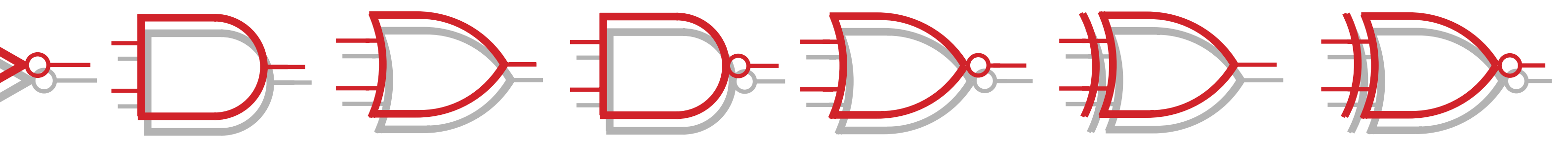
FGPA Course (1) Part 1:



This is the first instalment of an FPGA course based on the versatile FPGA unit and associated prototyping board described in the last issue. This series of articles describes how to design digital circuits and program an FPGA. In this article, which is the first instalment of the series, we discuss the basic components of digital electronics.

Paul Goossens

Digital electronics is based on circuits that operate using only two voltage levels. These levels are called ‘high’ and ‘low’. ‘High’ means that the signal potential (voltage) is higher than a previously defined value, and ‘low’ means that the signal level is lower than another previously defined value. The symbols ‘1’ and ‘0’ are also frequently used in place of ‘high’ and ‘low’. A ‘1’ corresponds to a high level, while a ‘0’ corresponds to a low level.
These digital signals can be processed by components that in turn generate digital signals.

AND gate

Digital electronics is entirely based on just three elementary components. Even highly complicated digital circuits are composed of just these three types of components.

These three elementary components (functions) are illustrated in **Figure 1**. The first two, AND and OR, can have any desired number of inputs, while the third one (INVERTER) always has only one input. Each of these components has a single output.

Let’s start with the AND function. The output of this component will be ‘1’ only if all its inputs are ‘1’. Another way to explain how it operates is to say that the output is ‘0’ if at least one input is ‘0’. That amounts to exactly the same thing, which can be clearly seen from Figure 1.

Notation

The name of each function is shown at the top in Figure 1. The next row shows the Boolean operator. Boolean notation can be used to describe digital circuitry in a mathematical manner. You will be seeing a lot of Boolean notation in this series of articles. It’s actually a relatively simple notation, as you can see from the examples of Boolean equations in the next row.
The associated truth tables are shown below the equations. They are a handy way to describe the behaviour of a digital circuit in an easily understandable manner. The input levels are shown in the left-hand column, and the associated output levels are shown in the right-hand column.
If you look at the AND gate, for example, you can see right away that the output is ‘1’ only if all the inputs are ‘1’.
It is often possible to reduce the size of a truth table and make it easier to interpret. The ‘X’ symbol is used for that purpose. It stands for ‘doesn’t matter’ or ‘don’t care’. If you look at the truth table at the bottom of the figure, you can see that the states of inputs of A and B in the first row don’t matter if input C is ‘0’, because the output is always ‘0’ in that case.
As a final remark, we’d like to point out that although the AND and OR gates in Figure 1 are shown with three inputs, the number of inputs can range from 2 to any desired number in actual practice.

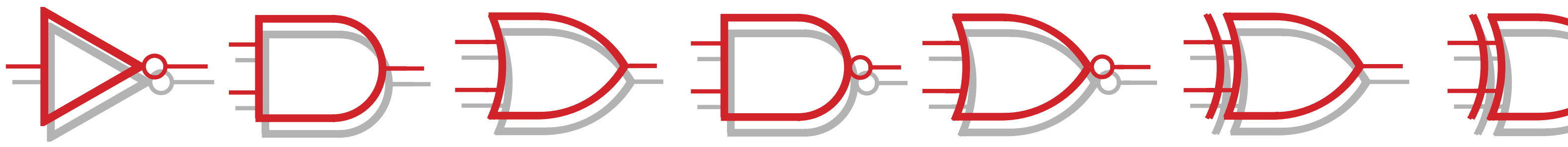
OR gate and inverter

The output of an OR gate is ‘1’ if at least one of its inputs is in the ‘1’ state. In other words, the output of an OR gate is ‘0’ only if all its inputs are in the ‘0’ state.
The final type of gate is the inverter or NOT gate. The output of this gate is the reverse of the signal at its input. In other words, a ‘0’ on the input causes a ‘1’ at the output, and vice versa. That is also shown in Figure 1.

Prototyping board

Now let’s try these gates out on the prototyping board.
Figure 2 shows the schematic diagram of a simple circuit. We’re going to implement it in the FPGA. We’ll use the signals from the four pushbuttons on the prototyping board as the inputs, and we’ll use the seven LEDs on the prototyping board to visibly indicate the states of the various outputs.
The files you need for this example are included in the software for this article, which can be downloaded free of charge from www.elektor-electronics.co.uk, follow Magazine _ April 2006 _ FPGA Course (1). The software can be installed conveniently using the installer program (Setup.exe) included in the download.
In the rest of this article, we assume you have installed the software in the default folder (C:\altera\FPGA_course\1\ex1). We also assume you have installed the FPGA unit and the prototyping board according to the instructions in the associated articles.
Double-click on file ‘ex1.qpf’ in the C:\altera\FPGA_course\ex1 folder. That will launch the Quartus program, which in turn will open the project belonging to the example. You will see a schematic diagram on your monitor that

The basic components of digital electronics



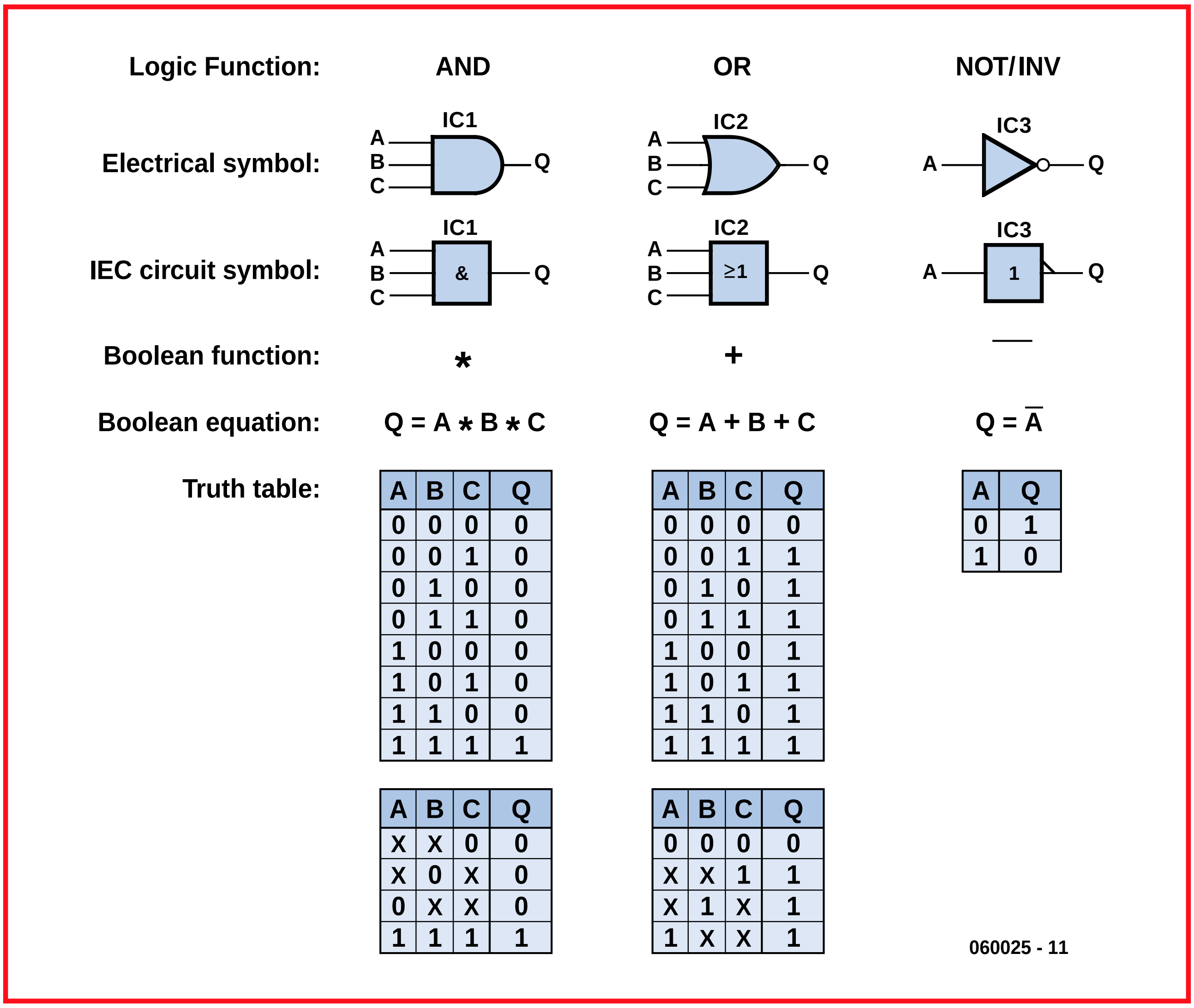
matches the one shown in Figure 2. Signals SWITCH1–SWITCH4 are connected to input pins. The software connects them to the associated FPGA contacts on FPGA unit. Naturally, the same holds true for the LED1–LED7 lines.

The Quartus program can use the schematic diagram to create a programming file that in turn can be used to configure this circuit **in** the FPGA.

To make this happen, select ‘Start compilation’ in the ‘Processing’ menu. It may take a while to create the file, but you will ultimately see a message stating that compilation was completed successfully.
Follow the instructions in the article on the FPGA unit to program the FPGA using this file. The only difference here is that you use file ‘ex1.sof’ as the programming file.

Trying it out

After you have programmed the FPGA, your circuit is ready for use. Now you can use the four pushbuttons to drive inputs SWITCH1–SWITCH4. If you press a pushbutton, the corresponding signal will be forced to a high (‘1’) level. When the pushbuttons are not pressed, pull-down resistors (R5–R8) cause a low (‘0’) level to be applied to



Logic Function:	AND	OR	NOT/INV
Boolean function:	*	+	—
Boolean equation:	Q = A * B * C	Q = A + B + C	Q = $\overline{A}$

Truth table:

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

A	B	C	Q
X	X	0	0
X	0	X	0
0	X	X	0
1	1	1	1

A	B	C	Q
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

A	B	C	Q
0	0	0	0
X	X	1	1
X	1	X	1
1	X	X	1

A	Q
0	1
1	0

Figure 1. The basic components on parade.

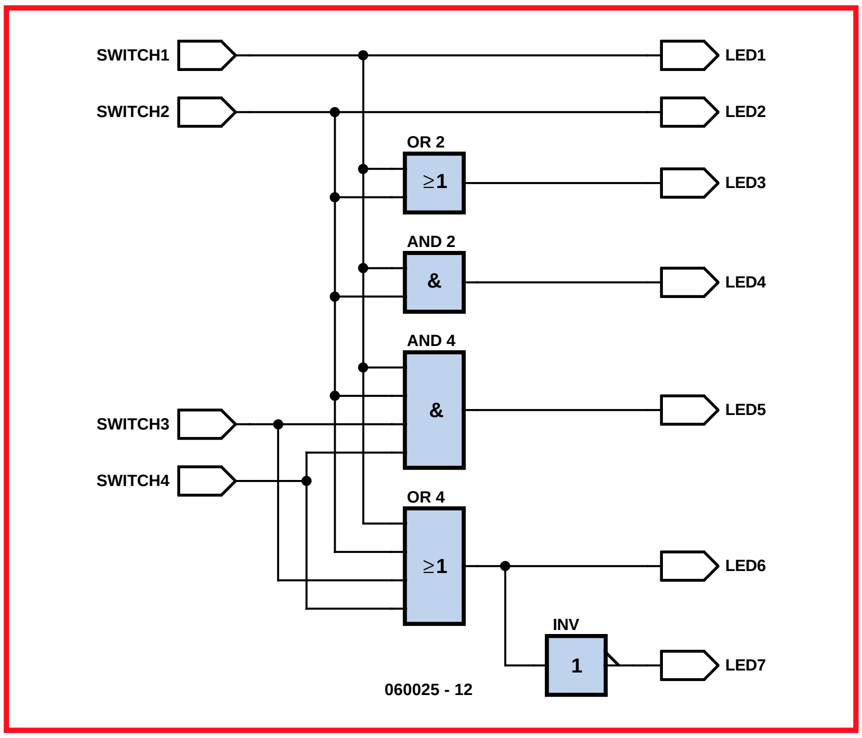


Figure 2. The schematic diagram of our first example.

the associated inputs.
Outputs LED1–LED7 drive LEDs D8–D14 via a buffer (IC7). The LEDs are shown as '1' to '7' on the schematic diagram.

From the schematic diagram in Quartus, you can see that LED1 is connected directly to the SWITCH1 input. When the SWITCH1 signal goes to '1', outputs LED1 will also change immediately to '1'. In practice, that means LED 1 will light up immediately when you press pushbutton 1. You can control LED 2 with pushbutton 2 in the same manner.
The situation is different with LED 3. It is connected to the output of an OR gate whose inputs are connected to SWITCH1 and SWITCH2. This LED will thus light up if pushbutton 1 or pushbutton 2 is pressed (or if both of them are pressed).
You can use LED 4 to study the operation of an AND gate. It will only light up if pushbutton 1 and pushbutton 2 are pressed at the same time. Try it for yourself – you'll see that the LEDs light up according to the rules described above.
You can use LED 5 to test a four-input AND gate. According to the rules, LED 5 should only light up if you press all four pushbuttons at the same time.
LED 6 demonstrates an OR gate with four inputs. It will light up when any of the inputs (the four pushbuttons) is high (button pressed).
Finally, there is LED 7. It demonstrates the NOR function. LED 7 will be lit when LED 6 is dark, which means when none of the buttons is pressed.
To convince yourself that the circuit is working properly, try pressing the pushbuttons individually and in combination and observing whether the LEDs actually respond as described above.

LED1 = SWITCH1

LED2 = SWITCH2

LED3 = SWITCH1 + SWITCH2

LED4 = SWITCH1 * SWITCH2

LED5 = SWITCH1 * SWITCH2 * SWITCH3 * SWITCH4

LED6 = SWITCH1 + SWITCH2 + SWITCH3 + SWITCH4

LED7 = $\overline{\text{LED6}}$
OR :
LED7 = $\overline{\text{(SWITCH1 + SWITCH2 + SWITCH3 + SWITCH4)}}$

'A*B' = 'A AND B'

'A+B' = 'A OR B'

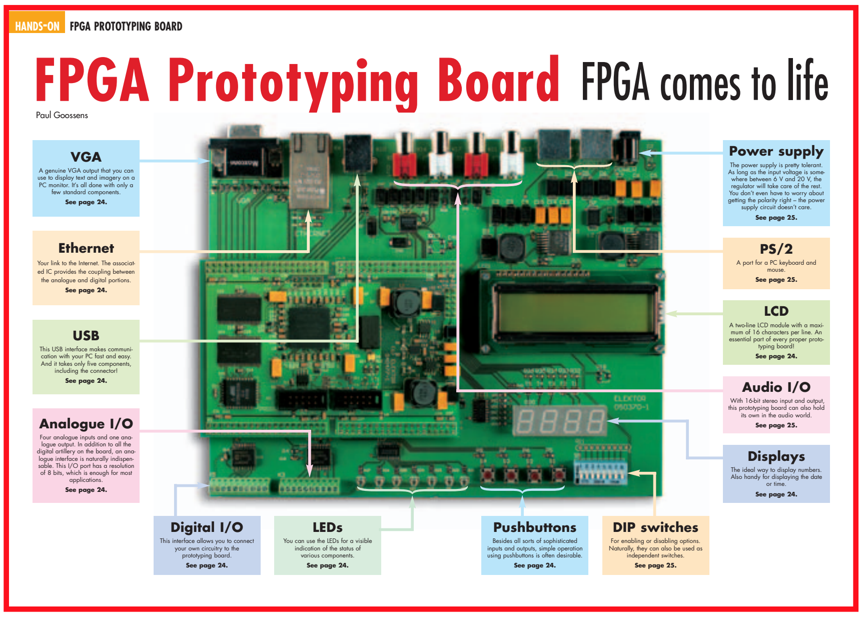
$\overline{\text{A}}$ = 'not(A)'

060025 - 13

Figure 3. Example 1 in Boolean notation.

Creating your own designs

The ultimate objective of this course is enable you to design your own digital circuits and program them in the FPGA. That means that addition to understanding how digital circuitry works, you need to know how to use the Quartus program.
The only way to learn how to use Quartus is to actually work with it. The best way to get started is to work through the tutorial for the program. There is also a PDF document in the C:\altera\FPGA_course\ex2 folder with step-by-step instructions for drawing the first example circuit. It is important that you read this document thoroughly and use it to draw the example circuit shown in Figure 2 for yourself. That way you will not only learn how to use the program, but also how to configure the software for use with the *Elektor Electronics* FPGA hardware and programming circuit.



Boolean algebra

The functions of the first example can also be described using Boolean algebra. **Figure 3** shows how the seven functions of the example circuit are described in Boolean notation. You can also use these equations to describe the circuit in Quartus.
That involves using a language called 'VHDL', which is intended to be used to describe the operation of digital circuits. The Quartus program can use such descriptions to create programming files. As far as the program is concerned, it doesn't matter whether you draw a circuit in graphic form or describe it using VHDL.

An example

The Quartus files for this example are located in C:\altera\FPGA_course\ex3. Simply click on 'x3.qpf'' to launch Quartus and load the project.
The difference from the previous example is immediately apparent. As you can see in ex3.bdf, which is the graphic representation of the circuit, the inputs and outputs of the design are connected to a large rectangle. This rectangle processes the input signals and generates the outputs.
Double-click on the rectangle to open the document that specifies how the rectangle operates. That can be another schematic diagram, but it can also be a bit of VHDL code. In this case, we defined the functions using VHDL.
In VHDL, everything after a '—' is regarded as a comment. Comments do not form part of the actual design, but they can be very handy for documenting the design. A detailed explanation of the structure of a VHDL file will have to wait until a later instalment – after all, this is just the first article in the series!
On line 29, we give the design a name (ex3_VHDL), which is followed by definitions of the inputs and outputs of the design. Here you can see that signals IN1–IN4 are inputs (**IN**). Signals OUT1–OUT7 are all outputs (**OUT**). That's all we want to say about the structure of this document for the time being.

Equations

However, lines 57–63 deserve further explanation. Here you see equations similar to the ones in Figure 3, with the difference that here we use '<=' ('becomes') instead of '=' as in Figure 3. You'll also notice that the '*' operator can be replaced by the word 'and', and that '+' can be replaced by 'or'.
In line 63, we wrote out the full equation instead of using the equation 'OUT7 <= not (OUT6);', although the latter equation is perfectly correct in theory. Unfortunately, VHDL does not allow output signals to be used as inputs for internal logic. Later on, we'll tell you why that is and show you a simple trick you can use to utilise output signals as inputs despite this restriction.
Once again, you can compile the example and try it out on the prototyping board.

Modifications

Folder 'ex4' contains a document that explains, step by step, how to draw the above example in Quartus. You can use it to familiarise yourself with drawing functional blocks and building up a design in a hierarchical manner. This document also describes some other features of Quartus, so you should make a point of working through the exercise.

VHDL is an extremely powerful language, and we've only touched on a few of its basic features here. Nevertheless, it's clear that designing circuits this way can save a lot of time. For example, suppose you decided that LED1 ('OUT1' in the VHDL file) should only light up when button 1 and button 4 are both pressed. That would require modifying the design. You could do so by replacing line 57 in the example with the following line:
OUT1 <= IN1 and IN4;
You can try that for yourself by recompiling the modified design file and programming it into the FPGA.
If we had generated the design in schematic form as for the first example, it would take more work to incorporate the modification into the design. What's more, the schematic diagram of a relatively large design can easily turn into a thicket of lines and logic gates, which makes it difficult to quickly understand how the design works. The operation of the design is much easier to understand if you describe it in VHDL, especially if you use comments.

Coming up next

As promised, we'll put the 7-segment displays to work in the next instalment. That will involve using memory elements and some rudimentary computations. In the meantime, you can visit the forum on our website to discuss FPGAs, our development kit, and of course the FPGA course.

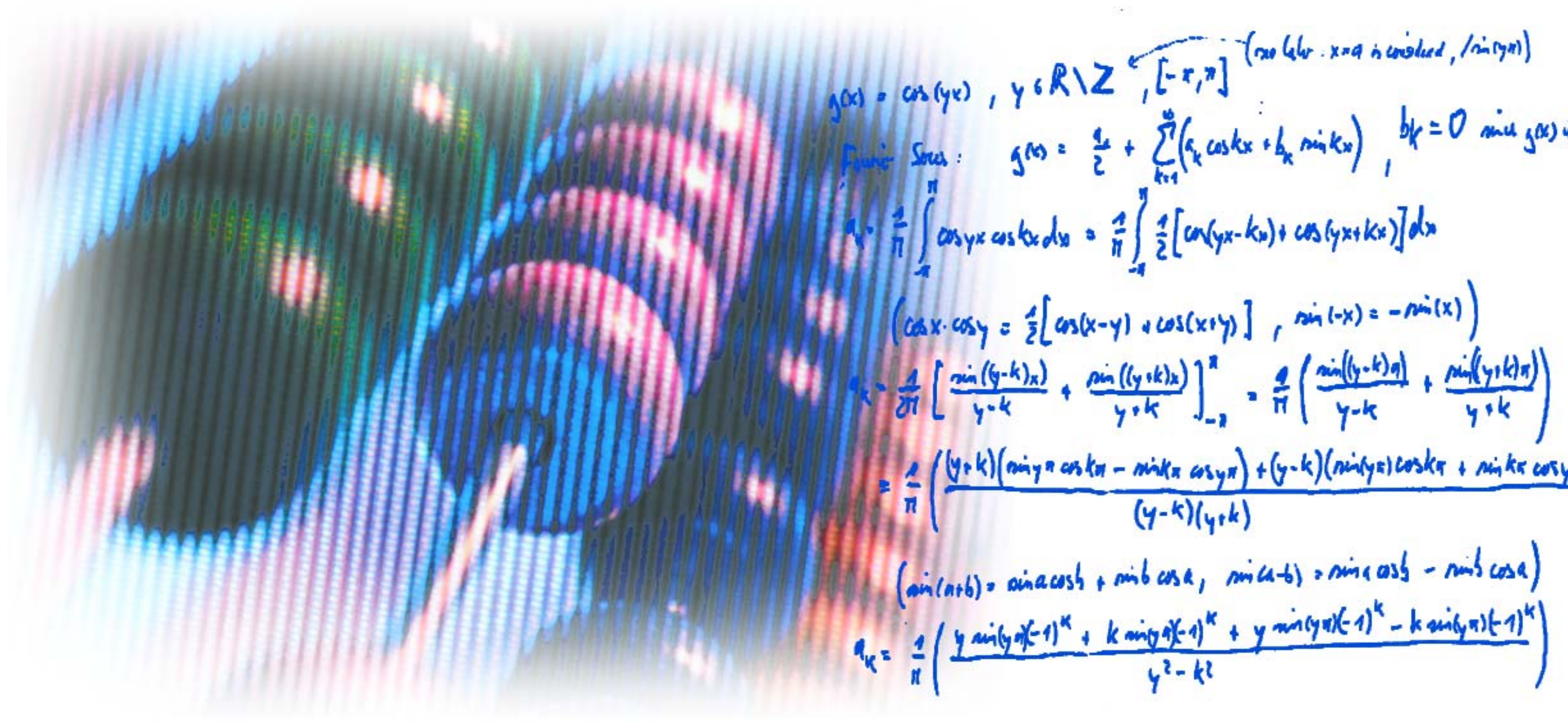
(060025-1)

The software for this instalment is available free of charge on the Elektor Electronics website under item number 060025-1-11.zip. From the homepage, go to Magazine → April 2006 → FPGA Course (1).

Earlier in this series
Versatile FPGA Module, *Elektor Electronics* March 2006.
FPGA Prototyping Board, *Elektor Electronics* March 2006.

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.

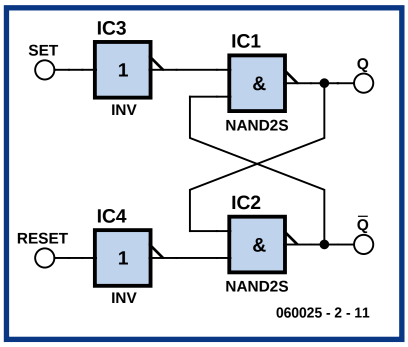


Figure 1. Basic configuration of a flip-flop using four NAND gates.

Processes

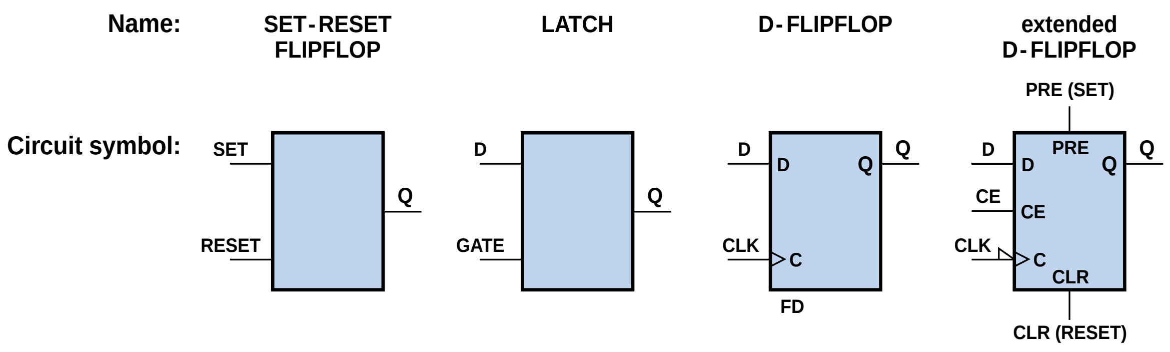
You have already seen that Boolean equations can be used to describe functions. An even more powerful approach is to use processes. In a process, you can specify the value(s) that one or more signals must assume under various circumstances.
The *process* keyword is followed by a sensitivity list. Each 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



Truth table:

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

060025 - 2 - 12

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

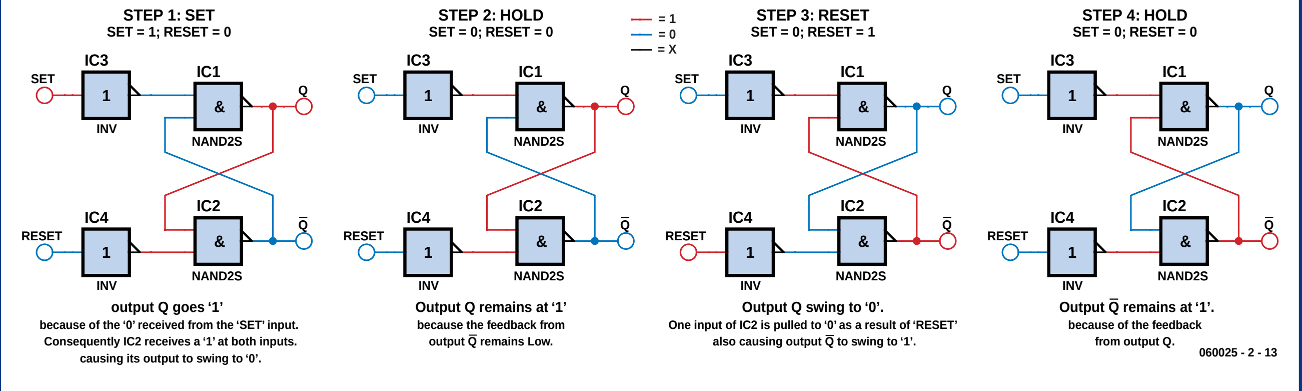

STEP 1: SET
SET = 1; RESET = 0
STEP 2: HOLD
SET = 0; RESET = 0
= 1
= 0
= X
STEP 3: RESET
SET = 0; RESET = 1
STEP 4: HOLD
SET = 0; RESET = 0
SET
IC3
1
INV
IC1
&
NAND2S
Q
RESET
IC4
1
INV
IC2
&
NAND2S
Q̄
output Q goes '1'
because of the '0' received from the 'SET' input. Consequently IC2 receives a '1' at both inputs. causing its output to swing to '0'.
Output Q remains at '1'
because the feedback from output Q̄ remains Low.
Output Q swing to '0'.
One input of IC2 is pulled to '0' as a result of 'RESET' also causing output Q̄ to swing to '1'.
Output Q̄ remains at '1'.
because of the feedback from output Q.
060025 - 2 - 13

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)

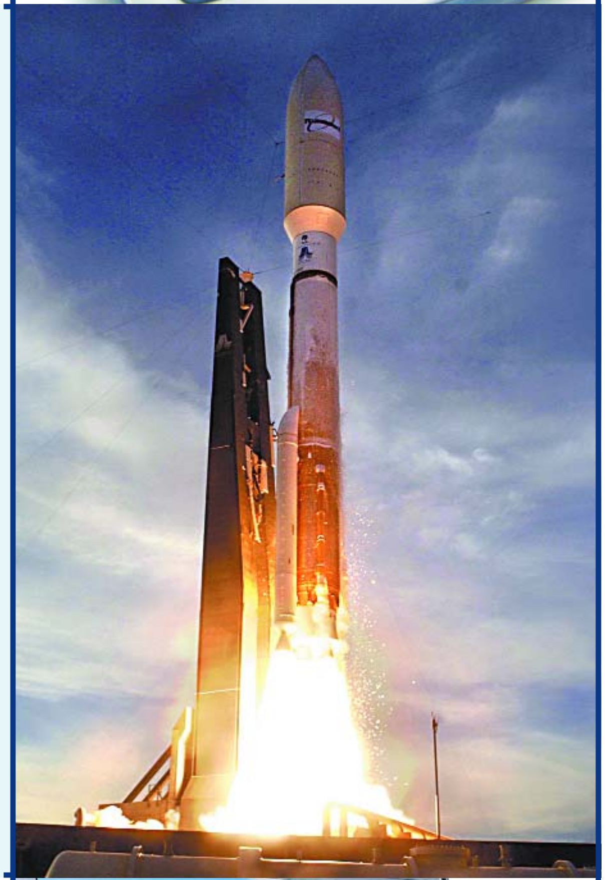
Digital arithmetic in practice

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.



FPGA Course (3)

Paul Goossens

This month we'll use the FPGA to construct a simple microcontroller system using a (free) 8052 microcontroller core. The microcontroller drives various peripheral elements of the prototyping board, including the I²C interface. Finally, we'll use it to build a four-channel multimeter.

This instalment uses an 8052 core derived from the T51 open-core design. We modified it slightly to make it easier to use in our FPGA module.
The internal operation of the microcontroller falls outside the scope of this course. However, we'll show you how to use the microcontroller and how to connect it the peripheral digital logic. That will be illustrated by building a four-channel voltmeter.

Cores

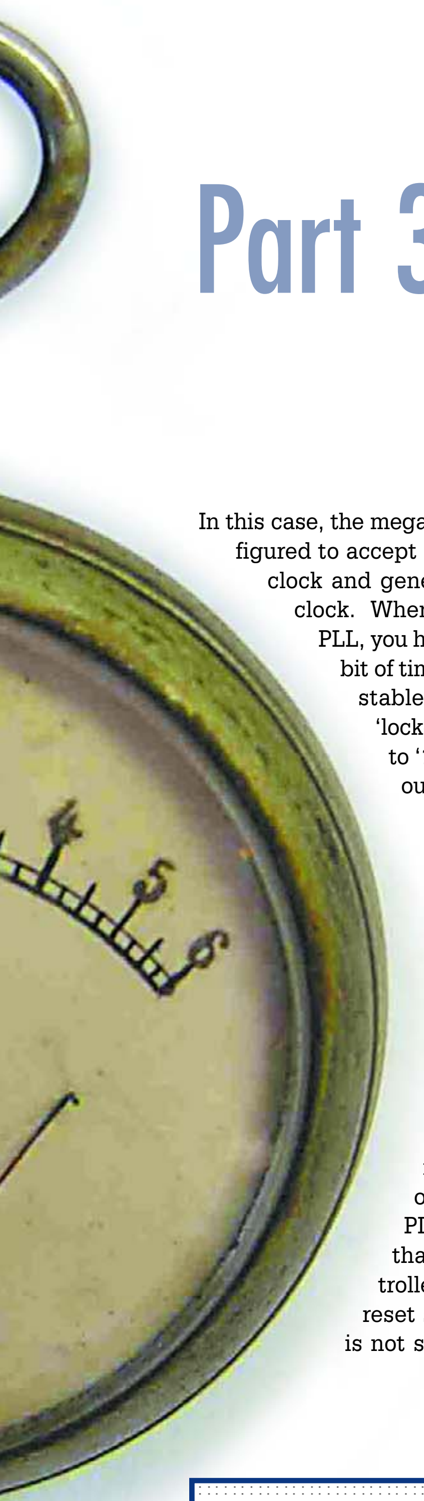
Our example here is built from various pieces of digital logic. You'll find the example project in the 'ex9' folder of the software for this instalment (060025-3-11). After opening the project file in that folder, you will see a schematic diagram as shown in **Figure 1**. The circuit is built around the block labelled 'T8052', which represents the 8052 microcontroller.

The associated files are located in the 'T51' and 'Altera Cyclone' folders. This microcontroller is derived from an 8052 core design that can be found at www.opencores.com. We modified the core slightly so it would use the internal memory of the FPGA more efficiently. We also added a wishbone interface. We'll have more to say about those changes in next month's instalment.

The important aspect of this microcontroller is that it has 8 KB of program memory and 4 KB of XRAM. It is also compatible with a standard 8052, so you can use regular 8052 development tools with it.

PLL

In this example, we operate the 8052 at a clock rate of 25 MHz. This clock signal could be generated using the 50-MHz clock on the board and a flip-flop. However, we decided to use the two PLLs in the FPGA for this example. The PLLs can be used to generate clock signals at frequencies that differ from the 50-MHz signal provided by IC7.
In the Quartus package, Quartus provides IP cores under the 'Megafunctions' designation that allow you to use the PLL. These megafunctions can be used as symbols via the same symbol toolbox that contains the input pins and so on. Refer to the Quartus manual if you want to know more about using megafunctions.



Part 3: Cores & System on Chip

In this case, the megafunction is configured to accept a 50-MHz input clock and generate a 25-MHz clock. When you use the PLL, you have to allow it a bit of time to generate a stable clock. The 'locked' output goes to '1' when the PLL output is stable.

8052

The 8052 in our circuit is clocked by the 25-MHz signal. Its reset input is a logic OR function of the 'switch1' input and the inverted 'locked' output of the PLL. That ensures that the microcontroller will be in the reset state if the PLL is not supplying a stable clock signal. The microcontroller can also be reset by pressing button S1 on the prototyping board.

The two external interrupt lines are tied to ground ('0') because they are not used in this example.

I/O ports PO–P3 require a bit more explanation. In a standard 8052, these ports can act as inputs as well as outputs. The pins are thus bidirectional. In an FPGA, it's often impossible to use bidirectional pins. For that reason, each port of the 8052 has an 8-bit input and an 8-bit output.

Unused inputs are simply connected directly to the corresponding outputs of the same port. For example, if bit 0 of port P3 is not used as an input, it must be connected to bit 0 of the output of port P3

The final thing that deserves mention is the XRAM_AC input, which forms part of the wishbone bus. We'll discuss that bus in next month's instalment.

I^2C

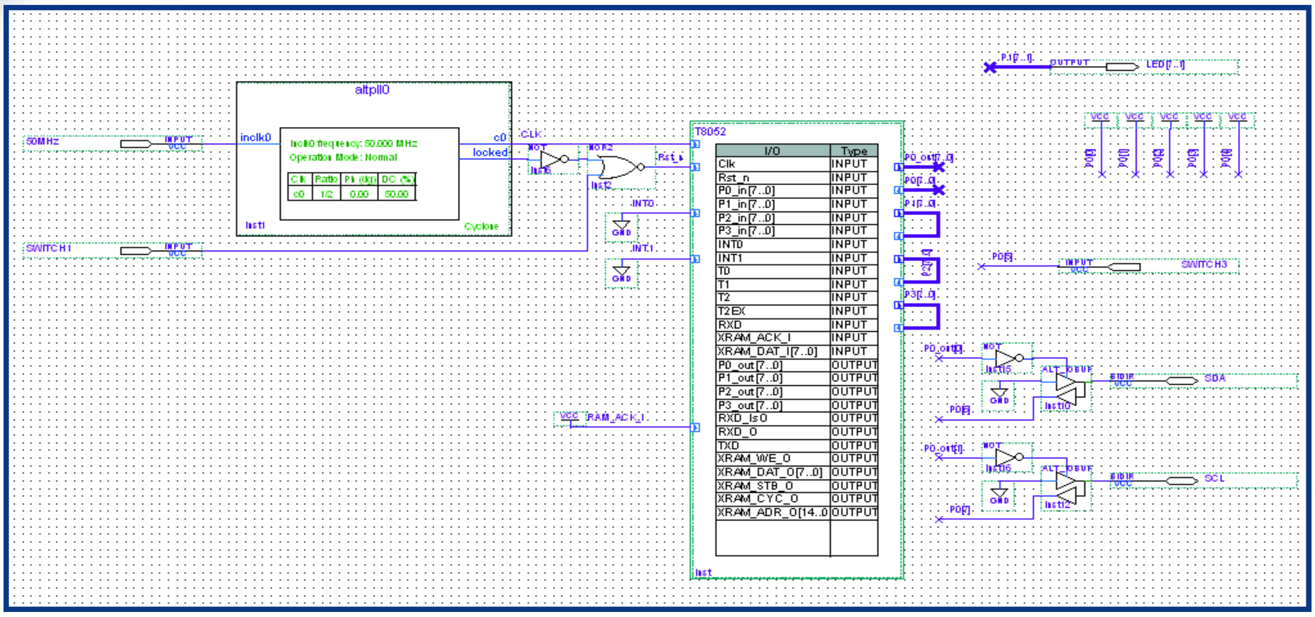
The I^2C bus requires two bidirectional signal lines. As we just mentioned, it's usually not possible to use bidirectional signal lines with an FPGA. Fortunately, the type of FPGA used here does allow I/O pins to be used for bidirectional signals. Two such pins are shown in the schematic drawing. These bidirectional pins are located in the same list as the input and output pins, with the designation 'bidir'.

We also need a bidirectional buffer in order to use these bidirectional pins. In this case, we used the 'ALT_OPBUF' symbol, which consists of an output buffer with an Enable input and an input buffer. When the Enable input is set to '1', the outputs can source or sink current according to the signal levels at the buffer inputs. The outputs are in a high-impedance state if the Enable input is '0'. In that state, an external device can drive the pin with a high or low level without causing a short-circuit condition.

The outputs of the input buffer can be used to read back the states of the pins, regardless of whether the output buffer is enabled.

As you already know, the I^2C lines must be connected to open-collector

Figure 1. Schematic diagram of the embedded system.



I²C bus

The prototyping board includes an I²C bus that can be used to drive various ICs. The ICs that provide the digital and analogue I/O signals on connectors K3 and K5 communicate with the FPGA via this bus, and the IC that drives the LCD uses the same I²C bus.

The general structure of the I²C signals is shown in the adjacent figure. There you can clearly see that all ICs connected to the bus drive the SDA and SCL lines via open-drain outputs. In other words, an IC can pull the level of a signal line to ground, but it can never connect it to the supply voltage. The pull-up resistors cause the level to be the same as the supply voltage when none of the connected ICs is pulling the line low.

Each communication is initiated by a 'Start condition', which can be recognised by a falling edge on the SDA (data) line while the SCL (clock) line is high. After that, the SCL line also goes low. The end of the communication is indicated by a 'Stop condition', which consists of a low-to-high transition on the SCL line followed by a high state on the SDA line. These two conditions are the only cases where the SDA line can change levels while the SCL line is high.

After a Start condition, the master first sends an I²C address. That is an 8-bit value, with the final bit indicating whether the communication is a read operation or a write operation. It is '1' for a read operation. The other seven bits are used to address a particular IC.

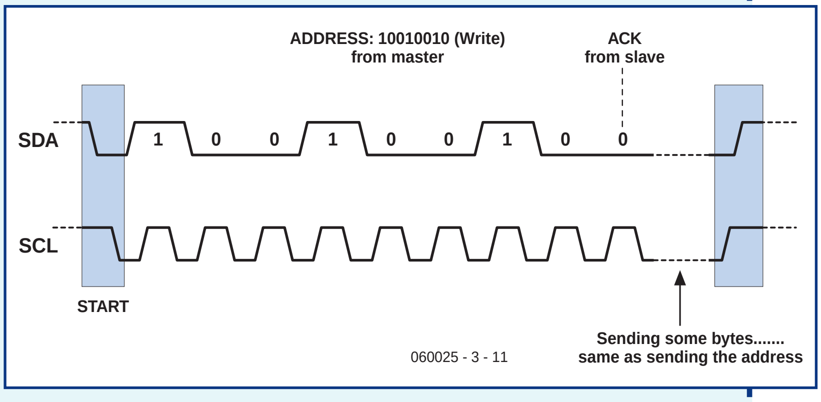
The IC that recognises its address will pull the SDA line low on the ninth clock pulse as a sign that it has been addressed. This is called 'ACK', which is short for 'Acknowledge'.

If the eighth bit is a '1', the master will then start sending its first data byte. It does this by placing the most significant bit of the first byte on the SDA line and then issuing a clock pulse. It then places the next bit on the SDA line and issues the next clock pulse. That process is repeated up to and including the least significant bit. On the ninth clock pulse, the slave pulls the SDA line low again as an ACK signal. After this, the master can send another byte if desired or terminate the communication with a Stop condition.

The situation is different with a read operation. In that case, the master generates a Start condition and then sends the address of the desired IC with the least significant bit set to '0'. The addressed slave device generates an ACK, just as before.

Next, the slave places the first bit (most significant bit) of the byte to be sent on the SDA line. The master then generates a clock pulse and receives that bit. After the master receives the eighth bit, it generates an ACK. The master can then decide to receive another byte or terminate the communication with a Stop condition.

It's important to bear in mind that the master always generates the clock signal. However, the slave can affect this by delaying the clock. That works as follows: after the master returns the SCL line to the high-impedance state, it must wait until the level on the SCL line goes high. The slave can hold the SCL line low during this interval if it needs extra time, for instance for processing data. The slave can thus use this mechanism to slow down communications.


ADDRESS: 10010010 (Write)
from master
ACK
from slave
SDA
1
0
0
1
0
0
1
0
0
SCL
START
060025 - 3 - 11
Sending some bytes.......
same as sending the address

outputs. That means the ICs connected to the lines are only allowed to sink current in order to pull the signal lines to a low level. For that reason, the input of the output buffer is connected to ground (GND) in the example. The signal line can then be pulled to ground by setting the Enable input to '1'.

Here the Enable line for the SDA signal is connected to the inverted bit-0 output of port P0. The Enable input will thus go to '1' when that bit set to '0', which will cause the SDA line to be pulled low.

The state of the SDA signal line of the I²C bus can be sensed at any desired time via bit 6 of the input buffer of port P0.

The SCL line is handled in the same way, but using other bits of port P0. This arrangement allows the I²C protocol to be implemented in software.

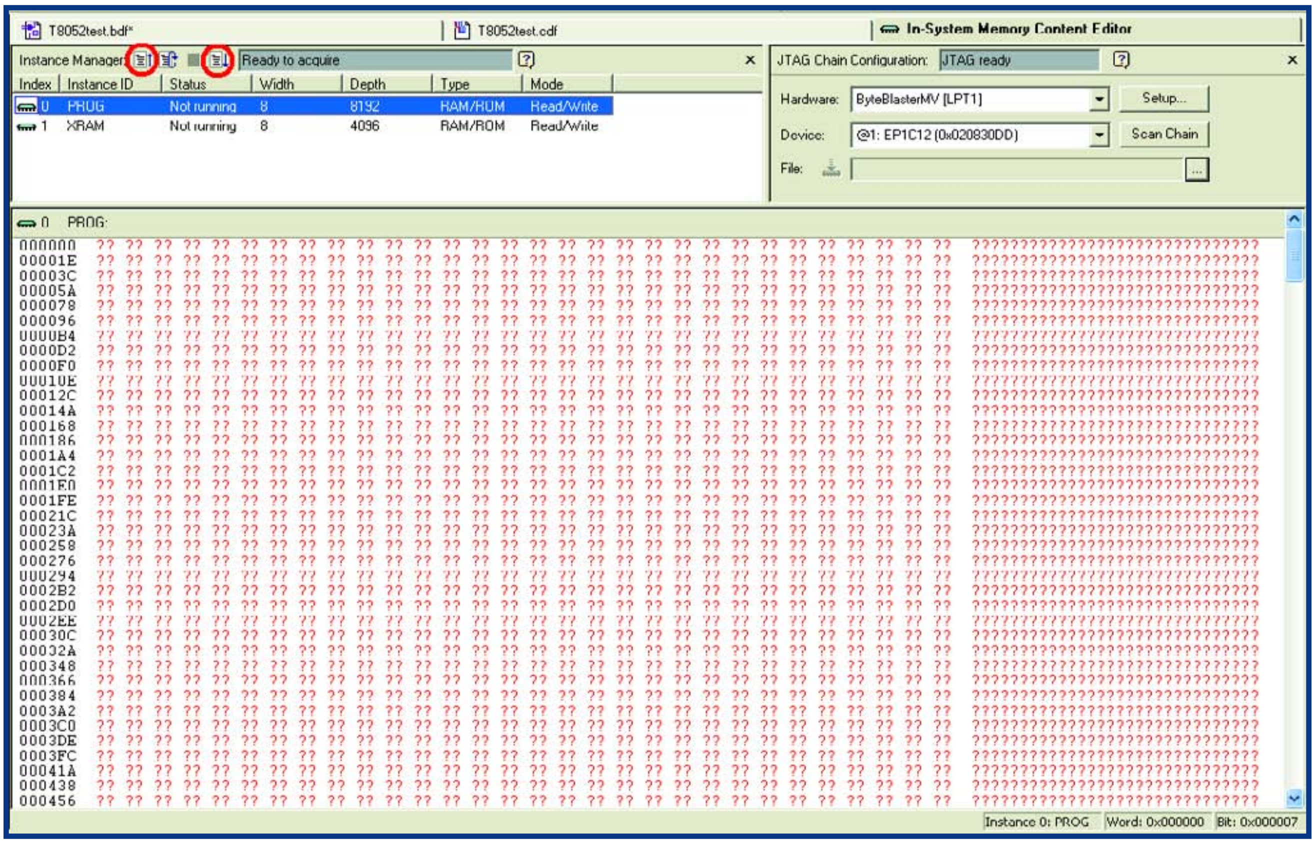
Firmware

By now, you may be wondering where the program for the microcontroller is located.

The program memory is incorporated in the T8052 core. This memory is also a megafunction. During compilation, the megafunction uses the 'firmware.hex' file in the 'firmware' folder to populate the program memory. The firmware can be modified quite easily, as you will soon see.

We used the free SDCC compiler to generate the firmware for this example. You can find the compiler on the home page of SDCC. You will need this compiler if you want to modify the firmware according to your own wishes. After downloading the compiler software, run the setup program. We recommend leaving the standard settings unchanged. When the installer asks whether you want to install SDCC using the default path, confirm this by clicking on 'Yes'.

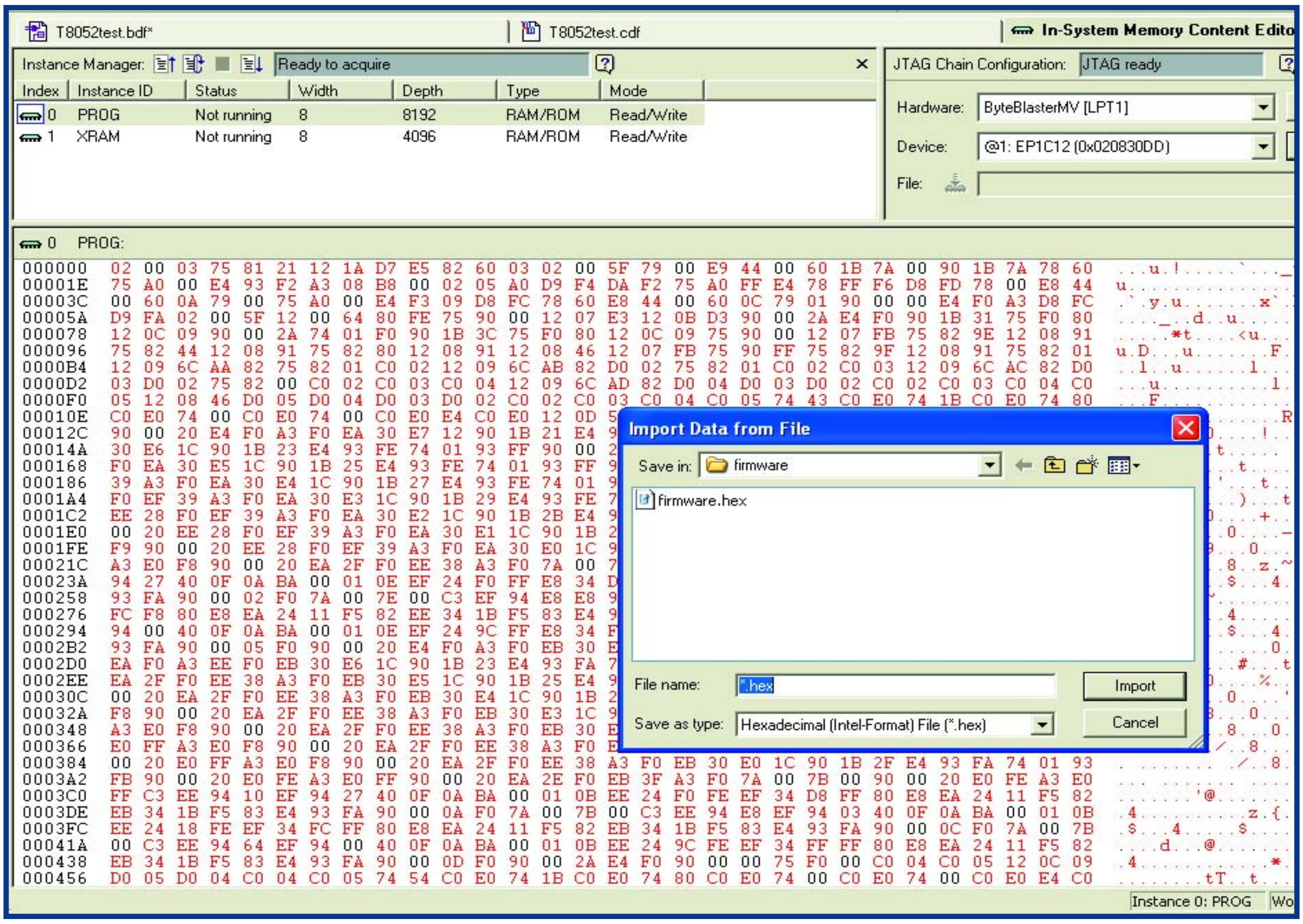
The firmware for this example is located in the folder under 'ex9/firmware'. That folder also contains a file named 'make.bat'. To



```
T8052test.bdf*                          T8052test.cdf                 In-System Memory Content Editor
Instance Manager:        Ready to acquire                             JTAG Chain Configuration:  JTAG ready
Index  Instance ID  Status       Width  Depth  Type     Mode          Hardware:  ByteBlasterMV [LPT1]       Setup...
0      PROG         Not running  8      8192   RAM/ROM  Read/Write    Device:    @1: EP1C12 (0x020830DD)    Scan Chain
1      XRAM         Not running  8      4096   RAM/ROM  Read/Write    File:

0  PROG:
000000  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00001E  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00003C  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00005A  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000078  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000096  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0000B4  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0000D2  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0000F0  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00010E  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00012C  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00014A  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000168  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000186  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0001A4  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0001C2  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0001E0  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0001FE  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00021C  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00023A  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000258  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000276  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000294  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0002B2  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0002D0  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0002EE  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00030C  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00032A  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000348  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000366  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000384  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0003A2  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0003C0  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0003DE  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
0003FC  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
00041A  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000438  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
000456  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??   ??????????????????????????????
                                                                                   Instance 0: PROG  Word: 0x000000  Bit: 0x000007
```

Figure 2. Memory peeking via JTAG.

Figure 3. Downloading new firmware.



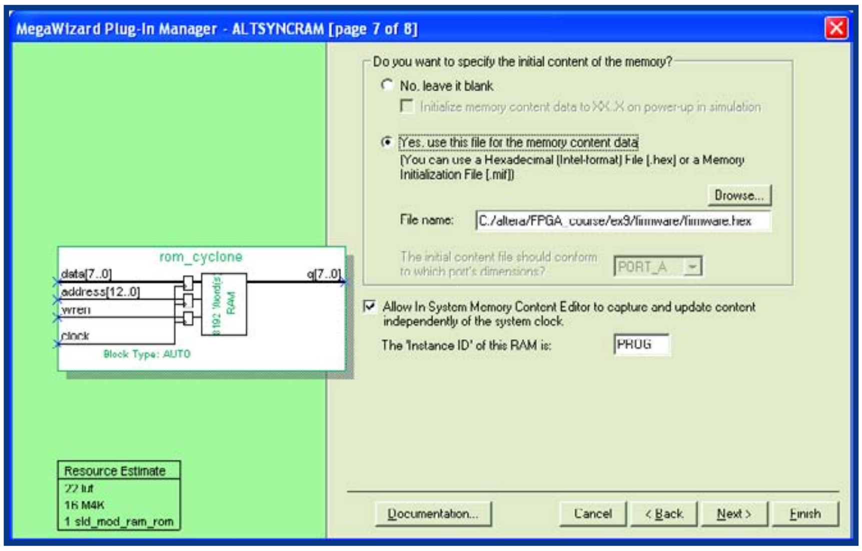
```
T8052test.bdf*                          T8052test.cdf                 In-System Memory Content Edito
Instance Manager:        Ready to acquire                             JTAG Chain Configuration:  JTAG ready
Index  Instance ID  Status       Width  Depth  Type     Mode          Hardware:  ByteBlasterMV [LPT1]
0      PROG         Not running  8      8192   RAM/ROM  Read/Write    Device:    @1: EP1C12 (0x020830DD)
1      XRAM         Not running  8      4096   RAM/ROM  Read/Write    File:

0  PROG:
000000  02 00 03 75 81 21 12 1A D7 E5 82 60 03 02 00 5F 79 00 E9 44 00 60 1B 7A 00 90 1B 7A 78 60   ...u.!.....`..._
00001E  75 A0 00 E4 93 F2 A3 08 B8 00 02 05 A0 D9 F4 DA F2 75 A0 FF E4 78 FF F6 D8 FD 78 00 E8 44   u...............
00003C  00 60 0A 79 00 75 A0 00 E4 F3 09 D8 FC 78 60 E8 44 00 60 0C 79 01 90 00 00 E4 F0 A3 D8 FC   .`.y.u.......x`.
00005A  D9 FA 02 00 5F 12 00 64 80 FE 75 90 00 12 07 E3 12 0B D3 90 00 2A E4 F0 90 1B 31 75 F0 80   ...._..d..u.....
000078  12 0C 09 90 00 2A 74 01 F0 90 1B 3C 75 F0 80 12 0C 09 75 90 00 12 07 FB 75 82 9E 12 08 91   .....*t....<u...
000096  75 82 44 12 08 91 75 82 80 12 08 91 12 08 46 12 07 FB 75 90 FF 75 82 9F 12 08 91 75 82 01   u.D...u.......F.
0000B4  12 09 6C AA 82 75 82 01 C0 02 12 09 6C AB 82 D0 02 75 82 01 C0 02 C0 03 12 09 6C AC 82 D0   ..l..u......l...
0000D2  03 D0 02 75 82 00 C0 02 C0 03 C0 04 12 09 6C AD 82 D0 04 D0 03 D0 02 C0 02 C0 03 C0 04 C0   ...u..........l.
0000F0  05 12 08 46 D0 05 D0 04 D0 03 D0 02 C0 02 C0 03 C0 04 C0 05 74 43 C0 E0 74 1B C0 E0 74 80   ...F............
00010E  C0 E0 74 00 C0 E0 74 00 C0 E0 E4 C0 E0 12 0D 5                                               R
00012C  90 00 20 E4 F0 A3 F0 EA 30 E7 12 90 1B 21 E4 9                                             0....!..
00014A  30 E6 1C 90 1B 23 E4 93 FE 74 01 93 FF 90 00 2                                             .t....
000168  F0 EA 30 E5 1C 90 1B 25 E4 93 FE 74 01 93 FF 9                                             ...t...
000186  39 A3 F0 EA 30 E4 1C 90 1B 27 E4 93 FE 74 01 9                                             '...t.
0001A4  F0 EF 39 A3 F0 EA 30 E3 1C 90 1B 29 E4 93 FE 7                                             ...)..t
0001C2  EE 28 F0 EF 39 A3 F0 EA 30 E2 1C 90 1B 2B E4 9                                             0...+..
0001E0  00 20 EE 28 F0 EF 39 A3 F0 EA 30 E1 1C 90 1B 2                                             ..0...
0001FE  F9 90 00 20 EE 28 F0 EF 39 A3 F0 EA 30 E0 1C 9                                             9...0..
00021C  A3 E0 F8 90 00 20 EA 2F F0 EE 38 A3 F0 7A 00 7                                             .8..z.~
00023A  94 27 40 0F 0A BA 00 01 0E EF 24 F0 FF E8 34 D                                             ..$...4.
000258  93 FA 90 00 02 F0 7A 00 7E 00 C3 EF 94 E8 E8 9                                             .
000276  FC F8 80 E8 EA 24 11 F5 82 EE 34 1B F5 83 E4 9                                             .4.
000294  94 00 40 0F 0A BA 00 01 0E EF 24 9C FF E8 34 F                                             ..$...4.
0002B2  93 FA 90 00 05 F0 90 00 20 E4 F0 A3 F0 EB 30 E                                             ......0.
0002D0  EA F0 A3 EE F0 EB 30 E6 1C 90 1B 23 E4 93 FA 7                                             ...#...t
0002EE  EA 2F F0 EE 38 A3 F0 EB 30 E5 1C 90 1B 25 E4 9                                             0....%..
00030C  00 20 EA 2F F0 EE 38 A3 F0 EB 30 E4 1C 90 1B 2                                             .0....'
00032A  F8 90 00 20 EA 2F F0 EE 38 A3 F0 EB 30 E3 1C 9                                             8...0..
000348  A3 E0 F8 90 00 20 EA 2F F0 EE 38 A3 F0 EB 30 E                                             .8...0.
000366  E0 FF A3 E0 F8 90 00 20 EA 2F F0 EE 38 A3 F0 E                                             /...8..
000384  00 20 E0 FF A3 E0 F8 90 00 20 EA 2F F0 EE 38 A3 F0 EB 30 E0 1C 90 1B 2F E4 93 FA 74 01 93   . ....... ./..8.
0003A2  FB 90 00 20 E0 FE A3 E0 FF 90 00 20 EA 2E F0 EB 3F A3 F0 7A 00 7B 00 90 00 20 E0 FE A3 E0   ... ....... ....
0003C0  FF C3 EE 94 10 EF 94 27 40 0F 0A BA 00 01 0B EE 24 F0 FE EF 34 D8 FF 80 E8 EA 24 11 F5 82   .......'@.......
0003DE  EB 34 1B F5 83 E4 93 FA 90 00 0A F0 7A 00 7B 00 C3 EE 94 E8 EF 94 03 40 0F 0A BA 00 01 0B   .4..........z.{.
0003FC  EE 24 18 FE EF 34 FC FF 80 E8 EA 24 11 F5 82 EB 34 1B F5 83 E4 93 FA 90 00 0C F0 7A 00 7B   .$...4.....$....
00041A  00 C3 EE 94 64 EF 94 00 40 0F 0A BA 00 01 0B EE 24 9C FE EF 34 FF FF 80 E8 EA 24 11 F5 82   ....d...@.......
000438  EB 34 1B F5 83 E4 93 FA 90 00 0D F0 90 00 2A E4 F0 90 00 00 75 F0 00 C0 04 C0 05 12 0C 09   .4............*.
000456  D0 05 D0 04 C0 04 C0 05 74 54 C0 E0 74 1B C0 E0 74 80 C0 E0 74 00 C0 E0 74 00 C0 E0 E4 C0   ........tT..t...
                                                                                                   Instance 0: PROG  Wo

Import Data from File
Save in: firmware
firmware.hex
File name:     *.hex                                        Import
Save as type:  Hexadecimal (Intel-Format) File (*.hex)      Cancel
```

recompile the firmware, simply double-click on the 'make.bat' file.
All the intelligence of the circuit is contained in the firmware. Among other things, it looks after driving the I²C bus, the A/D converter and the LCD module. The source code is relatively simple and should not present any problem for anyone with a reasonable knowledge of C.

Hands-on

Start the program and configure the FPGA using the example project for this instalment, which you will find in the 'ex9' folder. After you configure the FPGA, you will see a welcome message on the LCD. Starting around 4 seconds later, the LCD will continuously display the voltages measured at the four analogue inputs of the prototyping board. Congratulations – you just built a four-channel multimeter!
As you know, the FPGA now contains a microcontroller with program memory and working memory. Thanks to the modifications we made to the microcontroller, you can use Quartus to view and modify the memory contents.
In the 'Tools' menu, click on 'In-System Memory Content Editor'. That will open a new window as illustrated in **Figure 2.** Two memories are listed at the top left, with the names 'PROG' and 'XRAM'. PROG contains the firmware that is executed by the 8052. At this point, Quartus does not know the content of that memory. Consequently, the contents of the memory locations shown in the box under 'PROG' are filled with question marks. Select the program memory by clicking on the word 'PROG'. You will see 'Instance manager' somewhere near the top of the window, with several buttons next to it. Click on the leftmost button (an icon showing a document with an upward-pointing arrow next to it), which is called 'Read'.

Quartus will read in the content of the program memory and display it in the box below. That all happens without causing any problem for the 8052. In fact, there's no sign of any sort of slowdown.


MegaWizard Plug-In Manager - ALTSYNCRAM [page 7 of 8]

Do you want to specify the initial content of the memory?
No, leave it blank
Initialize memory content data to XX..X on power-up in simulation
[✓ Yes, use this file for the memory content data]
(You can use a Hexadecimal (Intel-format) File [.hex] or a Memory Initialization File [.mif])
File name: C:/altera/FPGA_course/ex9/firmware/firmware.hex
The initial content file should conform to which port's dimensions? PORT_A
[✓] Allow In System Memory Content Editor to capture and update content independently of the system clock
The 'Instance ID' of this RAM is: PROG

Resource Estimate
22 lut
16 M4K
1 sld_mod_ram_rom

Figure 4. Specifying the location of the firmware on the hard disk.

New firmware

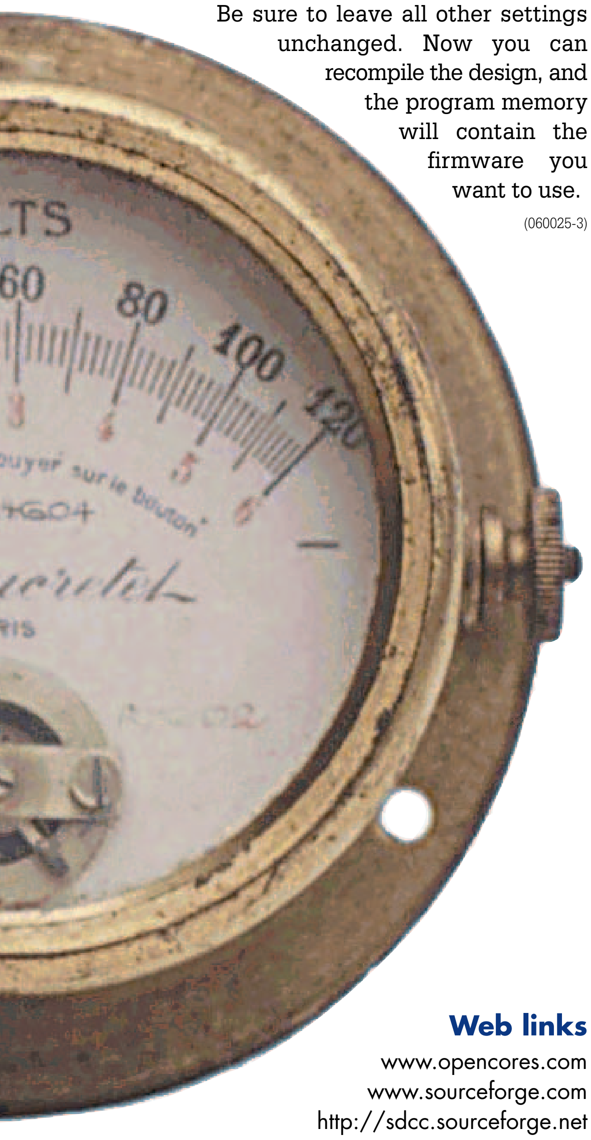
The next exercise is to try making changes to the firmware. For example, you can change the string in line 57 to 'Test'. Save this change, and then double-click on 'make.bat' in the previously mentioned folder in the Windows Explorer window. SDCC will recompile the program and generate a new 'firmware.hex' file.
Return to the Memory Content Editor window and position the mouse cursor somewhere in the data area. Now right-click with the mouse and select 'Import Data from File…' in the popup menu. Navigate to the 'firmware' folder and select the new 'firmware.hex' file. The program will now load the content of the selected hex file into the memory buffer. A warning message will also be displayed to indicate that the length of the hex file is not the same as the length of the memory buffer. You can simply ignore that message.
Next, press button S1 on the prototyping board to reset the microcontroller in the FPGA. While holding this button depressed, click on the 'Write' button on the screen. You can recognise it by its icon (a document with a downward-pointing arrow). That causes the program to transfer the content of the memory buffer to the FPGA memory via the JTAG interface.
Now you can release the reset button. The welcome message on the LCD will be different now, which proves that the new firmware has been loaded into the 8052.

On your own

While you're waiting for the next instalment, it's a good idea to work on your own to acquire more experience with the 8052. The easiest approach is to copy the 'ex9' folder to a new folder. You can then experiment to your heart's content with the copied version.

Don't forget that the compiler expects to find the firmware at a particular location, but it's easy to change that location. In the Project Navigator window, select the 'Hierarchy' tab. Click on the + sign next to 'T8052'. A list will be displayed underneath, with among other things the entry 'rom_cyclone'. Double-click on this entry. That will bring up a 'MegaWizard' window. Click on 'Next' to proceed to the next window of the wizard. Keep going in the same way until you reach window 7 (**Figure 4**). In this window, enter the exact location and name of the hex file you want to use for the firmware. Finally, click on 'Finish'. Be sure to leave all other settings unchanged. Now you can recompile the design, and the program memory will contain the firmware you want to use.

(060025-3)



Web links

www.opencores.com
www.sourceforge.com
http://sdcc.sourceforge.net

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).

FPGA Course (2), Elektor Electronics May 2006.

STD_LOGIC

Starting with part 1 of this course, we have been using signals of type STD_LOGIC. That signal type can have various states, but up to now we have only used the '1' and '0' states.

Signals of type STD_LOGIC can also have other states, of which 'Z' and '-' are the most important. The 'Z' state indicates that the signal is in a high-impedance state. An example of where that can happen is the output of a buffer with an Enable input. It is recommended **not** to use this state if you are generating a design to be implemented in an FPGA, because most FPGAs cannot generate internal high-impedance signal states. However, most FPGAs (including the one we use) can generate high-impedance states on their I/O pins. It is thus possible to use the 'Z' state for such signals.

Another state is '-', which is called 'don't care'. If you set a signal to '-' in a particular case, you effectively tell the compiler that you don't care whether the output is '1' or '0' and it has no effect on the overall operation of the design. That allows the compiler to simplify functions where possible, with the result that the circuit may take up less space in the FPGA.

We used an ALT_IOBUF in Quartus for the I^{2}C interface. That is a buffer with an Enable input, which means that the buffer outputs go to a high-impedance state when the Enable input goes to '0'. The level on the output (I/O pin of the FPGA) can be read via a second buffer.

Open cores

This 8052 core we used here is derived from the T51 core. That core is available free of charge from the www.opencores.com website. We modified the T51 core slightly to suit our purposes.

All cores (which are also called 'IP', which stands for 'intellectual property') on this website are free. That means they can be used without paying any licence fee. We made a few modifications to the 8052 core, primarily to enable it to work properly with the internal memory of the Cyclone FPGAs.

We don't intend to describe the internal operation this 8052 here. That's anyhow not particularly relevant with a core (aside from the educational value). A designer can simply regard a core as a black box that performs a particular function. It's the same as with an IC – you need to know how you can use it, but the details of its internal operation are not important.

We placed all the necessary files for this core in the 'T51' subfolder. The VHDL files in that folder collectively constitute an 8052 microcontroller. The top-level document for this core is the 'T8052.vhdl' file, and that is the only file you have to use directly. It's not particularly difficult to use this core in a new design (such as your own design). Nevertheless, we recommend that you copy the 'ex9' folder to a new folder if you want to use it in your own designs, after which you can modify the copied files as you see fit.

Most of the signal names of the 8052 core are self-explanatory, but the ones that start with 'XRAM' need a bit of explanation. These signals collective form what is called a 'wishbone bus', and that's something we'll explain in the next instalment. For now, it's quite important to know that the 'XRAM_ACK' signal must always be set to '1' if this bus is not used, since otherwise the microcontroller can hang unexpectedly.

FPGA Course (4)

Paul Goossens

Welcome back to the beginner-friendly course we run in support of our extremely popular FPGA Development System. This month we examine the simulation capabilities of Quartus. Simulation makes it a lot easier to design circuits and track down errors in your designs. The accompanying examples show how you can use the audio interface of the prototyping board.

It's handy to be able to test your design during the design process. VHDL allows you to create test benches, which make it a lot easier to test and simulate VHDL designs.
Unfortunately, Quartus does not support VHDL test benches. It has a graphic simulator instead. Although the simulator provides less user functionality, it is easier to use. The simulator is more than adequate for most of the sample applications in this course.

Virtual

The simulator is actually a combination of a virtual signal generator and a logic analyser. Here we use it to simulate the operation of an audio interface implemented with the hardware.

Codec

The 'ex10' sample application described in this instalment uses the audio codec (IC12) on the prototyping board. From the schematic diagram, you can see that a 12.288-MHz clock signal is applied to this IC. The clock signal is also routed to an I/O pin (B12) of the FPGA. Unfortunately, that was not shown in the original schematic diagram.

The data transfer clock (BCKIN) is supplied by the FPGA. This clock signal must be synchronised to the 12.288-MHz clock signal.

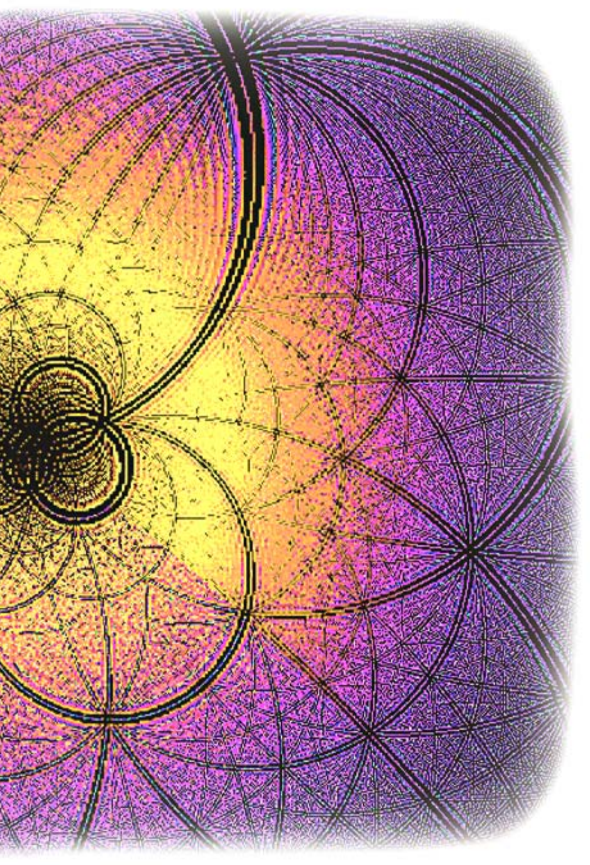
The timing diagram for data transfers between the codec and the FPGA is shown in **Figure 1**. As you can easily see, it takes 156 clock pulses of the CODECCLK signal to transfer a set of samples for the two channels.

The frequency of that signal on the prototyping board is 12.288 MHz. If you divide that figure by 256, you arrive at a sampling rate of 48 kHz.

Counter

All communications are synchronised to this clock. The simplest approach is to first create a counter that counts how many pulses of the CODECCLK signal have been generated. This signal is assigned the name '12_288MHz' in our design.
This counter is also represented by the COUNT signal in the PCM3006.VHD file, which is declared on line 54 as an 8-bit unsigned number. This signal can thus take on values the range of 0–255, which is exactly what we need.
We let this number increment in synchronization with the CLK signal (12.288 MHz).
The NEW_COUNT signal always contains the value that COUNT must assume on the next clock pulse. This is done by the following line:

```
NEW_COUNT <=
(COUNT+1) MOD 256;
```



Part 4: Simulation

This line is not in a process, so the function is evaluated each time COUNT changes.
The value of NEW_COUNT is loaded synchronously into COUNT in line 71.

Simulation

Now you can test this code using the Quartus simulator. For that purpose, we created a simulation file named 'ex10-1.vwf'. If you open this file, you will see several signals in the left-hand column and plots of the input signals versus time on the right.
Before you can use this file, you must configure Quartus so the simulator can use the file.

To do that, select 'Settings' in the 'Assignments' menu. In the new window that appears, select 'Simulator Settings'. Enter the file name 'ex10-1.vwf' in the Simulation Input box.
The simulation starts as soon as you select 'Start Simulator' in the 'Processing' menu. The result of the simulation (**Figure 2**) is displayed after the simulation is completed.
What matters at this point is the COUNT signal. As you can see from the simulation results, this counter is indeed incremented by 1 each time a rising edge occurs on the 12.288-MHz signal line. Note that COUNT has been changed to a 7-bit number due to optimization. Later on we'll explain why that is possible. What matters now is that the counter is synchronized to the clock.

Alias

The next step is to generate the data transmission clock (BCKIN). This signal must be low for two clock intervals of the main clock signal, after which it must be high for two clock intervals. That corresponds exactly to the third bit of the COUNT signal. This signal (called BCKOUT in the VHDL file) can thus be used at the output without any further processing.

The same holds true for the LRCK signal. This signal must be low for the first 128 clock pulses and then high for the following 128 clock pulses. That corresponds exactly to the most significant bit (highest-order bit) of the COUNT signal. That means you can use bit 7 of the COUNT signal as LRCKOUT.
The new status of LRCKOUT can be defined using the following line:

NEW_LRCKOUT <= NEW_COUNT(7);

Another way to do this is to use the 'alias' keyword, which allows a signal to have more than one name. If you write

ALIAS NEW_LRCOUT : STD_LOGIC is NEW_COUNT(7);

you can use the signal NEW_LRCOUT in the rest of the source code. The compiler will know that this signal is identical to NEW_COUNT(7).

Synchronous

Data bit reception is synchronised to the rising edge of BCK. The signal POSEDGE_BCK is used to detect the rising edge of BCK. It must indicate whether the BCK signal changes from low to high on the next rising edge of the system clock.

That requires knowing the current status of BCK and the status after the next clock pulse. These signals are COUNT(2) and NEW_COUNT(2), which are also assigned the names BCK_INT and NEW_BCK by alias statements in lines 62 and 63.
The POSEDGE_BCK signal is generated by the following line:

POSEDGE_BCK <= NEW_BCKOUT AND (NOT BCKOUT_INT);

Data must be sent on the falling edge of BCK as shown in **Figure 2**. The signal NEGEDGE_BCK is generated in a similar manner for his purpose.

Glitches

Now it's time to look at these new signals in more detail. First configure the settings to have the simulator use the file *ex10.vwf*, and then start the simulation.

You will see the signals NEGEDGE_BCK, POSEDGE_BCK and LRCIN in the simulation results. The last signal of this group is the same as

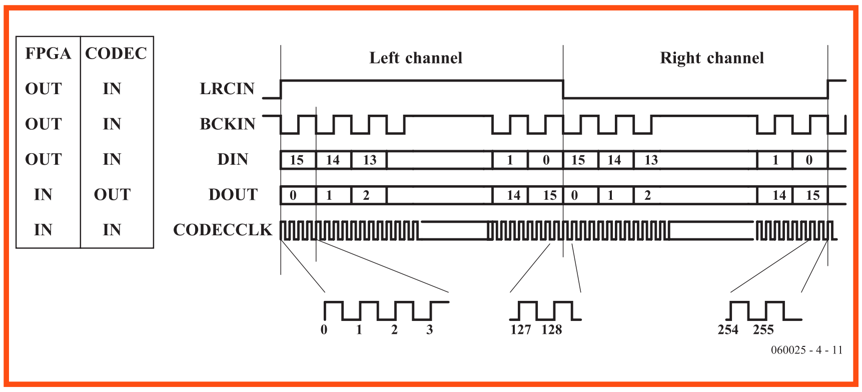


Figure 1. Timing diagram for data transfers between the codec and the FPGA.

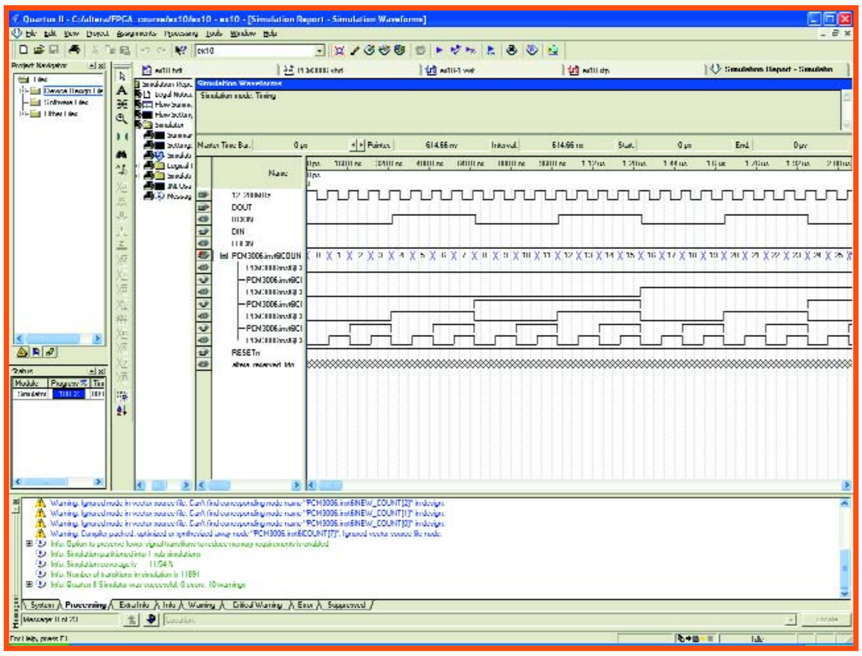


Figure 2. **Data is transmitted on the falling edge of BCK, as can be seen from the simulation.**

Each time a new set of samples must be transmitted, this shift register is loaded using the R_IN and L_IN signals. The contents of the receive shift register are also loaded into the LEFT_OUT and RIGHT_OUT registers.

Interface

At the same time, the NEW_SAMPLE output is set high for one clock interval. This signal indicates that a new set of samples has arrived. The peripheral logic can use this signal to process the new data.

Data must be applied to the inputs of the RIGHT_IN and LEFT_IN inputs in order to be transmitted. A high level at the LOAD input causes the data on the inputs to be stored, and it will then be sent with the next transmission.

Example

In the example, you can see that the data outputs are connected directly to the associated data inputs. The NEW_SAMPLE output signal is connected to the LOAD input.
This causes the received samples to be sent back to the codec on the next transmission. In other words, the signals at the inputs appear unchanged at the outputs after a short delay.

the LRCOUT signal in the VHDL design.
The POSEDGE_BCK signal and its counterpart NEGEDGE_BCK are generated using combinational logic. That means these signals are not synchronised by flip-flops. The disadvantage of this is that these signals can briefly assume an incorrect level if the input signals have different path delays. That phenomenon appears in the simulation in the form of short pulses. The technical term for these short pulses is 'glitches'.

Shift registers

The incoming and outgoing data are read in and 'pushed out' by shift registers. On each rising edge of the BCK signal, the contents of the SHIFTIN register are shifted left by one position. The incoming data is stored in bit 0.
The transmit shift register, SHIFTOUT, operates in a similar manner. It shifts the bits by one position on each falling the edge of BCK. The most significant bit of this shift register is also the serial data output.

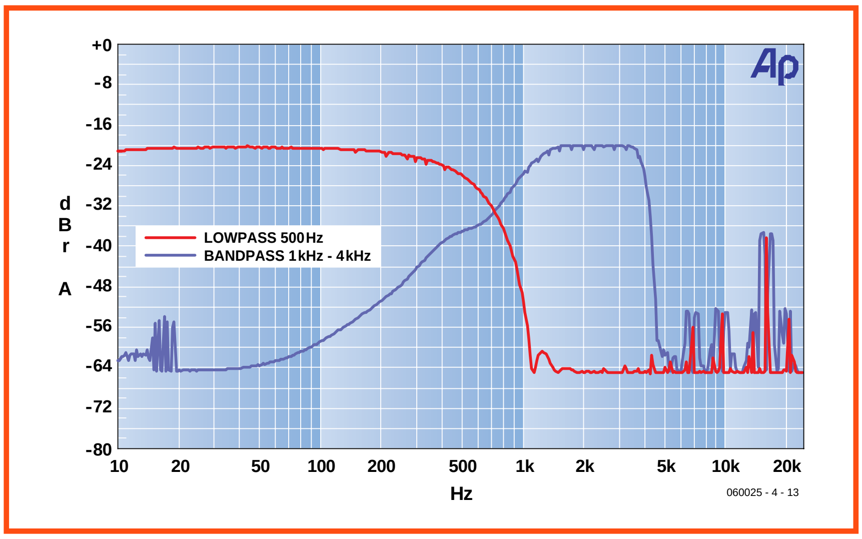


Figure 3. **The characteristics of the various filter sets.**

Another simulation

Overall operation of the circuit is illustrated by the file *ex10-3.vwf* . In this simulation, we used a random bit pattern for DOUT. The simulation clearly shows that this bit stream appears at the DIN output after approximately 31 μs. The received data is thus sent back to the codec without any modification.

As you can easily see, several signals are missing in this simulation. The reason for this is that these signals 'disappeared' in during compilation, because the compiler attempts to generate a design that is compact a possible. As a result, certain signals may become redundant, and the compiler will not implement these signals in the FPGA. As a result, Quartus cannot simulate the signals during the simulation session.

Filter

The bypass function described above is not particularly useful in practice. A more useful technique is to process the input signal and then output the results via the codec.

The example file *ex11* implements an audio filter. Communication with the codec takes place via the previously described PCM3006.VHD core.
The samples are processed in the FIR.VHD code segment. The filter operates on the FIR principle. ‘FIR’ stands for ‘finite impulse response’. Filters of this sort are often used in digital signal processing.

In a FIR filter, the passband characteristics are determined by a set of parameters called coefficients. These coefficients can be modified as desired in Quartus. Just as in the previous instalment, the Memory Content Editor is the tool for that purpose. The coefficients are stored in the memory segment named ‘COEF’. The memory segment named ‘IN’ holds the most recent 128 audio samples.
Several hex files with coefficients are available for the project so you can easily try out various filters. The characteristics of the various filter sets are shown in **Figure 3**.

Signal generator

The final example, *ex12*, implements a simple sinewave generator. It products sinusoidal signals on the outputs. The signal on the right channel lags the signal on the left channel by 90 degrees.

The sinusoidal signals are generated using an arithmetic unit that can compute sine and cosine values. This unit needs an angle (*phase*) and an amplitude (*mag*) for this purpose. The unit then calculates the corresponding *X* (cosine) and *Y* (sine) values.

The arithmetic unit uses the CORDIC algorithm, which is suitable for calculating goniometric functions using simple operations. CORDIC has the unique property that it multiplies the length of each vector by approximately 1.645. That means you have to ensure that the results fit within the range of a 16-bit signed number, so the input value must not exceed 4DDO. If you use a larger value, the resulting sinewave will be highly distorted.

Signal

To obtain a sinusoidal signal, a constant value must be applied the *mag* input. In addition, the phase must be increases slightly for each sample. That is the function of the *mag_phase_accu* block. Each time a new sample is sent, the signal *new_sample* goes high briefly. That tells this block that it must increment the value of *phase* by a certain amount. That amount can be set using the dip switches. The larger the amount, the higher the frequency at the output of the codec. The block *cordic* then performs the calculation and sends the result to the codec.

(060025-4)

CORDIC

CORDIC (**Co**ordinate **R**otation **Di**gital **C**omputer) is a method that can be used to implement goniometric functions efficiently in digital systems. It describes how to calculate goniometric functions using only add and shift operations.

CORDIC uses vectors. These vectors can be described as a combination of real and imaginary numbers (corresponding to their X and Y coordinates), or as a combination of a length and an angle.

If two vectors (A and B) are multiplied together, the length of the resulting vector (C) is equal to the length of vector A times the length of vector B. The angle of the resulting vector is the sum of the angles of vector A times and vector B. This multiplication takes the following form in X,Y notation:

$X_c = (X_a \times X_b) - (Y_a \times Y_b)$

$Y_c = (Y_a \times X_b) + (X_a \times Y_b)$

As you can clearly see, this requires using multiplications. If we ensure that these multiplications are all powers of 2, everything becomes very simple. Multiplication by a power of 2 (such as 2^{-2}) is equivalent to shifting bits. That is very easy to do using digital logic.

The CORDIC method describes how to calculate goniometric functions by multiplying an initial vector by vectors with X coordinates equal to 1 and Y coordinates that are always powers of 2. As a result, this method can be implement very efficiently in digital circuitry.

Example

Consider the following expression as an example: 100 × cos(30°). As our starting point, we take the vector (100, 0), which has length of 100 and an angle of 0°. The desired angle is greater than the current angle, so the first operation is to multiply the vector by the vector (0, 1). Our vector now has an angle of 90°. That is greater than the desired angle.

The next step is thus to reduce the angle. For that purpose, we multiply by the vector (1, –1). Note that the Y value of this vector is negative, so angle of this vector is also negative.

After this multiplication, the angle of our vector is (90° – 45°) = 45°. This is still greater than 30°, so the next step is to multiply by the vector (1, –0.5). This causes to angle of our vector to become (45° – 26.57°) = 18.43°.

The angle is now smaller than what we want. We thus use the vector (1, 0.25) in the next step. That increases the angle by 14.04°. After this step, the angle of our vector is 32.47°. We get closer to 30° with each step, so the result is more accurate with each step.

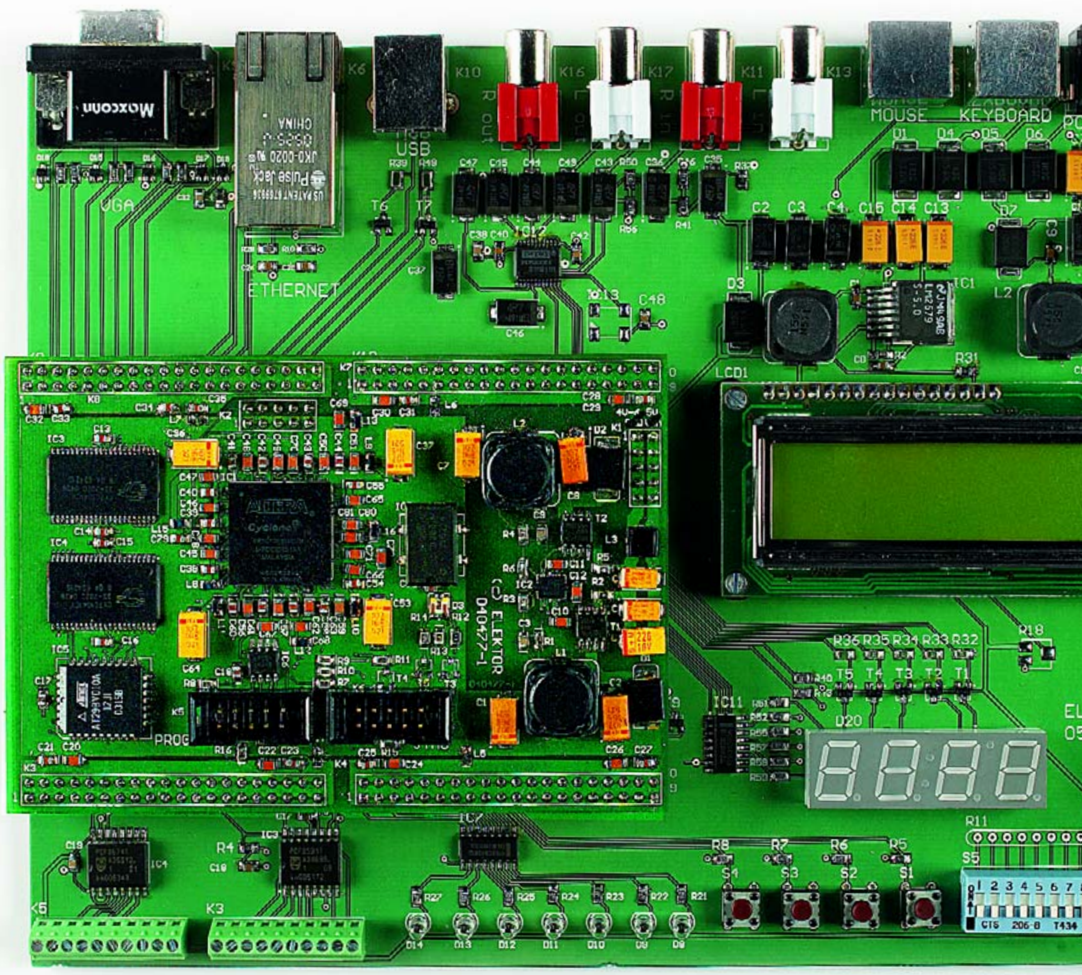
Each multiplication changes the length of our vector as well as its angle. In the present case, we now have a vector with a length of 100 × (1 × 1.41 × 1.12 × 1.03) = 162.66. That means we have to multiply this value by a correction factor. Another option would be to make the length of the initial vector 61.5 in order to finally obtain a vector with a length of 100. This multiplication factor is always the same, regardless of which angle we want.

No matter which option we choose, the desired cosine value is given by the X coordinate of our vector, while the Y coordinate represents the sine value. We get those values for free!

X	Y	Angle	Length
0	1	90°	1
1	1	45°	1.41
1	1/2	26.57°	1.12
1	_	14.04°	1.03
....	....	....	....

FPGA Course (5)

Paul Goossens



Every embedded system uses a system bus to transport data between the various components. This also applies to systems implemented in an FPGA. However, a different sort of bus system is commonly used in FPGAs. This month's article introduces a bus system that is often used in FPGAs.

A typical bus system in a 'normal' microprocessor circuit consists of a data bus, an address bus and several control signals, such as RD/$\overline{\text{WR}}$. The peripheral ICs put their data on the bus when requested to do so. During the rest of the time, their inputs are in a high-impedance state to give other ICs a chance to put data on the bus. These data ports are tristate ports, which means they can be set to a high-impedance state.

Different

In many FPGAs, it is not possible to put internal signals in a high-impedance state. In addition, a mistake in the design can cause short-circuits or data corruption on the bus. Tristate ports are thus not used for the system bus in such systems. Another factor is that the peripheral electronics often runs at a different clock speed than the processor. This makes handshaking necessary to ensure correct data transport.

Several standard system busses have been developed to avoid problems of this sort in FPGA designs. This instalment focuses on a system bus called the 'Wishbone bus', which is used quite often. It is used a lot at *www.opencores.com*, a handy site where you can download free designs and subdesigns.

Minimum system

A minimum Wishbone bus with a single master and a single slave is shown in **Figure 1.** The dual data bus is clearly visible. Each bus is unidirectional – one bus carries data from the master to the slave, while the other bus carries data in the opposite direction.

The STB (strobe), CYC (cycle) and ACK (acknowledge) signals provide handshaking for each data transmission. The slave can only respond to the Wishbone signals if the STB_I and CYC_I signals are both high. The master sets WE (write enable) high to indicate that it wants to write data to the slave. If this signal is low, it means that the master wants to read data from the slave.

When the slave has finished processing the data, it signals this by setting the ACK signal high. The master pulls the STB signal low in response. The slave must then return its ACK output to the low state.

This handshake protocol makes it possible to connect a slow slave device to

Part 5: bus systems and interconnections

a much faster master, since the slave can set its ACK signal high sometime later. This gives a relatively slow slave enough time to process the data. **Figure 2** shows a read operation of this sort, in which the slave needs two extra clock cycles to complete the transaction.

Example

We have prepared a simple example in *ex13*. Here the 8051 microcontroller has a master interface for the Wishbone bus. The bus is connected to a simple slave device that enables the master to drive eight outputs. The slave interface causes the ACK signal to appear with a delay of 10 clock pulses. This makes it possible to display the handshake process using the logic analyser built into Quartus.

The processor used here (T8052) uses the Wishbone bus for all transactions with XRAM memory in the region starting at address 0x1000. The only extension unit on this Wishbone bus is an 8-bit output named *wish_output*. This extension also has an internal address decoder. This is normally placed in a separate bit of hardware, but for this simple example we placed it in the core instead.
Seven of the eight outputs are connected to the LEDs on the extension board. The software causes the LEDs to light up sequentially for a 'running-light' effect.

Internal

Processing the Wishbone signals is fairly simple. The *sel* signal detects whether the address on the system bus matches the address of the extension (0x8000).
The code starting on line 63 causes the outputs to go high after a reset. When a valid address appears (*sel* = '1') while a valid write cycle is in progress (STB = '1', CYC = '1' and WE = '1'), the data present at the DAT_I input is stored in the output register by the rising edge of the clock signal.
Generating the ACK signal is somewhat more complicated in this case because it has to be delayed. The COUNT signal keeps track of how many clock pulses have occurred since the last write operation to this core. When the value of this counter reaches 10, ACK_OK goes high. This signal indicates that an ACK signal can be generated now.
The ACK output signal is finally defined in line 101. This core also generates an ACK if an invalid address is placed on the bus (*sel* = '0'). This is designed to prevent the processor from hanging if the software accidentally uses an incorrect address.
Note that the ACK signal is an asynchronous signal. In other words, it is not generated using a flip-flop. This is one of the requirements of the Wish-

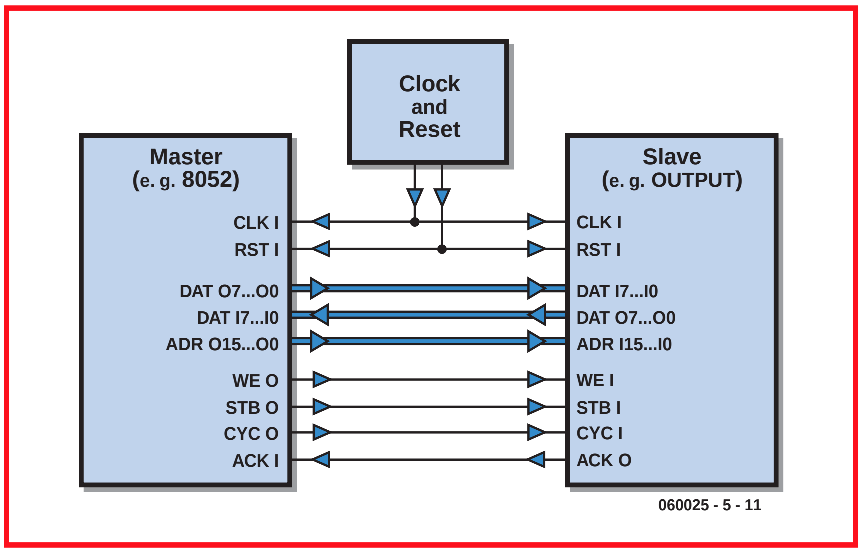


Figure 1. A minimum Wishbone bus with a single master and a single slave.

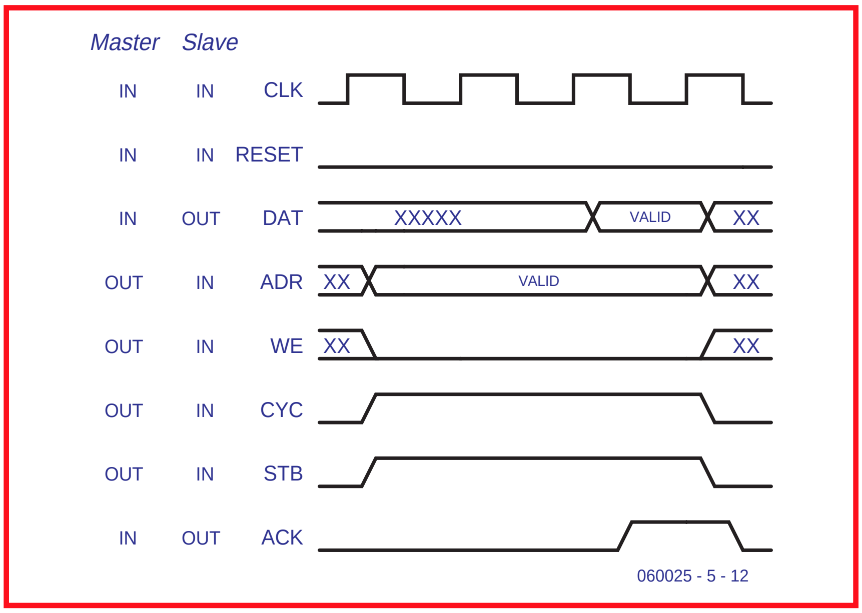


Figure 2. Here you can see that slave needs two extra clock cycles to complete the transaction.

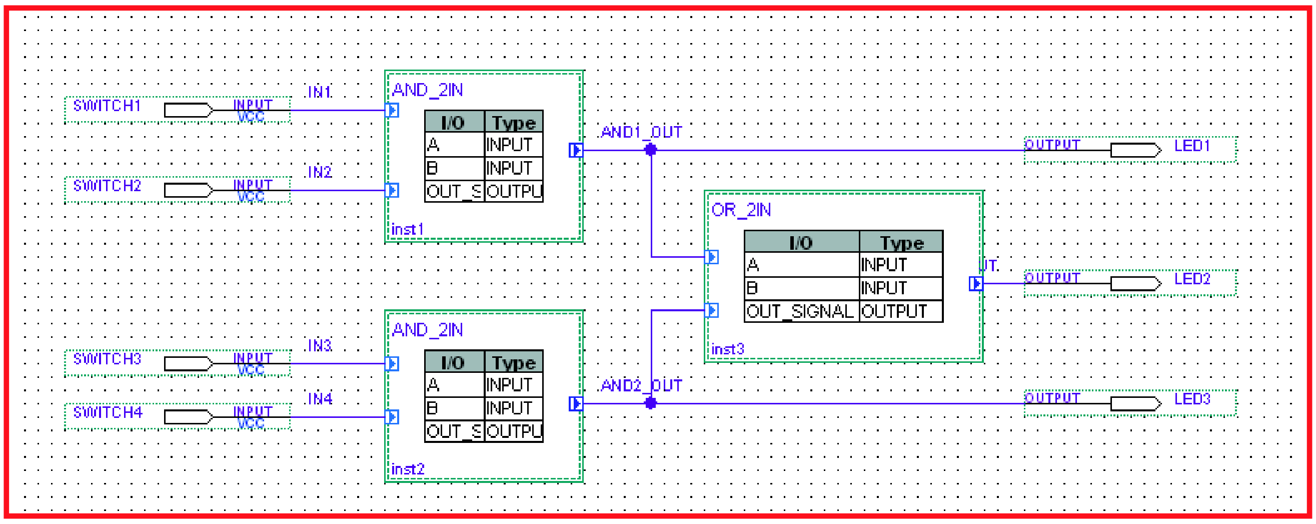


Figure 3. A simple design consisting of two VHDL files and a graphic file.

bone specification. ACK must go low in response to setting STB or CYC low.

Experiment

In the software, we send the same value to the output 20,000 times. This slows the running light down to the point that you can observe the effect visually.
ACK is delayed by 10 clock cycles in *wish_output*. If you increase this delay, the running light will also slow down. You can easily make this experiment yourself. Simply change line 87 of *wish_output.vhdl* to the following line:

IF (COUNT=200) THEN

Recompile the project and load it into the FPGA. Now the running light will run quite a bit slower than before. This proves that a slow slave on the Wishbone bus causes the master to run slower. This slowdown only occurs during read and write operations with the slave. All other instructions in the microcontroller are executed at full speed.

Multiple slaves

In practice, microcontroller circuits usually have more slaves than just a single I/O slave. All these slaves must communicate with the microcontroller via the same bus. This makes it necessary to add another piece of hardware that uses the address to determine which slave is being addressed.
In *ex14* the microcontroller is connected to two slaves. They are nearly the same as the slave used in the previous example. The address input has been omitted because there is only one write register and one read register. The slave also has eight inputs.

The job of the address decoder (*wishbone_decoder*) is to pass the signals to one of the two slaves depending on the address. We use two signals for this purpose (S1_SEL and S2_SEL), which go high when the right address appears on the Wishbone bus. The corresponding code for S1_SEL is:

```
S1_SEL<='1' WHEN ADR_I=x"8000"
ELSE '0';
```

In this case address 0x8000 was selected for slave 1.
A master–slave transaction can only occur when the CYC and STB signals are both high. It's easy to generate these signals now for slave 1:

```
S1_STB_O <= STB_I AND S1_SEL;
S1_CYC_O <= CYC_I AND S1_SEL;
```

The above lines of code ensure that the STB and CYC signals for slave 1 do not go high unless the slave is addressed. Finally, the data bus from the master to the slave has to be modified. If slave 1 is addressed, data must be sent from slave 1 to the master, and of course the same applies to slave 2. This is provided by the following line of code:

```
DAT_O_MASTER <= S1_DAT_I WHEN
(S1_SEL='1') ELSE
S2_DAT_I WHEN (S2_SEL='1')
ELSE x"00";
```

The same considerations apply to the ACK signal. It is passed on to the master in a similar manner.

Versatile

The handshake protocol makes the Wishbone bus very versatile. Besides the features already described, the bus can be extended with other signals such as an error signal, it can be configured so several masters can drive a single bus, and so on. If you want to know more about this, you can download the bus specification from the Opencores website.
There are also several other SoC busses. Most of them also use a handshake protocol, which makes it easy to implement a bridge between different bus systems.

Hierarchic VHDL

Up to now we have used graphic representations in our course to interconnect various blocks. However, you can also describe a complete design in Quartus using only VHDL.
We have prepared two examples to show how this can be done in VHDL. The first example (*ex15*) is a simple circuit consisting of two VHDL files and a graphic file. The graphic file is the 'top-level entity', which means it is the highest level in the hierarchy. The purpose of this file is to couple the subdesigns to each other and link the signals to the outside world (in other words, the FPGA pins). This is the method we have used up to now in all the examples. **Figure 3** shows this in schematic form.

The second example (*ex16*) contains the same design, but here the top-level document has been replaced by a VHDL file. The first statement in the *ex16.vhdl* file (see inset) is a standard ENTITY declaration. The inputs and outputs of this entity are ultimately connected to the pins of the FPGA, since this is our top-level document.
The input and output signals of the AND_2IN subdesign are described in lines 13-19. The signal names in this description must be the same as the names used in the AND_2IN.VHDL file. The same information must be provided for the OR_2IN subdesign.
Next we declare the signals used in this design. The signal names are the same as the names already used in example 15. In that example, these signals were drawn and labelled. In VHDL, this corresponds to signals of type STD_LOGIC.
A component with the name *inst1* is instanced in line 38. This reference is comparable to the designation 'IC1' or the like in a normal schematic diagram. The type of component to be placed here is described after the colon (:). In this case it is the component AND_2IN.
Finally, the inputs and outputs of this component are connected to signals starting with line 41.
If you compare the two examples, the principle involved will quickly become apparent.

Compatible

The advantage of describing a design entirely in VHDL is that the resulting source code is compatible with other CAD programs. Such a design can thus be used with the software of a different FPGA manufacturer without too much trouble. It's even possible to produce a real ASIC using exactly the same source code.

Another advantage is that it is often faster to modify a VHDL file than to make the corresponding changes in a graphic design, especially if there are a lot of signals between the various subdesigns.

(060025-5)

Web links

Opencores homepage:
www.opencores.org

Wishbone specification:
www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf

Listing ex16.vhdl

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ex16 IS
PORT
  (
    SWITCH1, SWITCH2, SWITCH3, SWITCH4 : IN STD_LOGIC;
    LED1, LED2, LED3 : OUT STD_LOGIC
  );
END ex16;

ARCHITECTURE arch OF ex16 IS
  COMPONENT AND_2IN
  PORT
    (
      A,B : IN STD_LOGIC;
      OUT_SIGNAL : OUT STD_LOGIC
    );
  END COMPONENT;

  COMPONENT OR_2IN
  PORT
    (
      A,B : IN STD_LOGIC;
      OUT_SIGNAL : OUT STD_LOGIC
    );
  END COMPONENT;

  SIGNAL IN1,IN2,IN3,IN4 : STD_LOGIC;
  SIGNAL AND1_OUT,AND2_OUT, OR_OUT : STD_LOGIC;

BEGIN
  IN1 <= SWITCH1;
  IN2 <= SWITCH2;
  IN3 <= SWITCH3;
  IN4 <= SWITCH4;

  inst1 : AND_2IN
  PORT MAP
  (
    A => IN1,
    B => IN2,
    OUT_SIGNAL => AND1_OUT
  );

  inst2 : AND_2IN
  PORT MAP
  (
    A => IN3,
    B => IN4,
    OUT_SIGNAL => AND2_OUT
  );

  inst3 : OR_2IN
  PORT MAP
  (
    A => AND1_OUT,
    B => AND2_OUT,
    OUT_SIGNAL => OR_OUT
  );

  LED1 <= AND1_OUT;
  LED2 <= OR_OUT;
  LED3 <= AND2_OUT;

END;
```

FPGA Course (6)

Part 6: connecting a keyboard

Paul Goossens and Andreas Voggeneder (FH Hagenberg)

You no doubt have an old PS/2 keyboard gathering dust somewhere. Now you can put it to good use again as an input device for the FPGA prototyping board. This instalment of our FPGA course tells you how.

Most prototyping systems use an RS232 link to send and receive data. This link normally goes to a PC running a terminal emulator program. The user thus uses the monitor of the PC as the actual output device and the keyboard as the input device.

You can dispense with the PC if you connect a monitor and keyboard directly to the prototyping board.

PS/2

Until recently, PC keyboards were always connected to the PC via the PS/2 bus. The FPGA prototyping board also has two PS/2 ports. Besides the power supply lines, a PS/2 port has a data line and a clock line. Both of the these lines are bi-directional, and here they are connected directly to several pins of the FPGA on the circuit board.
Data is transmitted from a device (such as a keyboard) to the host (in this case the FPGA) as follows. First, the device transmits a start bit. The start bit is always a 0. It then sends the eight data bits starting with the least significant bit. Next comes a parity bit with odd parity, which means the parity bit is a 1 if an even number of 1 bits are present in the transmitted byte. Finally there is a stop bit, which is always a 1.
The clock signal for the transmission is generated by the device. The signal level on the data line changes only when the clock signal is high (see **Figure 1**).

From host to device

Communication from the host to the device is somewhat more complicated. First, the host must indicate that it wants to transmit data. It does this by setting the clock line low, which terminates any communication that may already be in progress. After a brief delay, it sets the data line low. Finally, it sets the clock line high again.
As a result, the device knows that the host will be transmitting data. In response to this, the device generates a clock signal. The host then sends one data bit for each clock pulse. This starts with a start bit (0) followed by eight data bits. These data bits are also sent starting with the least significant bit. The next bit is a parity bit (odd). Finally, the host sends a stop bit (1), but the communication session is not yet complete. The final action is that the device sends an 0 as an acknowledgement (ACK) if the data was received correctly (see **Figure 2**).

Software versus hardware

Of course, it is possible to implement the PS/2 protocol in software. This approach was taken with the I^{2}C bus in a previous instalment. A disadvantage of this method is that the processor must spend some of its time processing the signals. It also doesn't make the software any simpler.
An alternative approach is to design a hardware interface that looks after generating and processing the signals. Here we describe how the PS/2 interface can be implemented in hardware. With a hardware interface, the microcontroller (8051) does not have to be concerned with the details of how the PS/2 interface works.
Before examining the hardware the implementation of the PS/2 interface, let's have a look at the T8052 microcontroller.

T8052

We've already used the T51 softcore processor several times in this course. The microcontroller consists of a processor portion called T51 and several peripheral devices, such as a UART, timers, and so on. Like every MCS51 processor, the T51 uses a special system bus to drive the peripheral devices. The registers on this bus are called Special Function Registers (SFRs). The original microcontroller (8051) did not use all available addresses. This was done intentionally to allow room for other peripheral devices. This possibility can be used to add a PS/2 interface to the microcontroller as shown in our *ex17* example.

The code in *T51_Glue.VHDL* looks after decoding the addresses of the SFRs. We add the following lines here:

```
ps2_data_sel      <= '1' when IO_Addr = "1011001" else '0'; -- 0xD9
ps2_data_wr       <= '1' when IO_Addr_r = "1011001" and IO_Wr = '1' else '0'; -- 0xD9
ps2_ctrl_stat_sel <= '1' when IO_Addr = "1011000" else '0'; -- 0xD8
ps2_ctrl_stat_wr  <= '1' when IO_Addr_r = "1011000" and IO_Wr = '1' else '0'; -- 0xD8
```

This bit of code makes register *PS2_DATA* available at SFR address 0xD9. Register *PS2_CTRL_STAT* is available at address 0xD8.

PS/2 interface

The actual PS/2 interface is described in *PS2Keyboard-a.VHDL*. The interface with the SFR bus is defined starting with line 218. Line 218 is part of a process that is evaluated on a rising clock edge. If a rising clock edge occurs when *data_wr_i* is high, the content provided by the processor is loaded into register *CmdReg*. In addition, a 1 is loaded into bit 8 (which is the ninth bit!).

If *ctrl_stat_wr_i* is high, several registers are loaded with the corresponding bits present on the data bus. As you can clearly see, the interface is returned to the quiescent stat if bit 7 is a 1. It waits for data from the device when it is in this state.

The content of *SFR_Data_o* is read by the processor in case of a read operation. Line 236 causes the content of the internal *DataReg* register to be passed if it is selected by the *data_sel_i* signal. Otherwise the content of the status register is passed.
The actual data transfer to the processor is handled by the *T8051.VHDL* file.

The remainder of the *PS2Keyboard-a.VHD* file describes the communication between the FPGA and the PS/2 bus. The comments in the source code should make it reasonably easy to understand how this works. If you can't figure it out, you can always use the simulator to examine the interactions between the internal signals.

Example

The first example of how to use the PS/2 interface is *ex17*. Start by connecting a keyboard to the connector labelled 'KEYBOARD', and then use the accompanying configuration file to configure the FPGA.
If everything goes as it should, a message will appear on the LCD. From now on, all data sent by the keyboard will

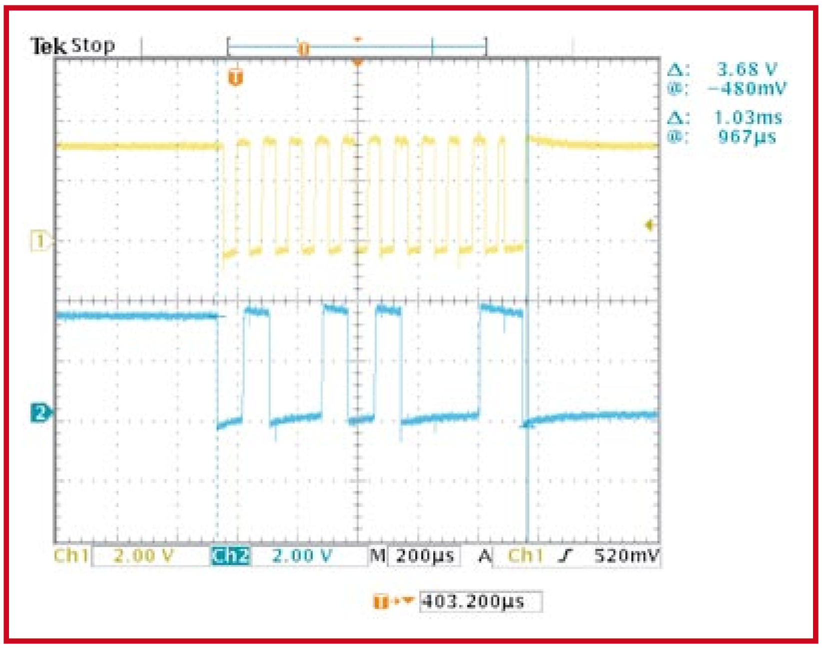


Figure 1. Data transmission from the device to the host. The upper trace shows the clock signa

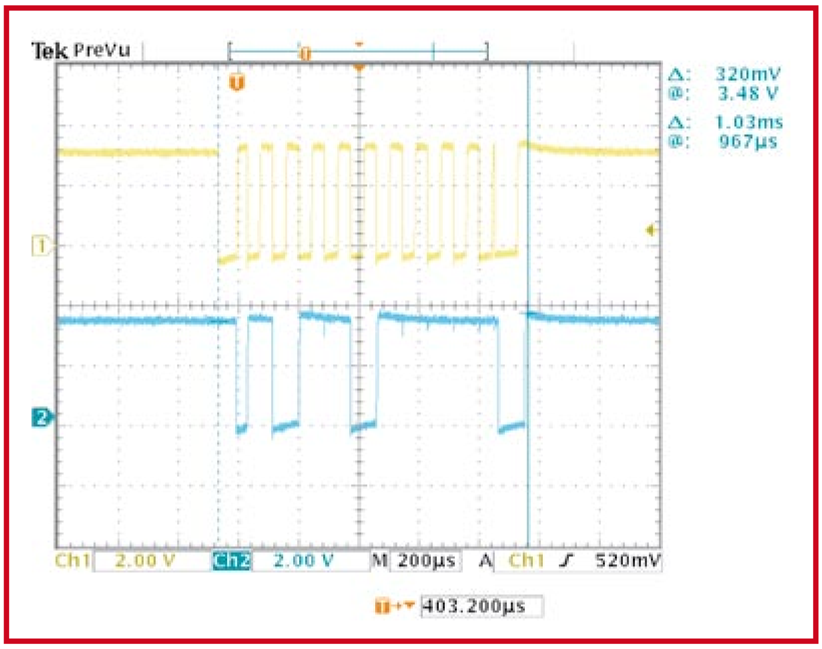


Figure 2. A data block transmitted from the host to the device.

be shown on the LCD in hexadecimal form. The keyboard will not send any data when it is not being used.
Start by pressing any desired key of the keyboard. The corresponding data will be sent immediately to the FPGA to report this action. If you hold a key pressed for longer than a certain interval, the keyboard will transmit the corresponding code repeatedly.
Let's suppose you pressed the 'w' key. According to the LCD, the keyboard sent the code 0x1D. The keyboard actually sends two codes when a key is released. The first code it sends is '0F', which indicates that the user has released a key. After this, it sends the code of the key concerned.

Firmware

How is all of this handled in the software? First, the LCD is initialised as usual. After this, the *init_kb()* routine is called. This is located in *kb.c*. Initial values are assigned to the variables here. After this, the final action is to call *InitKbd()*, which is located in *fpga_lib.c*. It shows how data can be sent to the keyboard and read from the keyboard. The first action is to send the code 0xFF to the keyboard to perform a reset. The keyboard returns an ACK (0xFA) if this code is received correctly. If the keyboard returns 0xFE, an error has occurred during the communication process.
If everything has gone as it should up to now, the keyboard will execute a self-test. This is indicated by briefly lighting the LEDs on the keyboard. If this test is completed successfully, the keyboard sends 0xAA as a sign that it is OK. From this point on, the keyboard operates the way it is supposed to.
The rest of the code has been kept very simple. Whenever data is received, the interrupt routine *ext_int2_isr()* causes the data to be stored in a buffer.

Scan codes

OK, now you can communicate with the keyboard. Keypresses are also being sent to the processor, but what you actually want to receive is normal ASCII characters. For this reason, we extended the interrupt routine of *kb.c* in a new example (*ex18*). The new version of the interrupt routine converts the scan codes into ASCII characters.
Our more inquisitive readers will have certainly also tried the CapsLock key in the previous example. If you did so, you would have seen that the associate LED did not respond at all. That's also the way it should be, since the LEDs are driven by the host. The routine *Set_LED()* in example 8 shows how you can drive the keyboard LEDs. In this example, the number of typed characters is counted and the result is indicated by the LEDs on the keyboard. As only three LEDs available, only numbers in the range of 0 to 7 can be indicated.

This instalment clearly shows that the full power of the FPGA can only be utilised if you make use of the advantages of configurable logic relative to conventional microcontroller systems. Of course, you could also implement the entire PS/2 protocol in software, but that would cost extra processing time. With the hardware implementation described here, the processor does not have to look after handling the electrical signals on the PS/2 bus, so it has more time for other tasks.
The I^{2}C protocol, which was implemented in software in a previous instalment of this course, could also be implemented in hardware in a similar manner.

In the next instalment, we will add a fully functional VGA output to the microcontroller system of this month's instalment. Naturally, it will also be implemented in hardware.

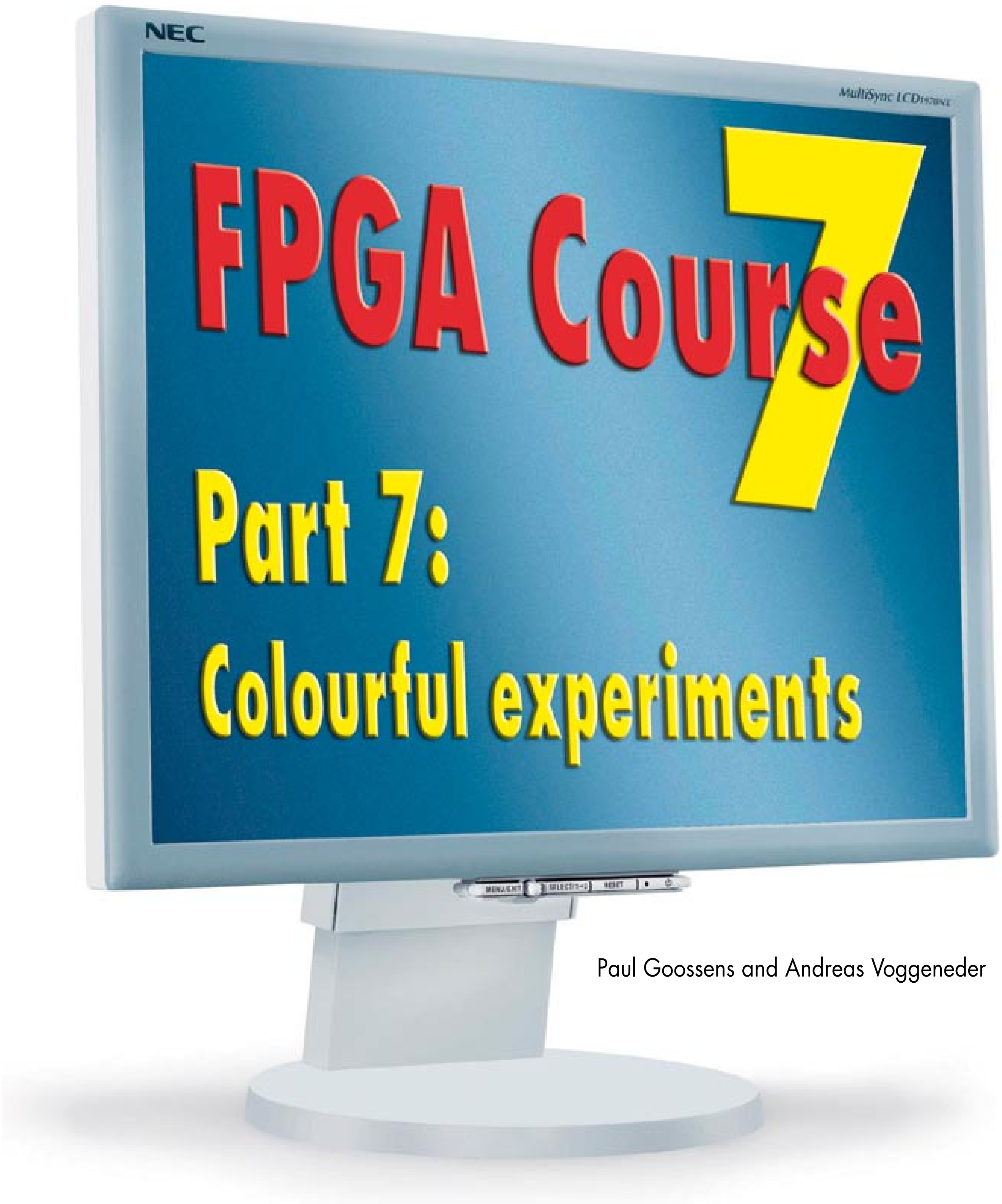
(060025-VI)

Join the FPGA Course with the Elektor FPGA Package!

The basis of this course is an FPGA Module powered by an Altera Cyclone FPGA chip, installed on an FPGA Prototyping Board equipped with a wealth of I/O and two displays (see the March 2006 issue).

Both boards are available ready-populated and tested. Together they form a solid basis for you to try out the examples presented as part of the course and so build personal expertise and know-how in the field of FPGAs.

Further information may be found on the shop/kits & modules pages at www.elektor.com



FPGA Course 7

Part 7: Colourful experiments

Paul Goossens and Andreas Voggeneder

A picture is worth a thousand words, which is probably why alphanumeric displays have given way to graphic displays in the latest mobile telephones. Although our FPGA prototyping board is not a small as a mobile phone, it can still generate video. That's why it has a VGA port.

As you have probably already guessed, the VGA port can be used to display video imagery on a VGA monitor. We've kept the VGA hardware on the prototyping board quite simple. Three D/A converters are implemented using several resistors. Each converter has a resolution of 3 bits. They generate the red, green and blue (RGB) signals. The horizontal and vertical sync signals come straight from the FPGA.

VGA signals

Naturally, all five VGA signals are generated by the FPGA. Before you delve into the details of the implementation, you need to know how theses signals ultimately create a video image.
Here we limit ourselves to generating an image with a resolution of 640 × 480 pixels and a frame rate of 60 Hz. Every VGA monitor can easily display this standard resolution.
Each frame consists of a number of lines of video information. A typical video line is shown in **Figure 1**. A video line can be decomposed into a certain number of pixel intervals. One image pixel can be generated in each pixel interval by the combined R, G and B signals.
The R, G and B signals must be low during the first 96 pixel intervals. The HSYNC signal is also low during this time. The monitor recognises the start of a new line from this set of signal states.
This is followed by 48 pixel intervals during which HSYNC is high while the colour signals (R, G and B) remain low.
The actual image line starts next. For each pixel interval, the R, G and B signals determine the colour and intensity of the associated pixel. As you might expect, this portion of the line consists of 640 pixel intervals – one for each pixel in a horizontal line. At the end of the line, there are another 16 pixel intervals during which the colour signals are all low. HSYNC remains high during this time. A single video line consists of 800 pixel intervals in total.
It takes another 479 video lines after this one to send all the pixels to the monitor. They are followed by ten video lines that are fully black (RGB = 0). These lines are followed by another two lines (also black) with the VSYNC signal low. This tells the monitor that the next frame is coming. Finally, there are another 33 black video lines with VSYNC high again. The total number of lines in a frame is thus 525.

Example 1

The pixel frequency (dot rate) in this VGA mode is 25.167 MHz. The length of one pixel interval is thus the inverse of this frequency, which is approximately 39.7 ns. For this example, we can round off the frequency to 25 MHz, which is easy to generate from the 50-MHz clock signal already present on the board. This differs from the desired frequency by approximately 0.7%, which is fairly small and lies within tolerance.
Example 19 (*ex19*) generates a VGA signal that produces a colourful image on the monitor.
The code in file *ex19.vhd* contains a process named *vid*, which generates the *VSYNC* and *HSYNC* signals. This process also keeps track of which column and line of the screen image is 'at bat'. This information is stored in the *Col* and *Line* registers.
VidEn indicates whether a pixel is being generated or the R, G and B signals must be set to 0. Note that the *Col* and *Line* signals are only valid when *VidEn* is set to 1.
The *qen* signal divides the 50-MHz clock signal by 2.
The rest of this process is very simple. Try to figure out for yourself exactly how it works!
The *fractal* process uses the current coordinates (*Col* and *Line*) to determine the intensities of the red, green and blue signals. The algorithm used here is actually not significant – its only purpose is to generate a particular pattern on the monitor screen.
After you compile the example and load it in the FPGA, connect a VGA monitor

VGA functions

Various low-level routines that make it easier to use the VGA interface are located in fpga_lib.c. The most important routines are:

Init_screen(): initialises the VGA controller

SetCurrentColor (Color):
sets the new foreground colour

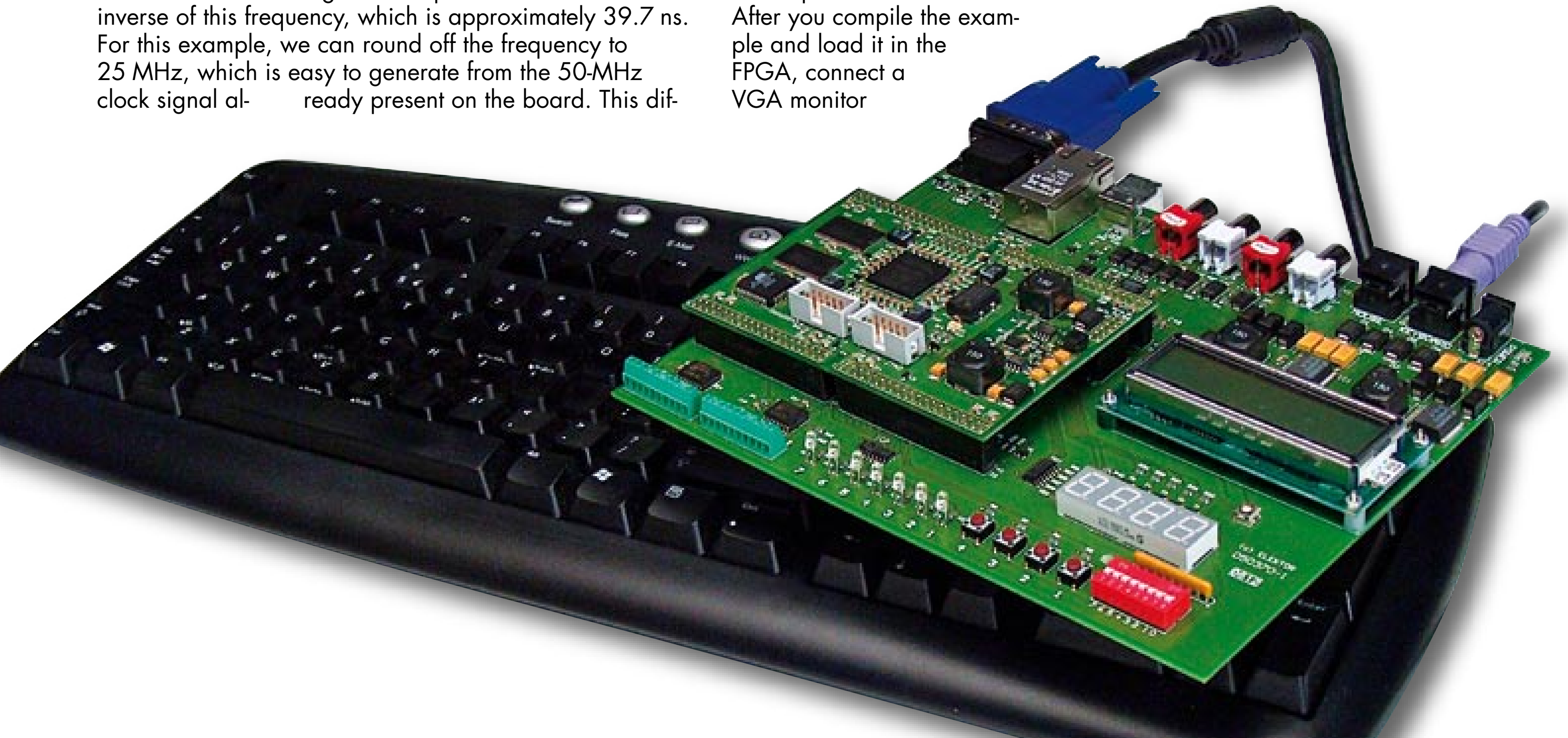
SetCurrentBkColor (Color):
sets the new background colour

Gotoxy(x,y): places the cursor at the specified position

WriteScreen(Text): writes text to the screen starting at the current cursor position

Clrscr(): erases the screen

Several constants that can come in handy when you're programming are located in infpga_lib.h and fpga_reg.h. They include constants for the various colours.



DIY experiments

Example 19 forms a good basis for some simple DIY experimenting. Be sure to make a copy of the file first so you can restore things to the original situation.

To get you started, try making the following changes:

- Delete lines 147–149
- Replace the graph process with the following code:

```
graph : process (clk, clr)
begin
if clr = '1' then
  Red  <="00";
  Blue <="00";
  Green <="00";
elsif (clk'event) and (clk = '1') then
  if qen = '1' then
    if VidEn = '1' then
      if Col(3 downto 0)="1000" then
        Green<="11";
      else
        Green<="00";
      end if;
    else
      Red   <= "00";
      Green <= "00";
      Blue  <= "00";
    end if;
  end if;
end if;
end process graph;
```

The above modification is quite simple. Now try making your own modification to add a blue horizontal line to the image. (Tip: the Line register keeps track of the currently active line.)

Once you've mastered this, how about generating a dashed line instead?

Timing

Signal timing is very important in digital circuit design. For instance, changes to a design can affect its maximum operating frequency.

Registers (flip-flops) are often used in digital designs. They impose timing requirements on the associated signals to ensure that they work properly. The most commonly used timing parameters are:

Tsu (setup): specifies how long the data must be valid at the input of the flip-flop before the clock pulse arrives

Tco (clock to output): indicates how long it takes for the data on the input to appear at the output after the clock pulse arrives

Th (hold): specifies how long the data on the input must remain unchanged after the clock pulse arrives

An operation of some sort (such as AND, OR or even addition) usually occurs between the output of one flip-flop and the input of the next one. These combinational circuits also have delays between their inputs and the outputs. The interconnects inside the FPGA also cause delays.

These delays and requirements have certain tolerance ranges. For example, a flip-flop with a specified Tco of 2 ns may be able to work even faster in practice. The longest possible time is called the 'worst-case time', and the best possible time is called the 'best-case time'.

After Quartus compiles a design, it analyses the timing of the design. This yields a figure for the maximum operating frequency of the design. To arrive at this result, the program calculates the maximum delay between the clock pulse and when the signal arrives at the flip-flop data input for each signal line. This delay is often called the 'arrival time'. The Tsu of the flip-flop concerned is included in this delay. Worst-case times are used for this calculation.

The longest delay determines how quickly successive clock pulses can follow each other. If you set a desired minimal clock rate in Quartus (refer to Quartus Help), it can determine whether the design meets your requirement.

Quartus also analyses the Th requirement of the flip-flops. Here again it calculates the delay between the clock pulse and when the data signal arrives at each flip-flop. The best-case times are used for this second analysis. They must always be greater than the Th requirement of the flip-flop concerned.

You can view the results of the timing analysis in Quartus in the Compilation Report. This report includes a section named 'Timing Analyzer', which provides detailed information about the timing of your design. The term 'slack' appears frequently in this report. Slack designates the difference between the calculated delay and the desired delay based on your desired clock frequency. A positive slack for Tsu indicates that the signal arrives at the data input of a flip-flop earlier than the required time (by the amount of the slack value). In the case of Th, the slack indicates how much longer the signal remains unchanged than the required (minimum) time. Positive slack is good news, and negative slack is bad news.

If your design is too slow, you have two options. The first option is to see whether Quartus can help you compile the design better. To do this, select 'Timing Optimization Advisor' under Advisors in the Tools menu. It will help you configure the Quartus compiler to produce a better result.

The second option is to examine the timing report to see which signals are slowing things down. Once you know this, you can try to modify your design to make these signals somewhat faster. You might find that using a different algorithm helps.

There are also techniques that can be used to adapt a design to a higher clock frequency, but that's a different subject!

to connector K9 on the prototyping board. If everything went right, you will be treated to a pretty design on the monitor.

Dynamic

Conjuring up a nice figure like this on your monitor is a lot of fun, but you'll get tired of looking at it after while. It would be better (and more useful) to be able to display information on the screen. If you combine this with a bit of intelligence, you might just have something.
This is the purpose of **Example 20** (*ex20*). Here the trusty T51 microcontroller has been extended to include the PS/2 interface described in the last instalment and a VGA interface. The VGA interface is linked to the microcontroller via the XRAM bus. From the perspective of the microcontroller, this interface is simply a block of memory starting at address 0x8000 and a set of eight memory locations starting at address 0xAA00.
As you can see from the size of the *Graffikkarte-a.vhd* file, this is a fairly sizeable design. For this reason, we don't intend to describe it in detail here. Based on what you've already learned from the previous instalments, you should be able to figure it out on your own.
However, it is good idea to say something about how the interface is driven by the software.

Driver

Thanks to the C files provided with the course, the driver routine that runs in the microcontroller is fairly simple. The first thing *ex20.c* does is to call *Initscreen()*. This causes the VGA controller to be configured properly so it is ready to be used.
Next, *SetCurrentColor* and *SetCurrentBkColor* set the image colours. Each character can be displayed in a different colour. Theses functions only change the colour that is used when the data is written to the screen.
Next, *Gotoxy(28,1)* positions the (invisible) cursor to coordinates (28, 1).
Writescreen("text") places the first text on the screen. Here the colour is determined by the previous *SetCurrentColor* and *SetCurrentBkColor* functions.
Finally, the *putchar2()* function displays individual characters on the screen. This function also uses the previous colour information.

Use

A welcome message will appear on the monitor after you configure the FPGA and connect a PS/2 keyboard. You can then use the keyboard to type in text, which will appear immediately on the monitor. You can use the function keys (F1 to F4) to change the colour of the text.
As an exercise to familiarise yourself with the VGA interface, you can experiment with modifying the software. For instance, try integrating the I^{2}C interface described in one of the previous instalments so the output of the A/D converter can be displayed on the monitor. Another idea would be to display the scan codes received on the PS/2 interface instead of the decoded characters. If you can't get that to work, you can always ask other readers of *Elektor Electronics* for help via the Forum on our website.

Let's have fun!

After all this hard work, it's time for a bit of relaxation. As a special treat, we've developed a fun application for you. The hardware is exactly the same as for the previous example (*ex20*). The only difference is in the software, which has been modified 'slightly'. **Example 21** (*ex21*) implements the familiar 'four in a row' game, but this time entirely in the FPGA.
This example also shows how you can define your own characters – that is, if you can tear yourself away from the game long enough to have a look at the source code.

(060025-7)

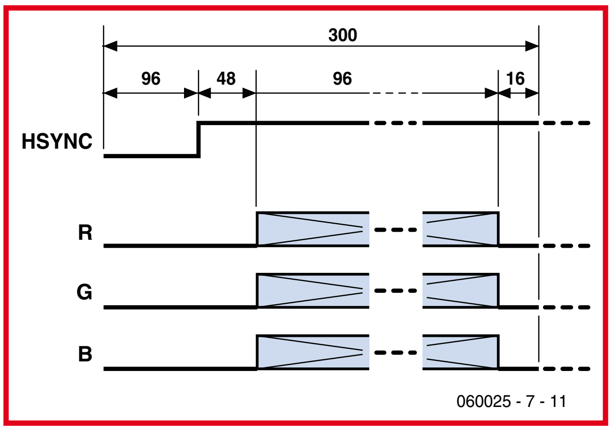


Figure 1.
Structure of a video line.

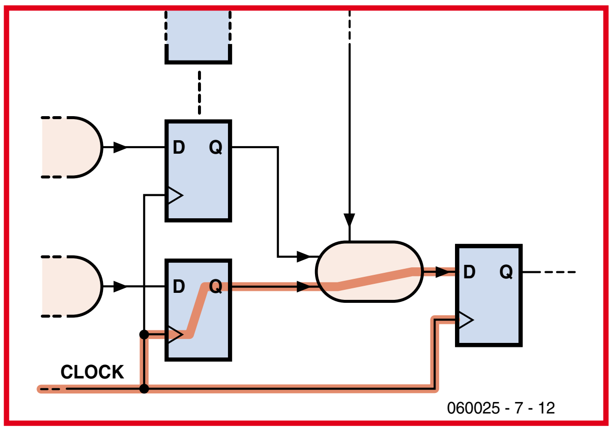


Figure 2.
Signal flow for a timing analysis.

Join the FPGA Course with the Elektor FPGA Package!

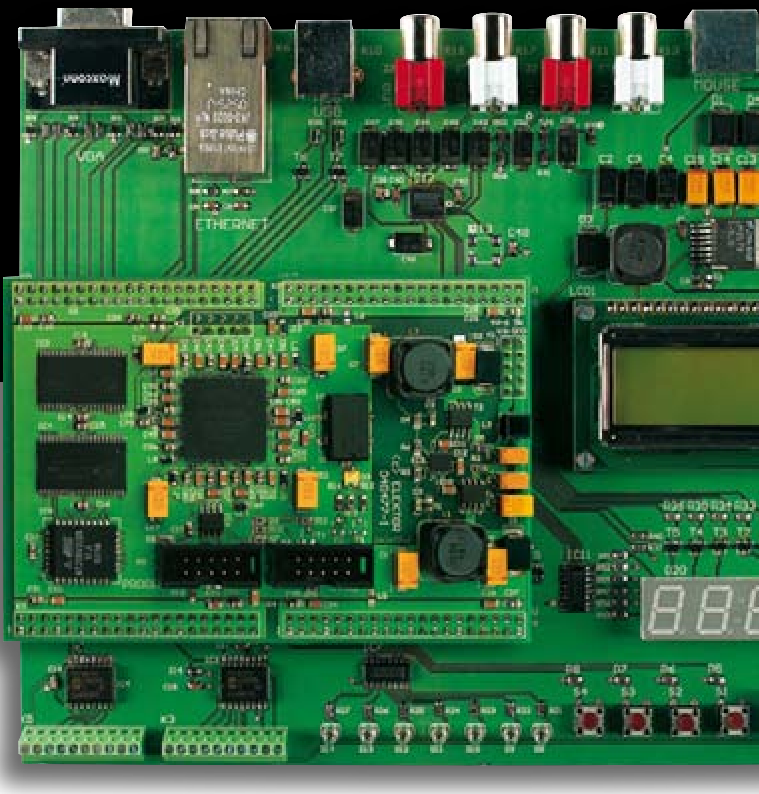
The basis of this course is an FPGA Module powered by an Altera Cyclone FPGA chip, installed on an FPGA Prototyping Board equipped with a wealth of I/O and two displays (see the March 2006 issue).

Both boards are available ready-populated and tested. Together they form a solid basis for you to try out the examples presented as part of the course and so build personal expertise and know-how in the field of FPGAs.

Further information may be found on the shop/kits & modules pages at www.elektor.com

FPGA Course (9)

Part 9 : The final act



Paul Goossens &
Andreas Voggeneder

We've used nearly every I/O port of the prototyping board at least once in this FPGA course. We say 'nearly' because we haven't used the Ethernet port up to now. The final instalment of the course sets this right with a full-fledged webserver application.

The prototyping board has an Ethernet connector and associated Ethernet PHY. This makes it possible to connect the FPGA to an Ethernet network, and ultimately to the Internet.

Ethernet

An Ethernet network can be used to interconnect several computers and other devices. The Internet is the largest and best-known example of an Ethernet network. This article shows you how to connect the FPGA module to an Ethernet network. This can be a small local area network in you home, but it can also be the Internet.

Various sorts of logic in the FPGA module are necessary for this purpose. Although the FPGA prototyping board has an Ethernet connector and Ethernet PHY, it takes quite a bit more than this to make an Ethernet connection. The task of the Ethernet PHY is to transmit and receive data in serial format according to the Ethernet standard.
Besides this PHY, you also need a media access controller (MAC). The functions of the MAC component are:

- automatic data flow control
- detecting data collisions and retransmitting data if something goes wrong (CSMA/CD)
- adding a CRC checksum
- adding a preamble on transmission and deleting the preamble on reception

For the experts among our readers, this is Layer 2 of the OSI model. Layer 1 consists of the combination of the PHY, the transformer and the connector.
You also need a microcontroller with the right software to add the TCP/IP protocol and the application to the system. Just as in the previous instalments, we use the T51 core for this purpose.
The software is based on the software for our MSC1210 webserver. We used the uC51 compiler and associated TCP/IP stack to compile the code. This compiler is available from Wickenhäuser (see inset).

MAC

The media access controller (MAC) comes from www.opencores.com, and it is written in Verilog. The MAC uses a 32-bit data bus, while the T51 works with an 8-bit data bus. The *ethernet_verilog.vhd* file was modified so these two busses can be connected together.
The details of how this MAC works fall outside the scope of this article. In any case, you don't need to understand the details to use the MAC properly. With the help of the software provided with the course, you can quickly write your own applications for the Ethernet port.

First example

Our first example (**ex24**) uses the T51 core, the MAC core, and some peripheral logic. The Ethernet core eats up a fair amount of the scarce memory in the FPGA, so the maximum amount of ROM available to the T51 is 14 kB. This is a rather small amount of memory for a webserver. Among other things, the entire TCP/IP stack, the control logic for the MAC and the PHY chip, and the webserver application must all be fitted into this 14 kB.
The RAM consists of an external SRAM IC that is already present in the FPGA module. This IC is connected to the microcontroller via the wishbone bus. The memory layout is shown in **Figure 1**. The first 52 kB of RAM are provided by the SRAM IC. Addresses 0xD000 through 0xDFFF are reserved for the Ethernet MAC. Finally, the remaining 8 kB is used as buffer memory for transmitted and received data packets.
Now the FPGA prototyping board has to be connected to a network. You can use a router or a switch for this. The TCP/IP address of this application is set to a fixed value of 192.168.0.1. Ensure that your network is configured such that address 192.168.0.1 is valid on the network. If you use a router, the TCP/IP address of the router must be configured such that the first three numbers are 192.168.0, and the last number must **not** be 1 since this number is already used by the FPGA module. If the router has a DHCP server, you will have to adapt it to the new TCP/IP addresses.
If everything goes the way it should, after you program the FPGA you can use the web browser on your PC to view the web pages in the FPGA by entering the following address (URL):

http://192.168.0.1

If this doesn't work, check the network settings of your PC and router.

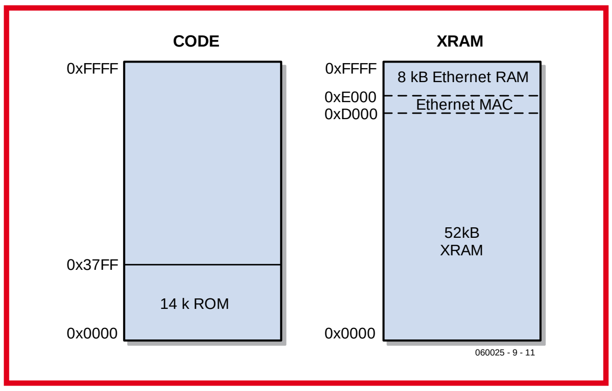


Figure 1.
Memory layout for example ex24.

Even more

The webserver of example 24 has just enough oomph to publish a couple of HTML pages using the FPGA module. This is rather on the meagre side, and it certainly isn't what we really want. The limiting factor here is program memory.

If you can expand this memory, you can then write more 'meaningful' applications for the webserver. Fortunately, the FPGA module has some extra RAM and flash memory. If you use the flash memory to store the program code, in theory you can use programs up to 128 kB in size. This brings us up against a problem: you can program the memory in the FPGA using Quartus, but you can't program external memory.

Bank registers

Bank_En (0xF8)

Bit 4: 1 = VGA enabled; 0 = VGA disabled
(0x8000–0xBFFF)

Bit 5: 1 = SRAM always bank 0;
0 = SRAM depends on Bank_Sel

Bit 6: 1 = SRAM enabled; 0 = flash enabled

Bit 7: 1 = Ethernet disabled; 0 = Ethernet enabled
(0xD000–0xFFFF)

Bank_Sel(0xF9)

This register selects the current bank.

Features of example 25

- 8052 controller running at 25 MIPS
- 10/100 Mb/s Ethernet interface
- 256 kB RAM
- 128 kB flash
- VGA interface
- PS/2 interface
- Firmware updating via BOOTP

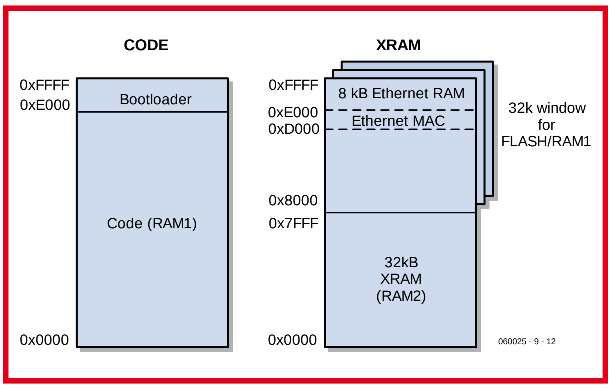
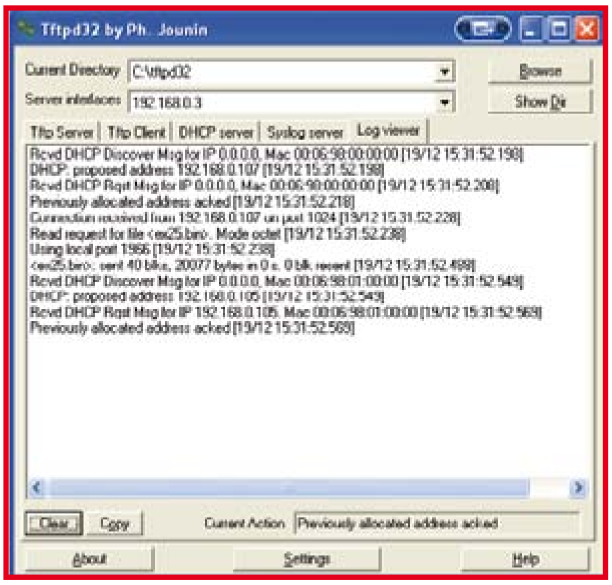


Figure 2.
Memory layout for example ex25 when the bootloader is NOT executed.



Current Directory: C:\tftpd32
Server interfaces: 192.168.0.3

```
Rcvd DHCP Discover Msg for IP 0.0.0.0, Mac 00:06:98:00:00:00 [19/12 15:31:52.198]
DHCP: proposed address 192.168.0.107 [19/12 15:31:52.198]
Rcvd DHCP Rqst Msg for IP 0.0.0.0, Mac 00:06:98:00:00:00 [19/12 15:31:52.208]
Previously allocated address acked [19/12 15:31:52.218]
Connection received from 192.168.0.107 on port 1024 [19/12 15:31:52.228]
Read request for file <ex25.bin>. Mode octet [19/12 15:31:52.238]
Using local port 1986 [19/12 15:31:52.238]
<ex25.bin>: sent 40 blks, 20077 bytes in 0 s. 0 blk resent [19/12 15:31:52.498]
Rcvd DHCP Discover Msg for IP 0.0.0.0, Mac 00:06:98:01:00:00 [19/12 15:31:52.549]
DHCP: proposed address 192.168.0.105 [19/12 15:31:52.549]
Rcvd DHCP Rqst Msg for IP 192.168.0.105, Mac 00:06:98:01:00:00 [19/12 15:31:52.569]
Previously allocated address acked [19/12 15:31:52.569]
```

Current Action: Previously allocated address acked

Figure 3.
DHCP server settings in the program tftpd32.

μC-51

The software used in examples 24 and 25 was generated using the Wickenhäuser μC51 C compiler (www.wickenhaeuser.de).

We used this compiler on account of the TCP/IP stack that comes with the compiler. The free version of the compiler can generate programs with up to 8 kB of code, which is not enough for our software. If you wish to modify and recompile the software, you will need the full version of the compiler. The price of the compiler was € 84.50 (approx.£ 59) at the time of writing.

Up to now we have not been able to find a suitable TCP/IP stack for SDCC. If at some point we manage to find an open-source TCP/IP stack that is suitable for SDCC, we will make this known via our FPGA forum.

Naturally, we would also be grateful to hear about any suitable TCP/IP stack written by one of our readers or encountered by our readers somewhere on the Internet.

Boot loader

In example 25 (**ex25**), we connect the microcontroller to the flash memory and both SRAM chips. We also place a boot loader program in the internal ROM. If you set dipswitch 6 to the ON position and then reset the circuit by pressing button 1, this program will run.
The boot loader uses the DHCP protocol to try to obtain a valid TCP/IP address, such as the address of the router or PC. If this succeeds, it then uses the BOOTP protocol to request data. The data that the microcontroller receives in response to this request is stored in the external flash memory.

If the system is started up with dipswitch 6 in the OFF position, the content of the flash memory is copied to the SDRAM (IC4 on the prototyping board). The program code in the SDRAM is then executed. The reason for copying the program code is that the flash memory is relatively slow (access time 150 ns). The SDRAM has an access time of 10 ns and is thus much faster. The long access time of the flash memory makes it necessary to insert three wait states for each read instruction. As a result, the microcontroller effectively runs at 6.25 MHz instead of 25 MHz. This restriction is not necessary if the program code is executed from the SDRAM, so the microcontroller is free to operate at full speed.
It is not essential to store the program code directly in the flash memory when you are developing new software. For this reason, we added the option of storing program code in external SRAM (IC4) instead of flash memory. To do so, set dipswitch 7 to the ON position. The program will then be executed from SDRAM after the firmware has been downloaded.

RAM

The RAM memory is also expanded in example 25. To help clarify the following explanation, the memory layout of example 25 is shown in **Figure 2**. This layout is only valid if the boot loader is ***not*** running. A different memory layout is used when the boot loader is running, so that data can be written to the SRAM chip (IC4).
The memory layout can be configured using two SFR registers: Bank_En (0xF8) and Bank_Sel (0xF9).
Bit 4 of Bank_En determines whether VGA memory is addressable. If it is set to '1', VGA memory can be accessed in the address range 0x8000–0xBFFF.
Bit 7 of the same register determines whether the memory of the Ethernet interface is accessible as XRAM in the address range 0xD000–0xFFFF.
Bit 6 selects either flash or RAM for the upper 32 kB. If bit 6 is set to '0', flash memory can be accessed in the memory range 0x8000–0xFFFF. If this bit is set to '1', the content of the SRAM (IC3) is accessible in this address range.

Note that the VGA interface and the Ethernet interface have the highest priority. If either of these interfaces is enabled, the associated addresses cannot be used for the SDRAM or flash memory.
Bank-switching capability
is also provided to enable using the entire SDRAM and flash memory. You can use the Bank_Sel register to choose which part of the flash memory or SDRAM is selected. In case of the flash memory, this register determines which of the 32-kB blocks is currently in use. With the SDRAM, this register determines which 64-kB block is currently in use.

Down to work

That's enough explanation – now let's try example 25 in practice! Besides the usual software, you need one more program file for this, which is named *tftpd32*. It can be downloaded from the Internet free of charge (http://tftpd32.jounin.net). This program file contains a DHCP server and a BOOTP server. You need both of them in order to load software into the flash memory.
To start off, you have to ensure that your PC uses a fixed IP address. As already mentioned, we selected address 192.168.0.1 for this.

If you router has a DHCP server, you must disable it, since only one DHCP server is allowed in a network. If you use a router, it must also be configured with a fixed IP address (in our case, we used address 192.168.0.2 for the router).
After starting up *tftp32*, you have to configure the right DHCP server settings. In our example (**Figure 3**), we reserved addresses 192.168.0.105 through 192.168.0.115 for dynamic addresses. We also selected file *ex25.bin* as the BOOTP file.
The next step is to use Quartus to configure the FPGA. Make sure that dipswitch 6 is ON and dipswitch 7 is OFF. After the configuration is completed, the boot loader will start running and request an IP address. It will then use BOOTP to ask for a program. You can see this in the Log Viewer of the *tftp32* window.
If everything goes the way it should, the IP address of the FPGA will appear on the LCD screen. In our case, the FPGA was assigned address 192.168.0.107. Now you can locate the FPGA module in your web browser at address http://192.168.0.107. If your FPGA module is assigned a different address, simply enter the corresponding address in your web browser.

Pitfalls

The file that is sent vial BOOTP (the firmware for the microcontroller) must be a binary file. The µC51 compiler will generate a binary file automatically if you use the 'make' files provided with the course.
If you want to use software generated by a different compiler, make sure that it generates a binary output file. Most compilers can generate binary files. If you compiler can't do this, there are several freeware programs available that can convert hex files into binary files.

Conclusion

You can refine and extend the final embedded system of example 25 to generate your own applications. The features of this system are quite impressive, especially considering that the complete source code is available in VHDL. You can use this source code to enlarge your knowledge of VHDL.
We hope this course has helped you carry out your first practical experiments with programmable logic and VHDL.
The course is naturally far from complete – the subject of FPGAs and VHDL is so broad that you could easily write a whole set of books about it. Nevertheless, we think the course provides enough material – in part because of the numerous examples – to enable you to start developing your own designs in VHDL.

We have set up a separate topic in the forums section of our website so users of the FPGA module can share their experience. There's a good chance that you can pick up new ideas there, and you can also ask other readers for help with any problems you may encounter.
In any case, we hope you have a lot of fun (and learn something) with the FPGA module!

(060025-9)

DHCP and BOOTP

The DHCP protocol can be used to assign dynamic addresses to devices with Ethernet ports. Each device in a network requires a unique IP address so it can communicate with other devices. This can be achieved by manually assigning each PC or device a unique address. However, people quickly realised that it would be convenient to have IP address configuration be handled automatically by a server.

The protocol for this is called the DHCP protocol. It requires a DHCP server. Most routers for home use have built-in DHCP servers.

Another protocol that has been developed is the BOOTP protocol. This protocol makes it possible to boot PCs from a network if they do not have internal hard disks. If a BOOTP request is sent to a server, it uses the same protocol to send a small program back to the PC. The PC runs this program after receiving it. This program usually enables the PC to continue starting up from the server. This makes it a lot easier for system managers to supply new software to all PCs on the network in a single operation or install updates.

In our example, we use BOOTP to send new firmware to the FPGA module. This firmware is loaded into flash memory or SDRAM (depending on the position of dipswitch 7) and then executed.

FPGA Course (8)

Part 8: Playing with the USB port

Paul Goossens and Andreas Voggeneder

In this instalment of the course We're going to pu together our own USB microcontroller. In terms of technical features, it can certainly hold its own against commercial USB controllers such as the Cypress IC.

The specifications of our home-grown USB controller embedded into an FPGA are certainly nothing to be ashamed of:

- 8052 processor running at 48 MHz with single clock cycle per byte fetch
- full-speed USB controller with five endpoints
- 8 KB of ROM
- 4 KB of RAM

We built the USB controller by extending the familiar 8052 core with a USB controller core, which is also freely available from OpenCores (www.opencores.com). This is not the place to delve into the details of how the USB core works or its internal structure. What matters here is how it is linked to the 8052 core and how to use it in practice. The USB core is the USBHostSlave core from OpenCores (www.opencores.org/projects.cgi/web/usbhostslave/overview). It was designed by Steve Fielding, and it has the following properties:

- register-configurable low-speed / high-speed mode (1.5 Mbps or 12 Mbps)
- 4 freely available endpoints, each with a 64-byte FIFO
- support for control, bulk, interrupt, and isochronous transfers
- host mode
- 8-bit Wishbone bus interface

The USBHostSlave core is coupled to the 8052 microcontroller via the Wishbone bus. As a result, the registers of the USB core appear in the XRAM memory area (in other words, they are addressed as external SRAM), where the USB core occupies 256 addresses. **Figure 1** shows the block diagram of the overall system, while **Figure 2** depicts the memory organisation.

Registers

The USB core has a set of four registers for each endpoint:

- EP[0-4]_Control
- EP[0-4]_Status
- EP[0-4]_Transtype_Status
- EP[0-4]_NAK_Transtype_Status

The Endpoint_Control registers can be used to control or drive endpoints for purposes such as sending data, setting or clearing stall states, or setting the operating mode of an endpoint (isochronous or bulk). The Endpoint_Status registers provide information about the current status of the endpoints – for instance, whether data has been received or an error has occurred.
The Endpoint_Transtype_Status register contains information about the last successful transfer between the PC and the USB core, such as whether a setup package was received. The Endpoint_NAK__Transtype_Status register displays any errors that may have occurred during the data transfer.

Besides these four primary control registers, each endpoint has an Rx FIFO buffer and a Tx FIFO buffer, each with a depth of 64 bytes. These two FIFOs can be controlled using an additional set of six registers for each endpoint:

- EP[0–4]_Rx_Fifo_Data
- EP[0–4]_Rx_Fifo_Data_Count_MSB
- EP[0–4]_Rx_Fifo_Data_Count_LSB
- EP[0–4]_Rx_Fifo_Control
- EP[0–4]_Tx_Fifo_Data
- EP[0–4]_Tx_Fifo_Control

Among other things, these registers are used to read data received via the endpoint from the associated Rx FIFO and check the fill level of the FIFO, to write data to a Tx FIFO, and to empty (flush) the FIFOs.
There are also several general Slave Control (SC) registers, including:

- SC_Control
- SC_Line_Status
- SC_Address
- SC_Interrupt_Status / Mask

The SC_Control register can be used to enable or disable the USB slave portion and configure it in either full-speed mode (12 Mbps) or low-speed mode (1.5 Mbps). The SC_Line_Status register can be used to check the line status (link to the PC) to determine whether a link actually exists (plug connected) or the link is interrupted somewhere. The SC_Address register is used to set the slave address assigned by the PC during the enumeration process. The SC_Interrupt_Mask register specifies which interrupts are allowed (which means they can generate an 8051 interrupt) and which ones are not allowed. The microcontroller can use the SC_Interrupt_Status register to determine the source of an interrupt.

Besides these slave-specific registers, there is also a large set of registers for the USB host portion, but they are not described here. As described later on in this article, the FPGA board cannot act as a host.

Hardware

The pull-up resistance that defines the connect condition can be controlled by the microcontroller via port line P3.7. P3.7 is high after a reset. MOSFET T6 (which connects the pull-up resistor to D+) is cut off under this condition, so the PC does not see any connected device. When the software has initialised the USB core and is ready to communicate with the PC, it pulls P3.7 low. This causes the D+ to be connected to +3.3 V via the 1.5-kΩ resistor. As a result, the PC sees that a full-speed device (12 Mbps) is connected and initiates communication with the FPGA board.

The USBHostSlave core needs a 48-MHz clock. This is stipulated by the USB specification. A clock signal at this frequency is derived from the 50-MHz clock signal by a PLL in the FPGA. The USB core and the T8052 core are both driven by the 48-MHz clock signal. As the T8052 core can execute all instructions in 1 to 4 clock cycles (except the divide instruction, which takes 12 clock cycles), this provides a considerable amount of computing power (at least for an 8051).

The USB core in the FPGA handles all the tasks necessary for USB communication with the PC. All that remains to be provided is the USB physical interface (USB Phy). Its job is to provide the electrical interface (level adjustment). This task is also handled by the FPGA.

Mixed Language

Besides VHDL, there are several other ways to design a digital circuit. One of them is to use a graphic drawing package. We described this option in earlier instalments of this course.

Other entry methods include Verilog, AHDL, SystemC, and so on. Fortunately, the Quartus compiler can also handle some of these languages. Using two or more different languages is called 'mixed-language design'.

The USB core used in this instalment was designed in Verilog. Some of the other components were designed in VHDL. Our design can thus be regarded as a mixed-language design (part Verilog and part VHDL).

The main advantage of this is that you are not limited to using only VHDL cores or only Verilog cores. You can use them in combination in your designs without any problems. However, you do need to know the Verilog equivalents of some VHDL terms, such as STD_LOGIC signals.

The link between the VHDL part and the Verilog part is located in the USB.vhd file. This file is what is called a 'wrapper', and its purpose is to export Verilog signals as VHDL signals.

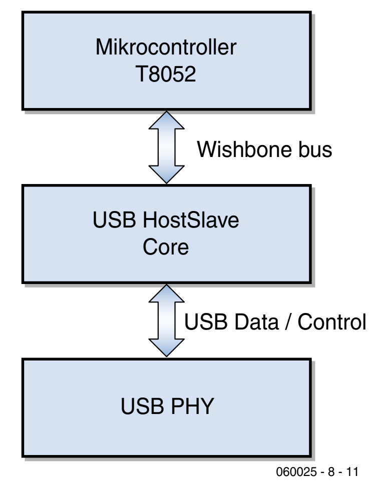


Figure 1.
The USBHostSlave core in the T8052 system.

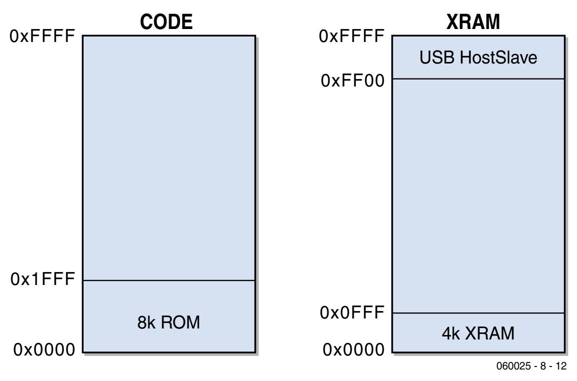


Figure 2.
T8052 memory organisation.

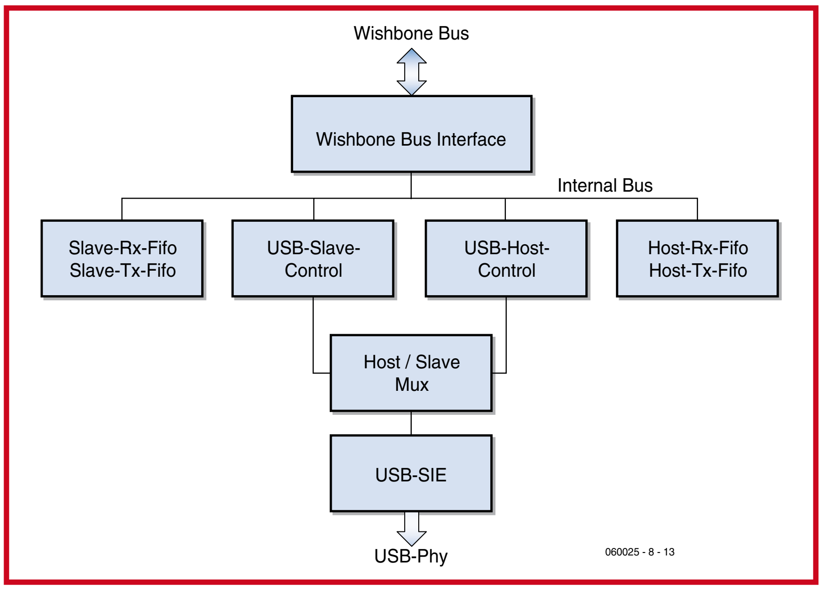


Figure 3. Block diagram of the USB core

Join the FPGA Course with the Elektor FPGA Package!

The basis of this course is an FPGA Module powered by an Altera Cyclone FPGA chip, installed on an FPGA Prototyping Board equipped with a wealth of I/O and two displays (see the March 2006 issue).

Both boards are available ready-populated and tested. Together they form a solid basis for you to try out the examples presented as part of the course and so build personal expertise and know-how in the field of FPGAs.

Further information may be found on the shop/kits & modules pages at www.elektor-electronics.co.uk

The USB core also supports host mode, which means it can act as a PC for functions such as enumerating and driving a USB mouse or other type of USB device. However, host mode cannot be used with the FPGA prototyping board because it lacks the necessary hardware (external to the FPGA). Here the USB core can only be used in slave mode with low-speed (1.5 Mbps) or full-speed (12 Mbps) data transfer.

Sample application

Our demo application here (*ex22*) involves connecting an HID keyboard. It identifies itself to the PC as a keyboard, which causes the PC to use the Human Interface Device (HID) protocol defined in the USB standard. The main purpose of the HID protocol is controlling input devices such as keyboards, mice and joysticks. The principal advantage of this protocol is that all major operating systems (Window, Linux and MacOS) incorporate drivers for it, so they do not have to be written for individual devices.

Endpoints 0 and 1 are used for communication with the PC. Endpoint 0 is a control endpoint, which must be present in every USB device. It is used for basic communication with the PC (enumeration). Enumeration is a process that takes place immediately after a USB device is connected to a host (PC). Each new device is recognised by the 1.5-kΩ resistor connected to the D+ or D– line in every USB device. If the resistor is connected to the D+ line, the device is a full-speed device (12 Mbps); otherwise it is a low-speed device (1.5 Mbps). The FPGA prototyping board has a switchable resistor for each of these lines, so you can configure it as desired as a full-speed or low-speed device. The PC detects the type of device during enumeration and assigns a slave address to the device. The appropriate driver is then loaded based on the Device_Descriptor data sent by the device. After this has been completed, an application can communicate with the device. In this case, the driver is the HID driver provided by the operating system. Other endpoints are not enabled until after the driver has been loaded. In the case of the HID keyboard, this is endpoint 1, which acts as an interrupt endpoint (IN = device to PC) and is used for transmitting keystrokes.

On the *Elektor Electronics* FPGA prototyping board, pushbuttons S2–S4 and the eight DIP switches in S5 are used as 'keys', with S4 serving as a Shift key. S1 acts as the reset button for resetting the 8052 and the USB core. LED 7 acts as the indicator for the Number Lock function, while LED 6 acts as the Caps Lock indicator and LED 5 as the Scroll Lock indicator.

Bonus application

Our second sample application (*ex23*) is a PS/2 to USB adapter. You can use this adapter to connect a keyboard with a PS/2 connector to a USB port of a PC. The PC will see the keyboard as a USB keyboard. The FPGA causes the keystrokes of the PS/2 keyboard to be transferred via the USB interface just as though they originated from a real USB keyboard.

Naturally, this example uses the PS/2 interface as well as the USB interface. This means the associated firmware is more extensive, but if you read the PS/2 instalment you should be able to understand how this application works As usual, the software for the examples described in this instalment is available on the *Elektor Electronics* website under item number 060025-8-11 (go to Magazine → January 2007 → FPGA Course Part 8).

(060025-8)

Andreas Voggeneder

Andreas Voggeneder studied Hardware/Software Systems Engineering (HSSE) at the University of Applied Sciences in Hagenberg, Austria (**http://hsse.fh-hagenberg.at/**), where he presently still gives occasional lectures.

As an enthusiastic electronics specialist, he devotes part of his time to designing digital circuits with the help of VHDL and Verilog. He is also a moderator at OpenCores (**http://www.opencores.com/**), a forum dedicated to the freely available T51 8051 processor core used in several instalments of this course.