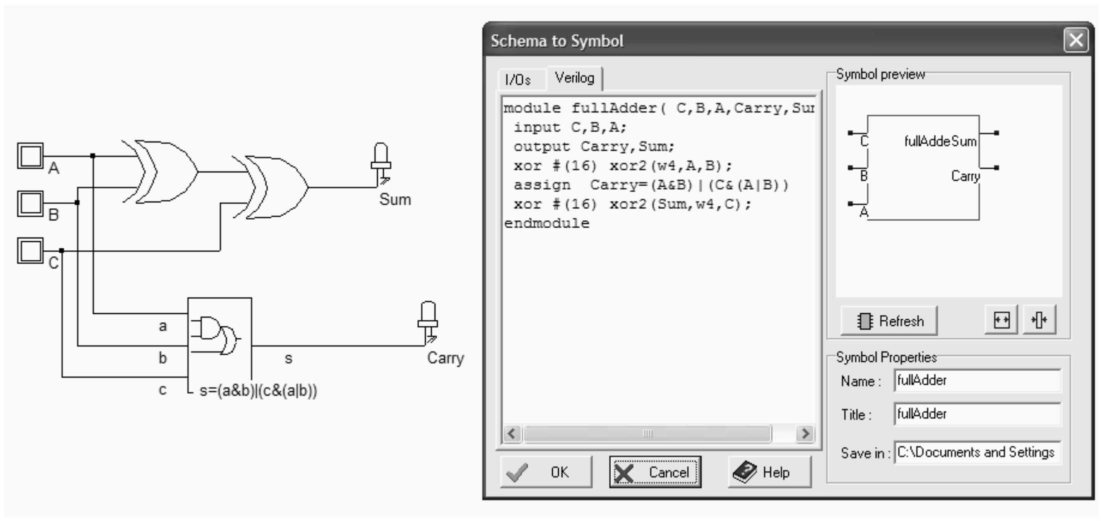
Fig. 7-14. Comparing the switching performances of the carry cells (Carry.SCH)

A switching speed comparison is proposed in figure 7-14: the upper cell is the fastest, as the number of stage is restricted to the complex gate itself, with short internal connections and small diffusion areas. The cell in the middle is the slowest due to the four internal stages (The AND is an AND plus an inverter). The lower circuit is a little faster, but not so. The reason is that DSCH automatically assigns a typical delay (Around 70ps) to each interconnect. At this stage, DSCH do not make the difference between a short and a long interconnect. Very probably, the interconnect between the NAND2 output and the NAND3 input will be very short and the delay is severely overestimated. But it could happen that the routing is quite large, in that case the supplementary delay is justified.

Full-Adder using Complex gate

The procedure to generate the symbol of the full-adder from its schematic diagram is the same as for the half adder. When invoking **File → Schema to new symbol**, the screen of figure 7-15 appears. Simply click **OK**. The symbol of the full-adder is created, with the name *FullAdder.sym* in the current directory. Meanwhile, the Verilog file **fullAdder.txt** is generated, which contents is reported in the left part of the window (Item **Verilog**).

We see that the XOR gates are declared as primitives while the complex gate is declared using the **Assign** command, as a combination of AND (&)and OR (|) operators. If we used AND and OR primitives instead, the layout compiler would implement the function in a series of AND and OR CMOS gates, loosing the benefits of complex gate approach in terms of cell density and switching speed.

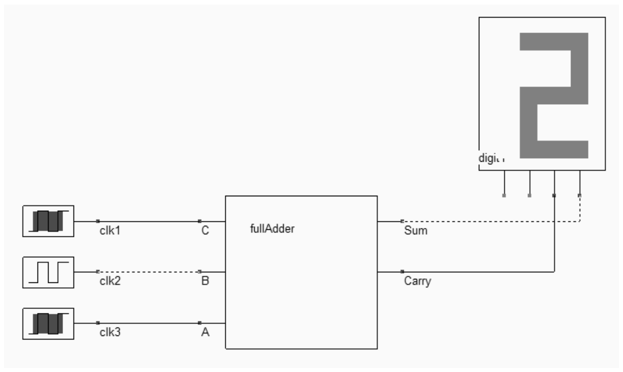


```
module fullAdder( C,B,A,Carry,Sur
 input C,B,A;
 output Carry,Sum;
 xor #(16) xor2(w4,A,B);
 assign  Carry=(A&B)|(C&(A|B))
 xor #(16) xor2(Sum,w4,C);
endmodule
```

Name : fullAdder
Title : fullAdder
Save in : C:\Documents and Settings

Fig. 7-15 Verilog description of the full adder (FullAdder.SYM)

Use the command **Insert → User Symbol** to include this symbol into a new circuit. For example, the circuit **FaddTest** includes the hierarchical symbol and verifies its behavior. Three clocks with 20,40 and 80ns are declared as inputs. Using such clocks is of particular importance to scan all possible combination of inputs, by following line by line the truth table. What we observe in the chronograms of figure 7-16 is the addition of three numbers, which follows the requested result given in the initial truth table. The addition of A,B and C appears in the output *sum*. For example, at time t=70ns, A=B=C=1, the output is 3 after a little delay.



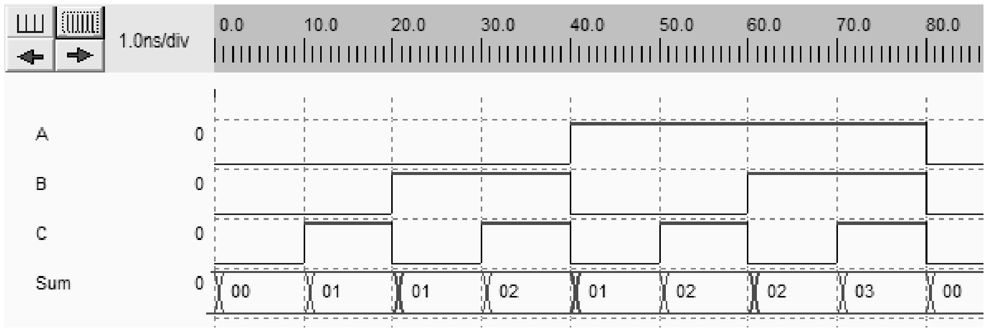


Fig. 7-16. Testing the new Adder symbol using clocks (FaddTest.SCH)

The Full-Adder Adder

You may create the layout of the full-adder by using the Verilog cell compiler of Microwind. It is recommended to compile the full-adder symbol which includes the complex gate, for an optimum result. Microwind handles complex gate descriptions and has the ability to convert AND/OR logic combinations into a compact layout. The full-adder compiled layout is shown in Figure 7-17. By default, all signals are routed to the left side of the logic cells, for clarity. In real-case designs, the routing creates connections in all directions, depending on the position of signals inputs and outputs.

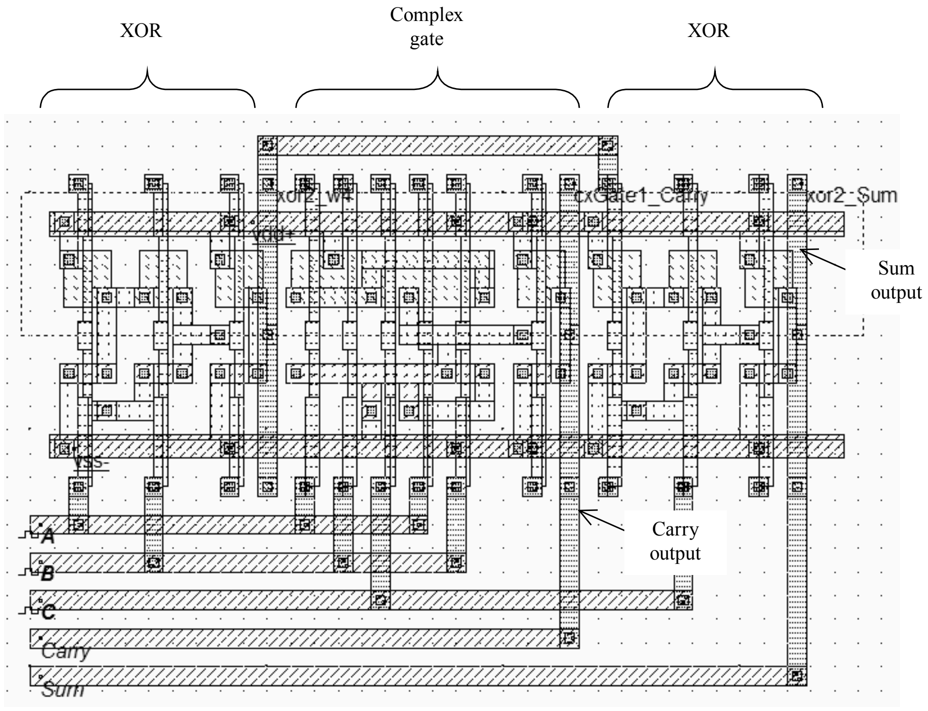


Fig. 7-17 The compiled full-adder (FullAdder.MSK).

The complex gate implementation includes some interesting design techniques to achieve a compact layout. The layout and schematic diagram are detailed in figure 7-18. The cell compiler has organized the contacts to ground and metal bridge in such a way that the n-diffusion and p-diffusion areas are continuous. The MOS arrangement is electrical equivalent to the schematic diagram of figure 7-13. Notice that the diffusion has been stretched to build an internal connection, in the n-MOS area. Diffusion is rarely used as an interconnect due to its very huge parasitic resistance. In that case, however, the role of this diffusion connection is not significant.

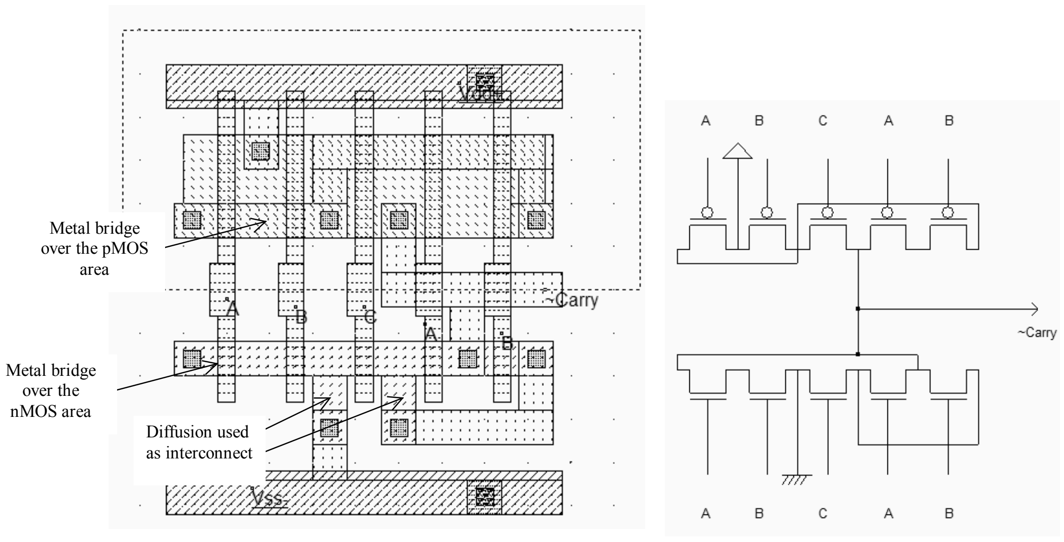

Vdd+
Metal bridge over the pMOS area
~Carry
A
B
C
A
B
Metal bridge over the nMOS area
Diffusion used as interconnect
Vss-
A B C A B
~Carry
A B C A B

Fig. 7-17 The carry cell layout and associated structure (CarryCell.MSK).

There is no need to add the clock properties in the layout as the clock signals properties have been extracted from the Verilog text, and automatically placed in the compiled layout. The clock description in Verilog format appears after the keyword "Always". Three clocks are declared: A,B and C. The text signifies that for a given time step, the logic signal is inverted. For example clock *C* switches from 0 to 1 at time 1000, from 1 to 0 at time 2000, etc.. There exist a time scale conversion between DSCH and Microwind. A logic time step of 1000 is transformed into 1.0ns.

```
module fullAdder( C,B,A,Carry,Sum);
 input C,B,A;
 output Carry,Sum;
 xor #(16) xor2(w4,A,B);
 assign   Carry=(A&B)|(C&(A|B))
 xor #(16) xor2(Sum,w4,C);
endmodule

// Simulation parameters in Verilog Format
always
#1000 C=~C;
#2000 B=~B;
#4000 A=~A;
```

Fig. 7-18: the declaration of clocks in the Verilog text (FullAdder.TXT)

The simulation of the full-Adder, shown in figure 7-18, complies with the initial truth-table and the logic simulation. Notice the glitch at time t=6.0ns due to internal transient switching of the logic circuit.

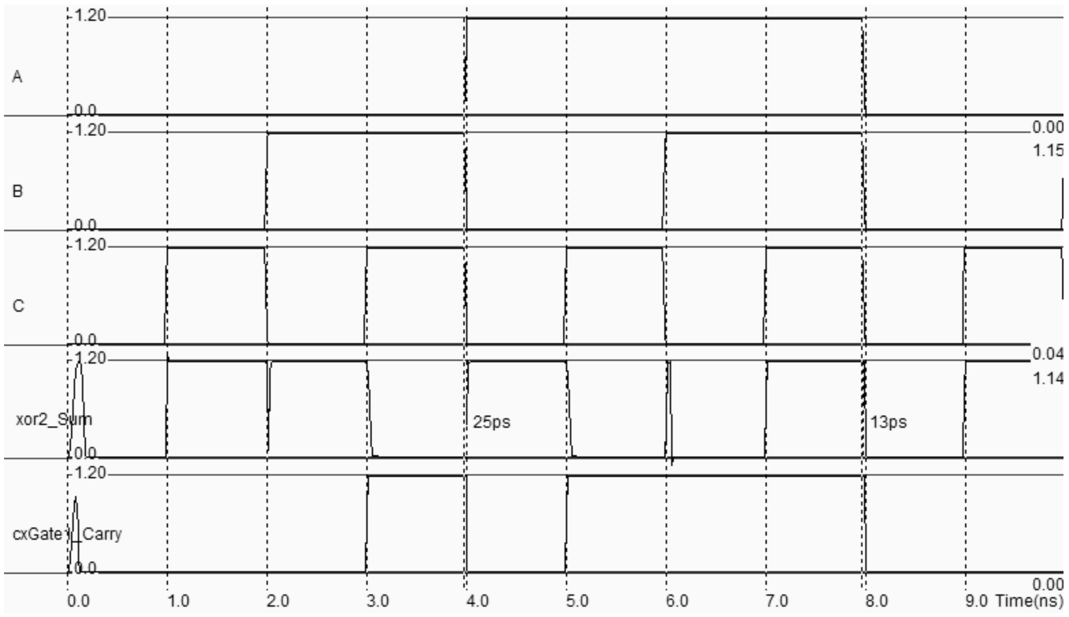


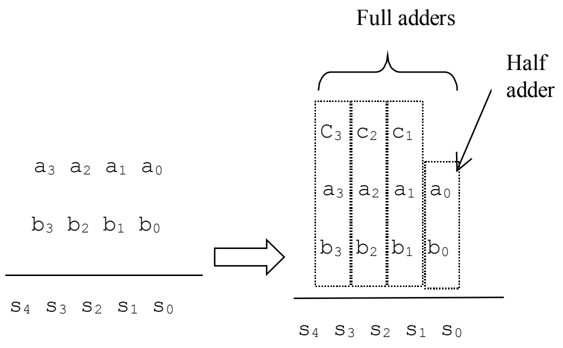
Fig. 7-19: Simulation of a full-adder (File fullAdder.MSK).

4. Ripple Carry Adder

In this section, the design of the adder is addressed in its simplest approach, where the carry of one stage propagates as an input for the next stage. This type of circuit based on a carry chain is called "ripple carry" adder.

Structure of the Ripple Carry Adder

The 4-bit ripple-carry adder circuit includes one half adder and 3 full-adders in serial, according to the elementary addition shown in figure 7-20 [Belaouar]. Each stage produces the Boolean addition of three logical information. For example, the Boolean output s[1] is the addition of input signals a[1] and b[1], together with the internal carry c[1]. Some examples of 4-bit addition are given in table 7-4. We give the output *s* in hexadecimal and decimal format.



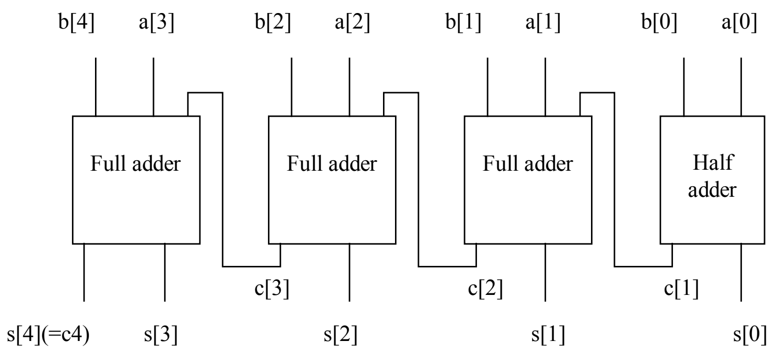


Fig. 7-20. Structure of the 4-bit ripple-carry adder

a[0..3]	b[0..3]	s[0..4] (hexa)	s[0..4] (Decimal)
0	0	00	0
0	F	0F	15
9	7	10	16
6	C	12	18
F	F	1E	30

Table 7-3. Adder result examples

The logic circuit shown in figure 7-4 allows a four-bit addition between two numbers a[0..3] and b[0..3]. The user symbols 'Hadd.sym' and 'Fadd.sym' are added to the design using the command **Insert → User Symbol**. In DSCH2, the numbers *a* and *b* are generated by keyboard symbols, which produces at the press of a desired key a 4-bit logic value. In the example shown in figure 7-21, the value of *a* is 1 (0001 in binary), and the value of *b* is F (1111 in binary form, or 15 in decimal). The result *s*, which combines s[0], s[1], s[2], s[3] and the last carry bit, is equal to 0x10 in hexadecimal, or 16 in decimal.

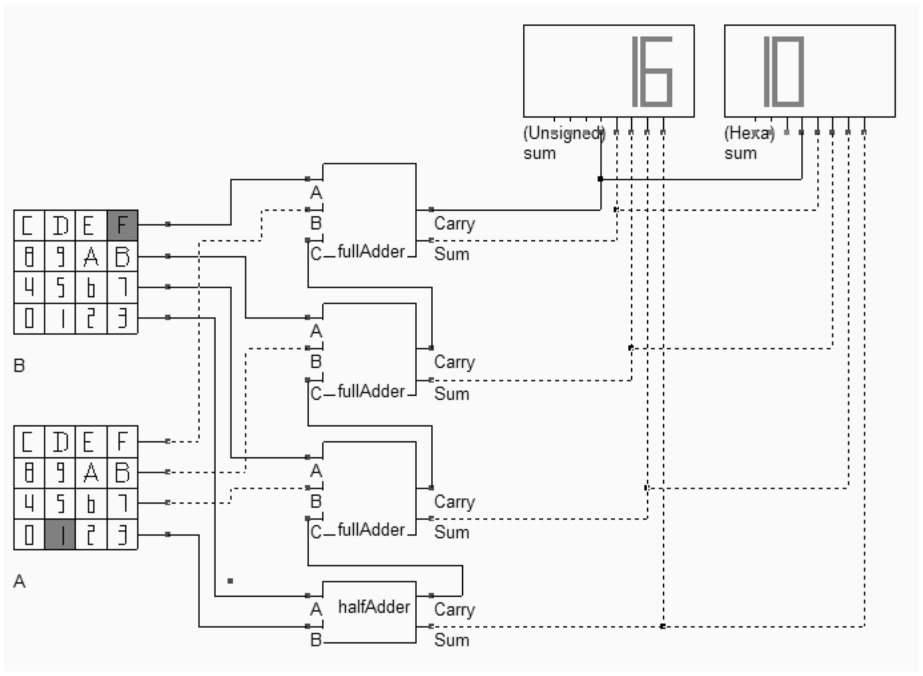


Fig. 7-21. Schematic diagram of the four-bit adder and some examples of results (Add4.SCH).

The two displays are connected to the identical data, but are configured in different mode: hexadecimal format for the right-most display, and integer mode for the left-most display. To change the display mode, double click inside the symbol, and change the format in the symbol property window, as shown in figure 7-22. The default option is the hexadecimal format. The unsigned integer format is used in the schematic diagram of figure 7-22 for the left display. Integer and fixed point display formats are used for arithmetic circuits. The ASCII character corresponding to the 8-bit input data may also be displayed.

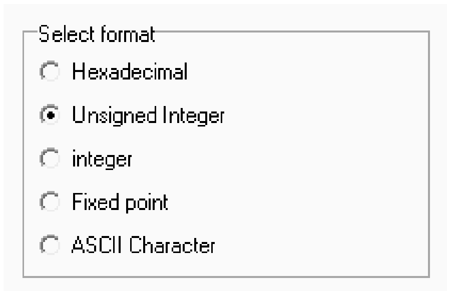

Select format
- Hexadecimal
- [✓ Unsigned Integer]
- integer
- Fixed point
- ASCII Character

Fig. 7-22. The display symbol has five format options (Add4.SCH).

Critical Path

The worst case delay of the circuit is calculated in Dsch by the command **Simulate→ Find critical Path→In the Diagram**. In the ripple carry circuit, the critical path passes through the carry chain. The predicted worst case delay is 0.8ns.

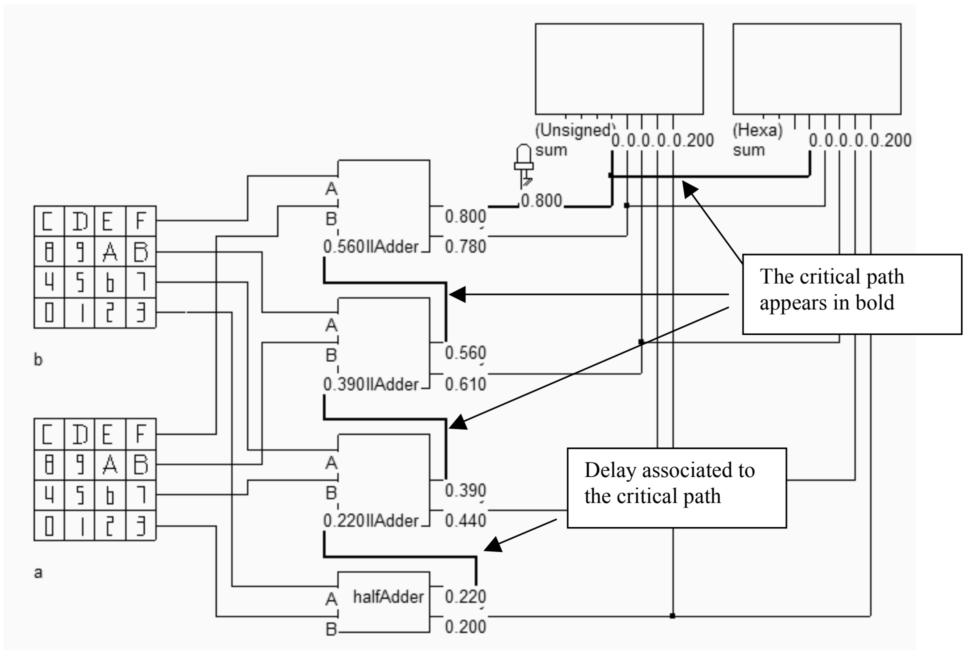


Fig. 7-23. The critical path of the four-bit adder (ADD4.SCH).

A manual design of the 4-bit Ripple Carry Adder

We described in this paragraph an example of manual design of a 4-bit ripple carry adder. The elementary adder circuit **FullAdder.MSK** from figure 7-17 is reused. We tried to compact the interconnect network by rerouting the input interconnects over the cell, and shortened the carry wires. The modified layout is **fadd.MSK**. A proposed strategy for connecting adders together in order to build a ripple carry adder is detailed in figure 7-24. The cell placement, supply and I/O routing positioning ease the connection between blocks, leading to a compact design and short connections. Each adder is supplied by VDD and VSS rails that are connected all together on the right side of the bloc. The carry propagation is realized by a short interconnect flowing from the bottom to the top of each cell. The *a* and *b* data are routed on the left side of the cell, the result on the right side.

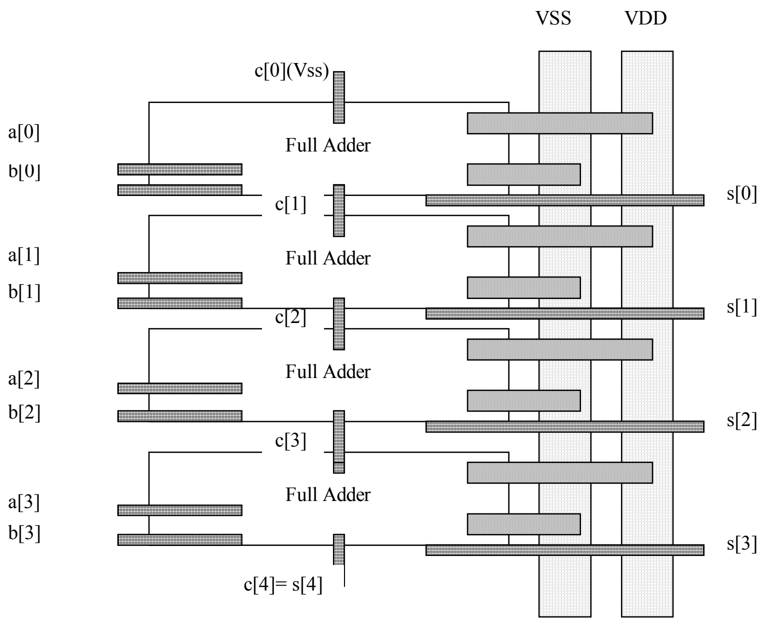


Fig. 7-24. The floor planning of the 4-bit ripple carry adder (ADD4.MSK).

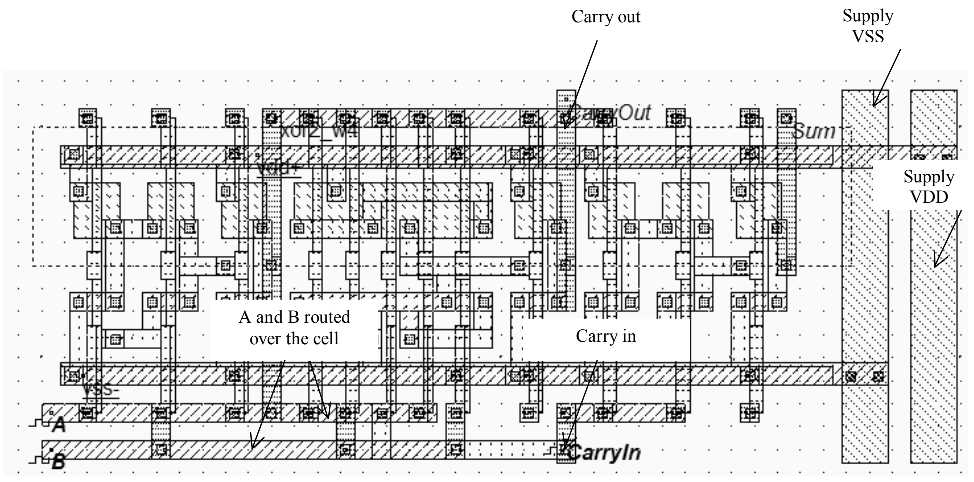


Fig. 7-25. Arranging the full adder to create a compact layout and to ease further connection (fadd.MSK)

Figure 7-26 details the four-bit adder layout based on the manual cell design of the adder. In Microwind2, the command **Edit → Duplicate X,Y** has been used to duplicate the full-adder layout vertically. The input and output names use brackets with an index (A[0]..A[3], B[0]..B[3] and Sum[0].. Sum [4]), so that Microwind automatically displays the logic value corresponding to A,B and Sum. For example, at time t=1.5ns (Figure 7-27), A=3,B=1, sum=4.

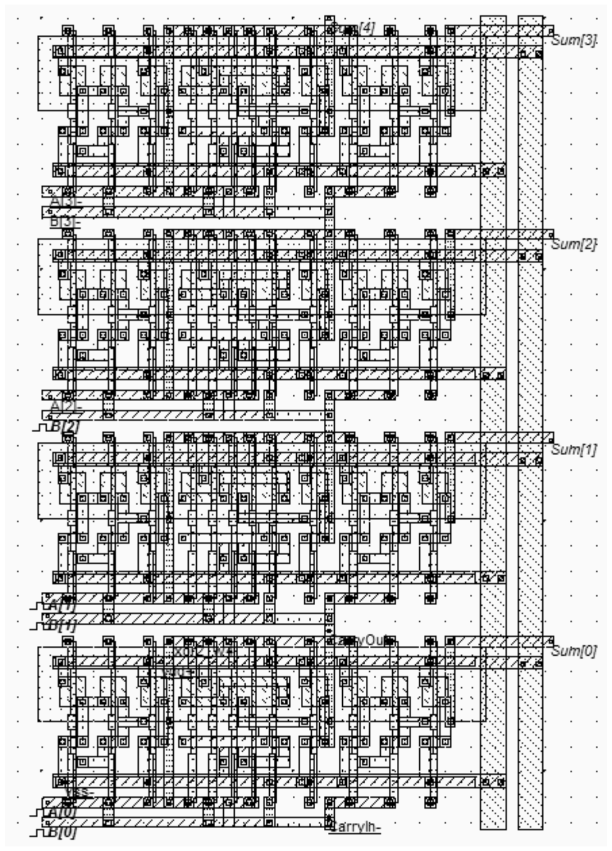


Fig. 7-26. Implementation of the four-bit adder (Add4.MSK).

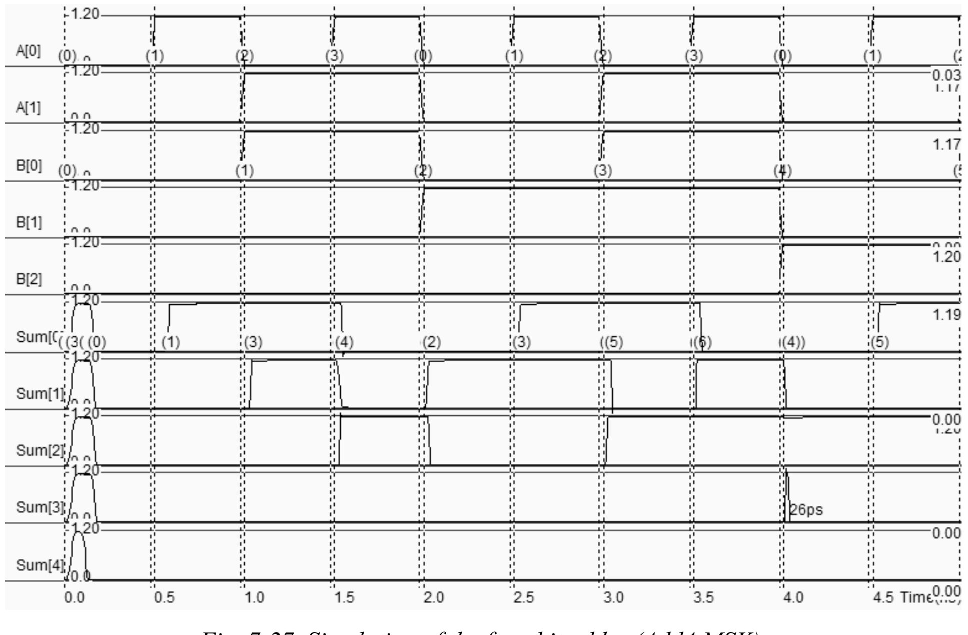


Fig. 7-27. Simulation of the four-bit adder (Add4.MSK).

A convenient method for characterizing the worst case circuit delay is to use the **Eye Diagram** simulation. This simulation mode superimposes output signals at any rise or fall edges of inputs A0..A3 or B0..B3. The cumulative drawing of the outputs is called the eye diagram. Figure 7-28 gives the eye diagram of the 4-bit ripple carry adder. It can be seen that the worst case delay is around 80ps. Remember that the critical delay was 0.8ns in the logic simulation. How do we explain that difference? First of all, because the layout level simulation to not investigate all input configurations, specifically the ones where the critical delay is involved. The inputs A2, A3,and B3 are inactive, and set to 0. Secondly, the logic simulation assigned a default 70ps delay per interconnect. This average delay is overestimated in our case, as most interconnects are routed very short.

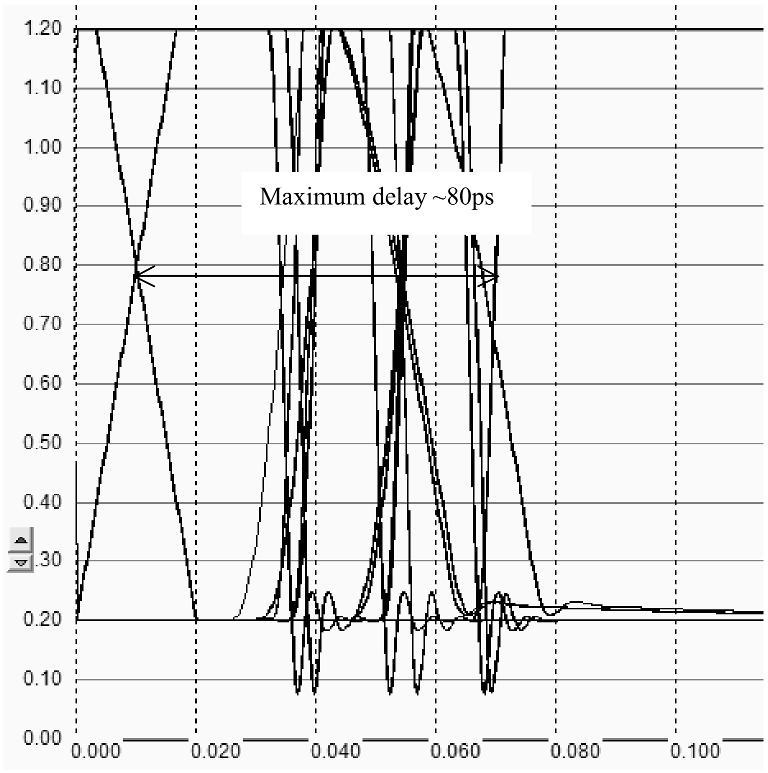
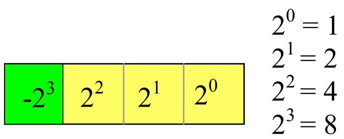


Fig. 7-28. Eye diagram of the four-bit adder (Add4.MSK).

5. Signed Adder

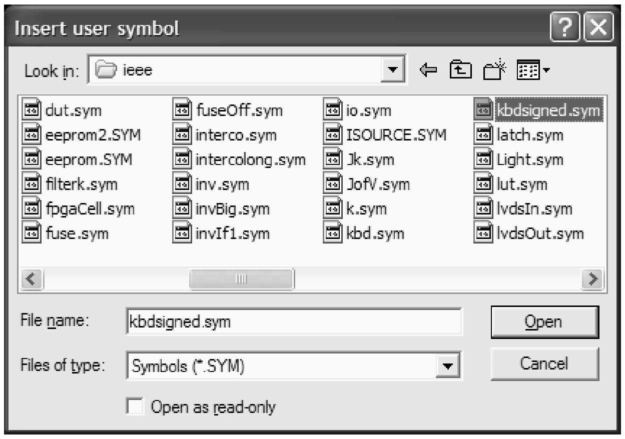
The signed integer format for a 4-bit input data is specified in table 7-6. The correspondence between unsigned and signed data for a 4-bit information is also reported. A specific symbol, namely **KbdSigned.SYM**, is available to support the generation of 4-bit signed input. The symbol may be loaded using the command Insert → User Symbol. Select the symbol **KbdSigned.SYM** in the list, as shown in figure 7-29. The display symbol on the left displays the 4-bit input data in unsigned integer form. The other symbol displays the same input information as a signed integer.



4 bit signed integer

I[3]	I[2]	I[1]	I[0]	Unsigned integer	Signed integer
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	-8
1	0	0	1	9	-7
1	0	1	0	10	-6
1	0	1	1	11	-5
1	1	0	0	12	-4
1	1	0	1	13	-3
1	1	1	0	14	-2
1	1	1	1	15	-1

Table 7-6. Correspondence between signed and unsigned 4-bit data


Insert user symbol

Look in: ieee

dut.sym, eeprom2.SYM, eeprom.SYM, filterk.sym, fpgaCell.sym, fuse.sym, fuseOff.sym, interco.sym, intercolong.sym, inv.sym, invBig.sym, invIf1.sym, io.sym, ISOURCE.SYM, Jk.sym, JofV.sym, k.sym, kbd.sym, kbdsigned.sym, latch.sym, Light.sym, lut.sym, lvdsIn.sym, lvdsOut.sym

File name: kbdsigned.sym

Files of type: Symbols (*.SYM)

Fig. 7-29: Inserting the specific keyboard symbol which supports signed input (KbdSigned.SYM)

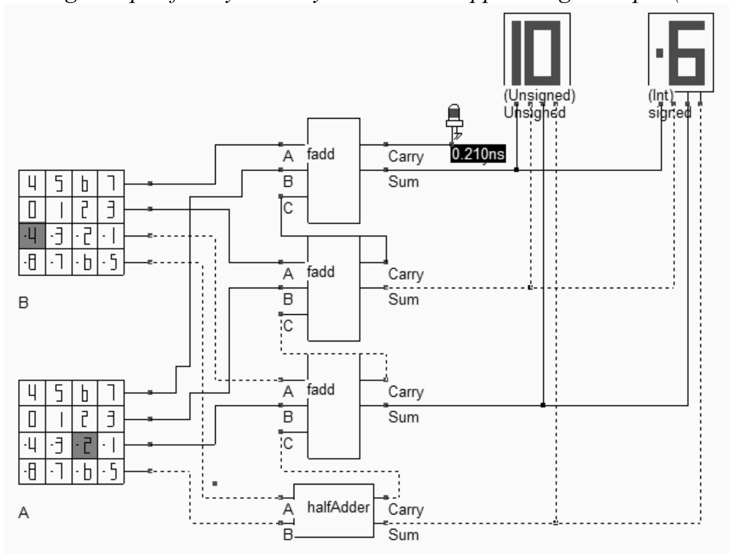


Fig. 7-30. The signed addition uses the same hardware as for the unsigned addition (ADD4Signed.SCH)

It can be seen from figure 7-30 that the unsigned adder circuit can be reused without any hardware modification for the addition of signed integers. The addition of *a=-2* with *b=-4* gives *sum=-6*.

6. Fast Adder Circuits

The main drawback of ripple carry adders is the very large computational delay due to the carry chain built in series. Several techniques have been proposed to speed up the addition, at the cost of more complex and power consuming circuits. We construct in this paragraph one type of high-speed adder and compare its performances to the ripple carry adder.

Carry Look Ahead Adder

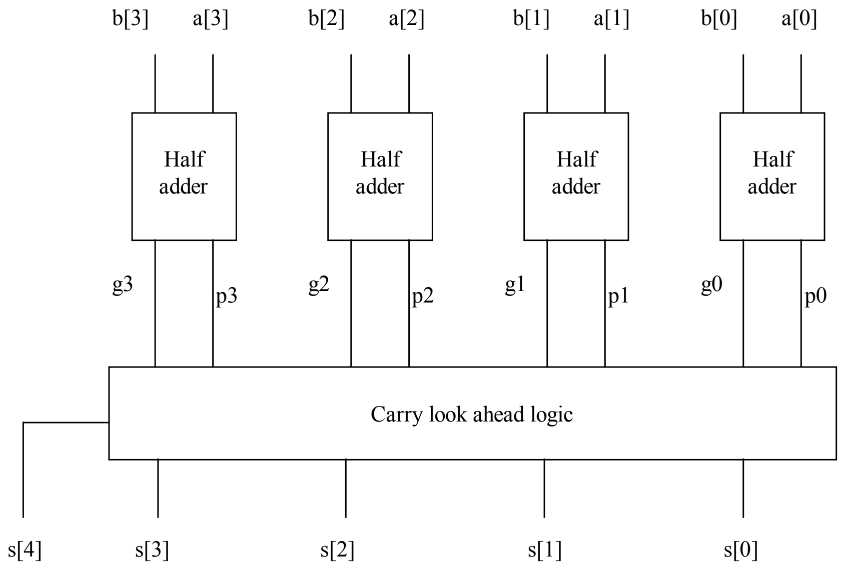


Fig. 7-31. The carry look-ahead logic circuit principles

The carry look-ahead adder computes the *c2* and *c3* carry functions by an optimized hardware, based on a complex gates. The start circuit is the elementary half-adder which produces *pi* and *gi* at each stage (equations 7-10 and 7-11). The sign "&" accounts for logical AND operator, "|" for OR and "^" for XOR. Then, the complex gate is built for each carry function, according to the Boolean expressions reported in equations 7-12.

$$\begin{aligned} g0 &= a[0] \,\&\, b[0] \\ g1 &= a[1] \,\&\, b[1] \\ g2 &= a[2] \,\&\, b[2] \\ g3 &= a[3] \,\&\, b[3] \end{aligned} \qquad \text{(Equ 7-10)}$$

$$\begin{aligned} p0 &= a[0] \wedge b[0] \\ p1 &= a[1] \wedge b[1] \\ p2 &= a[2] \wedge b[2] \\ p3 &= a[3] \wedge b[3] \end{aligned} \qquad \text{(Equ 7-11)}$$

$$\begin{aligned} s[0] &= p0 \\ c1 &= g0 \\ s[1] &= c1 \wedge p1 \\ c2 &= g1 \mid (p1 \,\&\, g0) \\ s[2] &= c2 \wedge p2 \\ c3 &= g2 \mid (p2 \,\&\, (g1 \mid (p1 \,\&\, g0))) \\ s[3] &= c3 \wedge p3 \\ c4 &= g3 \mid (p3 \,\&\, (g2 \mid p2 \,\&\, (g1 \mid (p1 \,\&\, g0)))) \\ s[4] &= c4 \end{aligned} \qquad \text{(Equ 7-12)}$$

The equations 7-12 can be translated into layout using XOR gates and complex gates. The schematic diagram of the carry look-ahead adder is given in figure 7-32. Each half-adder circuit produces the *gi* and *pi* data. Complex gates produce the internal carry *ci*, while XOR gates generate the outputs *s[i]*.

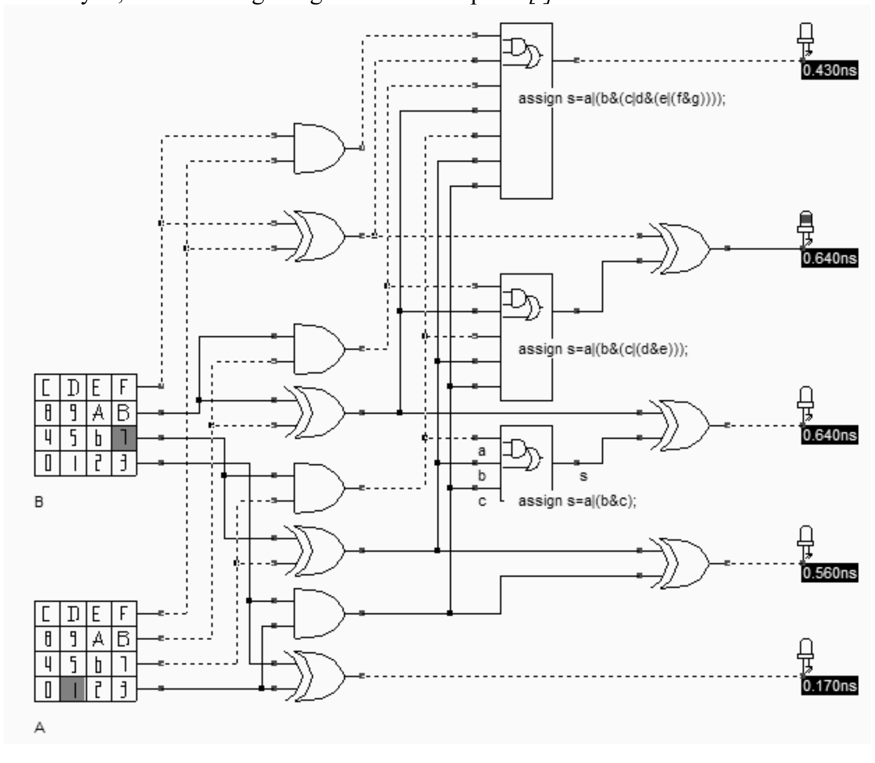


Fig. 7-32. The logic implementation of the carry look-ahead adder (Add4LookAhead.SCH)

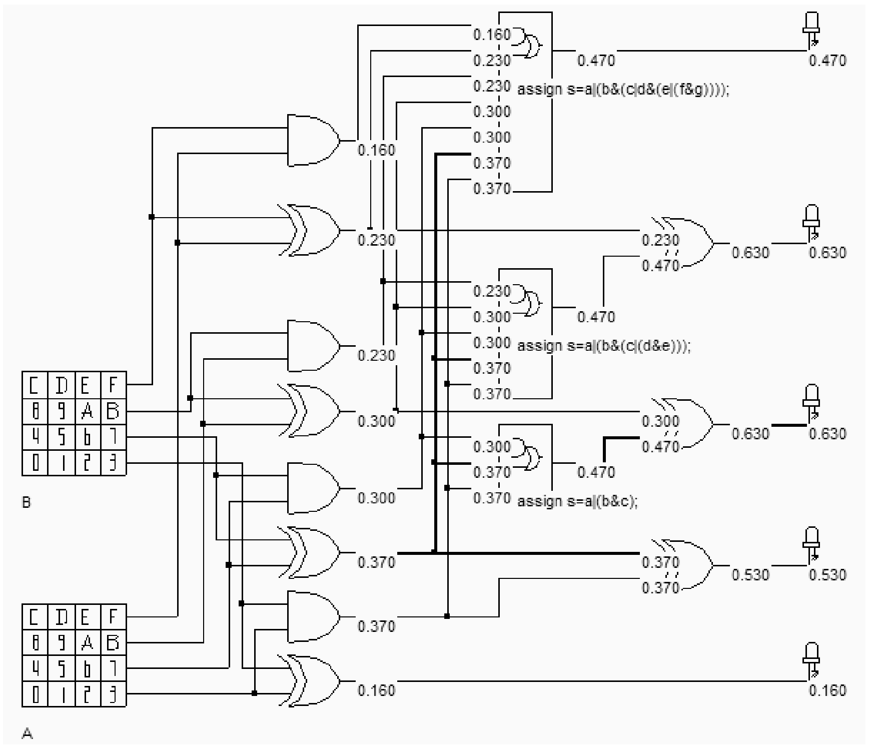


Fig. 7-33 Evaluation of the critical delay in the carry look-ahead adder (Add4LookAhead.SCH)

The key question is to know to which extend the critical delay has been reduced. The answer is given in figure 7-33. We notice a reduction of the critical delay from 0.81ns (Previous ripple carry circuit) down to 0.63ns (This look-ahead circuit). A remarkable point is the homogenous switching delay for most outputs, as compared to the ripple carry result which increases with the stage number.

Complex Gate Implementation

Most logic cells involved in the construction of the carry look-ahead adder are conventional AND and XOR gates. However, the internal carry signals C2, C3 and C4 are combinations of AND and OR operators that are perfect candidates for complex gate implementation.

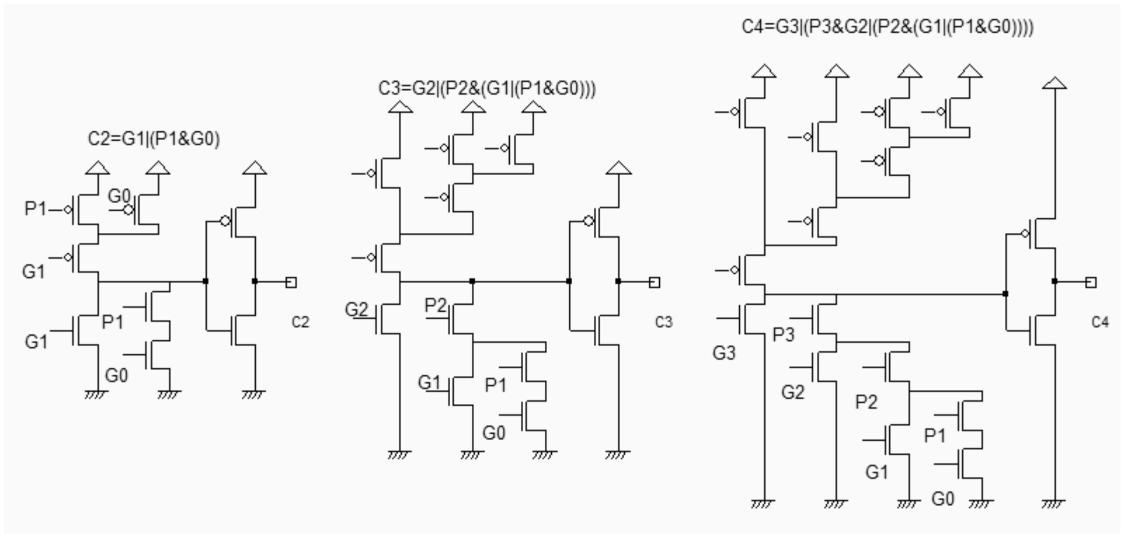
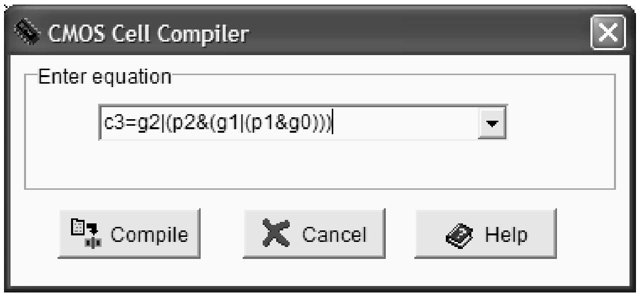


Fig. 7-34. The complex gates used in the carry look-ahead adder (Add4LookAheadCmos.SCH)

The circuit c3 can be generated by the CMOS cell compiler, through the command **Compile → Compile One Line**. The complex gate description is shown in figure 7-35. The layout generated from this description appears in figure 7-36. It is interesting to notice that the cell compiler could not implement the MOS devices using a continuous diffusion. Three separate p-diffusion areas have been used for implementing the pMOS devices.



c3=g2|(p2&(g1|(p1&g0)))

Fig. 7-35. Layout of the complex gate C3 (RippleCarryC3.MSK)

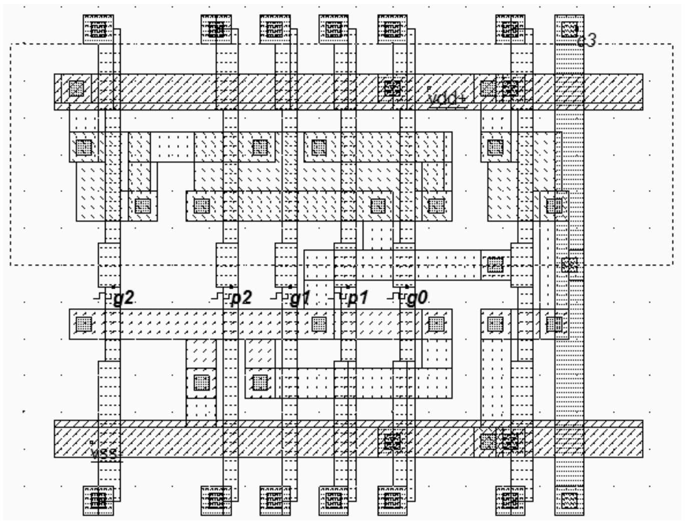


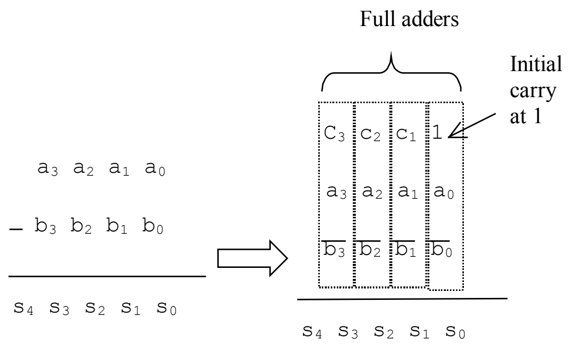
Fig. 7-36. Layout of the complex gate C3 (RippleCarryC3.MSK)

7. Subtractor Circuit

The substractor circuit can be built easily with a full adder structure as for the adder circuit. The main difference is the needs for a 2's complement circuit which inverts the value of *b*, and the replacement of the half adder by a full adder, as the initial carry must be 1 (Figure 7-37). The logic circuit corresponding to the 4-bit substractor is reported in figure 7-38. Some examples of substractor results are also listed.

a[0..3]	b[0..3]	s[0..4]=a-b (hexa)	s[0..4] (Decimal)
0	0	00	0
0	1	0F	-1
1	0	01	1
7	9	0E	-2
C	6	06	6
F	F	00	0

Table 7-7. Substractor result examples


Full adders
Initial carry at 1
C_3 C_2 C_1 1
a_3 a_2 a_1 a_0
$\overline{b_3}$ $\overline{b_2}$ $\overline{b_1}$ $\overline{b_0}$
a_3 a_2 a_1 a_0
$-$ b_3 b_2 b_1 b_0
s_4 s_3 s_2 s_1 s_0

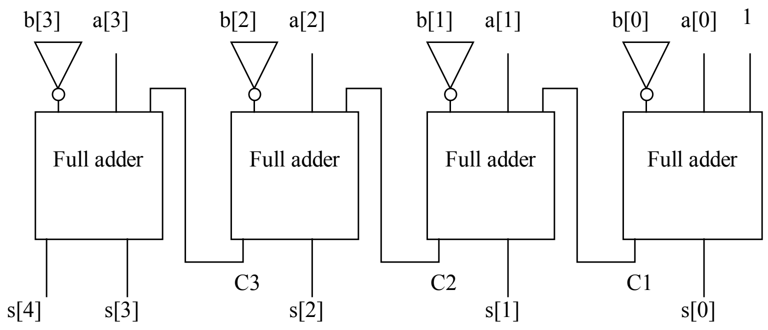


Fig. 7-36. Structure of the 4-bit substractor

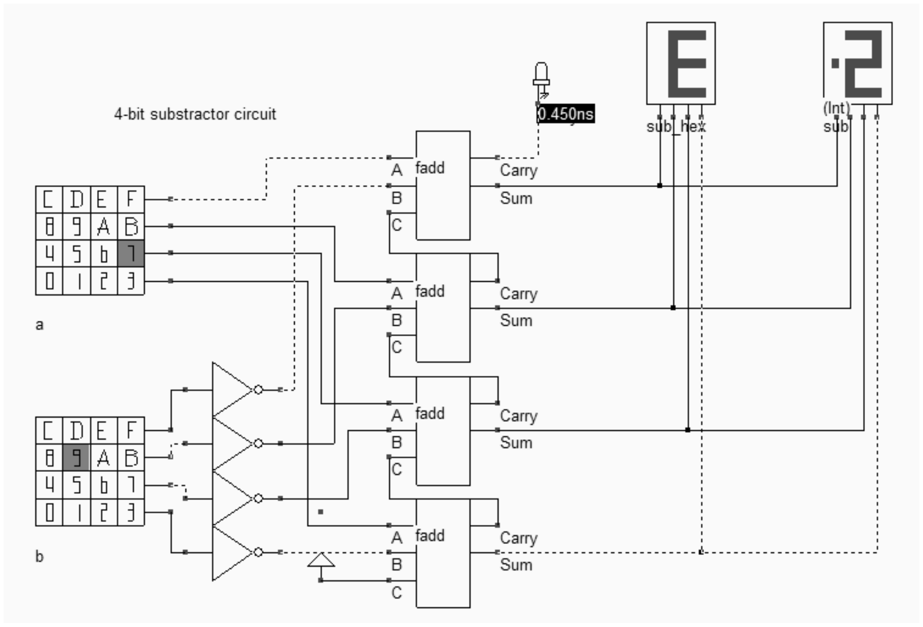


Fig. 7-37. Logic simulation of the 4-bit Substractor (Sub4.SCH)

8. Comparator Circuit

One-bit Comparator

The truth table and the schematic diagram of the comparator are given below. The A=B equality is built using an XNOR gate, and A>B, A<B are operators obtained by using inverters and AND gates.

A	B	A>B	A<B	A=B
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	1

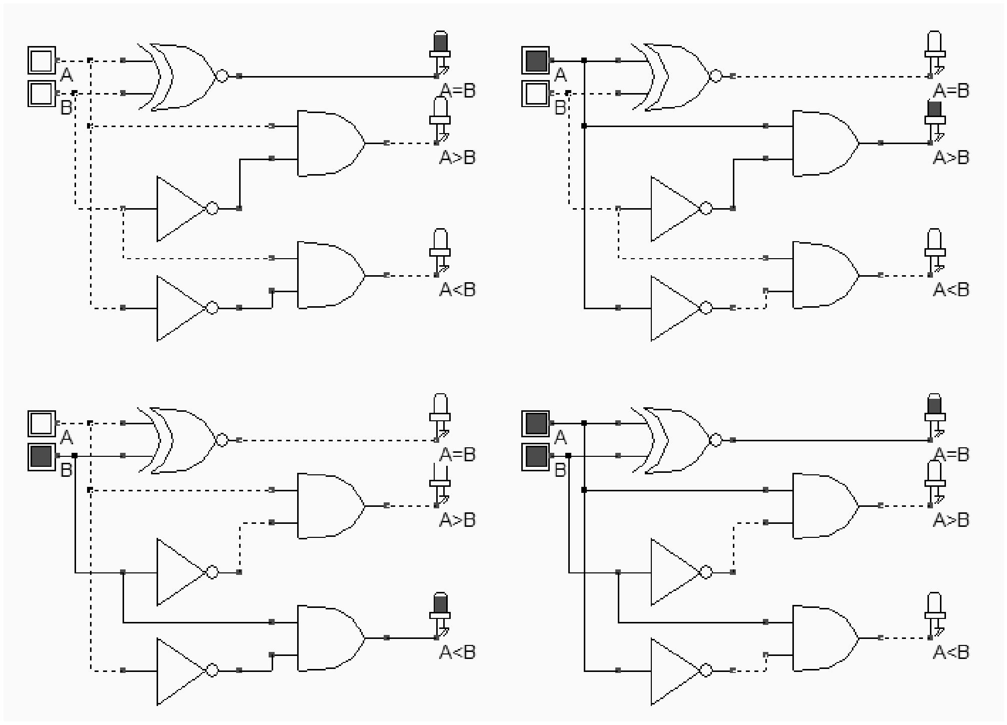


Fig. 7-38 The truth table and schematic diagram of the comparator (CompTest.SCH).

Once the logic circuit of the comparator is designed and verified at logic level, the conversion into Verilog is realized by the command **File → Make verilog File**. During the conversion, the node name of some signals has been changed. For example the node *A<B* has been modified into *AiB*. This is because some characters have another signification in Verilog, and cannot be part of a node name. Then, the Verilog text is converted into layout using Microwind. The layout of the comparator circuit is given in Figure 7-39. The XNOR gate is located at the left side of the design. The inverter and NOR gates are at the right side.

.

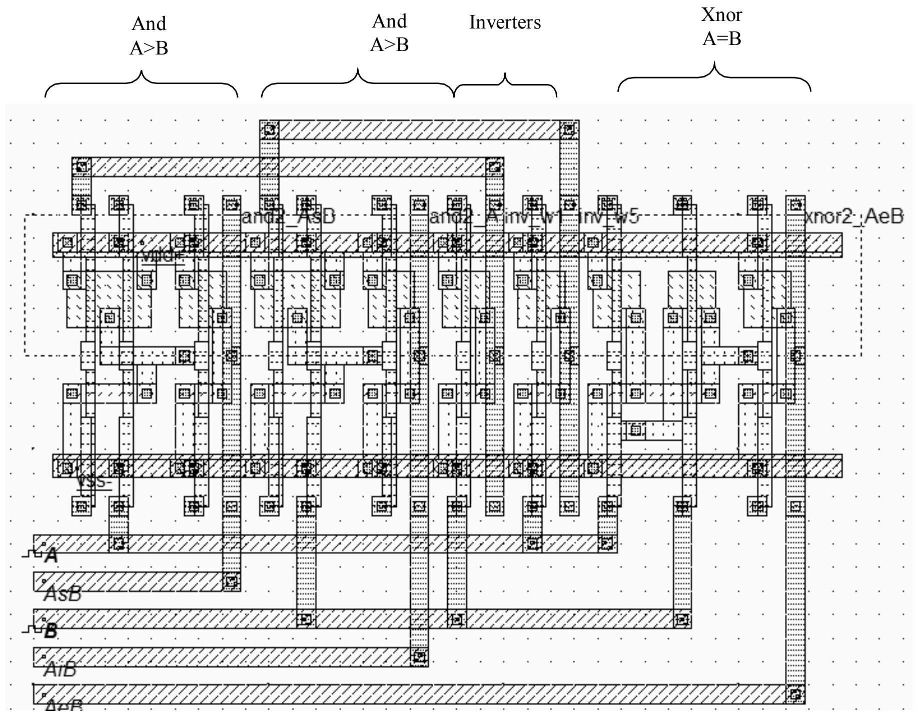


Fig. 7-39 The implementation of the comparator (Comp.MSK).

The simulation of the comparator is given in figure 7-40. After the initialization, A=B rises to 1. The clocks A and B produce the combinations 00,01,10 and 11.

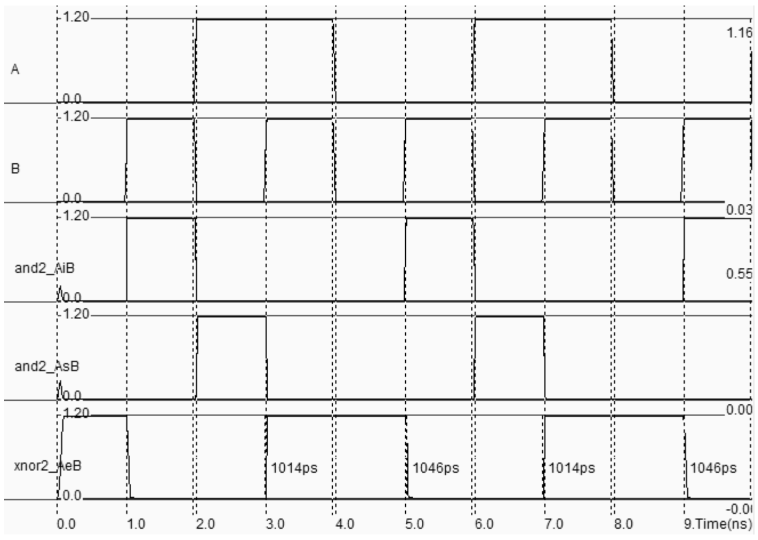


Fig. 7-40. Simulation of the comparator (COMP.MSK file).

n-bit Comparator

An efficient technique for n-bit comparison is based on the use of adder circuits. In the schematic diagram of figure 7-41, the adder system is modified into a comparison system, by computing the Boolean function A-B. The 'equal' operator is built using simple AND functions. Consequently, each elementary comparator includes the full-adder layout, one inverter and one AND gate.

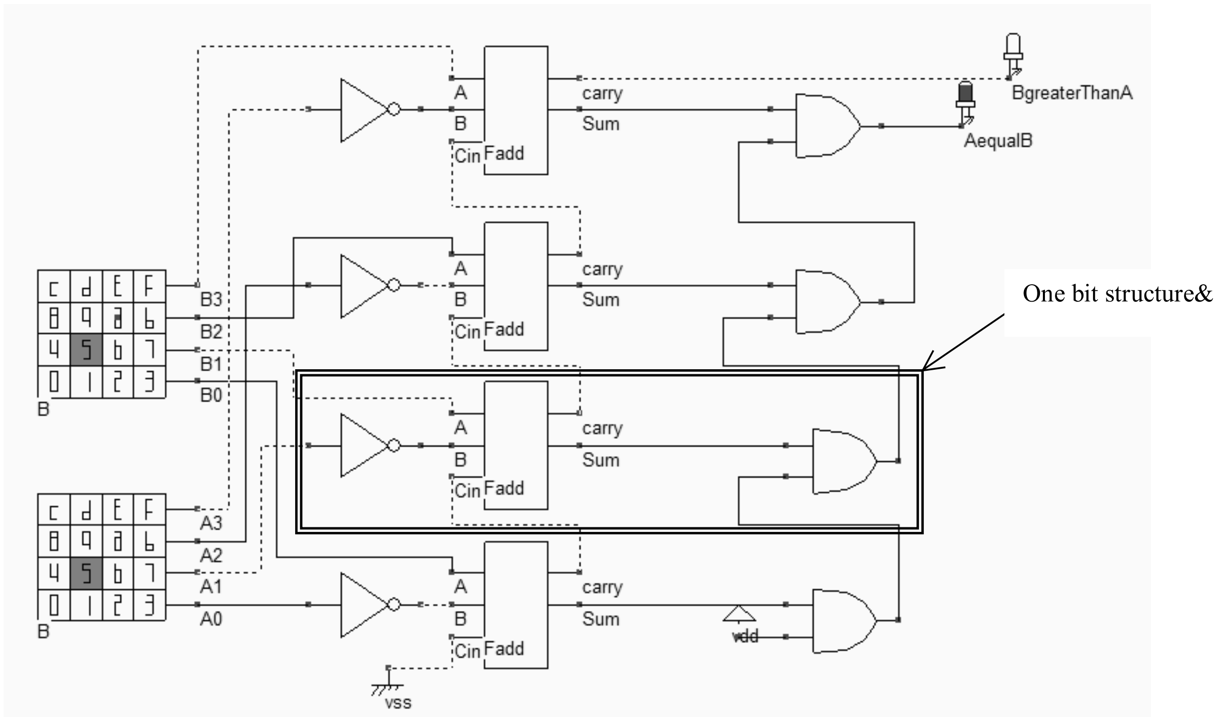


Fig. 7-41. Schematic diagram of a 4-bit comparator (COMP4.SCH).

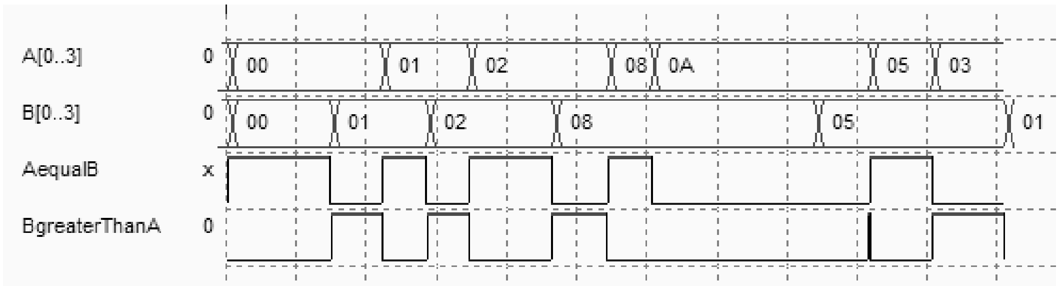


Fig. 7-42. Simulation of the 4-bit comparator (COMP4.SCH).

9. Student project: A Decimal Adder

This paragraph details a student project that performs the addition of two Binary Decimal Coded (BCD) numbers X and Y. The numbers range from 0 to 9, and the result (between 0 and 18) is visualized on hexadecimal displays. The specification of this project is described in the schematic diagram of figure 7-43.

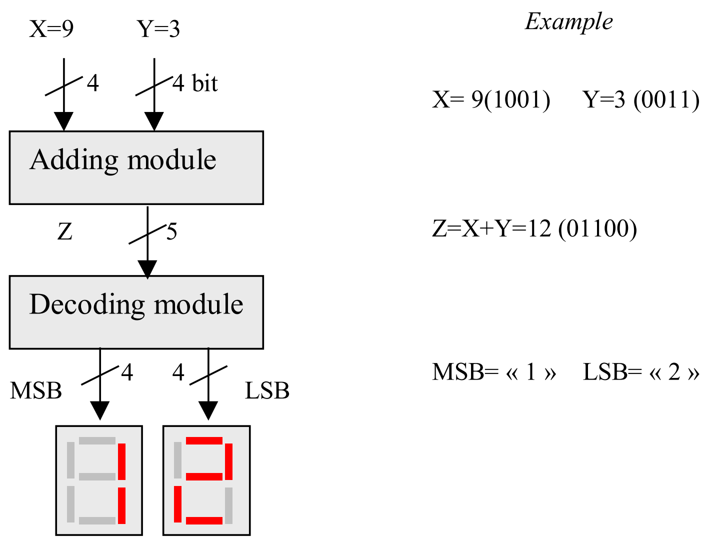


Fig. 7-43: Structure of the decimal adder project

4-BIT ADDER

Four one-bit adders linked in cascade construct the 4-bit adder. You may use the adder symbol created previously (Fadd.SYM). The 4-bit adder from figure 7-20 may be regrouped into a new user symbol Add4.SYM, which will be reused in the project.

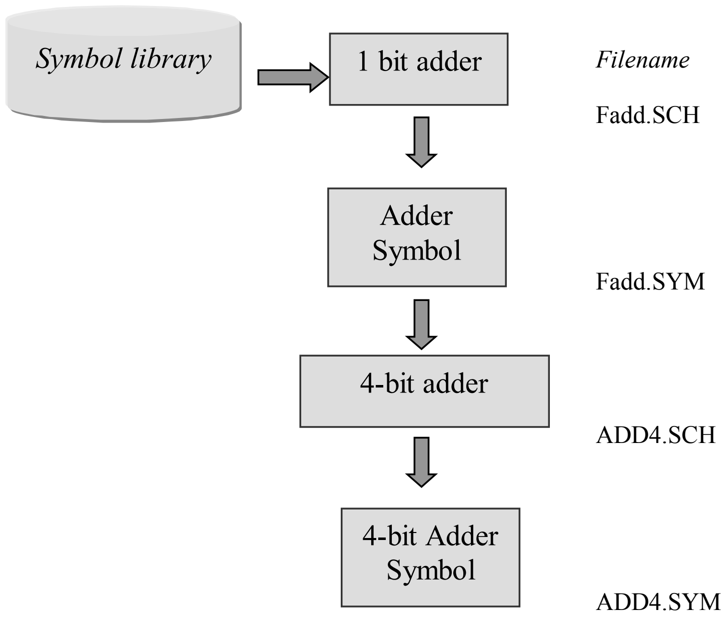


Fig. 7-44: Hierarchical construction of the 4-bit adder symbol

Use the command **File →Schema to New Symbol** to transfer the 4-bit adder schema into a user symbol. The schematic diagram is analyzed and a symbol is proposed regrouping all declared inputs and outputs. The button, clock, pulse and keyboard symbols are considered as inputs. The led and displays are considered as output. The user symbol with a name corresponding to the schematic diagram name, with the '.SYM' appendix, is stored in the current directory. In figure 7-45, the symbol ADD4.SYM is connected to two keyboards and one hexadecimal display to verify the correct behavior of the 4-bit adder.

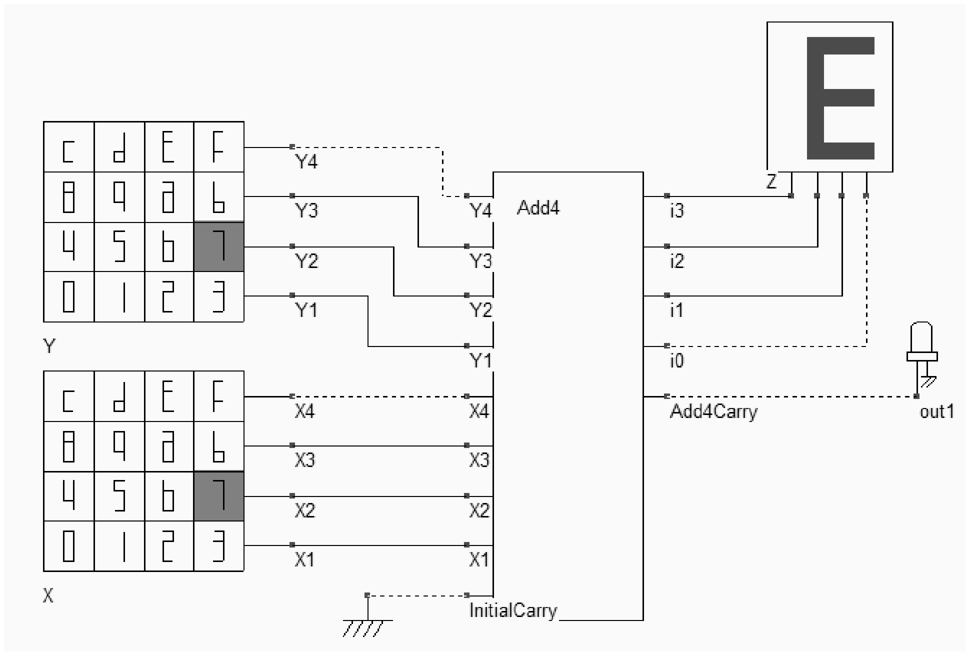
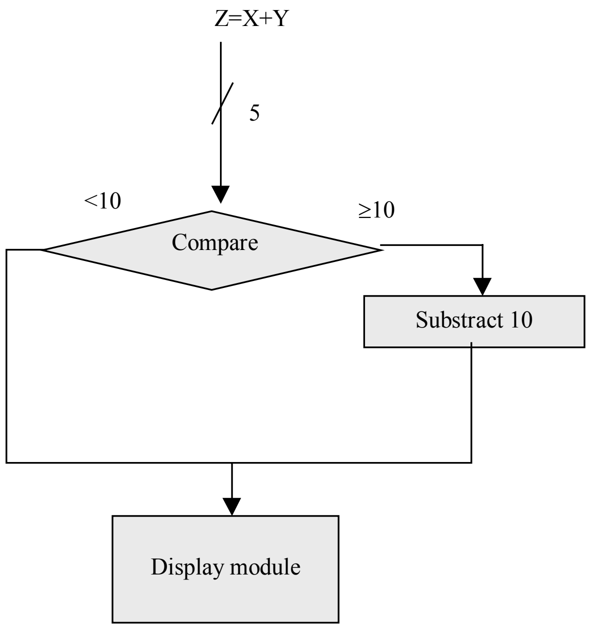


Fig. 7-45: Validation of the 4-bit adder symbol (ADD4Test.SCH)

DECODER MODULE

The objective of the decoding module is to split the binary result of the addition into two BCD codes, one representing the tenth bit ranging from 0 to 1, the other representing the unit bit ranging from 0 to 9. The principle of the decoding circuit is shown in figure 7-46. Firstly, the result Z= X+Y is passed through a comparator. IF Z<10, the result is sent directly to the visualizing module. If not, the result is adjusted by subtracting 10.



Z=X+Y (Hex Binary)		SupOrEqualTo10	LessThan10
0	00000	0	1
1	00001	0	1
2	00010	0	1
3	00011	0	1
4	00100	0	1
5	00101	0	1
6	00110	0	1
7	00111	0	1
8	01000	0	1
9	01001	0	1
A	01010	1	0
B	01011	1	0
	…		
1E	11110	1	0

Figure 7-46: Principles and truth table of the decoder module

COMPARE TO 10

The function *SupOrEqualTo10* may be written using the following Boolean equation. One possible implementation is reported in figure 4-47. The positive logic was used for clarity, although negative logic is a better choice from delay and power consumption points of view.

$$SupOrEqualTo10 = Z4 \,|\, (Z3 \,\&\, Z2) \,|\, (Z3 \,\&\, Z1) \qquad \text{(Equ 7-13)}$$

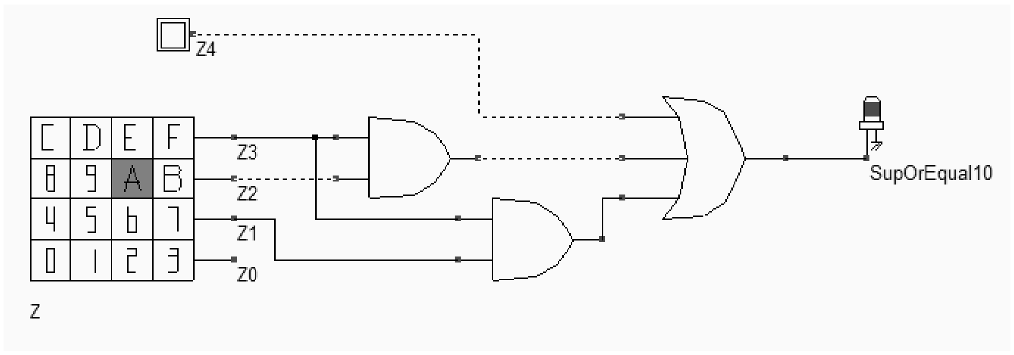


Fig. 7-47. A possible implementation of the function SupOrEqualto10 (SupEqu10.SCH)

SUBSTRACT 10

The substractor module is enabled when the result Z= X+Y is greater or equal to 10. In that case, a substract-by-10 operation is performed. The substractor circuit is simply an adder with inverted input B, and an input carry set to 1. Consequently, the substractor by 10 can be derived from the 4-bit Substractor circuit, as shown below. A **sub10.SYM** symbol is then created, with Z as inputs and LSB (Least significant bit) as outputs.

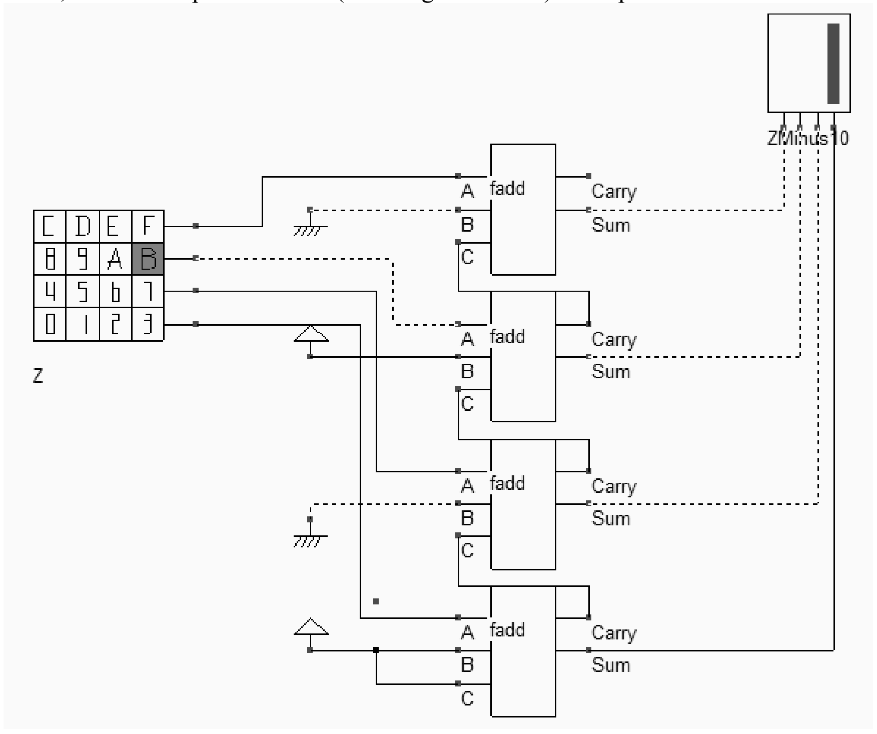


Fig 7-48. Subtraction by 10 (SUB10.SCH)

Final BCD Adder

To complete the circuit, a multiplexor circuit decides whether the *Z* result or the *ZMinus10* result is sent to the display. The multiplexor is made from 4 elementary multiplexor devices. When *SupOrEqu10* is asserted, the "1" appears in the left display, and the *ZMinus10* result appears in the right display. When *SupOrEqu10* is 0, a "0" appears in the left display and Z appears in the right display. The final circuit is shown in figure 7-49.

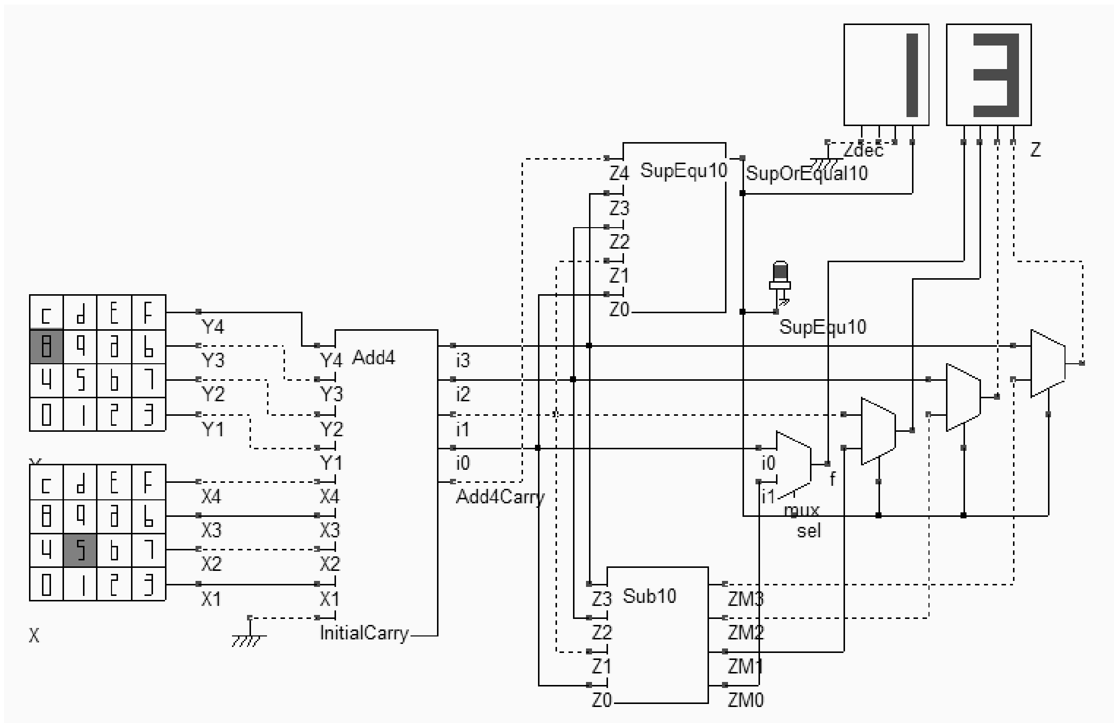


Fig. 7-49. Final circuit of the BCD adder (AdderBcd.SCH)

10. Multiplier

Let us illustrate the multiplication process through a small example based on 4-bit unsigned integer. The basic mechanism is a product A_i&B_j, combined with the addition which involves a carry. The result propagates down to the result.

```
A      0110  (6)
B      0111  (7)
       0110
      0110
     0110
    0000     .
  00101010 (42)
```

The multiplication of integer numbers A and B can be implemented in a parallel way using elementary binary multiplication circuits [Bellaouar]. Within each multiplication cell, the key idea is to compute the product P=Ai&Bi, and add the previous sum and previous carry. The next sum is propagated to the bottom, while the next carry is connected to the multiply cell situated at the left side (Figure 7-50).

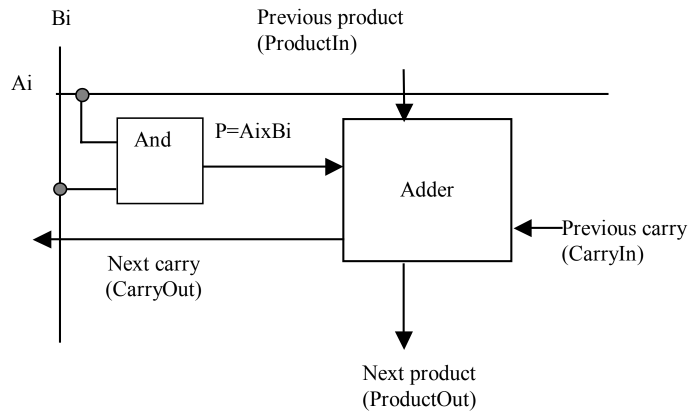


Fig. 7-50. Principles for the elementary multiplication

The elementary multiplication cell should verify the truth table given below. The cell can be made up of a full-adder cell and an AND gate, as shown in the schematic diagram given in figure 7-51.

Multiplier				
AixBi	ProductIn	CarryIn	CarryOut	ProductOut
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

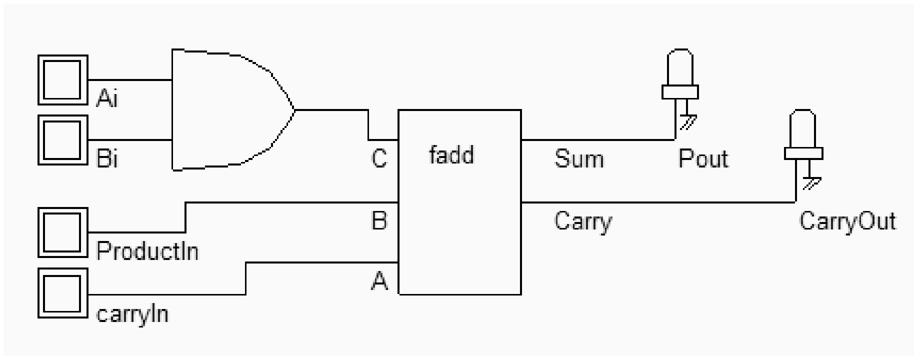


Fig. 7-51. Principles for the elementary multiplication (MUL1.SCH)

A 4x4 bit multiplication is proposed in Figure 7-52. The circuit multiplies input *A* (Upper keyboard) with input *B* (Lower keyboard) which produces an 8-bit result *P*. In the logic simulation, the 8-bit display is configured in decimal mode to ease the interpretation of the result.

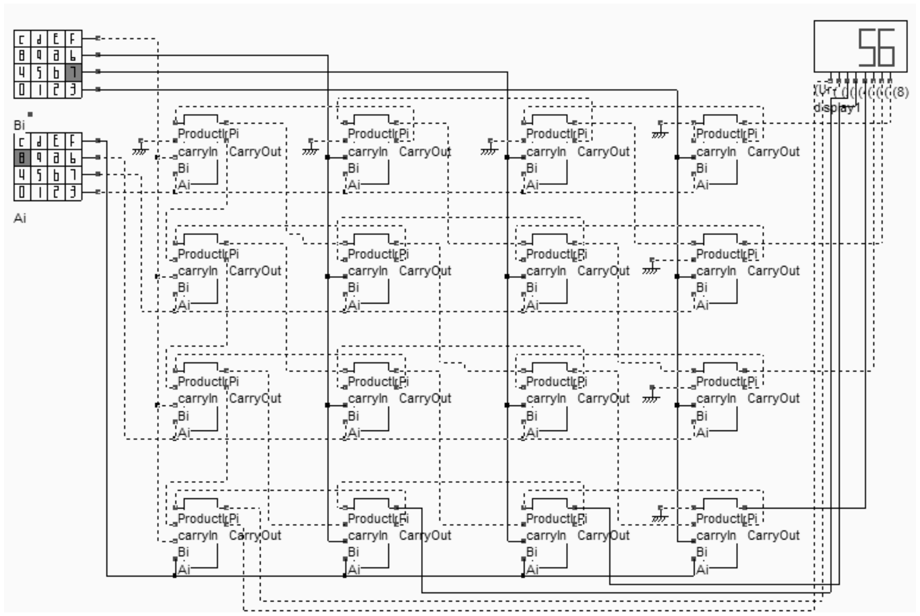


Fig. 7-52. Schematic diagram of the 4x4 bit multiplier (Mul44.SCH).

Compiled Multiplier

The 4x4 bit multiplier can be translated into layout thanks to the Verilog compiler. By default, the circuit is constructed by placing all basic gates on a single row and performing the routing. The input and output signals are routed below the active area, and the internal wire routing is performed on the upper part of the active area. Notice that the routing is performed using metal3 for horizontal connections and metal2 for vertical connections. Industrial routing tools take advantage of the other metal layers (metal4,5,6,etc..) to create a more layout-effective implementation.

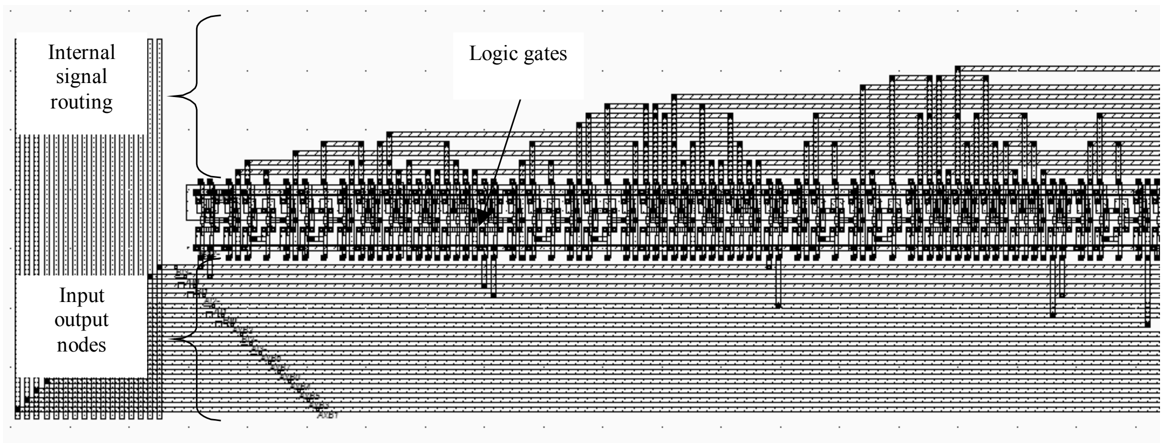
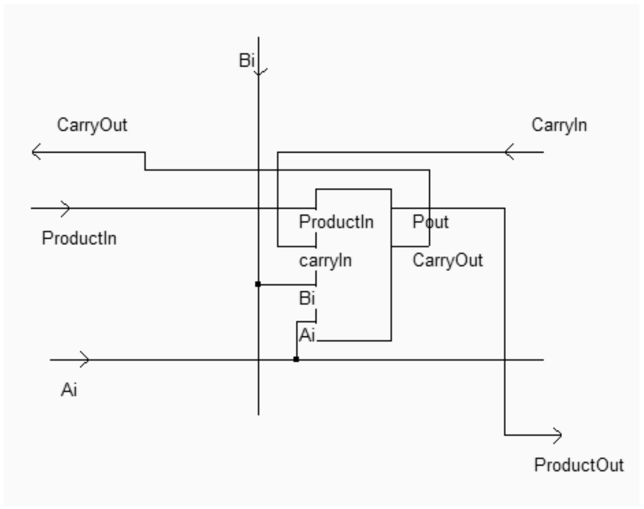


Fig. 7-53 Linear implementation of the 4x4 bit multiplier (Mul44.MSK).

An compact Multiplier layout

A more efficient approach consists in using an AND gate and a full adder, and placing the interconnects in order to facilitate an iterative implementation. The inputs *CarryIn*, *Ai*, *Bi* and *ProductIn* and the outputs *ProductOut* and

CarryOut are organized in such a way that the array can be extended to a larger format. The general floorplan of the multiplier is shown in figure 7-54. An *n*x*n* multiplier would require n^2 elementary multiplier cells, each based on one AND gate and one full adder.



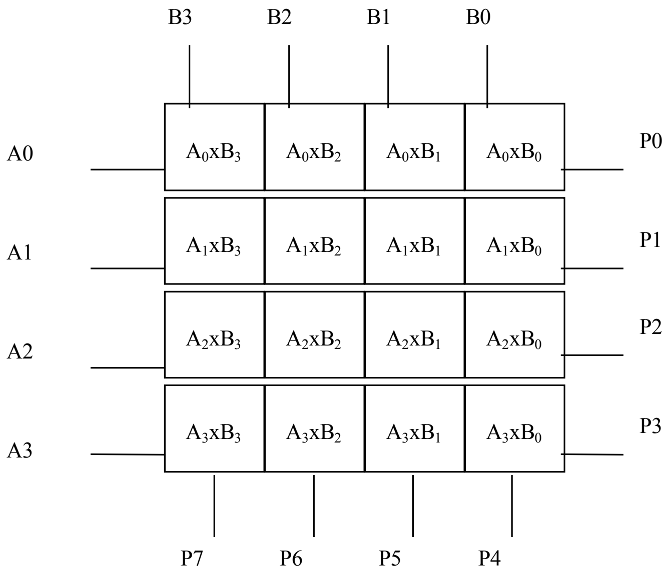


Fig. 7-54. Floor planning of the 4x4 bit multiplier

The main drawback of this architecture is the very important critical delay, which consists of the carry propagation through ten multiplier cells (Figure 7-55). Several algorithms exist for accelerating the multiplication (See reference [xxx]), more or less by a factor of two. Another problem is the design time needed to implement this array structure, and the severe risk of error due to manual arrangement of the cells and interconnects. In ultra-deep submicron technologies, compact 8x8 bit or 16x16 bit multipliers are proposed as intellectual property blocks (IP <Gloss>).

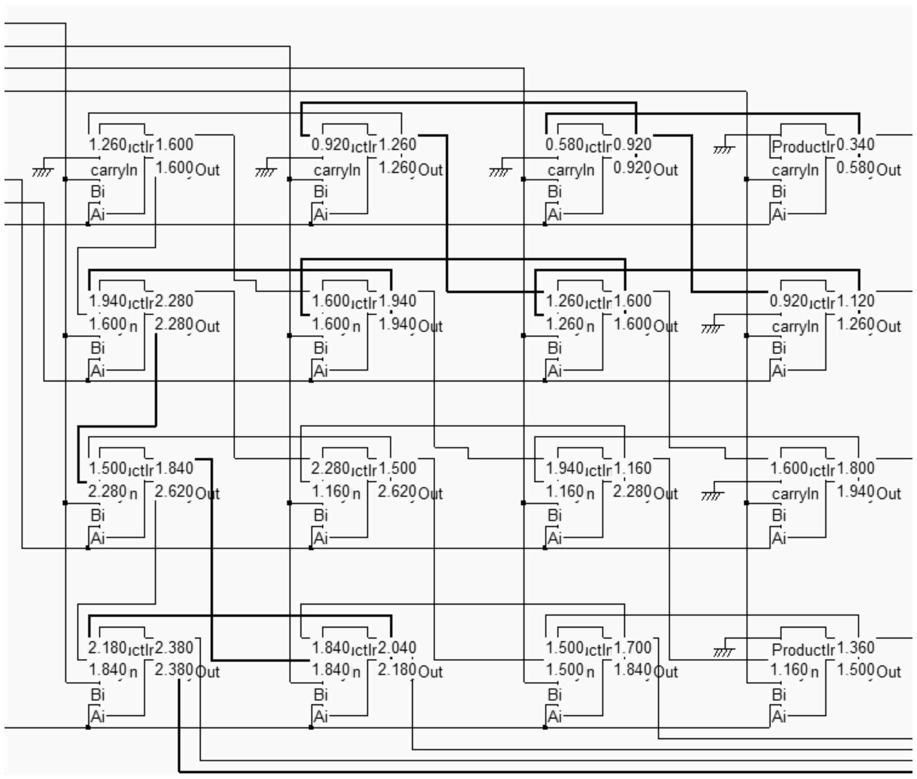


Fig. 7-55. Estimation of the critical delay path (Mul44.SCH).

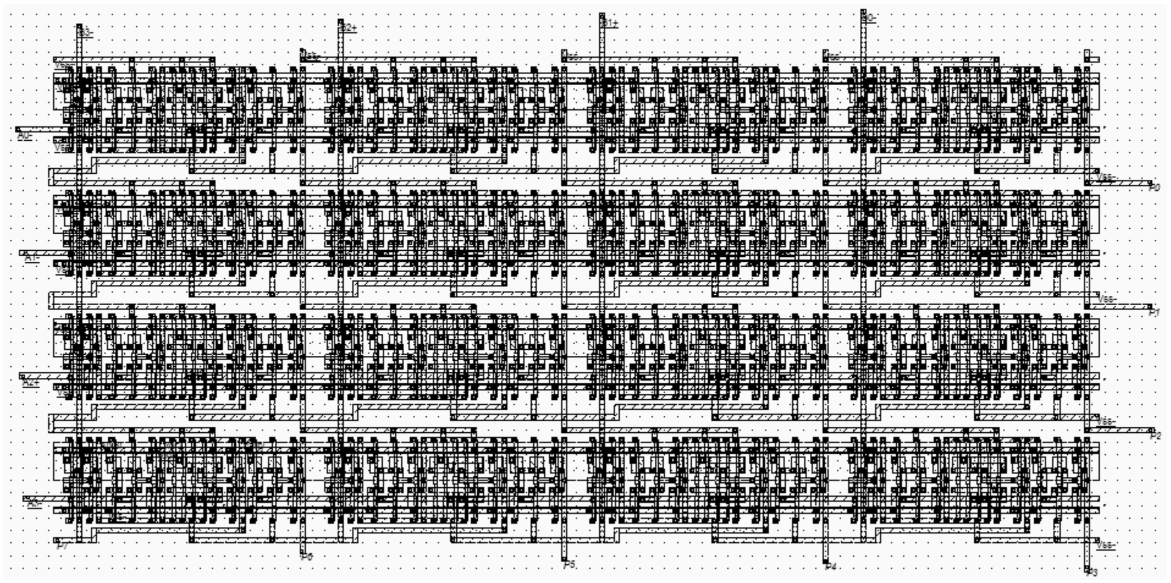


Fig. 7-56. Design of the 4x4 bit multiplier (Mul44Compact.MSK).

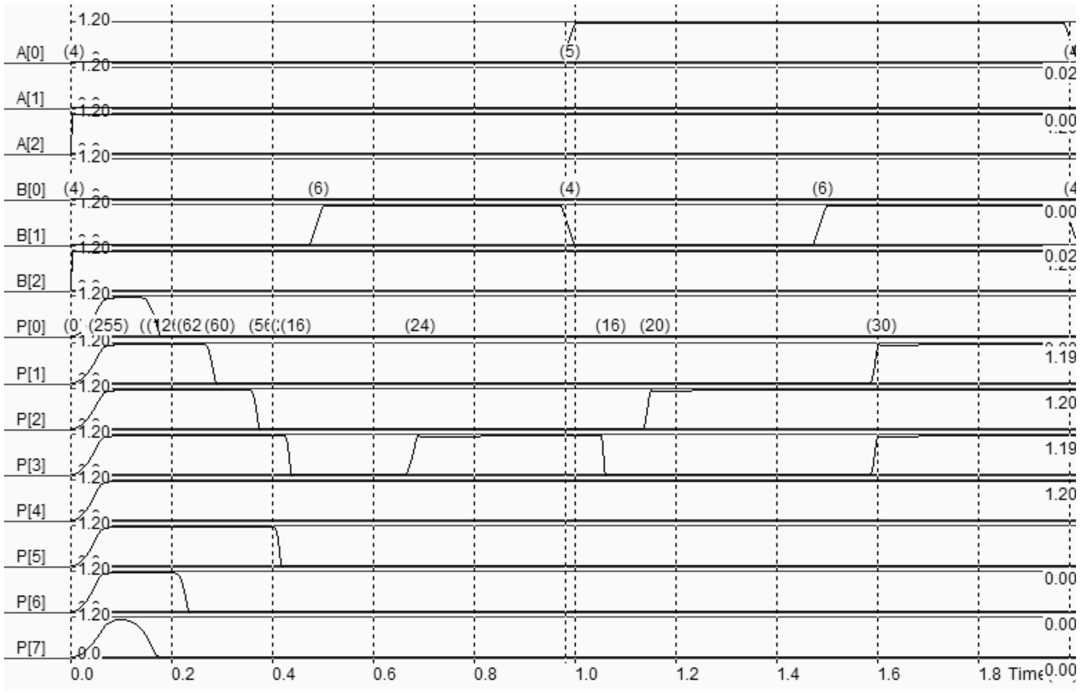


Fig. 7-57. Simulation of the 4x4 bit multiplier (Mul44Compact.MSK).

The simulation of the 4x4 parallel multiplier is given in figure 5-57. As for the adder, we use signal names with brackets, stating with index 0, to enable Microwind to compute the equivalent logic value of the A,B and P signals. Notice that the initialization phase is almost equal to 0.5nS. At time t=1.0ns, A=5, B=4, and P=20 after 150ps delay.

11. Arithmetic and logic Units (ALU)

The purpose of this chapter is to design and simulate an 8-bit arithmetic/logic unit (ALU). The ALU is a digital function that implements basic micro-operations on the information stored in registers. In figure 7-58, the ALU receives the information from the registers *A* and *B* and performs a given operation as specified by the control function F. The result is produced on *S*.

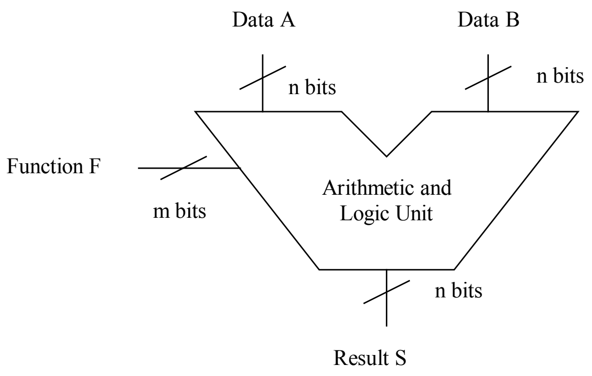


Fig. 7-58. Principles for the Arithmetic and Logic Unit

In DSCH, a simplified model of the Intel 8051 micro-controller is included. The 8051 core includes an arithmetic and logic unit to support a huge set of instructions. Most of the results are The data format is 8 bit. We consider here the following instructions, listed in table 7-8. Some instructions do not appear in this list, such as the multiplication and division.

Mnemonic	Type	Description
CLR	Clear	Clear the accumulator
CPL	Complement	Complements the accumulator, a bit or a memory contents. All the bits will be reversed.
ADD	Addition	Add the operand to the value of the accumulator, leaving the resulting value in the accumulator.
SUBB	Substractor	Subtracts the operand to the value of the accumulator, leaving the resulting value in the accumulator.
INC	Increment	Increment the content of the accumulator, the register or the memory.
DEC	Decrement	Decrement the content of the accumulator, the register or the memory.
XRL	XOR operator	Exclusive OR operation between the accumulator and the operand, leaving the resulting value in the accumulator.
ANL	AND operator	AND operation between the accumulator and the operand, leaving the resulting value in accumulator.
ORL	OR operator	OR operation between the accumulator and the operand, leaving the resulting value in accumulator.
RR	Rotate right	Shifts the bits of the accumulator to the right. The bit 0 is loaded into bit 7.
RL	Rotate left	Shifts the bits of the accumulator to the left. The bit 7 is loaded into bit 0.

Table 7-8. Some important instructions implemented in the ALU of the 8051 micro-controller

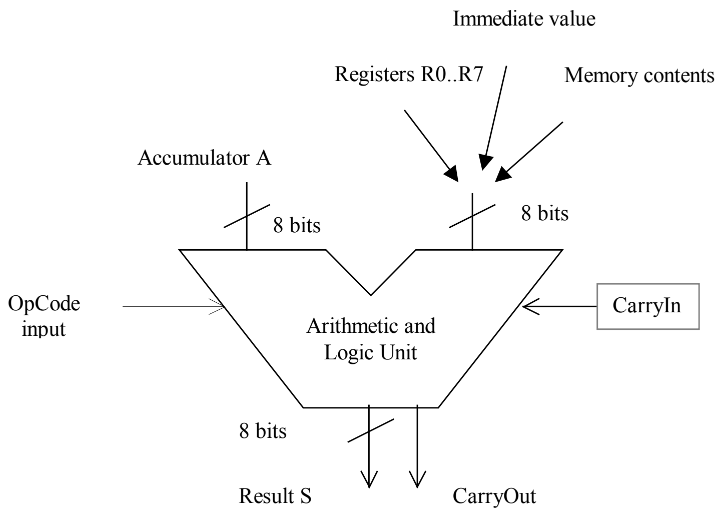


Figure 7-59. The arithmetic and logic unit of the 8051

For example:

- ADD A,R0 (Opcode 0x28) overwrites the accumulator with the result of the addition of A and the content of R0.
- SUBB A,#2 (Opcode 0x94 0x02) overwrites the accumulator with the result of the subtraction of A and the sum of the Carry and the byte 0x02.
- INC A (0x04) increments the content of the accumulator.
- DEC A (0x14) Decrements the content of the accumulator.
- ANL A,#10 (0x54) overwrites the accumulator with by the AND-gating of A and the constant 0x10.
- ORL A,R7 (0x4F) overwrites the accumulator with by the OR-gating of A and the content of R7.
- XRL A, R1 (0x69) overwrites the accumulator with the result of the XOR-gating of A and the content of the internal register R1.

The description of the 8051 arithmetic and logic unit in DSCH2 is outlined below. The description language, originally in DELPHI [Borland], has been translated for clarity into VHDL [John]. The declaration of the ALU always starts by a description of the variables (figure 7-60) in the *Entity*, where we recognize the operation code *op_Code*, with a 4 bit format fitted to the number of cases (The statement *When* appears 12 times in figure 7-60), the accumulator input *source_a*, the other input *source_b*, the previous *carry_in*. The inputs *source_a* and *source_b* are in an 8-bit format. The input *carry_in* is declared in the most common logic format (STD_LOGIC). The standard logic values are mainly the two logic values '0' and '1'. We also use 'X' for unknown, 'Z' for high-impedance and '-' for "don't care". The result *result_s* is in an 8-bit format, and *carry_o*ut is a standard logic output.

```
entity ALU_8051 is
port(op_code   : in  UNSIGNED (3 downto 0);
     source_a  : in  UNSIGNED (7 downto 0);
     source_b  : in  UNSIGNED (7 downto 0);
     carry_in  : in  STD_LOGIC;
     result_s  : out UNSIGNED (7 downto 0);
     carry_out : in  STD_LOGIC;
end ALU_8051;
```

Figure 7-60. The ALU interface described in VHDL

```
architecture of ALU_8051 is
begin
   case op_code is
    when CLR => result_s <= "00000000";
    when CPL  => result_s(7 downto 0) <= not source_a(7 downto 0);

    when ADD => DO_ADD(source_a,source_b, carry_in, result_s,carry_out);

    when SUBB => DO_SUBB(source_a,source_b, carry_in, result_s,carry_out);

    when INC  => DO_ADD(source_a, "00000001",'0' , result_s,carry_out);

    when DEC  => DO_SUBB(source_a, "00000001" ,'0' , result_s,carry_out);

    when XRL =>  result_s(7 downto 0) <=
                  source_a(7 downto 0) xor source_b(7 downto 0);

    when ANL  => result_s(7 downto 0) <=
                    source_a(7 downto 0) and source_b(7 downto 0);
    when ORL =>  result_s(7 downto 0) <=
                  source_a(7 downto 0) or source_b(7 downto 0);

    when RL =>  result_s(0) <= source_a(7);
                result_s(7 downto 1) <= source_a(6 downto 0);

    when RR =>  result_s(7) <= source_a(0);
                result_s(6 downto 0) <= source_a(7 downto 1);

    when others => carry_out <= '-';
   end case;
end process;
```

Figure 7-61. The ALU circuit described in VHDL

The ALU circuit code is given in figure 7-61. For each value of the operation code, a specific circuit is constructed which links the inputs *source_a* and *source_b* to the output *result_s*. The physical connection is created by the assignment "<=". The logical operators XOR, AND, OR and NOT are basic keywords of the VHDL language. The addition is described through a procedure called DO_ADD, the substraction through DO_SUB. The complete code used to describe the ADD and SUBB operation is quite complex, due to the treatment of the carry. Part of that code is given in figure 7-62. We notice the use of arithmetic operators such as "+" for addition (And "-" for substraction) and the formatting of the data to remove the most significant bit and to take into account the input *carry_in*.

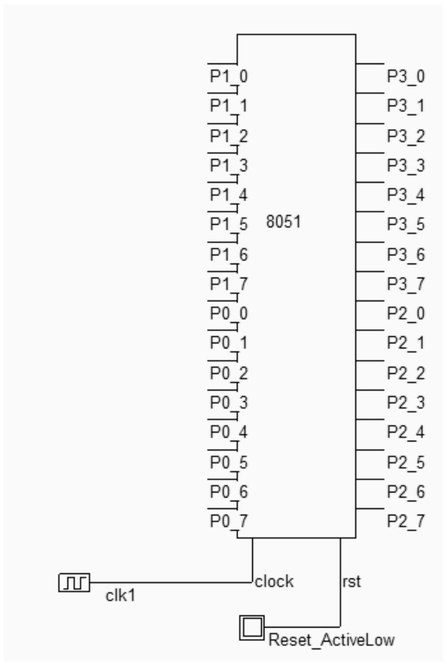
```
Result_1 := "0" & source_a (6 downto 0);
Result_2 := "0" & source_b (6 downto 0);
Result_3 := "0000000" & carry_in;
Result_s := Result_1+ Result_2+ Result_3;
```

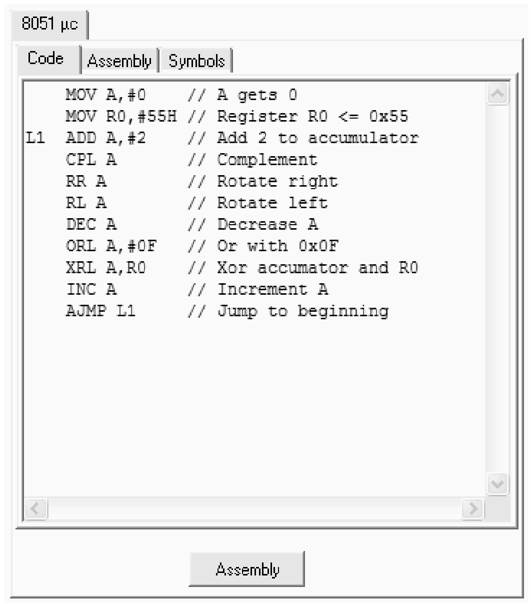
Figure 7-62. The basic instruction included into the DO_ADD procedure

Several synthesis tools are able to convert the VHDL architectural description into gate-level design, which can then be translated easily into layout. Neither DSCH nor Microwind can support such a description. The ALU may be designed using basic cells such as AND, XOR, Adder, Multiplexor using DSCH, and then be converted into layout.

The ALU in DSCH2

A simplified model of the 8-bit micro-controller 8051 exists in DSCH2. You can add the corresponding symbol (8051.SYM) using the command **Insert → User Symbol**, as the symbol is not directly accessible through the symbol palette. The symbol consists mainly of general purpose input/output ports (*P0,P1,P2* and *P3*), a *clock* and a *reset* control signals. The basic connection consists of a clock on the *Clock* input and a button on the *Reset* input (Figure 7-63).





```
    MOV A,#0     // A gets 0
    MOV R0,#55H // Register R0 <= 0x55
L1  ADD A,#2     // Add 2 to accumulator
    CPL A        // Complement
    RR A         // Rotate right
    RL A         // Rotate left
    DEC A        // Decrease A
    ORL A,#0F    // Or with 0x0F
    XRL A,R0     // Xor accumator and R0
    INC A        // Increment A
    AJMP L1      // Jump to beginning
```

Figure 7-63. The 8051 symbol and its embedded software (8051.SCH)

After a double-click in the symbol, the embedded code appears. That code may be edited and modified (Figure 7-63). When the button **Assembly** is pressed, the assembly text is translated into executable binary format. Once the logic simulation is running, the code is executed as soon as the reset input is deactivated. The value of the program counter, the accumulator A, the current *op_code* and the registers is displayed. In the chronograms, the accumulator variations versus the time are displayed. It can be noticed that this core operates with one single clock cycle per instruction, except for some instructions such as MOV (Move data) and AJMP (Jump to a specific address).

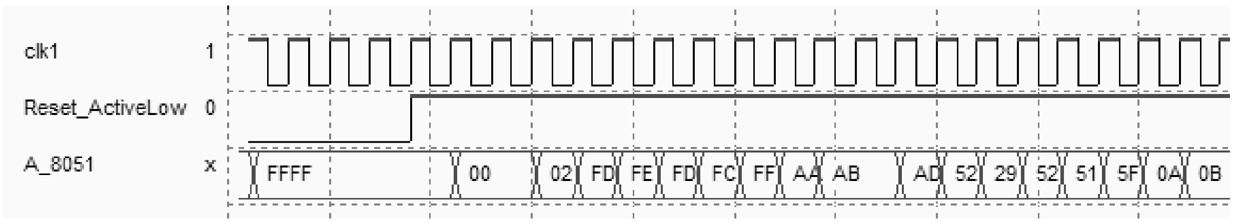


Figure 7-64. The simulation of the arithmetic and logic operation using the 8051 micro-controller (8051.SCH)

12. Conclusion

In this chapter, the design of basic arithmetic gates has been presented. The half adder and full adder have been presented. The full adder design has been conducted at layout level, with emphasis on advantages of manual design against automatic design regarding the silicon area efficiency. The comparator and multiplier circuits have also been described. A student project concerning the addition in binary-to-decimal format has been described in details, as well as an illustration of an arithmetic and logic unit used in the 8051 micro-controller.

Exercises

7.1 What is the 16 bit data equivalent to -3? What is the 32 bit data equivalent to 10^6?

7.2 <Etienne: alternative circuit for full-adder>

7.3 Design a Half Adder using the AOI technique described in chapter 6.

7.4 Implement the carry look ahead adder of figure 7-32, using the CMOS cell compiler of Microwind. Do you confirm the switching speed gains forecast by logic simulation?

7.5 Compile into layout the binary-decimal-coded adder described in section 10. What is the average power consumption per MHz? From the analysis of the logic circuitry, which part could be redesigned in order to reduce the dynamic power consumption?

7.6 Are there alternative circuits to reduce the standby current of the binary-decimal-coded adder described in section 10?

7.7 Dsch2 also includes the model of the PIC16f84 micro-controller [Pic]. An example file can be found in **16f84adder.SCH**. Double click the 16f84 symbol, and click **Assembly** to convert the text lines into binary executable code.

```
; Simple program to add two numbers
;
oper1 EQU 0x0c
oper2 EQU 0x0d
result EQU 0x0e

 org 0

 movlw  5
 movwf  oper1
 movlw  2
 movwf  oper2
 movf   oper1,0
 addwf  oper2,0
 movwf  result
 sleep
```

Then click **OK**, run the simulation. Click the *Reset* button to activate the processor. The default code realizes the addition of two numbers (Instruction `addwf`) and stores the result in the internal registers. Modify the code to perform the AND (Instruction `andwf`), OR (Instruction `iorwf`) , XOR (Instruction `xorwf`) and SUB (Instruction `subwf`) operations.

References

[Uyemura] John P. Uyemura "Introduction to VLSI Circuits & Systems", Wiley, 2002.

[Belaouar] <tbd>

[Weste] Chapter 8

[Borland] www.borland.com

[John] Michael John, Sebastien Smith, "Application Specific Integrate Circuits", Addison-Wesley, 1997, ISBN 0-201-50022-1

[Pic] <Add ref>

[8051] <Add ref>