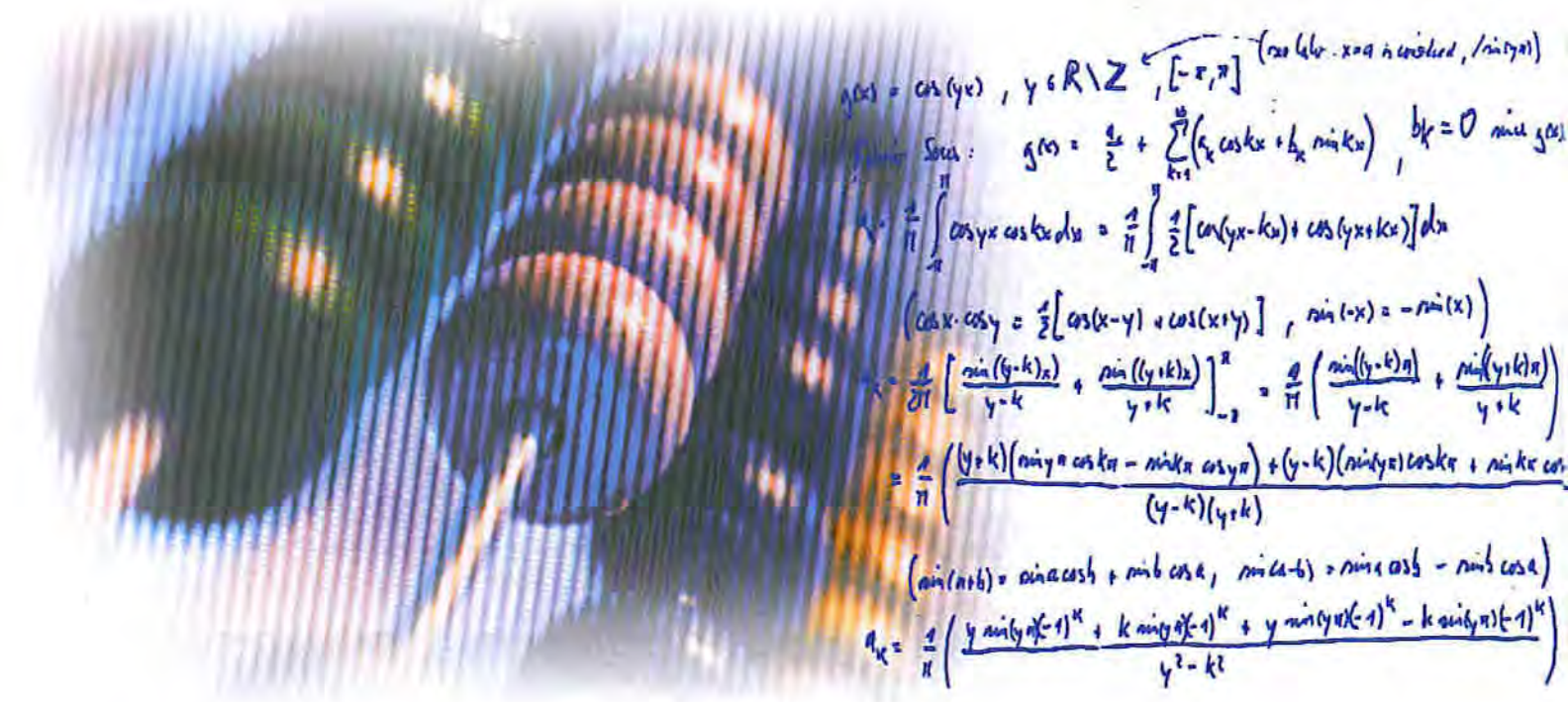
FPGA Course (2)

Paul Goossens



In the first article of this series, we described the basic components of digital electronics and put them to use. In this second instalment, we introduce some components that are a bit more complicated and perform a few simple calculations using digital logic.

After reading the first part of this series, you should know enough about the basic components of digital electronics. In this instalment, we use them to do things that are a bit more useful.

Memory

Let's start off by looking at the most commonly used type of memory element: the flip-flop.

The simplest type of flip-flop is the 'set–reset' (SR) flip-flop. It has two inputs (Set and Reset) and one or two outputs (Q and $\overline{Q}$). When the Set input goes to '1', the Q output also goes to '1'. That state remains unchanged even if the Set input returns to '0'. When the Reset input goes to '1', the Q output will go to '0'. That state also remains stable after the Reset input returns to '0'. The response of the flip-flop is undefined if the Set and Reset inputs are both '1'. That is regarded as a forbidden state that must never occur.

Figure 1 shows a schematic diagram for this type of flip-flop. It is constructed from standard components. IC1 and IC2 are NAND gates (a NAND gate is an AND gate with an inverter at its output, and an inverted output is marked by a small circle or diagonal line at the output).

Truth table

How can this be translated into a truth table? The answer is shown in **Figure 2**. The SR flip-flop is shown at the far left. The associated truth table shows output Q_{N+1} instead of output Q. That indicates that this column shows the state of the output after the input signals have been processed. In some cases, the output state also depends on the previous state of the output. That is indicated here by Q_N.

Advanced forms

A slightly more advanced form of flip-flop is the type known as a 'latch'. Its truth table is shown in **Figure 2**. This type of flip-flop has two inputs – D ('data') and Gate – and one output (Q). The Q output is the same as the D input as long as the Gate input is '1'. When the Gate input goes to '0', the Q

Part 2: Memories and calculations

output retains its value regardless of the state of the D input. It effectively stores the state of the D input at the time when the Gate signal became '0'.
The next step brings us to the D-type flip-flop. The output of this type of flip-flop assumes the state of the input when the signal on the CLK input changes from '0' to '1'. That is indicated in the truth table by an arrow. The output remains unchanged as long as the CLK input stays at '1' or '0'.
The D-type flip-flop can be expanded with a variety of additional inputs. **Figure 2** shows such an expanded D-type flip-flop. It has three inputs in addition to the D and CLK inputs: SET, RESET and CE. The SET and RESET inputs perform the same functions as with an SR flip-flop. The clock enable (CE) input of this flip-flop controls the response to the clock signal: a rising edge on the CLK input has no effect if the CE input is not in the High state.

Trying it out

The **ex5** folder (included in the downloads for the second instalment on the *Elektor Electronics* website under Magazine/May) contains an example with various types of flip-flops. The RS flip-flop and the latch are implemented in the example in the form of logic gates. You can use the example to convince yourself that these functions can be constructed using ordinary gates. The other types of flip-flops are taken from the Quartus library.
You can use the pushbuttons to experimentally test the operation of the various types of flip-flops.

VHDL

Things really start to get interesting when you use VHDL for flip-flops. The nice thing about VHDL is that you can describe a design instead of building it with small logic elements. The program uses the description to design logic that does exactly what your description says.
Before we get into the details of the design, you need to know how the VHDL compiler reads your description. A VHDL file describes how the outputs (and the internal signals, if any) have to respond to the inputs. For this purpose, the VHDL compiler applies every conceivable combination of input signals (in virtual form) to the inputs of your design. For each change to the input signals, the compiler attempts to determine how the outputs must respond.
That all sounds a bit abstract, but the following example should help clarify what it means.

Structure

The structure of a simple VHDL file is shown in **Figure 3**. The first thing you have to do is make the standard library 'visible' to the compiler. Several basic functions for digital logic are defined in the standard library.
After that, you must declare at least one entity. You can think of an entity as something like a particular type of IC. In the entity declaration, you give the entity a name (a 'type number') and define the inputs and outputs of your 'virtual IC'.
After that comes the architecture section, which describes how the entity functions.

An example

We can use an example to show exactly how all this works. The project for this example is located in the **ex6** folder. Double-click on the block named Latch_VHDL. That will open the VHDL file that describes how this particular bit of logic operates.
The declaration of an entity named *Latch_VHDL* starts on line 29. The inputs CLK and DATA and an output named Q are declared here. These signals are all of type *std_logic*. That data type indicates that they are digital signals. We'll describe some other types later on.

The description of how the *Latch_VHDL* entity has to respond to its input signals starts on line 44.

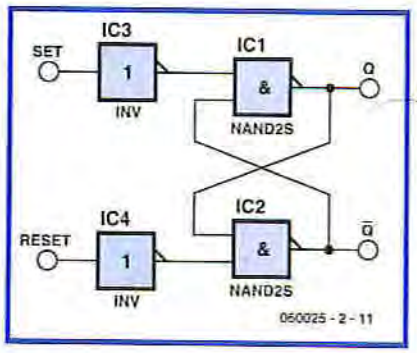


Figure 1. Basic configuration of a flip-flop using four NAND gates.

Processes

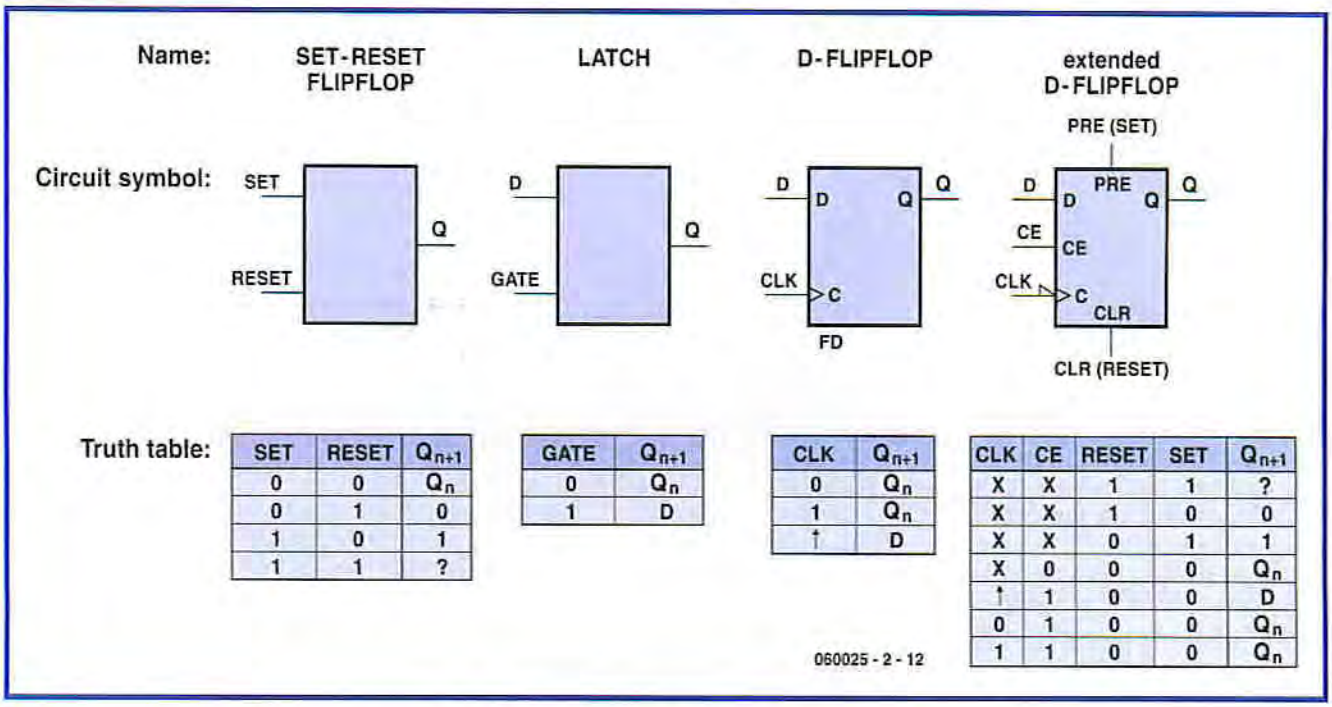
You have already seen that Boolean equations can be used to describe functions. An even more powerful approach is to use processes. In a process, you can specify the value(s) that one or more signals must assume under various circumstances.
The *process* keyword is followed by a sensitivity list. Each time the compiler changes the (virtual) value of any of the signal in this list, it must evaluate the code segment of the process. We'll explain this a bit later on.

If then else

The keyword *if* appears on line 51. It will doubtless be familiar to the programmers among our readers. This line says that if the signal on the CLK input is '1', the compiler must evaluate the code until it encounters an *end if* statement.
In this case there is only line in between, and it contains the statement *Q<=DATA;*. The whole process is terminated by an *end process* statement, and the end of the description is declared in line 56.

Evaluation

When the compiler evaluates the code segment, it discovers that the Q output must be the same as the DATA input as long as CLK is '1'. Nothing must happen when CLK is not '1', which



SET	RESET	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	?

GATE	Q_{n+1}
0	Q_n
1	D

CLK	Q_{n+1}
0	Q_n
1	Q_n
↑	D

CLK	CE	RESET	SET	Q_{n+1}
X	X	1	1	?
X	X	1	0	0
X	X	0	1	1
X	0	0	0	Q_n
↑	1	0	0	D
0	1	0	0	Q_n
1	1	0	0	Q_n

Figure 2. Various types of flip-flops and their truth tables.

means Q must not change. That shows how you can design a latch in VHDL.

D-type flip-flop

Now open the file D_ff_VHDL in the same way as before. Here you will see a similar file with a few crucial differences.

A new construction for the *if* statement appears on line 51. The construction *clk'event* is only true when the CLK signal changes. The construction *clk'event and clk='1'* is thus only true when a positive edge (low-to-high transition) is present at the CLK input. The Q output will only assume the value of the DATA input in that situation. In all other situations, Q will remains the same. This is thus a description of a D-type flip-flop.

Another D-type flip-flop

The second example of a D-type flip-flop has two additional inputs: SET and RESET. In the accompanying VHDL code, you can see that a test is first made to see whether *reset* is '1'. If it is, the output is set to '0'. Otherwise the state of the *set* input is examined. If it is '1', the output will go high.

If the set and reset inputs are both not '1', a test is made to see whether the *clk* signal exhibits a rising edge (just as in the previous example for a D-type flip-flop).

If you refer back at the start of the process, you will see that the signals *clk*, *set* and *reset* appear in the sensitivity list. Output Q can change if any one of these signals changes state. The *set* and *reset* inputs act asynchronously to the *clk* input. In other words, the device does not require a rising edge on the *clk* input to respond to a *set* or *reset* command.

You should also note that *reset* has a higher priority than *set* for this flip-flop. If *set* and *reset* are both '1' at the same time, *reset* will win the contest and the output will go to '0'.

Arithmetic

The examples up to now have used signals of type *std_logic*. An extension

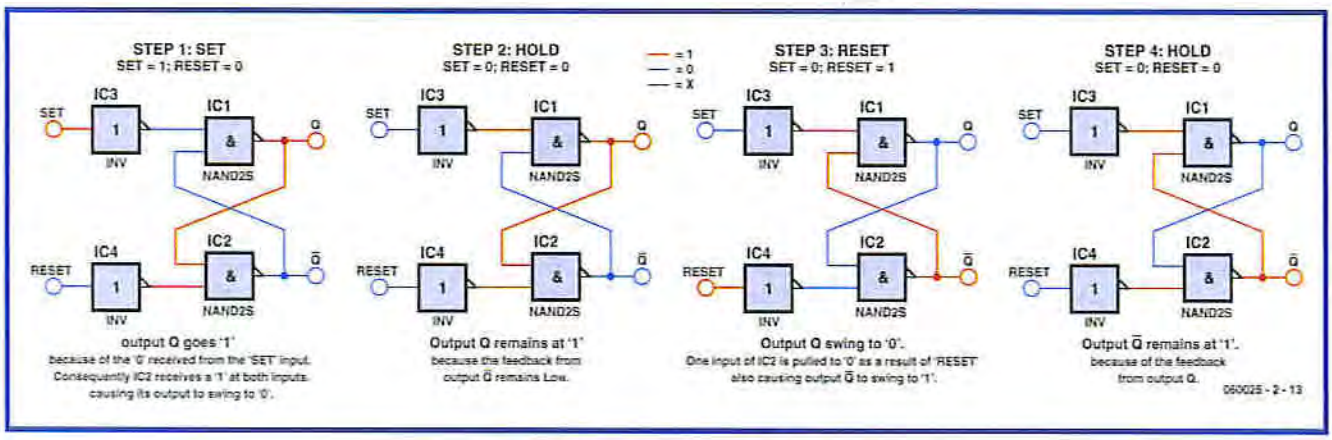


Figure 3. The various states of a flip-flop.

of that type is *std_logic_vector*, which we abbreviate as S_L_V for the remainder of this article. That type consists of a set of signals of type *std_logic*. You can use such a set of signals to represent a number (see inset).

There is also a type known as *natural*. It encompasses all positive whole numbers (integers). Doing arithmetic with signals of type *natural* is quite easy. You can use them for addition, subtraction, multiplication and division in VHDL.

That capability is used in **ex7** to create a pulse waveform with a frequency of 1 kHz, derived from a 50-MHz clock.

Open the example and double-click on the block named *pulse_generator*. In the associated VHDL code, you can see how a signal of type *natural* is used for counting. First the ports are defined: a input signal named *clk* and an output signal named *slow_clk*. In the associated VHDL code, you can see how a signal of type *natural* is used for counting. This signal must be able to hold the range of values from 0 to 500,000 inclusive. The VHDL code uses these numbers to determine how many bits are required.

In the associated **procedure**, a test is made on each rising edge of *clk* to determine whether the value of the *counter* signal has reached the maximum value (499,999). If it has, the new value is set to '0' and the *slow_clk* output is set to '1'. In all other cases, the value of *counter* is incremented by 1 and the *slow_clk* output is set to '0'.

The net result is that the output goes to '1' after 500,000 clock pulses. On the next clock pulse, it returns to '0' and the cycle starts again from the beginning. If a 50-MHz clock signal is applied to the *clk* input, the output will briefly go to '1' a thousand times per second.

The VHDL code for *calculate_sum* demonstrates something else that's new. First, line 25 shows that an additional library is necessary – the numeric_std library. A variety of arithmetic operations and conversions are defined in that library.

The input signal a is declared in line 34. The expression *STD_LOGIC_VECTOR (3 downto 0)* says that this signal set consists of four signals: *a(3)*, *a(2)*, *a(1)* and *a(0)*. You already know that a signal set can be used to represent numbers. Making calculations with *S_L_V* is a bit more roundabout. The functions of addition, subtraction and so on are not defined for type *S_L_V* in

Binary arithmetic

Numbers can be represented using one or more digital signals. As binary signals can have only two states (1 or 0), the binary number system must be used in such cases. In normal life, we use the decimal number system (base 10). In the decimal system, a set of three numerals can be used to represent 10^3 (1000) different numbers (0–999).

In the binary system, a set of three digits (signals) can represent a total of 2^3 ($2 \times 2 \times 2 = 8$) values ranging from '000' to '111', or 0 to 7 in decimal notation.

Sample calculation

The number '821' in decimal notation consists of $8 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$. Similarly, the number '101' in binary notation consists of $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1 \times 4 + 1 \times 1 = 5$ in decimal notation.

You may find the following table useful for converting between binary and decimal numbers.

$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$	Decimal	Hexadecimal
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	9
1	0	1	0	10	A
1	0	1	1	11	B
1	1	0	0	12	C
1	1	0	1	13	D
1	1	1	0	14	E
1	1	1	1	15	F

Hexadecimal notation

Relatively large binary numbers are generally difficult to comprehend due to the large number of ones and zeros. Hexadecimal notation (base-16 number system) can be used to make them easier to understand. That notation uses the numerals 0–9 and the letters A–F, with A representing the decimal value 10, B the decimal value 11, and so on. See also our Hexadoku puzzle! A single character can thus be used to represent 16 different values. That corresponds to four bits in the binary system.

Active High and Active Low

Many components have 'active Low' inputs. That means the input is 'active' when the logic level at the input is Low.

For instance, a flip-flop with an active-low Reset will be reset when a logic Low signal is applied to the Reset input. Active-low inputs can be recognised by the small circle or triangle at the input concerned. An 'inversion bar' can also be placed above the name of the input to indicate that it is active low, such as $\overline{\text{RESET}}$

It's also possible for outputs to be active Low. Such outputs can similarly be recognised by the inversion bar over the name or a small circle or slanted line at the output.

VHDL. However, they are defined in the numeric_std library for some other types, including *unsigned*. Several useful conversion routines are also located in that library.
The expression *UNSIGNED (a)* converts the set of signals *a(3)...a(0)* to type *unsigned*.
A set of signals of type *unsigned* can be converted to type *S_L_V* in the same manner.
You can see all that on line 49. There the two input signals *a* and *b* (each of which is a signal set) are converted to type *unsigned*. The two values are added together, which yields a result of type *unsigned*. Finally, that result is converted to type *S_L_V*. These signals are also linked to the *SUM* output. This thus amounts to a simple addition function.
A subtraction function is described in the same manner in *calculate_dif*.

Busses

The two blocks are interconnected in the Quartus schematic by several signal lines originating from dipswitch S5. The input port, *DIPSWITCH[3..0]*, consists of four independent input lines with the designations *DIPSWITCH[3]*, *DIPSWITCH[2]*, etc. This deviates from the VHDL standard with regard to the notation for a set of signals.
This set of signal lines can also be connected using a bus instead of individual signal lines. You can draw a bus in Quartus by using the Bus Tool instead of the Node Tool. A bus is shown in a schematic diagram with a thicker line than individual signal lines.
The schematic diagram here has several busses, some of which are connected to a port at only one end. Quartus regards all signals with the same name as being linked together, so it isn't necessary to connect all the associated lines together.

Multiplexing

The 7-segment displays must be driven one at a time. Only one segment can be on at any given time. This sequential drive is provided by the *sequencer* block.
In the *count* process in sequencer.vhd, you can see that the *internal_select* signal is incremented by 1 each time each time a rising edge occurs on the *clk* line when the *clk_en* input is '1' . In the schematic, *clk_en* is connected to a signal line that goes to '1' once every 500,000 clock pulses. As a result, the *internal_select* signal is incremented 100 times per second (50 MHz ÷ 500,000).
Line 66 shows a new feature of VHDL. It says that the *sel1* signal must go to '1' if the counter value is '0', and otherwise the *sel1* output must be '0'. Signals *sel2* to *sel4* are generated in the same way. As a result, these outputs go to '1' sequentially. The counter value is also output via the *sel* signal (line 72).
Signals *sel1–sel4* drive transistors T1–T4 on the prototyping board to enable the various groups of LED segments.

In step

This example is designed to display four different numbers. The numbers must be output sequentially, synchronised with the drive signals for transistors T1–T4. The *sel[1..0]* signals from the sequencer can be used for this purpose.
The *mux* block takes care of all this. The *current* output is enabled by the four inputs (*val1*, *val2*, *sum* and *dif*) according to the value of the *sel* signal. The keyword *when* appears in line 52 of the associated VHDL code. In that line, the *current* output is enabled by the *val1* input if the value of *sel* is '00' (0). Similarly, the output is enabled by *val2* if *sel* is equal to '01' (1), and so on. Once again, the final enable is provided here by the *else* keyword.

Decoding

The *current* output contains the value to be shown on the display. This value is also nicely synchronised with the drive signals for the individual groups of LED segments.
The value of *current* is available in binary form, so the individual LED segments must be driven based on this binary code. The value '1' must be shown on a 7-segment display by driving segments b and c. To display the value '0', segments a–f must be driven. That means the binary values must be converted into the proper drive signals for the various segments.
This last bit of processing is handled by the block named *to_seven_segment*. In the VHDL code for that block, you will see another new keyword: *case*. The line *CASE val IS* says that the value of *val* determines how the following bit of code is to be processed. You could read line 50 as 'if *val* has a value of '0000', then the following must happen'. The software will then evaluate the subsequent lines of code until it encounters the next *when* keyword.
In line 51, you can see that this causes the value '1111110' to be assigned to the *segments_out* output. In other words, segments a–f are driven on and segment g is not driven. The numerals 1–9 and the digits A–F (for hexadeci-

Earlier in this series

Versatile FPGA Module, *Elektor Electronics* March 2006.

FPGA Prototyping Board, *Elektor Electronics* March 2006.

FPGA Course (1), *Elektor Electronics* April 2006 (with free downloads).

mal display) are decoded in a similar manner.
Finally, the end of the *case* statement is indicated in line 84.

Testing

The best way to understand the previous explanation is to try it out in practice. To do so, program the FPGA unit with the example program in **ex7.** If everything goes right, you will see several numbers appear on the 7-segment displays.

You can use switches 0–3 of S5 to assign a value to *val1*. It will be shown on the display at the far left. You can similarly use switches 4–7 to assign a value to the second number.

The third number on the display shows the result of adding the two input values, and the last number shows the difference between the two values.

Experimenting

To familiarise yourself with this approach to designing, try making a few changes to the code of the example. For instance, you can change the order of the numbers, or instead of calculating the difference of two numbers, you can calculate and display the product of the two values (multiply = * in VHDL).
The **ex8** folder contains several other folders with even more examples. Now that you've read this article, you should be able to figure out how these examples work. Try out each of them in turn, and study the associated code to discover how they work. That's the only way to become fluent in VHDL!

(060025-2)

Digital arithmetic in practice

All digital devices calculate using the binary number system, even if their users are not aware of the fact. The arithmetic processes are often considerably more complex than simple addition.

Your personal calculator is a good example of what can be done with digital arithmetic. Besides normal arithmetic operations, it can also be used for relatively complex calculations such as sine, cosine, square root, and so on.

An example of a more practical application of this arithmetic capability is a digital control system for a rocket. A considerable amount of real-time calculation is necessary to accurately send a rocket into space along its intended path.

Yet another example is your DVD player. It takes a lot of mathematical calculations to transform the compressed data on a DVD into a nice picture on your screen. All those calculations are performed by a processor. Naturally, that's a digital processor, and it calculates (lightning fast) using only binary numbers.