

# Chapter 15

## Embedded Systems Development: Case Studies

Embedded systems are computers designed and programmed to meet the requirements of a specific application. Applications may not require an OS (Operating System) or may rely on a customized OS. The system architecture is usually composed of a low-cost microprocessor, memory and peripherals interconnected through busses. It may also include a coprocessor to speed-up a specific computation.

This chapter introduces the design of embedded systems on Xilinx FPGAs through a set of case studies. The studies focus on the hardware development of a peripheral and a coprocessor, as well as their software drivers. It assumes you are familiarized with the VHDL necessary on the hardware implementation, and with C/C++ which is required during the software development. In order to simplify the contents, they only expose the most relevant steps and interesting portions of the source files. They usually make references to specific documentation available in the Xilinx software, to get more details. The full examples can be downloaded from the authors' website.

### 15.1 Introduction to Xilinx EDK

The FPGA implementation of embedded systems requires a set of tools which permits the building of the hardware and the software (also named firmware). It also provides utilities to simulate and debug the embedded system.

The Xilinx Embedded Development Kit (EDK) suite facilitates the development of those systems. The system's hardware is composed of a general-purpose microprocessor connected to some peripherals through busses. The software development provides the firmware executed by the microprocessor. The EDK generates the bitstream of the system in order to program the FPGA, which can be connected to the debugger tools, in order to test it on a real scenario. EDK also

permits the building of a simulation model in order to check the hardware design through a Hardware Description Language (HDL) simulator.

The EDK is frequently updated and upgraded. We will focus this chapter on the ISE Design Suite 13.1 for Windows since it is probably the most popular operating system for PCs, but there are no significant differences with versions for other operating systems. Although the case studies included in this chapter can be implemented on other upgraded EDK versions, they might require some small changes.

The EDK is composed of a set of tools:

- Xilinx Platform Studio (XPS). A graphical user interface that permits the designing of the hardware of the embedded system from a set of interconnected IP cores. It is the top-level tool which takes care of the necessary files and steps needed to successfully complete the hardware design. The XPS implements the design flow which runs other low-level tools in order to compute the hardware synthesis and implementation (Platgen), the generation of the bitstream (Bit-Gen) and the simulation model (Simgen).
- Software Development Kit (SDK). An integrated environment based on the Eclipse/CDT to manage the development of the software. It launches the C/C++ cross-compiler and linker to build the binary which is executed by the embedded microprocessor. Moreover, it also provides a simple interface with the debugger and profiler tools used by more advanced users. SDK also builds the BSP (Board Support Package) through a low-level tool (Libgen). The BSP contains the set of software drivers used to control the hardware from the executable.
- IP cores. The library of configurable cores (microprocessors, peripherals, busses, etc.) that are used as basic building blocks of the embedded system. Most of these cores are licensed with the EDK, but there are also some cores that must be licensed separately. Many cores include a set of programming functions and drivers that can be used to facilitate the software development.
- GNU tools chain. The set of tools that generate the software libraries and the executable binaries. It includes the GNU C/C++ cross-compiler and linker. The GNU tools are developed for the Linux operating system, but EDK includes ported versions to the Windows OS.

Additionally, the EDK relies on other tools:

- ISE tools. They synthesize and implement the hardware, generate the bitstream and program the device. They also include other tools to generate the simulation model, implement the internal memory, the timing analysis and others. Platform Studio calls the required ISE tools, simplifying the design flow since the user can be abstracted from many specific details.
- Supported HDL simulator. It is recommended if the user designs a new IP, since it permits the simulation of the hardware of the embedded system. Some IP cores deployed with EDK are protected and encrypted, therefore, they can be simulated only on supported simulators. The ISE tools provide the ISim, but the user can choose a third-party simulator.

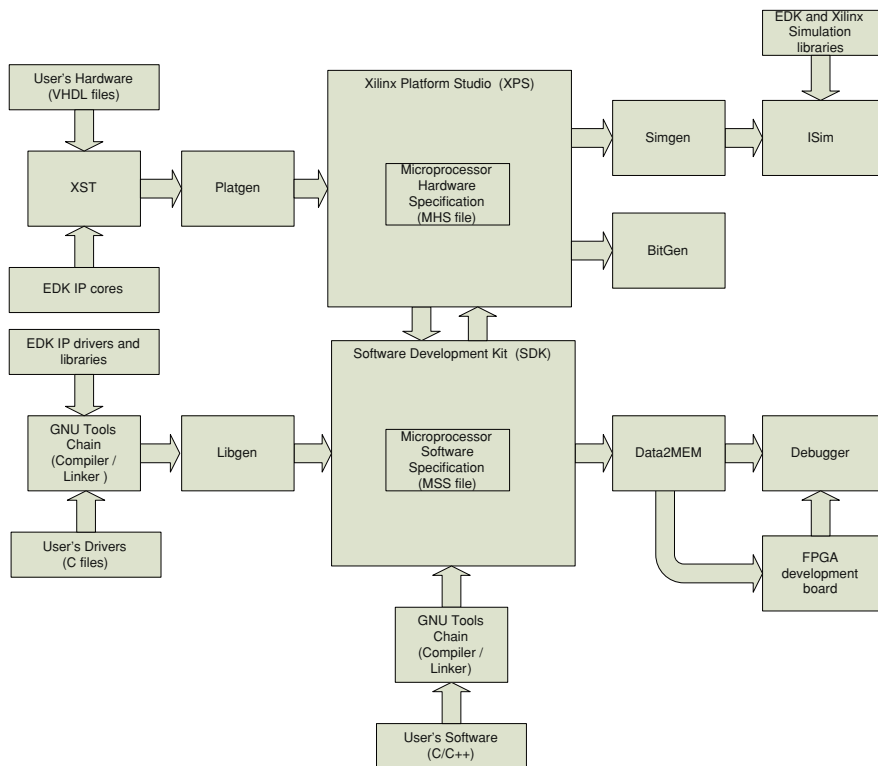
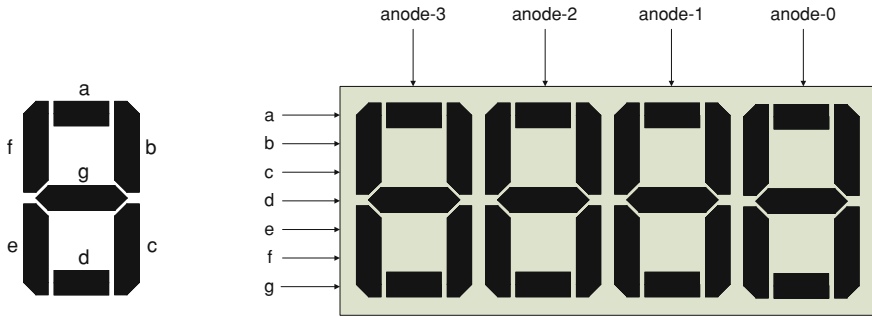


Fig. 15.1 Overview of the EDK design flow

- A development board with a Xilinx FPGA. There is a wide range of boards with different FPGAs (Spartan or Virtex series), memories, displays, communication interfaces and other elements.
- A Xilinx programmer, such as the Parallel III/IV or the Platform Cable USB I/II. Some development boards provide an embedded USB programmer. The programmers can be used to program the FPGA and to debug the executable through the Xilinx Machine Debugger (XMD) low-level tool.

Two files play an important role in the design flow (see Fig. 15.1): the Microprocessor Hardware Specification (MHS) and the Microprocessor Software Specification (MSS). The XPS manages the hardware design flow using a Xilinx Microprocessor Project (XMP) project file. The XPS project relies on the MHS file which configures a set of instances to IP cores that are interconnected as building blocks. The XPS can export the hardware design to SDK in order to generate the Board Support Package (BSP) for the embedded system. The BSP generation relies on the MSS file which configures the drivers and libraries that can be used by the executable to control the hardware.



**Fig. 15.2** The 4-digit, 7-segment led display

### 15.1.1 Case Study 1-A: A Basic Embedded System

This case study builds a basic embedded system, composed of the MicroBlaze processor [16], internal memory, and some peripherals. The system controls a 4-digit, 7-segment led display to visualize the hexadecimal content of a variable. The system attaches to two external switches that turn on/off the display and show/hide the left-side zeros. The display can also be controlled by an external computer connected through the serial interface.

The development FPGA board is the Xilinx Spartan-3 Starter Kit [2], a cheap board composed of a XC3S200 FPGA interconnected to a 7-segment display, a serial port and other elements. The four digits of the display share the segment inputs and each digit is enabled with a dedicated anode input, as depicted in Fig. 15.2. The system must continuously perform a time-multiplexed control to refresh the display, driving a single digit during a short time slot (a few milliseconds). Depending on the board, the inputs of the display are asserted to low or high logic levels. The case study can be easily adapted to any FPGA board which provides a similar display.

The display could be continuously controlled by the microprocessor in a loop to enable one digit per refresh cycle. This system is quite inefficient, since most of the execution time would be devoted to wait the refresh of the display, and to poll the serial interface and the state of the switches. A better approach uses two interrupt service routines (ISR) to attend the interrupts from two peripherals. A timer peripheral can, periodically, request an interrupt which triggers an ISR in order to read the switches and refresh the display. The Universal Asynchronous Receiver Transmitter (UART) peripheral requests an interrupt when a new character is received from the serial interface, and its ISR will process it.

The design of an embedded system involves the hardware and software development phases. The output of the hardware phase is a BIT (bitstream) file which configures the hardware resources of the FPGA except the contents of the internal BRAM (Block RAM) used as the microprocessor's local memory. The output of the software phase is the Executable and Linkable Format (ELF) file which must be



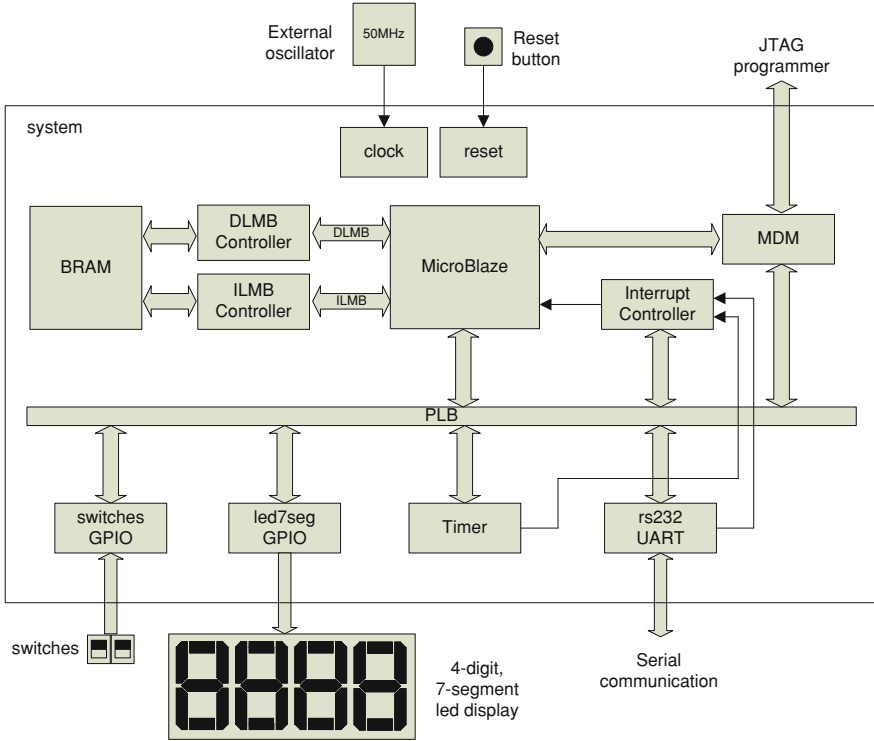


Fig. 15.3 Overview of the system hardware

allocated into the microprocessor’s memory. In order to program the FPGA the design flow executes the Data2MEM tool to generate a new bitstream file which configures the FPGA including the BRAM contents to store the executable binary.

### 15.1.2 Hardware

The system is composed of the MicroBlaze, the internal BRAM and a set of peripherals (see Fig. 15.3). The BRAM implements the microprocessor’s local memory and it is connected through two Local Memory Bus (LMBs). The peripherals are connected through a Processor Local Bus (PLB). The MicroBlaze can control the display and the two switches through General Purpose Input Output (GPIO) peripherals. The UART peripheral permits the serial communication with the external PC through the RS-232 interface. The timer and the UART will request the microprocessor to interrupt, therefore, the system includes an interrupt controller. Finally, the Machine Debug Module (MDM) permits the debugging of the application executed by the MicroBlaze through the FPGA programmer and the XMD tool.

The hardware generation involves three main steps:

- (1) Hardware specification in the MHS file
- (2) Synthesis of the hardware
- (3) Implementation of the FPGA layout and bitstream generation

### 15.1.2.1 Specification

The first step is to specify the hardware of the embedded system in the MHS file. The easiest way to start is by using the Base System Builder (BSB) wizard. It creates the project file and a valid MHS file [14] which describes the system composed of the microprocessor attached to local memory and peripherals. Open the Xilinx Platform Studio (XPS) and follow the steps:

- (1) XPS opens a dialog window. Choose the BSB wizard
- (2) The next dialog window configures the project file and directory. Change the path to *C:\edk13\Ned7seg* and the project name to *system.xmp*
- (3) Next, a new dialog configures the system bus. The AXI is currently supported only on the newer FPGA families (Spartan-6, Virtex-6). Therefore, select the PLB [4] since it is supported by all the FPGA families.
- (4) Select the option *Create a new design* in the dialog *Welcome*.
- (5) The dialog can configure a pre-built system for a supported FPGA board. Choose *Create a system for a custom board* to setup the system from scratch. Then select the Spartan-3 *xc3s200-ft256-4* device and any polarity for the reset button. These parameters can be easily modified later.
- (6) Choose a single processor system since it simplifies the design.
- (7) The next dialog configures the frequencies of the reference and system clocks as well as the capacity of the microprocessor's local memory. Leave the default parameters since they are changed later.
- (8) The BSB permits the connecting of a set of peripherals to the system. Just continue until the end of the wizard since they are added later. The BSP creates the hardware specification of a basic embedded system.

The XPS permits to display and modify the system in a graphical way using the tab *System Assembly View*, as shown in Fig. 15.4. The view *Bus Interfaces* shows the system composed of instances to IP components that are interconnected through busses. The microprocessor (*microblaze\_0*) is connected to the BRAM (*lmb\_bram*) through the data and instruction LMBs (*dlmb*, *ilmb*) [17] and their memory controllers (*dlmb\_cntrl*, *ilmb\_cntrl*). The peripherals attach to the system through the PLB as slaves. The instruction and data PLB sides of the microprocessor are the bus masters. The MDM (*mdm\_0*) peripheral attaches to the microprocessor through the PLB (*mb\_plb*). The MicroBlaze is master of the busses, meanwhile, the peripheral and memory controllers are the slaves. Finally the last two instances are the generators of the internal clock (*clock\_generator\_0*) and reset (*proc\_sys\_reset\_0*) that are required by the system.

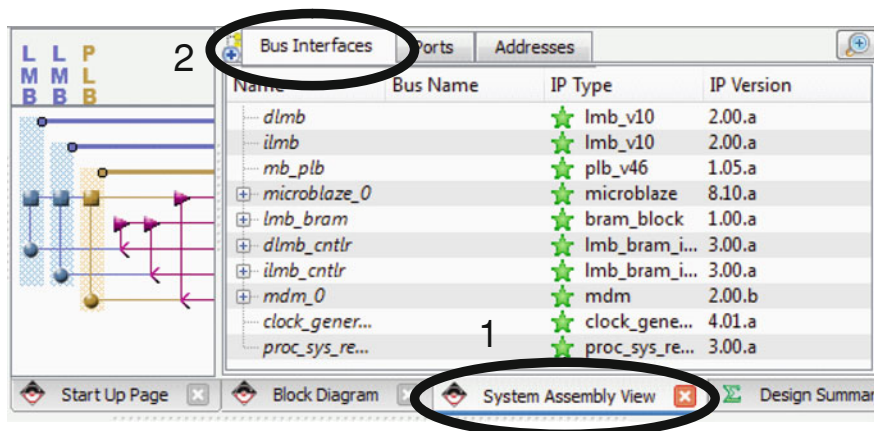


Fig. 15.4 Bus interfaces of the system in EDK

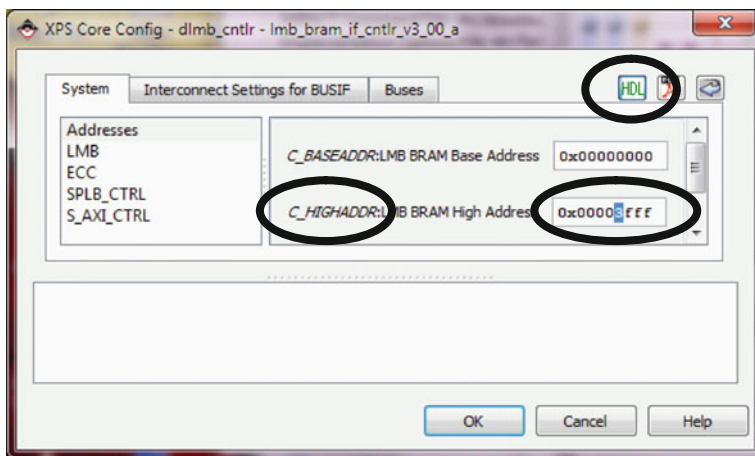


Fig. 15.5 Configuration parameters of the data LMB controller

The IPs provide a set of parameters that can be fixed, auto-computed or configurable. Select an instance and open the contextual menu (click the right button of the mouse) to configure the IP graphically (see Fig. 15.5). The HDL name of a parameter is the same as it appears in the MHS file. The configurable parameters *C\_BASEADDR* and *C\_HIGHADDR* of the LMB controllers setup the address space of the microprocessor’s local memory. Changing the *C\_HIGHADDR* to  $0 \times 3FFF$  of both LMB controllers increases the auto-computed parameter *C\_MEMSIZE* of the BRAM to 16 KB ( $0 \times 4000$ ). Every IP deployed by EDK provides a PDF file which details the parameters, the input/output ports and internal architecture.

The MHS file is a text file which can be manually edited, as shown in Fig. 15.6. It is synchronized with the *System Assembly View*. Therefore, they both update

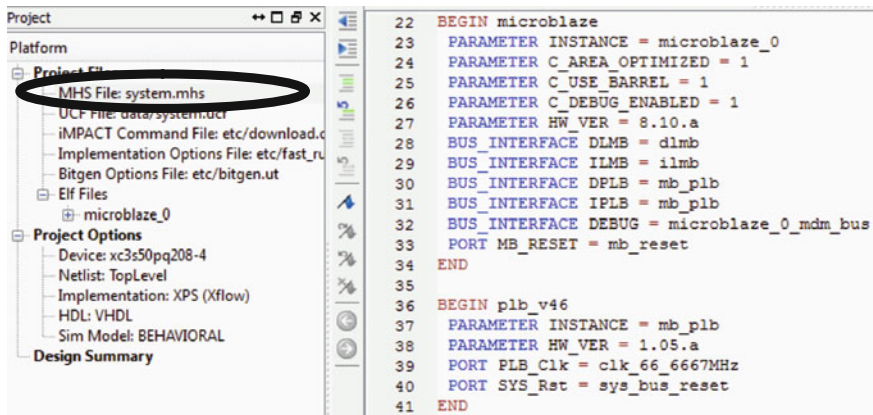


Fig. 15.6 MHS file in the XPS project

when the other one is modified. The MHS specifies the system's hardware as a set of interconnected instances and external FPGA ports. Each instance contains configurable parameters, interface to busses and other ports. The parameters that are not declared in the MHS take a default or the auto-computed value. The bus interfaces or ports that are not declared in the MHS are disconnected.

At the beginning of the MHS file there are the declarations of the external FPGA ports used to input the clock and the reset. The external ports are connected to the internal signals *CLK\_S* and *sys\_rst\_s* that are used by the clock and reset generators. The parameter *CLK\_FREQ* declares the frequency of the external oscillator and the *RST\_POLARITY* declares the logic level when the reset input asserts. Both parameters must be modified according to the FPGA board.

```

PORT fpga_0_clk_1_sys_clk_pin = CLK_S, DIR = I,
    SIGIS = CLK, CLK_FREQ = 50000000
PORT fpga_0_rst_1_sys_rst_pin = sys_rst_s, DIR = I,
    SIGIS = RST, RST_POLARITY = 1

```

The instance *proc\_sys\_reset\_0* generates the internal reset signals required by the system. The configurable parameter *C\_EXT\_RESET\_HIGH* must be modified according to the reset polarity which was declared in the external port.

```

BEGIN proc_sys_reset
  PARAMETER INSTANCE = proc_sys_reset_0
  PARAMETER C_EXT_RESET_HIGH = 1
  ...
  PORT Ext_Reset_In = sys_rst_s
END

```

The instance *clock\_generator\_0* infers a Digital Clock Manager (DCM) circuit to generate the system clock from a reference clock (external oscillator). The parameter *C\_EXT\_RESET\_HIGH* must be configured as in the reset generator. The parameter *C\_CLKOUT0\_FREQ* configures the generated frequency from the reference clock defined by the *C\_CLKIN\_FREQ*. The BSP generated a signal named *clk\_66\_6667 MHz* (or similar) which carries out the generated clock, but it can be renamed to *sys\_clk*. Also change the name of the signal which drives the clock port of the busses. An incorrect configuration of the clock frequencies will lead to errors during the synthesis or implementation of the hardware.

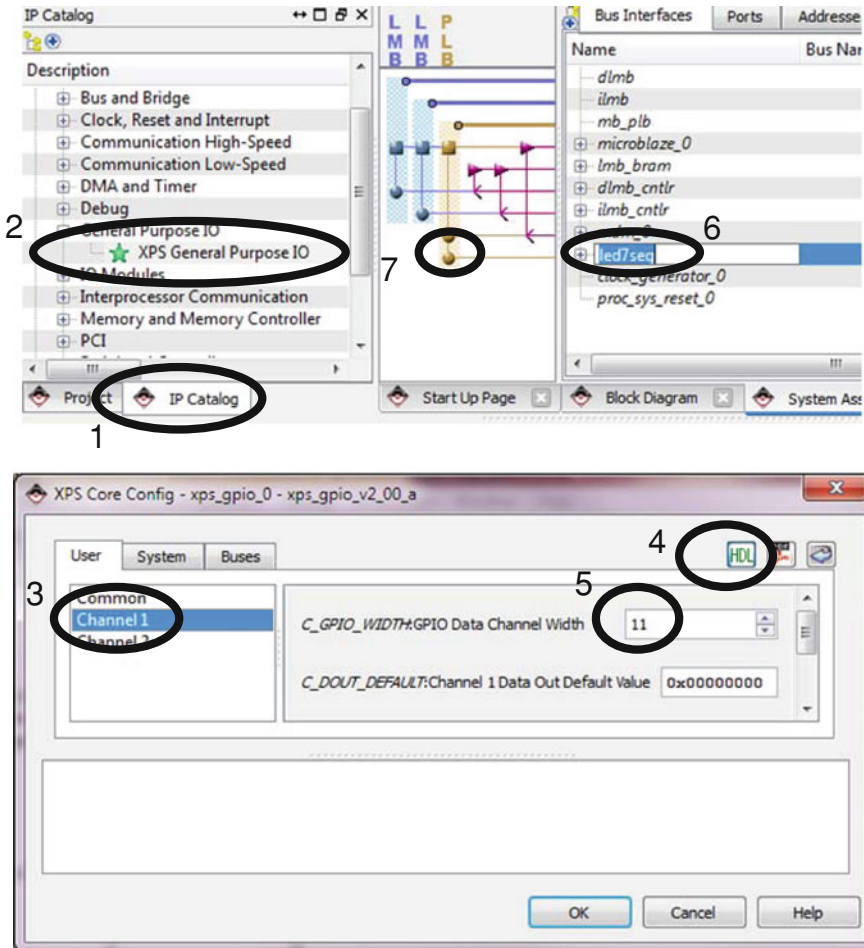
```
BEGIN clock_generator
  PARAMETER INSTANCE = clock_generator_0
  PARAMETER C_CLKIN_FREQ = 50000000
  PARAMETER C_CLKOUT0_FREQ = 50000000
  ...
  PARAMETER C_EXT_RESET_HIGH = 1
  ...
  PORT CLKIN = CLK_S
  PORT CLKOUT0 = sys_clk
  PORT RST = sys_rst_s
  ...
END

BEGIN plb_v46
  PARAMETER INSTANCE = mb_plb
  PORT PLB_Clk = sys_clk
  ...
END

BEGIN lmb_v10
  PARAMETER INSTANCE = ilmb
  PORT LMB_Clk = sys_clk
  ...
END

BEGIN lmb_v10
  PARAMETER INSTANCE = dlmb
  PORT LMB_Clk = sys_clk
  ...
END
```

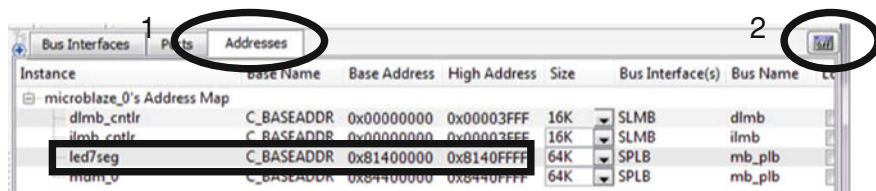
In order to accommodate it to the desired application, the hardware must add new peripherals. The first peripheral is a GPIO [5] which controls the 7 segments and the 4 anodes of the display. As depicted in Fig. 15.7:



**Fig. 15.7** Adding the new GPIO peripheral

- (1) Drag the *XPS General Purpose IO* from the *IP Catalog* to the *System Assembly View*.
- (2) XPS opens a dialog to configure it. Set the data width of the first channel (parameter *C\_GPIO\_WIDTH*) to *11* in order to control the display.
- (3) Click the created instance and change the name to *led7seg*.
- (4) Click the PLB interface to connect the peripheral's SPLB (Slave PLB).

Go to the view *Addresses* and configure automatically the addresses of the peripheral (see Fig. 15.8). The internal registers of the GPIO are accessible from the microprocessor within this addresses range. By default, XPS assigns the addresses range to *64 KB* ( $0 \times 10000$ ) from an address above the  $0 \times 80000000$ .



**Fig. 15.8** Automatic configuration of the addresses for the GPIO peripheral

The user can change the range, but they must not overlap to the other memory-mapped peripherals.

Finally, the 11-bit width GPIO's output port must be connected to the external FPGA ports that drive the display. Figure 15.9 shows the view *Ports* which permits the external connection of the output port *GPIO\_IO\_O*. Change the default name to *fpga\_0\_led7seg\_pin* in a similar fashion as the rest of the external FPGA ports and check that the direction is configured as output. Finally, set the range order to *[10:0]* to declare them in descending order. The MSB and the LSB are indexed as 10 and 0, respectively.

Open the MHS file to observe the previous changes. There is a new entry in the section of the external FPGA ports. It also contains the new GPIO instance including its configuration parameters and connections.

```
PORT fpga_0_led7seg_pin=led7seg_GPIO_IO_O, DIR=O, VEC=[10:0]
...
BEGIN xps_gpio
  PARAMETER INSTANCE = led7seg
  PARAMETER HW_VER = 2.00.a
  PARAMETER C_BASEADDR = 0x81400000
  PARAMETER C_HIGHADDR = 0x8140ffff
  PARAMETER C_GPIO_WIDTH = 11
  BUS_INTERFACE SPLB = mb_plb
  PORT GPIO_IO_O = led7seg_GPIO_IO_O
END
```

The next step adds a new GPIO to read the state of the two switches. The hardware can also be modified by editing the MHS file. Copy the previous GPIO instance and change the instance name to *switches* and the data width to 2. Set the addresses range to *64 KB* and do not overlap it to the other peripherals. Connect the input port *GPIO\_IO\_I* to a signal which is connected to an external FPGA port named *fpga\_0\_switches\_pin*. Save the MHS file to update the graphical view. The project will close if there is an error in the MHS file, which must be manually corrected.

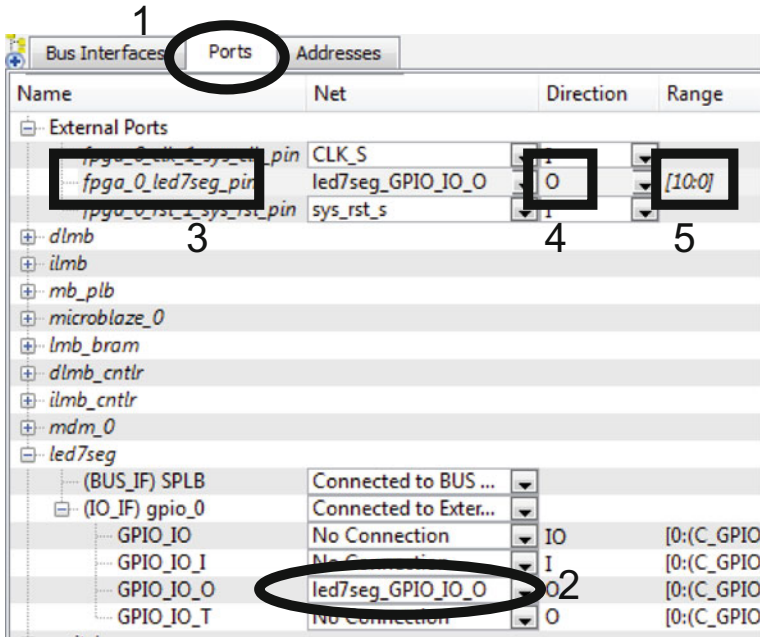


Fig. 15.9 External FPGA connection of the GPIO's output port

```
PORT fpga_0_switches_pin=switches_GPIO_IO_I, DIR=I, VEC=[1:0]
...
BEGIN xps_gpio
  PARAMETER INSTANCE = switches
  PARAMETER HW_VER = 2.00.a
  PARAMETER C_BASEADDR = 0x82400000
  PARAMETER C_HIGHADDR = 0x8240ffff
  PARAMETER C_GPIO_WIDTH = 2
  BUS_INTERFACE SPLB = mb_plb
  PORT GPIO_IO_I = switches_GPIO_IO_I
END
```

The next step adds the two peripherals to the PLB that will request interrupts. The timer [6] will be programmed to request a periodic interrupt. The parameter *C\_ ONE\_TIMER\_ONLY* configures a single timer to minimize the size of the peripheral.



The UART [7] will request an interrupt when it receives a new character from the RS232. It will also transmit messages to the user through the serial communication. Therefore the UART ports transmission (TX) and reception (RX) are connected to external FPGA ports. The parameters *C\_BAUDRATE* and *C\_USE\_PARITY* configure the speed and parity of the communication.

Some displays provide an extra input to turn on a dot placed beside the digit. This case study just turns off the dot connecting the *net\_gnd* or *net\_vcc* to its associated external port.

```
PORT fpga_0_RX_pin = rs232_RX, DIR = I
PORT fpga_0_TX_pin = rs232_TX, DIR = O
PORT fpga_0_led7seg_dot_pin = net_vcc, DIR = O
```

...

```
BEGIN xps_timer
  PARAMETER INSTANCE = timer
  PARAMETER HW_VER = 1.02.a
  PARAMETER C_BASEADDR = 0x83c00000
  PARAMETER C_HIGHADDR = 0x83c0ffff
  PARAMETER C_ONE_TIMER_ONLY = 1
  BUS_INTERFACE SPLB = mb_plb
  PORT Interrupt = timer_Interrupt
END
```

```
BEGIN xps_uartlite
  PARAMETER INSTANCE = rs232
  PARAMETER HW_VER = 1.01.a
  PARAMETER C_BAUDRATE = 115200
  PARAMETER C_USE_PARITY = 0
  PARAMETER C_BASEADDR = 0x84000000
  PARAMETER C_HIGHADDR = 0x8400ffff
  BUS_INTERFACE SPLB = mb_plb
  PORT RX = rs232_RX
  PORT TX = rs232_TX
  PORT Interrupt = rs232_Interrupt
END
```

The system requires an interrupt controller [8] since MicroBlaze provides a single input port for the interrupt requests. The interrupt controller attaches to the PLB in order to permit the MicroBlaze to enable/disable interrupts or to check the interrupt source. The interrupt controller (*int\_control\_0*) connects the interrupt requests from the timer and UART peripherals to Microblaze. The controller receives the concatenated signal from the two peripherals and drives the interrupt port of MicroBlaze. The interrupt priority is higher when the interrupt source is concatenated at the right side. The UART is assigned to the lower priority since it

provides a receiving First Input First Output (FIFO) memory which temporarily stores the characters. Therefore, a new character received can be processed when the display is not refreshing.

```
BEGIN xps_intc
  PARAMETER INSTANCE = int_control_0
  PARAMETER HW_VER = 2.01.a
  PARAMETER C_BASEADDR = 0x81800000
  PARAMETER C_HIGHADDR = 0x8180ffff
  BUS_INTERFACE SPLB = mb_plb
  PORT Intr = rs232_Interrupt & timer_Interrupt
  PORT Irq = int_control_0_Irq
END
```

MicroBlaze provides configurable parameters to optimize area/performance and to implement optional machine instructions. The parameter *C\_AREA\_OPTIMIZED* configures a 3-stage pipeline architecture which optimizes the area of the implementation. The parameter *C\_USE\_BARRELL* implements a barrel shifter and its related machine instructions. Therefore, the C/C++ compiler provides a set of flags to build the BSP and the executable for the configured microprocessor. By default, the MicroBlaze attaches to a MDM instance (*mdm\_0*) which permits the debugging of executables.

```
BEGIN microblaze
  PARAMETER C_AREA_OPTIMIZED = 1
  PARAMETER C_USE_BARREL = 1
  PARAMETER C_DEBUG_ENABLED = 1
  ...
  PORT INTERRUPT = int_control_0_Irq
END

BEGIN mdm
  PARAMETER INSTANCE = mdm_0
  ...
END
```

### 15.1.2.2 Synthesis

The XPS menu *Hardware* → *Generate Netlist* synthesizes the design to generate a set of NGC netlist files. It calls the Platgen tool [13] which starts performing a Design Rule Check (DRC). Then it calls the Xilinx Synthesis Technology (XST) [3] tool to synthesize the IP instances to get their NGC files. The embedded system is finally synthesized and optimized to get the top-level netlist file. A change in the MHS file will force Platgen to synthesise only the required

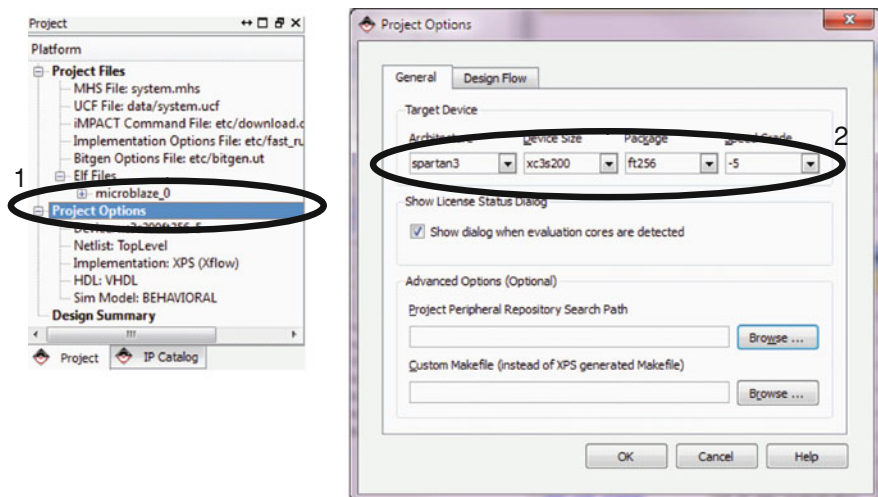


Fig. 15.10 Selection of the device in the XPS project

modules to speed-up the execution. If desired, the XPS can clean up the generated files to start the Platgen from scratch. Synthesis is dependent of the FPGA, therefore, the user must select the correct device in the *Project Options* (see Fig. 15.10) before proceeding.

Figure 15.11 shows the tab *Design Summary* which displays the report files generated by the Platgen and XST in order to check details about the design, such as the occupied FPGA resources or the estimated maximum frequency of the clock.

### 15.1.2.3 Implementation

The implementation computes the FPGA layout which is stored in a Native Circuit Description (NCD) file. The design flow executes three main tools: NGDBUILD, MAP and PAR [15]. The NGDBUILD translates the NGC files and annotates constraints from a User Constraints File (UCF). The following tools compute the layout based on the annotated constraints. The design flow continues with the MAP and PAR tools to map the netlist into the FPGA resources and to compute their placements and routings.

The BSB wizard generates the UCF for the selected prototyping board. The XPS project refers the UCF which must be edited to specify the attachment of the display and switches to the FPGA board. The UCF also specifies the clock frequency of the external oscillator.

The screenshot shows the XPS Reports and XPS Synthesis Summary windows. The XPS Reports table lists various log files generated during the design process. The XPS Synthesis Summary table provides a detailed breakdown of resource usage for each report.

XPS Reports				
Report Name	Generated	Errors	Warnings	Infos
Platgen Log File	Jun 19, May 12:22:10 2011	0	9 Warnings (9 new)	31 Infos (31 new)
Libgen Log File				
Simgen Log File				
BitInit Log File	Jun 16, May 11:44:35 2011			
System Log File	Jun 19, May 12:39:28 2011			

XPS Synthesis Summary (estimated values)						
Report	Generated	Flip Flops Used	LUTs Used	BRAMS Used	Errors	
system	Jun 19, May 12:22:24 2011		2121	3043	8	0
rst_control_0_wrapper	Jun 19, May 12:21:56 2011		132	96		0
rs232_wrapper	Jun 19, May 12:21:42 2011		142	133		0
timer_wrapper	Jun 19, May 12:21:25 2011		291	257		0
clock_generator_0_wrapper	Jun 19, May 12:21:09 2011		4			0
dmb_wrapper	Jun 19, May 12:21:05 2011		1	1		0
dmb_cntrl_wrapper	Jun 19, May 12:21:02 2011		2	6		0
lmb_wrapper	Jun 19, May 12:20:58 2011		1	1		0
lmb_cntrl_wrapper	Jun 19, May 12:20:54 2011		2	6		0
lmb_bram_wrapper	Jun 19, May 12:20:50 2011				8	0
mb_pib_wrapper	Jun 19, May 12:20:46 2011		154	437		0
mdm_0_wrapper	Jun 19, May 12:20:31 2011		126	148		0
microblaze_0_wrapper	Jun 19, May 12:20:22 2011		966	1773		0

Fig. 15.11 Report files from the Platgen and synthesis

```

Net fpga_0_clk_1_sys_clk_pin TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 50000 kHz;
Net fpga_0_rst_1_sys_rst_pin TIG;

Net fpga_0_clk_1_sys_clk_pin      LOC=T9;
Net fpga_0_rst_1_sys_rst_pin     LOC=L14;

Net fpga_0_RX_pin                LOC=T13;
Net fpga_0_TX_pin                LOC=R13;

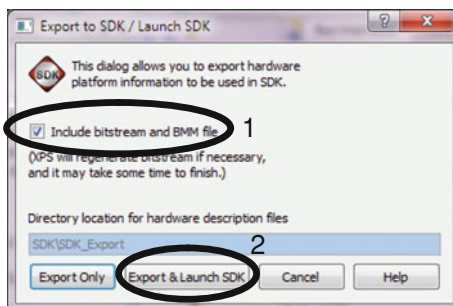
Net fpga_0_led7seg_pin<0>        LOC=E14;           #segment-A
...
Net fpga_0_led7seg_pin<10>       LOC=E13;           #anode-3
Net fpga_0_led7seg_dot_pin       LOC=P16;           #dot

Net fpga_0_switches_pin<0>       LOC=K13;           #switch-0
Net fpga_0_switches_pin<1>       LOC=K14;           #switch-1

```

The XPS menu *Hardware* → *Generate Bitstream* launches the BitGen tool [15] which generates the bitstream file from the FPGA layout. First, the XPS executes the design flow to implement the FPGA layout, if necessary. Then it generates the BIT (bitstream) file *system.bit* and the BlockRAM Memory Map (BMM) file *system\_bd.bmm*. The microprocessor's local memory is implemented on BRAMs,

**Fig. 15.12** Hardware export from EDK to SDK



but the generated BIT file does not initialize them, since the executable binary is not available at this stage. The file *system\_bd.bmm* annotates the physical placement of the BRAMs with the microprocessor's local memory. This file will be required later to update the BRAM contents of the bitstream. The tab *Design Summary* shows the reports generated by the implementation tools.

#### 15.1.2.4 Software

The XPS menu *Project* → *Export Hardware* opens a dialog window to export the required files to SDK, as shown in Fig. 15.12. Select the option to export the BIT and BMM files to permit SDK to program the FPGA. It creates a new directory which is allocated in the XPS project folder.

SDK starts opening a dialog to set the workspace folder. Write the path *c:\edk13\Ned7seg\SDK\workspace* to create it into the SDK folder which was generated by XPS during the hardware exportation. The software development involves two stages:

- (1) The BSP generation. Creates a set of headers and libraries to control the hardware from the microprocessor.
- (2) The executable ELF file. It builds the file executed by the embedded microprocessor.

#### 15.1.2.5 Board Support Package

The BSP provides the Application Programming Interface (API) for the target hardware and Operating System (OS). The BSP generates a set of libraries and header files that facilitates the development of software executables. The application C/C++ source files are compiled and linked using the API to build the binary ELF which is executed by the microprocessor.

The SDK menu *File* → *New* → *New Board Support Package Project* launches a wizard to create a BSP project (see Fig. 15.13). Choose the platform *hw\_platform\_0* which is the hardware exported from XPS. Then set the *standalone* OS [18] since it provides interrupts management and it does require a large memory capacity. Other

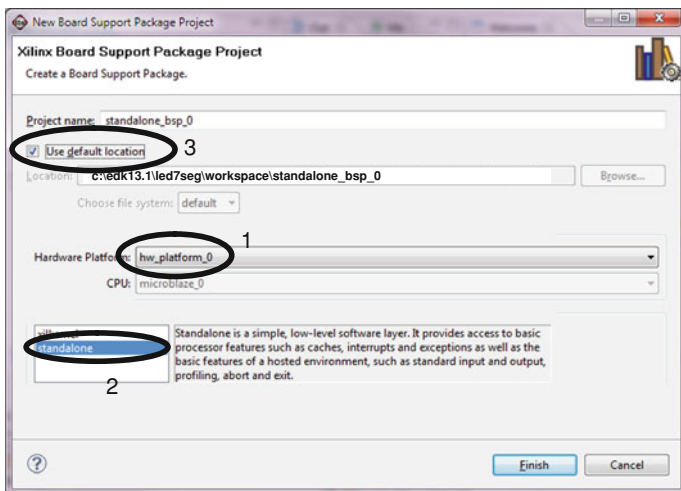


Fig. 15.13 Configuration of the BSP project

OS provide more advanced features, but they require external memory. The BSP project *standalone\_bsp\_0* is located by default in a folder contained in the SDK workspace.

The wizard generates the BSP project which is linked to a MSS file. The MSS [14] is a text file which list the drivers used by the peripherals and the OS for the microprocessor. The Libgen [13] tool reads the MSS file to generate the BSP. As with the MHS file, the MSS can be graphically or manually edited. Figure 15.14 shows the graphical view which configures the MSS. Change the standard input (*stdin*) and output (*stdout*) to the instance *rs232* in order to permit the console functions to use the UART peripheral.

The MSS file can also be manually edited, and it reflects the configuration changes done in the previous dialog.

```
BEGIN OS
PARAMETER OS_NAME = standalone
PARAMETER OS_VER = 3.01.a
PARAMETER PROC_INSTANCE = microblaze_0
PARAMETER STDIN = rs232
PARAMETER STDOUT = rs232
END
```

The rest of the MSS file shows the drivers and peripherals. A peripheral driver is a collection of declarations and functions that can be used to control it from the executable. By default the BSP wizard sets a specific driver to every peripheral, but the user can change it to set a generic driver or no driver. The generic driver can control any peripheral, but the user must have a deeper knowledge about its internal architecture. The system's hardware provides two GPIO peripherals: the

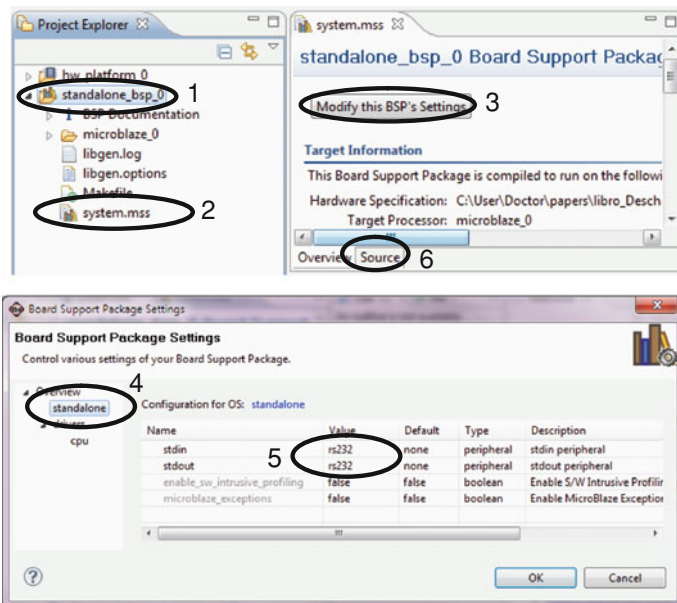


Fig. 15.14 Configuration on the BSP

switches and the led7seg instances. Select the generic driver for them in order to understand better the role of internal registers of peripherals.

```
BEGIN DRIVER
  PARAMETER DRIVER_NAME = generic
  PARAMETER DRIVER_VER = 1.00.a
  PARAMETER HW_INSTANCE = switches
END
```

```
BEGIN DRIVER
  PARAMETER DRIVER_NAME = generic
  PARAMETER DRIVER_VER = 1.00.a
  PARAMETER HW_INSTANCE = led7seg
END
```

The SDK automatically calls the Libgen tool [13] when the MHS is changed to build the BSP. The user may disable the *Build Automatically* behaviour in order to clean or build the BSP using the commands under the menu *Project*. The Libgen tool compiles the source files of the peripheral drivers and the OS, and it stores them into the A (archive) library files. It also generates the H (header) files that declare the functions contained in the libraries. The library and header files are stored in the folders *lib* and *include* of the instance *microblaze\_0*. The SDK can display the contents of both folders and open the header files.

An important header file is the *xparameters.h* which declares a set of parameters about the hardware. Every peripheral has its own parameters that are obtained from the exported hardware, as the addresses range of the GPIOs. The declarations can be used by the C/C++ source files to control the peripherals.

```
/* Definitions for driver GPIO */
#define XPAR_XGPIO_NUM_INSTANCES 2

/* Definitions for peripheral LED7SEG */
#define XPAR_LED7SEG_BASEADDR 0x81400000
#define XPAR_LED7SEG_HIGHADDR 0x8140FFFF

/* Definitions for peripheral SWITCHES */
#define XPAR_SWITCHES_BASEADDR 0x82400000
#define XPAR_SWITCHES_HIGHADDR 0x8240FFFF
```

### 15.1.2.6 Executable

The SDK will build the ELF executable from the source C++ files that are compiled and linked with the functions stored in the BSP libraries. Click the menu *File* → *New* → *Xilinx New C++ Project* which opens a wizard to create a C++ project for the BSP. Change the default project name to *app1* and select the previously generated BSP *standalone\_bsp\_0*, as depicted in Fig. 15.15. The wizard creates a nested folder *app1/src* in the SDK workspace to store the source files that will be compiled.

Using the Windows Explorer delete the template file *main.cc* which was created by the wizard, and copy the new source files: *ledseg7.cc*, *led7seg.h* and *application.cc*. Go to SDK and click the menu *Refresh* of the contextual menu (right button of the mouse) of the project *app1*, in order to update the list of source files. The SDK can open and display the source files in its integrated editor (see Fig. 15.16).

The source files *led7seg.h* and *led7seg.cc* declare and implement a C++ class named *CLed7Seg* which controls the display through the GPIO. The EDK peripherals implement a set of 32-bit registers that are used to control them. The peripheral's registers are memory-mapped, therefore, MicroBlaze can access them when it executes read/write instructions to the content of a C/C++ pointer. The application can directly control the peripheral, although, it is necessary to have a deeper knowledge about the internal architecture. The first register of a GPIO [5] is the *GPIO\_DATA* which is mapped at the base address of the peripheral. The register retrieves/sets the state of the input/output ports depending if the microprocessor reads/writes it.

The class constructor assigns the input argument to the integer (32-bit) pointer *GPIO\_Data*. Any pointer used to access a peripheral's register should be declared *volatile*. If not, the compiler may optimize a set of sequential memory accesses through the pointer, changing the order or deleting some of them. The *GPIO*



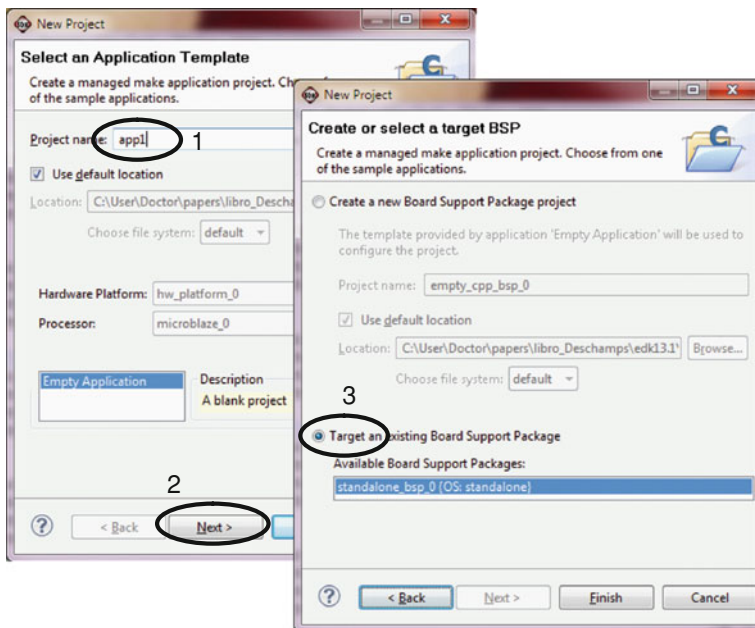


Fig. 15.15 Creating a new C++ project in SDK

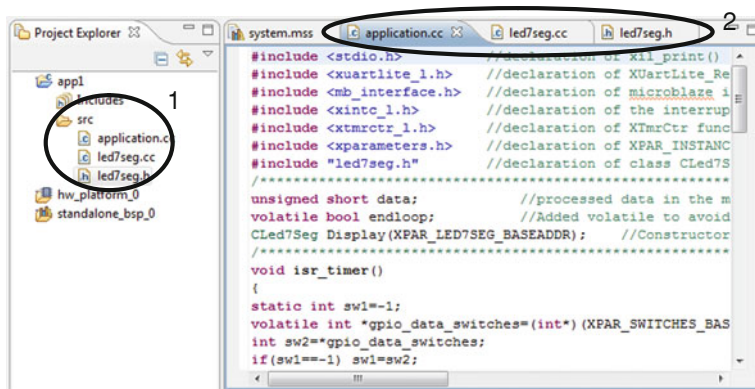


Fig. 15.16 Displaying the C++ source files

method concatenates the anodes and segments to write them into the *GPIO\_DATA* register through its pointer. The header file of the class declares the two parameters that configure the active state of the anodes and segments of the display, therefore, they can be easily changed to adapt it to another prototyping board.

```

volatile int* GPIO_Data;          //Pointer to register GPIO_DATA
...
CLed7Seg::CLed7Seg(int gpio_baseaddr) //Class constructor
{GPIO_Data=(int*) (gpio_baseaddr);} //Pointer assignment
...

void CLed7Seg::GPIO(unsigned char anodes,
                    unsigned char segments)
{if(LED7SEG_SEGMENTS_ACTIVE_LOW) segments=(~segments)&0x7F;
 if(LED7SEG_ANODES_ACTIVE_LOW) anodes=(~anodes)&0x0F;
 *GPIO_Data=(anodes<<7)|segments;} //Writes GPIO_DATA

```

The class declares two member variables: *Data* and *Config*. The *Data* is a 16-bit variable which stores the number which is displayed. The *Config* is an 8-bit variable which uses the two LSBs to turn on/off the display and to show/hide the left-side zeros of the number. The method *Refresh* is periodically executed since the timer's ISR calls it. It reads the member variables and calls the *Digit* method to display one of the digits starting at the left side. The *Digit* method first computes the segments and anodes of a digit and then it calls the *GPIO* method to display it.

```

void CLed7Seg::Refresh()          //Called from the timer's ISR
{static char idx=3;              //Start at the left-side digit
 char off=(Config&0x02)>>1;      //Turn on/off the display
 char zeros=(Config&0x01);       //Show/hide the left-side zeros
 Digit(off,zeros,Data,idx);      //Display a single digit
 idx=(idx==0)? 3 : idx-1;}       //Next call will get next digit

```

The application C++ file is composed of 4 sections. The first section opens the required header files. The application controls the GPIOs directly, but the rest of the peripherals are controlled through their drivers. Therefore, it opens the header files that declare the functions stored in the BSP libraries. The file *xparameter.h* declares base addresses that are necessary to use the driver functions.

The second section initializes the object *Display* for the class *CLed7Seg*. The object's constructor gets the base address of the GPIO which drives the display. The section also declares the global variables *data* and *endloop* that are managed by the ISR of the UART.

The third section is composed of the two ISRs. The timer's ISR periodically reads the state of the two external switches and refreshes the display. First, it reads the register *GPIO\_DATA* of the peripheral which attaches to the two external switches. Then, it compares the state of the switches against the previous call. A change in one of the switches will swap one of the two bits that configure the display, using the bitwise XOR operators. Finally, it refreshes the display. The other ISR is executed when the UART receives a new character which is read

using its driver function [9]. Depending on the received character, it changes the data or the configuration of the display, or it quits the application.

```
#include <xparameters.h>    //declaration of MHS parameters
#include "led7seg.h"        //declaration of class CLed7Seg
...
CLed7Seg Display(XPAR_LED7SEG_BASEADDR); //Object Display
unsigned short data; volatile bool endloop;
...
void isr_timer()           //The timer ISR
{...
    volatile int *gpio_data_switches
        =(int*)(XPAR_SWITCHES_BASEADDR+0); //Pointer GPIO_DATA
    int sw2=*gpio_data_switches;           //Read switches
    ...
    Display.Config^=(sw1^sw2);            //Swap configuration bits
    Display.Data=data;                    //Update the displayed data
    Display.Refresh();                    //Refresh the display
    sw1=sw2;                              //Store the switches state
    ...}

void isr_rs232()          //The UART ISR
{...
    char rs232_char=XUartLite_RecvByte(XPAR_RS232_BASEADDR);
    switch(rs232_char){ //Read received character
        case '+': data++; break; //Increment displayed data
        case 'z': Display.Config^=LED7SEG_ZEROS; break;
        case 'x': endloop=true; break; //Quit the app
        ...}
    ...}
```

The last section is the main function of the application. It configures and enables the interrupt sources, and then it executes a loop until the application quits. The loop can execute any computation without affecting the control of the display.

The timer peripheral [6] implements two registers to periodically generate an interrupt request: Timer Control/Status Register 0 (TCSR0) and Timer Load Register 0 (TLR0). The configuration of both registers asserts the interrupt signal every 5 ms (250,000 counts, 50 MHz clock). The constants and functions of the timer driver are declared in the header file *tmrctr\_1.h* [10] which was generated by the BSP. They are low-level API functions since the programmer knows the functionality of the registers. These functions compute the address of the registers and write data into them through a pointer.

The driver of the interrupt controller provides functions [11] to register ISRs and to enable the interrupts sources. Finally the application enables the MicroBlaze's interrupt input through an OS function [18].

```

int main()
{
...
  XTmrCtr_SetLoadReg(XPAR_TIMER_BASEADDR, 0,
    LED7SEG_REFRESH_COUNTS); //Writes TLR0 register
  XTmrCtr_SetControlStatusReg(XPAR_TIMER_BASEADDR, 0,
    XTC_CSR_LOAD_MASK); //Loads the counter register
  ...
  XIntc_RegisterHandler(XPAR_INT_CONTROL_0_BASEADDR,
    XPAR_INT_CONTROL_0_TIMER_INTERRUPT_INTR,
    (XInterruptHandler)isr_timer, NULL); //register Timer ISR
  ...
  XIntc_MasterEnable(XPAR_INT_CONTROL_0_BASEADDR);
  ...
  microblaze_enable_interrupts(); //standalone OS function
  ...}

```

By default, the wizard of the C++ project generates two targets: *Debug* and *Release*. They differ in the flags of the GNU compiler [13]. The target *Debug* compiles source files without optimizations and enabling debug symbols. The target *Release* compiles source files with optimizations to build smaller and faster code which is not suitable to debug. The targets configure other compiler flags that are derived from the MicroBlaze's configuration in order to use the optional machine instructions. The menu *Project* → *Build Project* builds the active target which can be changed anytime using the *Project* → *Build Configurations* → *Set Active*. Then, SDK runs the GNU tool chain which compiles the source files and it links the resulting object code with the BSP libraries. The executable ELF file is stored in a nested folder which is named as the target.

### 15.1.3 Programming and Debugging

The SDK menu *Xilinx Tools* → *Program FPGA* opens a dialog window which displays the BIT and BMM files that were imported from EDK, as shown in Fig. 15.17. Select the ELF file *app1.elf* of any of the two targets and press the button *Program*. It calls the Data2MEM tool [20] which generates a new bitstream *download.bit* from an existing bitstream and its annotated BMM file. The new bitstream configures the entire FPGA including the BRAM contents with the selected ELF file. Then, the new bitstream is programmed into the FPGA and the application begins to run.

The serial ports of the FPGA board and the PC are attached to test the application. The PC should execute a terminal configured with the same communication parameters as the embedded system. The terminal displays the received characters from the FPGA and sends the characters that are pressed on the keyboard. The SDK provides its own terminal which can be used for this

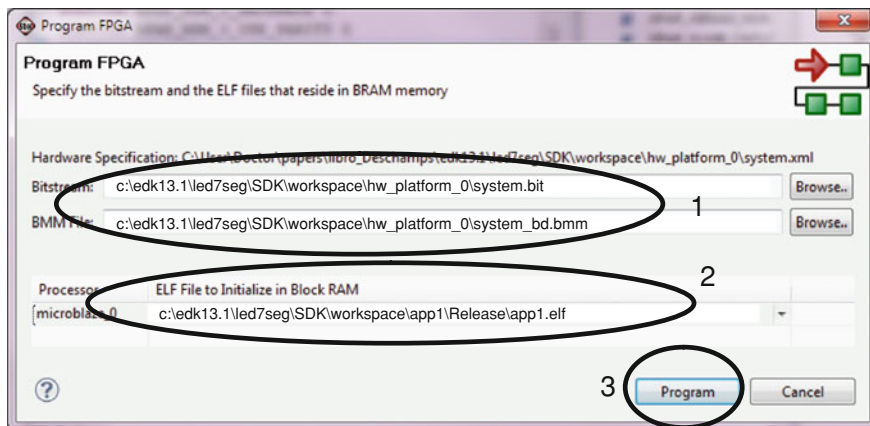


Fig. 15.17 Bitstream configuration to program the FPGA

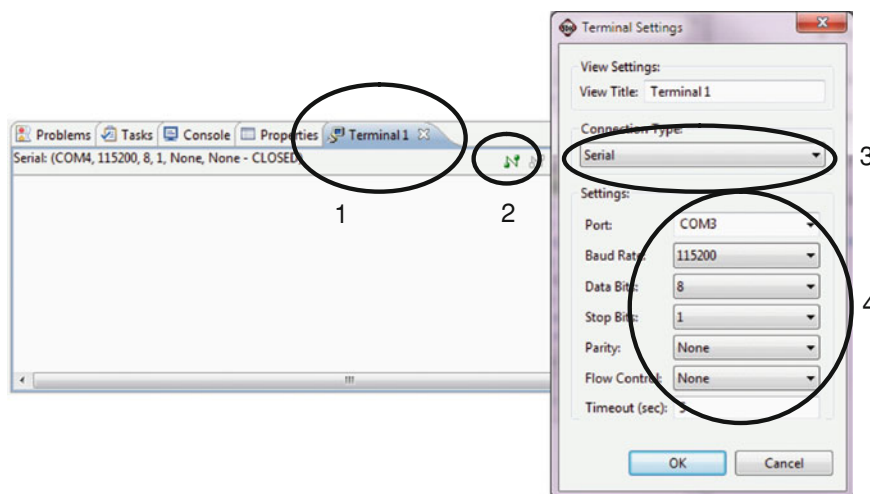
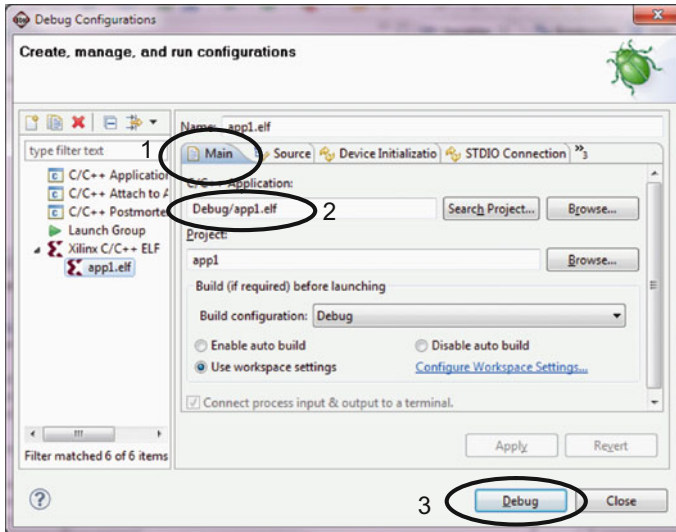


Fig. 15.18 Configuration of the terminal in SDK

purpose (see Fig. 15.18). Press the reset button of the FPGA board and play with the terminal.

The Xilinx Microprocessor Debugger (XMD) [13] is a low-level tool which manages the programming and debugging of the embedded system through the MDM peripheral and the JTAG programming cable. The user can interact with the XMD clicking the SDK menu *Xilinx Tools* → *XMD console*



**Fig. 15.19** Configuration of the debugger

```
XMD% fpga -f
C:/edk13.1/led7seg/SDK/workspace/hw_platform_0/download.bit

Fpga Programming Progress
.....10...20...30.....70....80...90....Done
Successfully downloaded bit file.
```

The debugging permits the programmer to inspect variables, insert breakpoints or execute step-by-step, in order to correct or improve the executable. Click the menu *Run* → *Debug Configurations* to open the dialog shown in Fig. 15.19. Ensure it selects the ELF under the target *Debug*. SDK will ask to switch to a new perspective which facilitates the debugging tasks, therefore it is recommended to confirm it. The debug perspective shows the source code, disassembly, variables, microprocessor's registers, memory, breakpoints, XMD console and more. The user can debug the executable, manually launching XMD commands which is quite uncomfortable. The SDK debugger relies on XMD to send or retrieve data through a graphical view. The debugger starts uploading the ELF into the BRAM and suspending it at the first executable line of the source file.

Set a breakpoint on a line of the timer's ISR and resume the application to observe the display refreshing (see Fig. 15.20). The tab *Variables* shows the local variables of the ISR that are updated when the user plays with the switches and resumes the application. The tab *Expressions* permits to display the object *Display* and the global variables *data* and *endloop*.

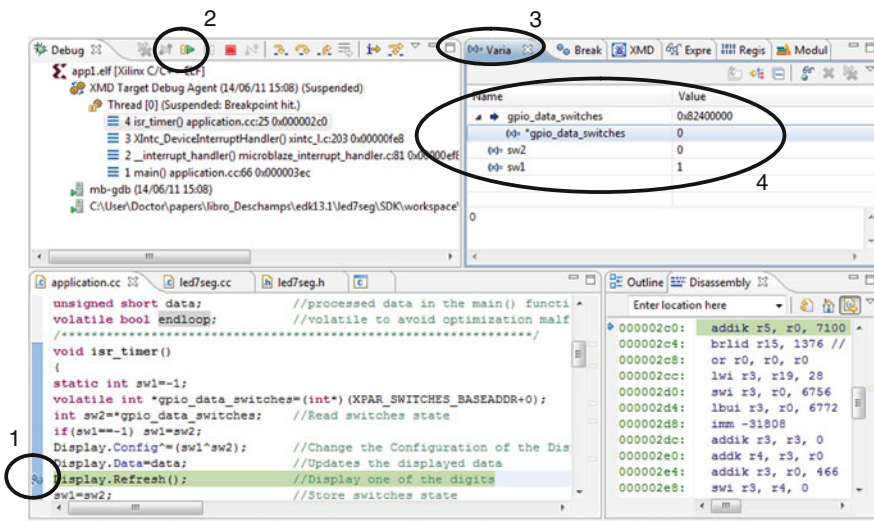


Fig. 15.20 Debugging the application

In a similar way the user can place another breakpoint at the assignment of the variable `rs232_char` in the ISR of the UART. The application will suspend when it receives a new character from the PC. Then, the ISR updates the data or the configuration of the display.

## 15.2 Case Study 1-B: Creating a Custom Peripheral

The previous embedded system devotes three peripherals (two GPIOs and a timer) to drive the display and to read the switches. The executable uses the timer's ISR to periodically read the switches and refresh the display. A dedicated peripheral can improve the design since it can replace the ISR and peripherals that are devoted to a specific task. The main disadvantage is the greater design effort since the designer must develop the source files of the hardware and the software driver. The designer will surely require simulating the source VHDL files, in order to verify and modify the hardware of the peripheral.

This case study starts from the previous one. It modifies the hardware design and the executable to incorporate the new peripheral. Copy the previous project folder and rename it to `led7seg_ip`. Then, open the XMP file to launch the XPS.

### 15.2.1 Design of a Custom Peripheral

A peripheral performs a specific task on hardware. Typically they are attached to the PLB bus as a slave in order to permit the microprocessor to access their



internal registers to control them. More sophisticated peripherals can be attached as PLB masters in order to access memory themselves. These kinds of peripherals are quite harder to develop and they are much less common, therefore, they are not covered in this example.

The hardware of a peripheral is implemented from a set of source VHDL files. The driver of a peripheral is compiled from a set of source C and H files to build the BSP. The source files must be organized in a set of named folders in order to use the new peripheral in the EDK and SDK.

### 15.2.1.1 Hardware Design

The hardware is described in a set of VHDL files that are synthesized during the design flow. EDK requires two files in order to permit the integration of the peripheral to the embedded system: the MPD and PAO. The peripheral wizard from XPS generates the folders and a set of template files that must be modified in order to develop the desired IP. Open the wizard by clicking the menu *Hardware* → *Create or Import Peripheral*.

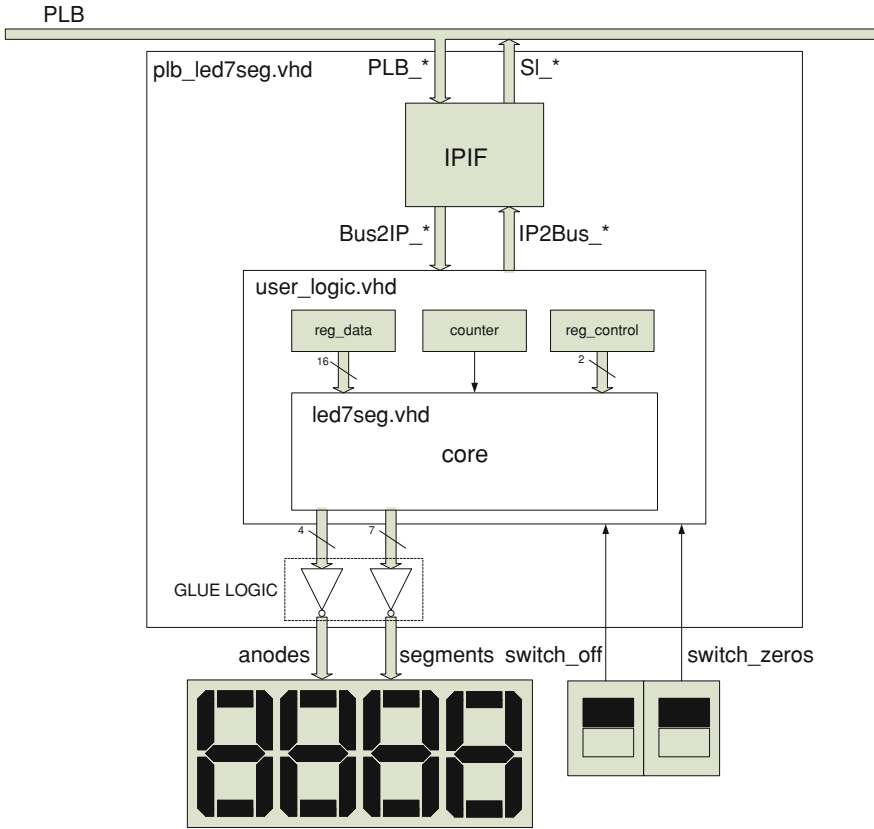
- (1) Select the option *Create templates for a new peripheral*.
- (2) Select the *XPS project repository* to create the template files into the local repository which is stored in the XPS directory. The folders *pcores* and *drivers* contain the local repository of the hardware and software.
- (3) Set the name the peripheral to *plb\_led7seg*, and the version to *1.00.a*. The wizard will create the folders *plb\_led7seg\_v1\_00\_a* in the local repositories.
- (4) Select the PLB attachment.
- (5) Unselect all the slave and master services of the IPIF (IP InterFace) to generate the simplest template files. The IPIFs are IPs deployed with EDK to facilitate the PLB interface.
- (6) Continue with the default options of the next three dialogs.
- (7) Select the option *Generate template driver file*. This option creates the template files for the driver in the software repository.

The MPD and PAO files are stored in the folder *data* which is nested in the local hardware repository *pcores*.

The Microprocessor Peripheral Description (MPD) file [14] is composed of four sections. The first section declares the description and implementation options. The second section sets the bus interface to Slave PLB (SPLB). The last two sections declare the parameters and ports of the SPLB. The data generated by the wizard in the MPD template must not be modified. However, the designer can add more parameters and ports to extend the peripheral. The parameters will be passed as VHDL generics and ports during the synthesis.

The new peripheral adds three parameters in order to configure the peripheral from XPS. The refresh period is an integer parameter which contains the number of microseconds, and its default value is set to *5,000*  $\mu$ s. The boolean data configure the active state of the ports that drive the display. The peripheral adds two output ports to





**Fig. 15.21** Hierarchical schematic of the peripheral drive the 7 segments and the 4 anodes of the display. Finally, it declares two input ports that attaches to the external switches that configure the display.

```

BEGIN plb_led7seg
...
## PERIPHERAL's SPECIFIC GENERICS
PARAMETER C_REFRESH_PERIOD_US = 5000, DT = integer
PARAMETER C_SEGMENTS_ACTIVE_LOW = false, DT = boolean
PARAMETER C_ANODES_ACTIVE_LOW = false, DT = boolean
## PERIPHERAL's SPECIFIC PORTS
PORT segments = "", DIR=0, VEC=[6:0]
PORT anodes = "", DIR=0, VEC=[3:0]
PORT switch_zeros= "", DIR=I
PORT switch_off= "", DIR=I
END
    
```

Figure 15.21 shows a hierarchical schematic of the peripheral and related files. The wizard created two VHDL files that are stored in the folder *hdl\vhdl* which is nested in the hardware repository. The *plb\_led7seg.vhd* is the top-level file which connects an instance of the user logic to the PLB. The *user\_logic.vhd* is a dummy peripheral, therefore, the file must be modified to perform the desired functionality. The tasks related to the timer's ISR are now described as hardware in this file. The computation related to the class *CLed7Seg* class is now described in the new hardware file *led7seg.vhd*.

The Peripheral Analyze Order (PAO) file [14] is the ordered list (bottom to top level) of libraries and files required to synthesize the IP. The first two entries refer to EDK libraries due to the selected IPIF. Then, it continues with the list of VHDL files that will be synthesized into the target library. The target library must be named as the repository folder of the peripheral.

```
lib proc_common_v3_00_a all ##EDK library
lib plbv46_slave_single_v1_01_a all ##EDK library
lib plb_led7seg_v1_00_a led7seg vhd1 ##Added
lib plb_led7seg_v1_00_a user_logic vhd1 ##user logic
lib plb_led7seg_v1_00_a plb_led7seg vhd1 ##top-level
```

The top-level file declares the entity *plb\_dec7seg* and its architecture. The template of the entity leaves space to add new generics and ports, therefore, the user must complete it. The VHDL generics and ports of the entity must be declared in the same way as in the MPD file.

```
ENTITY plb_led7seg IS
  GENERIC (
    -- ADD USER GENERICS BELOW THIS LINE -----
    c_refresh_period_us: INTEGER:=5000;
    c_segments_active_low: BOOLEAN:=FALSE;
    c_anodes_active_low: BOOLEAN:=FALSE;
    -- ADD USER GENERICS ABOVE THIS LINE -----
    ...);
  PORT (
    -- ADD USER PORTS BELOW THIS LINE -----
    segments: OUT STD_LOGIC_VECTOR(6 DOWNT0 0);
    anodes: OUT STD_LOGIC_VECTOR(3 DOWNT0 0);
    switch_off: IN STD_LOGIC;
    switch_zeros: IN STD_LOGIC;
    -- ADD USER PORTS ABOVE THIS LINE -----
    ...);
```

The architecture declares an instance to the user logic. It leaves space to map the new generics and ports. The two switches are directly connected from the top-level

input ports to the user logic. The glue logic attaches the output ports from the user logic to the top-level ports, in order to drive the display. The synthesizer will infer inverters between these ports if the configuration parameters are set to true. The user logic will provide its internal timer, therefore, the instance requires a new generic which configures the number of clock cycles to refresh the display. It is computed from the MPD parameters that define the refreshing period (microseconds) and the clock period (picoseconds). Finally, the user logic also requires a generic which configures the number of internal registers.

```

user_logic_i : ENTITY plb_led7seg_v1_00_a.user_logic
  GENERIC MAP (
    -- MAP USER GENERICS BELOW THIS LINE -----
    c_refresh_counts => (c_refresh_period_us*1000)
                        / (c_splb_clk_period_ps/1000),
    -- MAP USER GENERICS ABOVE THIS LINE -----
    c_num_reg => user_num_reg,
    ...)
  PORT MAP (
    -- MAP USER PORTS BELOW THIS LINE -----
    anodes=> user_anodes,
    segments=> user_segments,
    switch_off=> switch_off,
    switch_zeros=> switch_zeros,
    -- MAP USER PORTS ABOVE THIS LINE -----
    ...);
...
--Glue logic
segments <=NOT user_segments WHEN c_segments_active_low
        ELSE user_segments;
anodes <=NOT user_anodes WHEN c_anodes_active_low
        ELSE user_anodes;

```

The architecture also contains the IPIF instance which eases the PLB connection. It adapts the PLB handshake to/from IP signals named *ipif\_Bus2IP\_\*/ipif\_IP2Bus\_\**. The IPIF also decodes the PLB address bus to enable one of the peripheral's internal registers. The architecture declares the number of internal registers in the user logic, which is two in this case. This number affects the width of the signals *Bus2IP\_RdCE* (Read Chip Enable) and *Bus2IP\_WrCE* (Write Chip Enable) that arrives to the user logic. When MicroBlaze executes a read/write instruction to a peripheral's address range, the IPIF decoder sets one of the enable bits, in order to read/write one of the internal registers of the user logic. The IPIF maps registers on 32-bit boundaries from the peripheral's base address. The first register is mapped on *C\_BASEADDR*, the second register on *C\_BASEADDR+4*, and so on. In order to simplify the IPIF, the *C\_HIGHADDR* parameter is usually

configured much higher than necessary, but it leads to an incomplete address decoding. In this case, the peripheral's registers can be accessed from different addresses since the IFPF does not decode some bits of the address bus.

```
CONSTANT user_slv_num_reg : INTEGER := 2;
CONSTANT user_num_reg: INTEGER := user_slv_num_reg;
```

The template of the user logic is a dummy design which must be modified. The entity template declares an important generic which is the number of registers. The entity must be completed according to the top-level file. Therefore, the designer must add the new generics to configure the refreshing counts and the new ports to drive the display and to read the switches. The default values of the generics are overwritten since they are passed from the instance at the top-level file.

```
ENTITY user_logic IS
  GENERIC (
    -- ADD USER GENERICS BELOW THIS LINE -----
    c_refresh_counts:          INTEGER := 1000;
    -- ADD USER GENERICS ABOVE THIS LINE -----
    c_num_reg:                 INTEGER := 1 );

  PORT (
    -- ADD USER PORTS BELOW THIS LINE -----
    segments:                 OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
    anodes:                   OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
    switch_zeros:             IN STD_LOGIC;
    switch_off:               IN STD_LOGIC;
    -- ADD USER PORTS ABOVE THIS LINE -----
    ...);
```

The architecture of the user logic declares two registers to control the display that are accessible from the PLB. The first one (*reg\_data*) is a 16-bit data register which sets the 4 digits to display. The second one (*reg\_control*) is a 2-bit register which controls the display configuration: one bit to turn on/off the display and the other bit to show/hide the left-hand zeros. The input ports *Bus2IP\_RdCE* and *Bus2IP\_WrCE* provide a single bit for each register to read or write them. The data comes from the port *Bus2IP\_Data* during a write access to one of the registers. During a read access, one of the registers drives the port *IP2Bus\_Data*. The architecture uses the signals *bus\_data/ip\_data* to get/set the data bus, since they are declared in descending bit order as the registers. The PLB must acknowledge the access completion before MicroBlaze can continue with a new access. The user logic asserts the ports *IP2Bus\_RdAck/IP2Bus\_WrAck* when it completes the access to the registers. In this case, the user logic does not require wait states to read/write the registers.

```

SIGNAL reg_control: STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL reg_data: STD_LOGIC_VECTOR (15 DOWNTO 0);
...
bus_data <= Bus2IP_Data;           --Reads data bus
IP2Bus_Data <= ip_data;           --Writes data bus
IP2Bus_WrAck <= or_reduce(Bus2IP_WrCE); --Acknowledges write
IP2Bus_RdAck <= or_reduce(Bus2IP_RdCE); --Acknowledges read
IP2Bus_Error <= '0';

read_reg: PROCESS(Bus2IP_RdCE, reg_data, reg_control)
BEGIN
    ip_data<=(OTHERS=>'0');           --default value
    CASE Bus2IP_RdCE(0 TO 1) IS      --read a register
        WHEN "10"=> ip_data(15 DOWNTO 0)<=reg_data;
        WHEN "01"=> ip_data (1 DOWNTO 0)<=reg_control;
        WHEN OTHERS=> NULL;
    END CASE;
END PROCESS;

write_reg: PROCESS(Bus2IP_Reset, Bus2IP_Clk)
BEGIN
    IF Bus2IP_Reset='1' THEN          --reset registers
        reg_data<=(OTHERS=>'0'); reg_control<=(OTHERS=>'0');
    ELSIF RISING_EDGE(Bus2IP_Clk) THEN
        IF Bus2IP_WrCE(0)='1' THEN   --write reg_data
            reg_data<=bus_data(15 DOWNTO 0);
        END IF;
        IF Bus2IP_WrCE(1)='1' THEN    --write reg_control
            reg_control<=bus_data(1 DOWNTO 0);
        ELSIF refresh='1' THEN        --refresh reg_control
            reg_control<=reg_control XOR (sw1 XOR sw2);
        END IF;
    END IF;
END PROCESS;

```

The rest of the user logic performs the tasks executed in the timer's ISR in the software application. It implements a counter which periodically asserts the signal *refresh*. This signal is used to refresh the display and to capture the state of the switches in order to modify the control register. There is an instance, named *core*, of the IP *led7seg*, which drives the display from the registers contents.

```

core: ENTITY plb_led7seg_v1_00_a.led7seg (beh1) PORT MAP (
  rst=> Bus2IP_Reset,      --Reset input
  clk=> Bus2IP_Clk,       --Clock input
  refresh=> refresh,      --refresh input
  segments=> segments,    --Drive the display
  anodes=> anodes,        --Drive the display
  data=> reg_data,        --Input from data register
  zeros=> reg_control(0), --Input from control register
  off=> reg_control(1) ); --Input from control register

```

The file *led7seg.vhd* implements the functionalities described in the C++ class *CLed7Seg* in the software application. It generates the signals that drive the display, updating the displayed digit when the port *refresh* is asserted. The architecture is composed of two processes. The first process updates the index of the digit to refresh, starting at the left side. The second process computes the display's ports based in the input ports *off*, *zeros* and *data* that are driven from the peripheral's registers.

### 15.2.1.2 Driver Design

A peripheral's driver is built from header and implementation files. The C files implement functions that control the peripheral. The header H files contain the declarations required to use the functions. The SDK requires two additional files, the MDD and the TCL, and a structured folder organization in a repository in order to build the BSP with the peripheral's driver. The Microprocessor Driver Definition (MDD) file [14] declares the supported peripherals, dependencies of the driver and files to copy. The TCL file is a script used to compile the source files and build the library. The XPS peripheral wizard generates the template files and folders. The designer usually does not modify the MDD and TCL files, but he must rewrite the H and C files to develop the driver.

The MSS file controls the peripheral drivers that are built in the BSP. The Libgen tool creates a folder named as the microprocessor and copies the H files into the folder *include*. Then it compiles the C files and adds the functions into the library *libxil.a* which is stored in the folder *lib* of the BSP.

The header file contains the declarations that are used to build the BSP and the executable. First, it defines the masks to control the display. Next, it declares C macros to read or write the peripheral's registers through a pointer. Finally, there are the declarations of the visible functions.

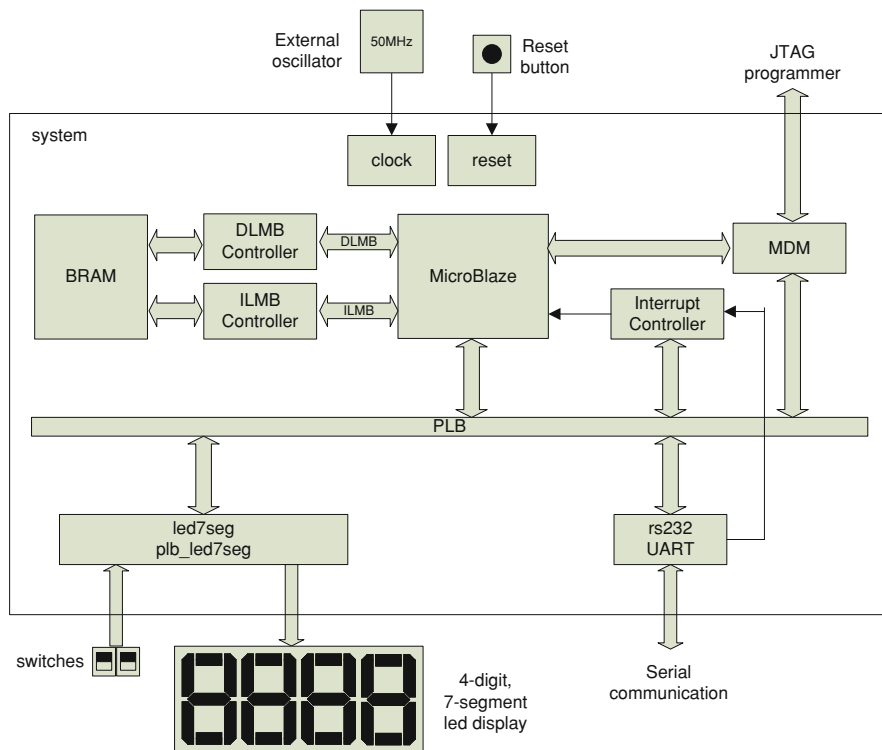
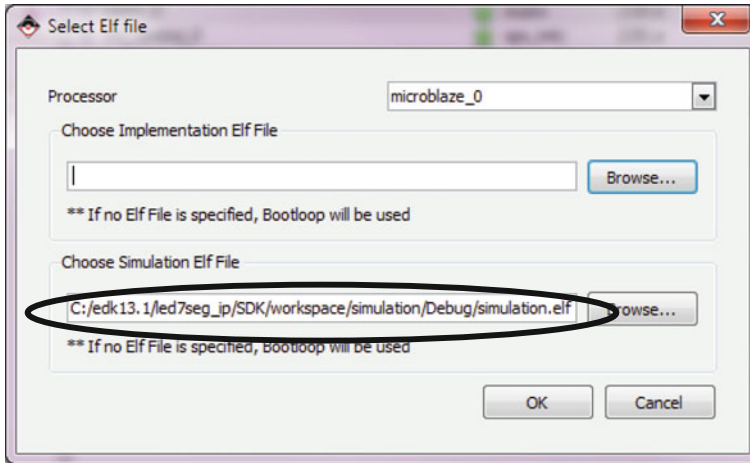


Fig. 15.22 Overview of the modified system architecture

```
#define LED7SEG_ZEROS_MASK (0x1) //Display left-side zeros
#define LED7SEG_OFF_MASK (0x2) //Turn-off display
...
#define Led7Seg_GetStatus(baseaddr)
    *((volatile int*)(baseaddr+4)) //Read macro
#define Led7Seg_SetControl(baseaddr, config)
    *((volatile int*)(baseaddr+4))=(int)(config) //Write macro
void Led7Seg_SwapOff(int baseaddr); //Function
void Led7Seg_SwapZeros(int baseaddr); //Function
```

The C file of the driver is compiled during the BSP generation. The library file stores the functions, but not the C macros. The library may store other internal functions that are not visible for the programmer since they are not declared in the H file. The function which swaps one of the bits of the control register executes two accesses to the register. First, MicroBlaze reads the register and changes one bit according to a mask. Next, it writes the resulting value into the same register.



**Fig. 15.23** Selection of the ELF to generate the simulation model

```
void Led7Seg_SwapControl(int baseaddr, char mask)
{Led7Seg_SetControl( baseaddr,
                    Led7Seg_GetStatus(baseaddr)^(mask) );}
void Led7Seg_SwapOff(int baseaddr)
{Led7Seg_SwapControl( baseaddr, LED7SEG_OFF_MASK );}
```

## 15.2.2 System's Hardware/Software Modification

The new peripheral replaces the timer and the GPIOs connected to the display and switches, as well as the software code related to them. Therefore, the hardware and software of the new system must be modified, accordingly.

### 15.2.2.1 Hardware Modification

The MHS file of the new system adds the instance *led7seg* of the PLB display controller *plb\_led7seg*, and it removes the timer and GPIOs, as depicted in Fig. 15.22. The external ports connected to the display and switches are assigned to the display controller. The parameters that configure the refresh period and active state of the anodes and segments can be easily modified to adapt them to other FPGA board. The MHS parameters overwrite the default values defined in the peripheral's MPD file. The modified system requires just one interrupt source for the UART, therefore, the interrupt controller could be eliminated. However, the system maintains it since it does not increase significantly the area and it provides the possibility to add new peripherals to request interrupts.



```

PORT fpga_0_led7seg_segments_pin=led7seg_segments, DIR = O,
    VEC = [6:0]
PORT fpga_0_led7seg_anodes_pin=led7seg_anodes, DIR = O,
    VEC = [3:0]
PORT fpga_0_led7seg_zeros_pin=led7seg_switch_zeros, DIR = I
PORT fpga_0_led7seg_off_pin=led7seg_switch_off, DIR = I
...
BEGIN plb_led7seg
    PARAMETER INSTANCE = led7seg
    PARAMETER HW_VER = 1.00.a
    PARAMETER C_BASEADDR = 0x84C00000
    PARAMETER C_HIGHADDR = 0x84C0FFFF
    PARAMETER C_REFRESH_PERIOD_US = 4000
    PARAMETER C_SEGMENTS_ACTIVE_LOW = true
    PARAMETER C_ANODES_ACTIVE_LOW = true
    BUS_INTERFACE SPLB = mb_plb
    PORT segments = led7seg_segments
    PORT anodes = led7seg_anodes
    PORT switch_zeros = led7seg_switch_zeros
    PORT switch_off = led7seg_switch_off
END

BEGIN xps_intc
    ...
    PORT Intr = rs232_Interrupt
END

```

The names of the external FPGA ports, connected to the display and switches, have changed, therefore, the UCF must be updated.

```

Net fpga_0_led7seg_segments_pin<0> LOC=E14; #segment-A
Net fpga_0_led7seg_anodes_pin<0> LOC=D14; #anode-0
Net fpga_0_led7seg_zeros_pin LOC=K13; #switch zeros
...

```

At this point, the designer can easily test the peripheral's hardware using the XMD tool. Using the XPS, implement the new hardware. Then, click *Device Configuration* → *Download Bitstream* to create the *download.bit* file and program the FPGA. If the XPS project has no ELF file, the bitstream configures the BRAM to store the default executable *bootloop* which runs a dummy loop. In order to program the FPGA, XPS calls the iMPACT tool with the commands of the batch file *download.cmd*. Check the number of the *-p* flag, which configures the FPGA position in the JTAG chain.

Once the FPGA is successfully programmed, click the *Debug* → *Launch XMD* to open the command shell. The XMD shell initially shows the microprocessor configuration when it connects to the MDM. Then, the user can test the peripheral when he runs the XMD commands: *mwr* (memory write) and *mrd* (memory read). Write the peripheral's base address ( $0 \times 84C00000$ ) to change the displayed number. Write the next register ( $0 \times 84C00004$ ) to change the configuration of the display. The peripheral's registers are accessible from different addresses due to the incomplete IPIF decoder.

```

...
MicroBlaze Processor Configuration :
-----
Version.....8.10.a
Optimization.....Area
Interconnect.....PLB_v46
...
Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0)...

XMD% mwr 0x84c00000 0x0AF4      (Display number 0AF4)
XMD% mwr 0x84c00004 0x1       (Display left-side zeros)
XMD% mwr 0x84c00004 0x2       (Turn off display)
XMD% mrd 0x84c00000 4         (Read 4 registers. It shows)
84C00000:  00000AF4           (The data register)
84C00004:  00000002           (The control register)
84C00008:  00000AF4           (The data register)
84C0000C:  00000002           (The control register)

```

### 15.2.2.2 Software Modification

At the software level, the executable has to be also modified in order to delete the code related to the GPIOs and the timer's ISR. Export the new hardware from XPS and create a SDK workspace in the *c:\edk13\Ned7seg\_ip\SDK\workspace* path. Check that the imported hardware presents the new display controller.

Create a new BSP project and select the standalone OS. By default, the MSS assigns the generic driver to the display controller *led7seg*. A more convenient way is to assign the specific driver to the peripheral in order to build the BSP.

```

BEGIN DRIVER
  PARAMETER DRIVER_NAME = plb_led7seg
  PARAMETER DRIVER_VER = 1.00.a
  PARAMETER HW_INSTANCE = led7seg
END

```

The Libgen tool generates the BSP according to the information described in the MSS file. It gets the source files from the folder *plb\_led7seg\_v1\_00\_a* which must be stored in the driver repositories. Then, it copies the header files and builds the libraries. The file *libgen.options* must be modified to configure the local repository to the XPS directory.

```
PROCESSOR=microblaze 0
REPOSITORIES=-lp ../../../../
HWSPEC=../hw_platform_0/system.xml
```

The C++ code related to the GPIOs and timer's ISR is deleted. Therefore, the files that declare and implement the class *CLed7Seg* are deleted in the project. The modified file *application.cc* calls the driver's functions to control the display.

```
#include <plb_led7seg.h>          //Driver's header file
...
void isr_rs232()
{...
    switch(rs232_char){           //Received key
        case '+':                //Increments displayed number
            Led7Seg_SetData(XPAR_LED7SEG_BASEADDR, ++data); break;
        case 'z':                //Show/hide the left-side zeros
            Led7Seg_SwapZeros(XPAR_LED7SEG_BASEADDR); break;
        ...}
    ...}
```

The application can be programmed and debugged following the same steps as in the previous case study. Check that the BIT and BMM files are the imported ones from the current XPS project, before proceeding.

### 15.2.3 Functional Simulation

The source VHDL files should be simulated if the peripheral is not working as expected. The easiest way to test the VHDL files is by performing a functional simulation of the embedded system. The system can be simulated with the ISim or other supported simulator. The XPS can select the simulator when you click the *Edit → Preferences*. The Xilinx ISim has some limitations when it is compared with other third-party simulators. However, the ISim is easy to use, and it provides the simulation libraries.

Instead of simulating the application executable, which depends on the serial communication, it is preferable to develop a much simpler executable. Create a

new Xilinx C++ project in SDK named *simulation*. Modify the C++ template file *main.cc* to add a sequence which just writes the peripheral's registers, as desired. The function *delay* is called to improve the observation on the simulation results. Build the target *Debug* to get the file *simulation.elf*.

```
Led7Seg_SetData(XPAR_LED7SEG_BASEADDR, 0x001E);
Led7Seg_SetControl(XPAR_LED7SEG_BASEADDR, 0x0);
delay();
Led7Seg_SetControl(XPAR_LED7SEG_BASEADDR, LED7SEG_ZEROS_MASK);
delay();
...
Led7Seg_SwapOff(XPAR_LED7SEG_BASEADDR);
delay();
...
```

The XPS project must import the ELF file in order to generate the simulation model. Click the menu *Project* → *Select Elf file* and choose the file *simulation.elf* which is stored in the SDK project (see Fig. 15.23)

XPS launches the Simgen tool [13] to generate the set of files required to simulate the embedded system running the selected executable. Click the XPS menu *Project* → *Project Options* which opens the dialog window shown in Fig. 15.24. Select the options: VHDL, generate testbench template and behavioural model. This is the recommended model since it simulates the peripheral's source files. The other models are more complicated, since they simulate the output files from the synthesis or the implementation stages. Click the XPS menu *Simulation* → *Generate Simulation HDL Files* to call the Simgen tool. The Simgen does not support some FPGA families, such as the Spartan-3, but there is a workaround. Choose any supported device since it does not affect the behavioural simulation of the peripheral, because it does not require the synthesizing of its source VHDL files. Finally, click the menu *Simulation* → *Launch HDL Simulator* which takes some time to compile the simulation files before starting the ISim [19].

Simgen creates the testbench file *system\_tb.vhd*. It declares the instance *dut* (Device Under Test) of the system and the stimulus applied to its inputs. ISim shows the hierarchical view of the instances to simulate. Open the contextual menu of the top-level instance *system\_tb* and click *Go To Source Code* in order to edit the testbench template, as shown in Fig. 15.25.

The template file drives the clock and reset signals, and it provides a user's section to write more stimulus. In order to simulate the external switches, change the state of the switch, which turns off/on the display, at 250 and 260  $\mu$ s.

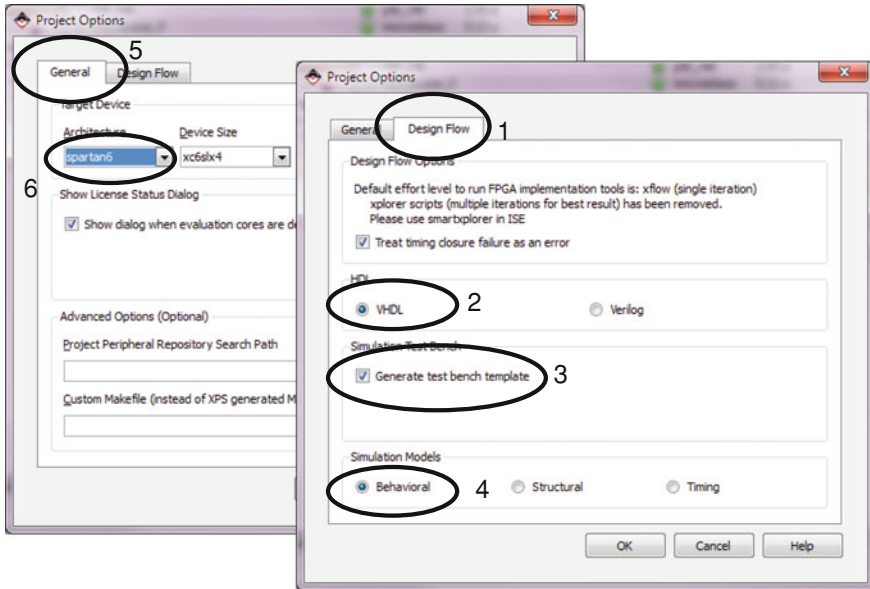


Fig. 15.24 Configuration of the simulation model

```

dut : system PORT MAP (
    fpga_0_clk_1_sys_clk_pin=>fpga_0_clk_1_sys_clk_pin,
    fpga_0_rst_1_sys_rst_pin=>fpga_0_rst_1_sys_rst_pin,
    ...
    fpga_0_led7seg_zeros_pin=>fpga_0_led7seg_zeros_pin,
    fpga_0_led7seg_off_pin =>fpga_0_led7seg_off_pin);
...
-- START USER CODE (Do not remove this line)
fpga_0_led7seg_zeros_pin<='0';
fpga_0_led7seg_off_pin<='0', '1' AFTER 250 us,
                        '0' AFTER 260 us;
-- END USER CODE (Do not remove this line)

```

The simulation will take a huge number of clocks to refresh the display. The display's ports are updated every 200,000 clock cycles when the refresh cycle is set to 4,000  $\mu$ s (50 MHz clock frequency). Each instance of a peripheral is simulated from a wrapper file which configures the IP according the MHS file. Go through the hierarchical view, and select the instance *led7seg*. Open the wrapper file *led7seg\_wrapper.vhd*, and change the generic *C\_REFRESH\_PERIOD\_US* to shorten the simulation of the refresh. This change does not affect the synthesis or the implementation since it does not modify the MHS file.

Fig. 15.25 Hierarchical view of the instances to open the testbench file

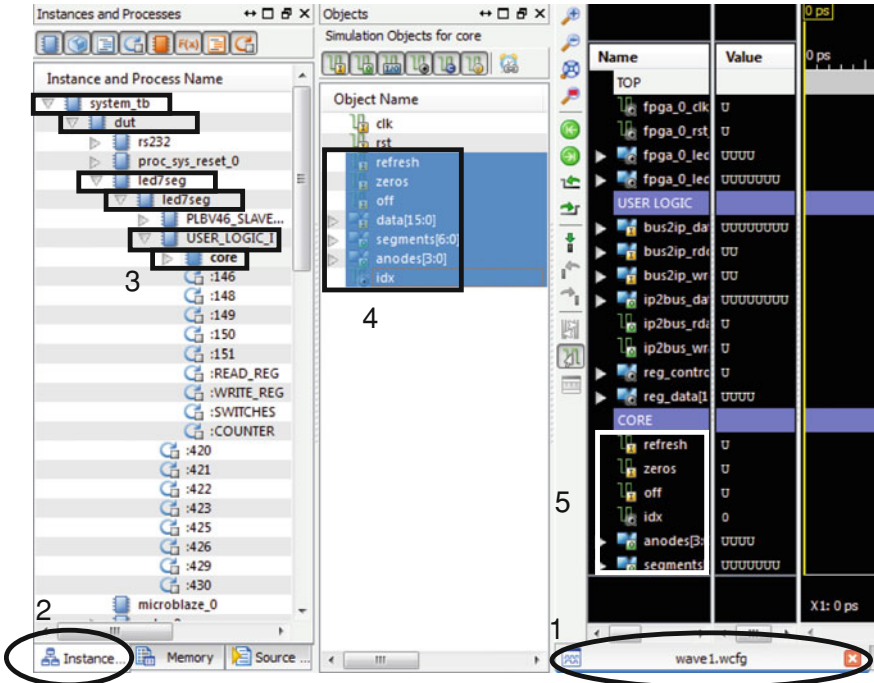
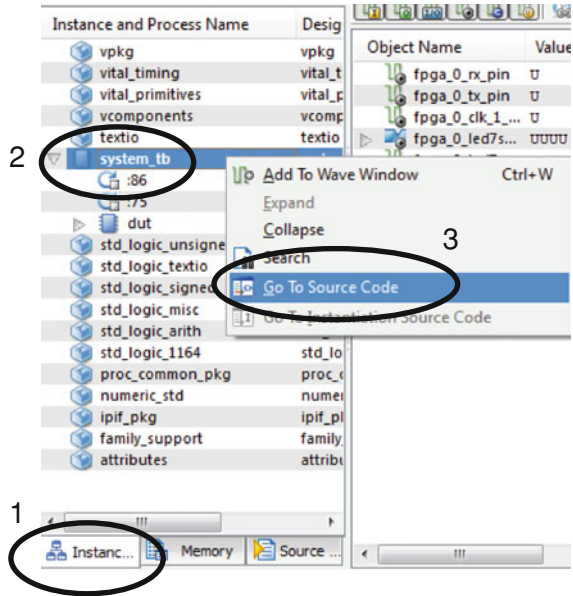


Fig. 15.26 Configuring the waveform window

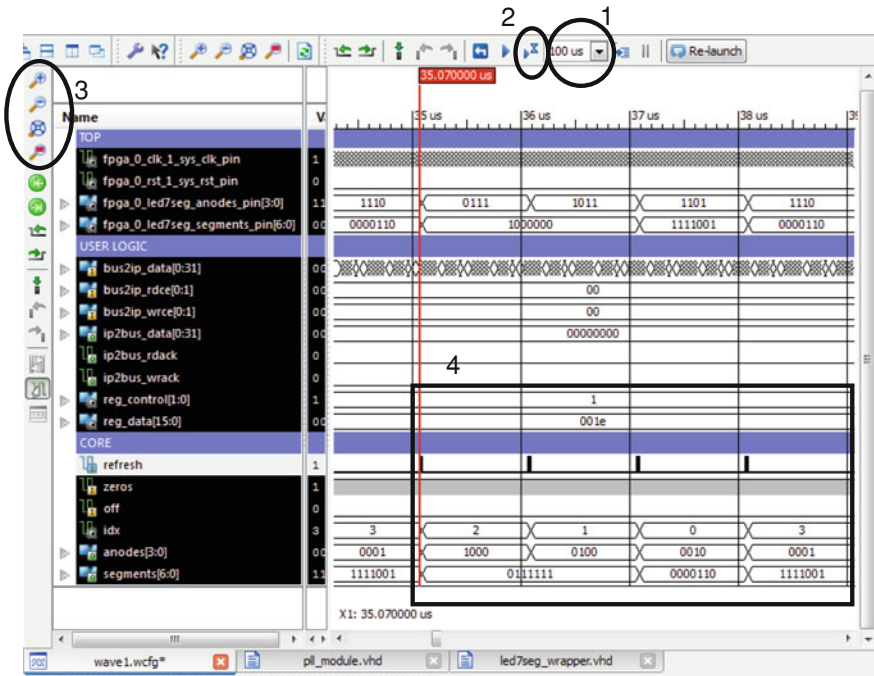


Fig. 15.27 Simulation waveform

```

led7seg : plb_led7seg
  GENERIC MAP (
    c_baseaddr => X"84C00000",
    c_highaddr => X"84C0FFFF",
    ...
    c_refresh_period_us => 1,
    ...)
  PORT MAP (...);
  
```

--Parameters from the MHS  
 --Changed to simulate

Any change in the VHDL files requires the compilation of the model before starting the simulation. Therefore, click the ISim menu *Simulation* → *Relaunch*. In order to display the external ports of the system, select the instance *system\_tb* and drag the desired signals into the waveform window. Add a divider, named *USER LOGIC*, in order to display a new group of waveforms separately from the previous ones. Go to the hierarchy of the instances *system\_tb* → *dut* → *led7seg* → *led7seg* → *USER\_LOGIC\_I* and drag the IPIF signals and the peripheral’s registers to the waveform. Repeat the previous steps to display the signals of the instance *core*, as shown in Fig. 15.26.

By default, signals are displayed in binary radix which is uncomfortable for some signals. Select the peripheral's registers and the IPIF busses to change the radix to hexadecimal.

Write 100  $\mu\text{s}$  on the simulation time, and run the simulation, as shown in Fig. 15.27. The resulting waveform can be zoomed in/out to show the desired time interval. Check that the signal *refresh* is asserted during a clock cycle every 1  $\mu\text{s}$ , due to the change done in the wrapper file. Display the peripheral's registers at 35  $\mu\text{s}$ . At this point MicroBlaze has already executed the first four lines of the C++ file, therefore, the peripheral's registers (*reg\_data*, *reg\_control*) contain the expected values. The ports *anodes* and *segments* that drive the display are refreshed according to the value stored in those registers.

The simulation should continue another 100  $\mu\text{s}$  to display how Microblaze executes the driver function *Led7Seg\_SwapOff*. The function executes two consecutive accesses to the register *control\_reg*: a read followed by a write access. The registers are accessed when the microprocessor executes read/write instructions to memory addresses within the peripheral's address range. The IPIF decodes the PLB address bus in order to assert one of the enabler bits of the signals *Bus2IP\_RdCE/Bus2IP\_WrCE* that arrive to the user logic, in order to read/write a peripheral's registers. The register *control\_reg* is associated with the index 1 of these signals. The waveform window permits a search for the next/previous transition of a signal. Search a rising edge of the bit *Bus2IP\_RdCE(1)*, as shown in Fig. 15.28. At this time, the MicroBlaze is reading the control register. The peripheral carries the register's content on the signal *IP2Bus\_Data* and it asserts the *IP2Bus\_RdAck* to complete the read access. The next access, MicroBlaze writes the control register. The peripheral gets the new content from the *Bus2IP\_Data* and asserts the *IP2Bus\_WrAck* to complete the write access.

Figure 15.29 shows the simulation of the user action on a switch at 250  $\mu\text{s}$  and 260  $\mu\text{s}$  to turn off/on the display, as described in the testbench file. The peripheral updates the control register when the signal *refresh* is asserted, after the user action.

If the C/C++ code of the simulation executable is changed it is necessary to close the ISim before executing Simgen. Save the waveform file before exiting ISim. Then, build the SDK project to get the new ELF file and generate the new simulation model with XPS. Finally, launch ISim, open the saved waveform file, and run a new simulation.

Once the simulation is finished, select the target FPGA in the project options and generate the bitstream. The system will be synthesized and implemented for the selected device.

### 15.3 Case Study 2: Implementation of a Custom Coprocessor

A coprocessor is a specific processor aimed to accelerate an algorithm. The hardware architecture is designed to efficiently speed-up a specific computation. The rest of computations and tasks are carried on the general-purpose



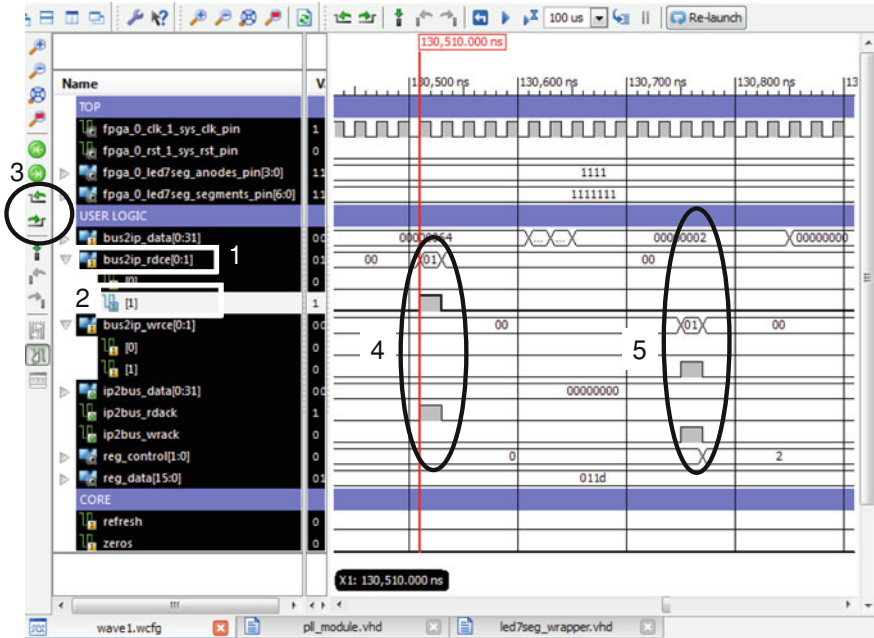


Fig. 15.28 Simulation of a read/write accesses to a peripheral's register

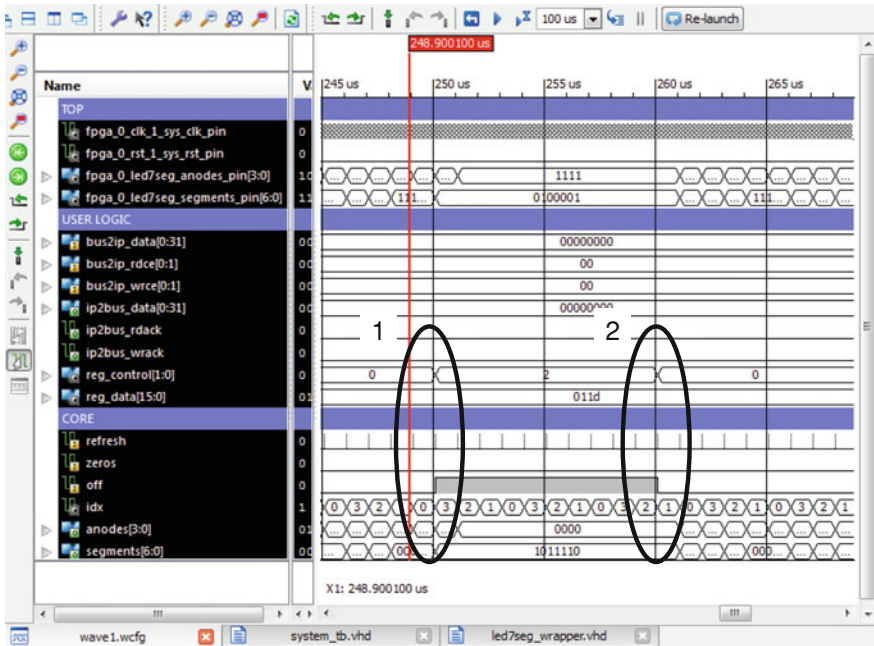
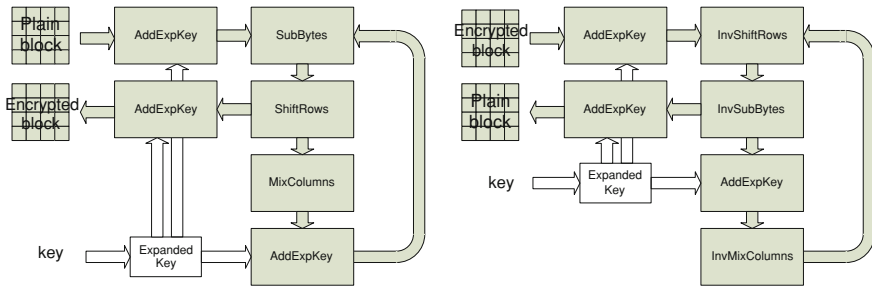


Fig. 15.29 Simulation of the user's action on a switch



**Fig. 15.30** Block encryption (*left*) and decryption (*right*) in the AES-128 cipher

microprocessor. The microprocessor writes commands and data to the coprocessor's registers in order to perform a computation. When the coprocessor completes, the microprocessor retrieves the computed data in order to continue the algorithm.

This case study presents an embedded system which communicates with an external PC in order to set the state of some led diodes and to read the state of some switches. The system is connected through the serial port, but the communications are ciphered using the Advanced Encryption Standard (AES-128). The system decrypts the commands received from the PC and encrypts the answer messages. The MicroBlaze cannot decrypt the messages at the desired speed, therefore, a coprocessor is developed to accelerate the AES-128 computation.

### 15.3.1 A Brief Introduction to the AES-128 Cipher

The AES [1] standard comprises three symmetric-key block ciphers: AES-128, AES-192 and AES-256, where the number denotes the key size. The cipher encrypts a fixed-size block of plain text to get a block of ciphered text. It also decrypts ciphered blocks in the reverse way using the same key. The block size of the AES ciphers is 128 bits independently of the key size.

All the steps executed during AES encryption/decryption are done using a variable named *state*. It is a 128-bit data arranged on an array of 4-rows x 4-columns of 8-bit elements (bytes). During the AES-128 encryption the *state* is initially written with a block of plain text. Then it is processed 9 rounds, and each round is composed of 4 steps, as shown in Fig. 15.30. The initial and the final rounds are slightly different since some steps are skipped. The AES *state* stores intermediary computations and the block of ciphered text at the end of the final round. In a similar way the decryption initializes the *state* with the ciphered block which is processed with inverted computations and steps to get the block of plain text.

One of the AES steps applies the expanded key which must be computed from the cipher key. The expansion of the key can be computed online or offline. The

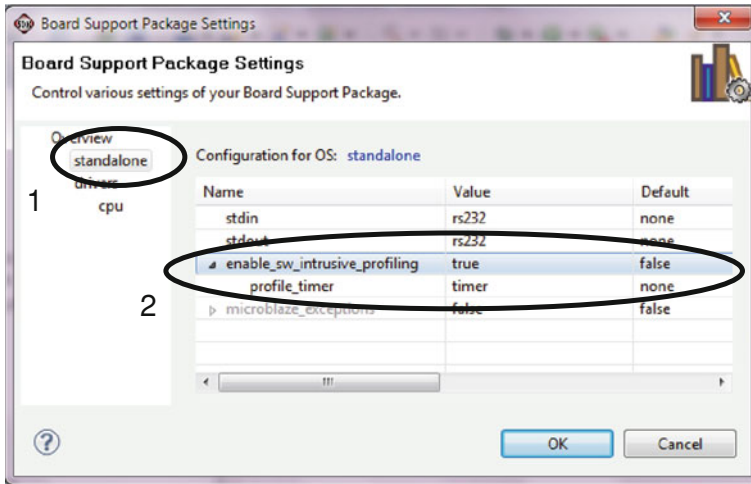


Fig. 15.31 Enabling and configuring the profiling in the BSP

offline mode computes the expanded key before starting the rounds, therefore, it is computed only when the cipher key changes.

The transmitter divides a message into blocks that are encrypted and transmitted. The receiver decrypts the received blocks to build the message. There are several operation modes and padding schemes to permit block ciphers to work with messages of any length [22]. This case study chooses the Electronic Code-Book (ECB) mode and null padding, due to its simplicity. The other modes require the IV (Initialization Vector) generator and a different block partitioning, but this fact does not affect the AES block cipher.

### 15.3.2 Software Implementation of the AES-128 Cipher

In order to study the applicability of a coprocessor on this application, the AES-128 cipher is implemented as a C++ class named *CAES128*. The system receives/transmits ciphered messages from/to the external PC and it uses the C++ class to decrypt/encrypt blocks according to the cipher key. Microblaze reacts to the messages to set the leds or to report the state of the switches.

The hardware specification is quite similar to the first case study, but it provides a single GPIO peripheral to read two switches and to set two leds. Copy the folder which allocates the XPS project of the first case study and rename it as *co-processor*. Edit the MHS file to add the 4-bit width GPIO and connect its bidirectional port to an external FPGA port. The timer will not only measure the time taken to encrypt/decrypt blocks, but it will also be used for the cipher's profiling. The profiler requires the timer to be able to interrupt the

microprocessor, therefore, the MHS connects their interrupt ports. The system removes the interrupt controller since there is a single interrupt source.

```

PORT fpga_0_gpio_pin=gpio_GPIO_IO, DIR=IO, VEC=[3:0]
...
BEGIN xps_gpio
  PARAMETER INSTANCE = gpio
  PARAMETER C_GPIO_WIDTH = 4
  PORT GPIO_IO = gpio_GPIO_IO
  ...
END

BEGIN microblaze
  PARAMETER INSTANCE = microblaze_0
  ...
  PORT INTERRUPT = timer_interrupt
END

BEGIN xps_timer
  PARAMETER INSTANCE = timer
  ...
  PORT Interrupt = timer_interrupt
END

```

The UCF file is modified to attach the external port to the switches and leds according the FPGA board. Implement the hardware and export it to SDK.

```

Net fpga_0_gpio_pin<0> LOC=K12;          #led-0
Net fpga_0_gpio_pin<1> LOC=P14;        #led-1
Net fpga_0_gpio_pin<2> LOC=K13;        #switch-0
Net fpga_0_gpio_pin<3> LOC=K14;        #switch-1

```

Open the SDK workspace `c:\vedk13.\ncoprocessor\SDK\workspace`. Modify the BSP settings to enable the profiling and configure the profile timer (see Fig. 15.31). Clean and build the new BSP which includes a new library which is required when the application is profiled.

Delete the previous C++ projects and create a new project named *server* which targets the new BSP. The server application is composed of four files.

The files *caes128.h* and *caes128.cc* are the source files of the C++ class *CAES128* which implements the cipher. It is a straightforward implementation in order to facilitate the next steps. The class can be improved, in terms of speed and security, but it requires a deeper knowledge of the AES. The class provides the method *SetKey* to compute the expanded key at the beginning of the application. Then, it provides methods to *Encrypt* or *Decrypt* a single 128-bit block.

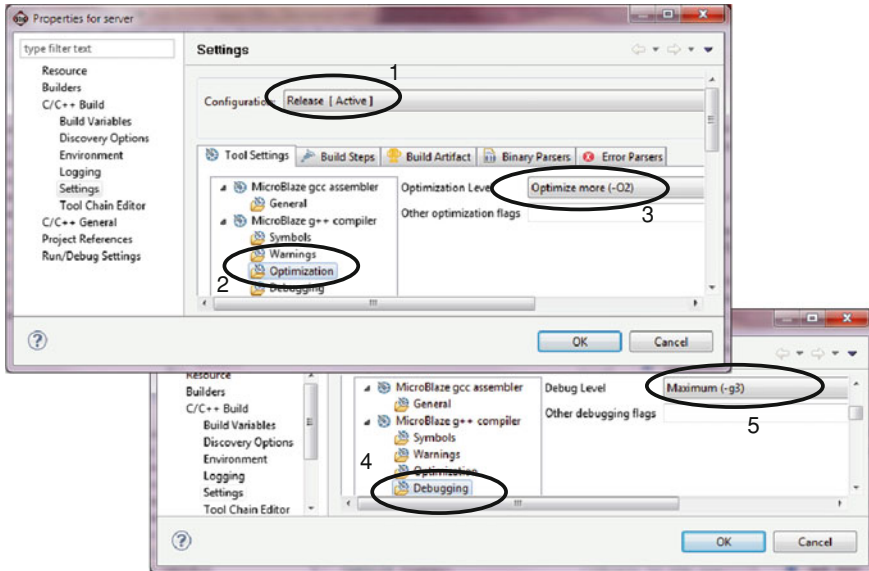
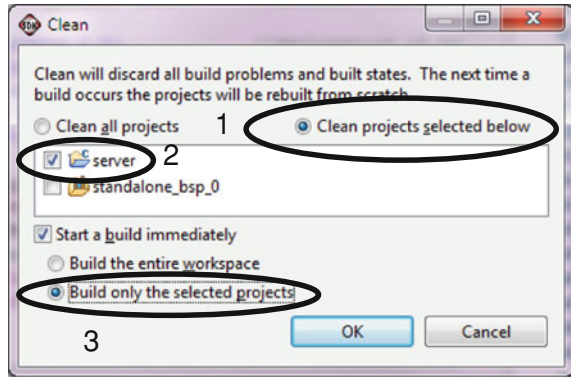


Fig. 15.32 Settings to build the release target

```
#define AES128_BLOCK_SIZE (128)           //128 bits
...
class CAES128
{
private:
    unsigned char State[4][4];           //The state variable
    ...
public:
    void SetKey(unsigned char key[AES128_KEY_SIZE/8]);
    void Encrypt(unsigned char in[AES128_BLOCK_SIZE/8],
                unsigned char out[AES128_BLOCK_SIZE/8]);
    void Decrypt(unsigned char in[AES128_BLOCK_SIZE/8],
                unsigned char out[AES128_BLOCK_SIZE/8]);
};
```

The file *app.cc* implements the application. It waits for commands launched from a PC in order to set the leds or to report the state of the switches. Any command produces an answer message which is transmitted to the PC. Commands and answer messages are encrypted during their transmission on the serial interface. The function *send\_rs232\_cipher* implements the ECB operation mode and null padding. It divides an answer message into 128-bit blocks that are individually encrypted and transmitted. In a similar way, the function *get\_rs232\_cipher* builds a command message from the received and decrypted blocks.

**Fig. 15.33** Project cleaning and building



```

CAES128 Cipher;                                //Object to the AES-128 C++ class
...
void send_rs232_cipher(char *string)
{unsigned char block[AES128_BLOCK_SIZE/8];
char c=0xFF;
do{
    for(unsigned char j=0; j<AES128_BLOCK_SIZE/8; j++)
        {if(c) c=string[k++];                //null padding
        block[j]=c;}
    ...
    Cipher.Encrypt(block,block);            //ECB mode
    ...
    for(unsigned char j=0; j<AES128_BLOCK_SIZE/8; j++)
        XUartLite_SendByte(XPAR_RS232_BASEADDR,block[j]);
    } while(c);                            //Repeat until the end of the string
}
...
int main()
{unsigned char key[AES128_KEY_SIZE/8]={'0','1',...,'f'};
Cipher.SetKey(key);                        //Computes the expanded key
...
while(!quit)
    {send_rs232_cipher("Enter CMD: ");
    XUartLite_SetControlReg(...); //Reset the receiving buffer
    get_rs232_cipher(input,INPUT_NCHARS);
    if(check_rs232())                //Check overrun or framing errors
        command(input);}           //Parse the received command
...}

```

The application stores into the variables *EncryptMaxCycles/DecryptMaxCycles* the maximum number of clock cycles taken to encrypt/decrypt a block. It relies on the file *cchronometer.h* which implements the C++ class to get the number of clock cycles from the timer. The compiler can skip the code related to the

chronometer when the *CIPHER\_CHRONOMETER* declaration is commented. The application also declares the *CIPHER\_DISABLE* which can be used to skip the cipher to facilitate debugging.

Click the menu *Project* → *Properties* to edit the settings of the build of the target *Release*, as shown in Fig. 15.32. The compiler can optimize the code to improve the execution speed, by changing the order of instructions or the unrolling of loops. The ELF may not fit in the BRAM memory if the optimization level at the maximum (*-O3*). In order to display the measured encryption/decryption time, lower the optimization level (*-O2*) and set the debug to the maximum level (*-g3*).

Set the *Release* as the active target. Then, click the menu *Project* → *Clean* to clean the C++ project in order to build it from scratch (see Fig. 15.33).

In order to test or debug the application, the PC must also use the AES-128 cipher. There are many PC programs that permit the ciphering of messages through a communication channel, but they usually employ sophisticated protocols (SSH, SSL, etc.) to authenticate users before transmitting the cipher keys. The presented application is much simpler, since the PC and the embedded system are programmed with the same cipher key. In order to test the application, the PC executes a Linux program to encrypt/decrypt messages from/to a console which redirects the input/output to the serial port. The folder *linux\_aes128* stores the executable files for Linux or Cygwin. Cygwin permits the execution of applications written for Linux on Windows PC. Cygwin cannot execute the Linux binaries, but it requires the building of the executable from the Linux source files.

Install Cygwin or a Linux virtual machine and open a shell to launch a set of commands. The first command changes to the directory which contains the path of the executable (it may change). The next one configures the communication parameters on the serial port (*/dev/com1* may change). The third command maps a file descriptor to the serial port. The last two commands decrypt or encrypt messages between the console and the serial port. The cipher key is set to the same value as the embedded system. Finally, program the FPGA and test the application. The Linux console shows unreadable text if the cipher keys are mismatched.

```
$ cd /cygdrive/c/edk13.1/linux_128
$ stty -F /dev/com1 115200 cs8 -parenb -cstopb -crtscts -isig
$ exec 9<>/dev/com1
$ ./aes128 -d -c -k "0123456789abcdef" <&9 &
$ ./aes128 -e -c -k "0123456789abcdef" >&9
```

In order to display the time taken to encrypt/decrypt blocks, debug the ELF of the target *Release* using SDK. Set a breakpoint at the last instruction of the C++ file to pause the execution when the user launches the quit command. Then resume the application and launch several commands to the embedded system through the Linux console. The variables *EncryptMaxCycles* and *DecryptMaxCycles* contain

the maximum number of clock cycles required to encrypt and decrypt a block. The decryption of a block takes 1.76 ms (87,338 clock cycles, 50 MHz clock frequency), but the serial communication requires 1.39 ms to transmit it (115,200 bps). The FIFO of the UART may overrun during the reception of large messages since the MicroBlaze cannot decrypt blocks at the required speed. To solve the problem, the transmission speed can be lowered or the system can implement a flow control. A better solution is to accelerate the most time-consuming computations.

### 15.3.3 Profiling

The profiler [12] is an intrusive tool which is used to test the application performances. It reports the number of calls and the execution time of every function. The profiling requires a dedicated timer able to interrupt the microprocessor in order to sample the program counter. The source files of the application must be compiled to add profiling ISR and code (compiler switch—*pg*). The linker attaches to the profiling library to build the executable.

The server application is not adequate to profile the since most of the execution time is devoted to wait for user messages from the serial port. A better approach is to execute a new application which continuously encrypts and decrypts messages. Therefore, create a new C++ project named *profiling*. The application file, named *profiling\_app.cc*, uses the class *CAES128* to encrypt/decrypt messages in a loop. Change the compiler switches of the target *Release* to enable the profiler and the same optimization level as the server application (switches *-pg -O2*). Next, clean the project in order to build the application from scratch.

The profiler collects data and stores it in memory during the execution. Once the application completes, the collected data is downloaded to the PC in order to analyze it. The SDK must set the profiler memory which cannot be overlapped to the application memory. Use the SDK to program the FPGA with the imported bitstream and BMM, and set the ELF to *bootloop*. Open the XMD console to launch some commands. The first command changes the working directory to the application's folder. The second command establishes the connection to the MicroBlaze's MDM. The last command tries to download the ELF file into memory. It fails since the profiler memory is not defined, but it reports the allocated memory of the ELF. The hardware implements the local memory from the address  $0 \times 0000$  to  $0 \times 3FFF$ , and the XMD reported there is available free memory from address  $0 \times 2248$ .



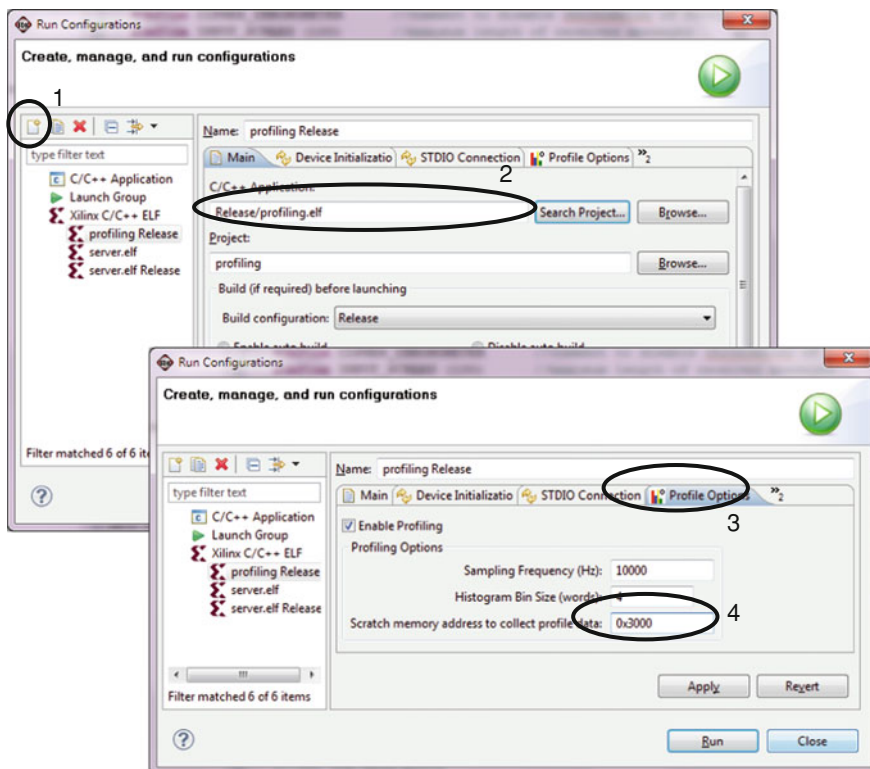


Fig. 15.34 Run configuration (to profile)

```
XMD% cd c:/edk13.1/coprocessor/SDK/workspace/profiling
XMD% connect mb mdm
XMD% dow Release/profiling.elf
```

```
Downloading Program -- Release/profiling.elf
section, .vectors.reset: 0x00000000-0x00000003
...
section, .stack: 0x00001e44-0x00002247
```

```
ERROR: Failed to download ELF file
Memory needed for Profiling not defined
```

The profiling data can be stored into any free space of memory. Click the menu *Run* → *Run Configurations* to open the dialog depicted in Fig. 15.34. Then add the application to profile and set the profile memory from the  $0 \times 3000$  address. Finally, run the application and wait until the application ends. The PC downloads

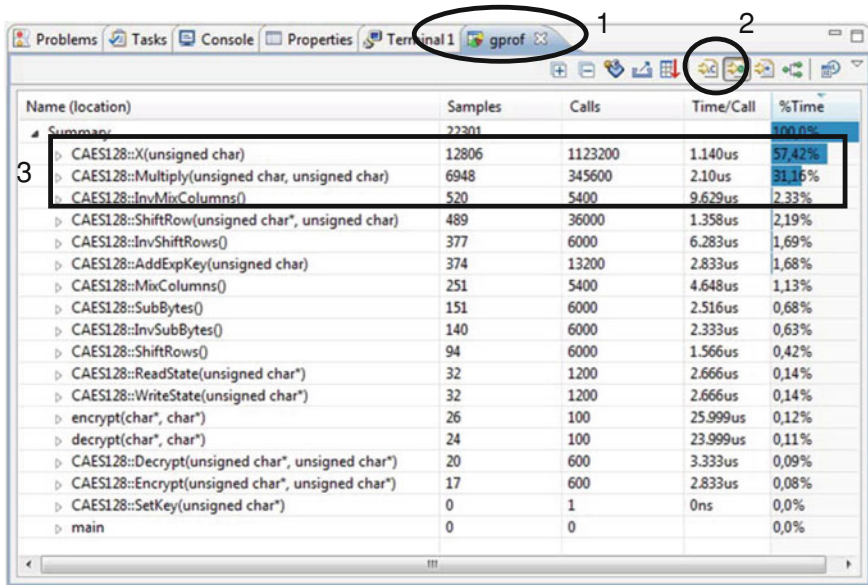


Fig. 15.35 Collected data from the application profiling

the collected data which is stored in the file *gmon.out*. It is a binary file which is interpreted by the GNU *gprof* tool.

Double click on the file *gmon.out* to display the results (see Fig. 15.35). The SDK shows a graphical view of the collected data which can be arranged in several ways, as the percentage of execution time devoted to each function. Two methods of the *CAESI28* class take the 88% of the processing time: *X* and *Multiply*. They are child functions called from the function *InvMixColumn* which is one of the steps executed during the decryption. The child function *X* is also called from the step *MixColumn* during the encryption.

The profiling information can be used to re-implement the most time-spending functions to improve the execution time. However, a coprocessor can greatly accelerate a specific computation.

### 15.3.4 Coprocessor Design

The coprocessor aims to accelerate the step *InvMixColumn* and its child functions. The step *MixColumn* is quite similar, therefore, the hardware design of both steps does not significantly increase the effort. The coprocessor will accelerate the computations that take the 92% of the processing time obtained from the profiler.

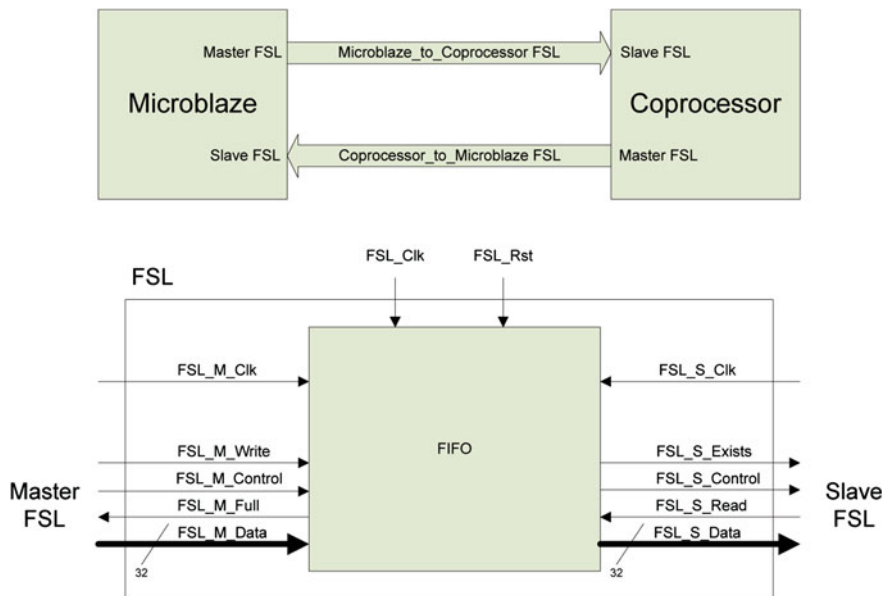


Fig. 15.36 Attachment of the MicroBlaze and coprocessor (top), and FSL schematic (bottom)

Copy the previous XPS folder and rename it as *coprocessor\_ip* in order to develop the coprocessor, its driver, and to update the hardware and software of the application.

### 15.3.4.1 Hardware Design

Coprocessors are attached to MicroBlaze through Fast Simplex Link (FSLs), as depicted in Fig. 15.36 (top). A MicroBlaze’s master-FSL connects to the coprocessor’s slave-FSL in order to write the coprocessor’s registers. In the reverse way, the MicroBlaze’s slave-FSL permits the reading of the coprocessor’s registers.

The FSL [21] is a point-to-point link which permits a low latency and fast communication due to its simple handshake. The FSL does not provide an address bus, as seen in Fig. 15.36 (bottom). Therefore, the coprocessor’s registers must be sequentially accessed in a predetermined order. Each FSL provides, by default, a 16-depth FIFO which can temporarily store data written from the master, when the slave is not ready to read it. The FSL provides a single 32-bit width bus (*FSL\_M\_Data*, *FSL\_S\_Data*) which can carry data or commands, depending on the control signal (*FSL\_M\_Control*, *FSL\_S\_Control*). In order to read data, the FIFO signals when data is available to read (*FSL\_S\_Exists*), and the slave acknowledges

when data is retrieved (*FSL\_S\_Read*). Similarly, in order to write data, the FIFO signals when there is no free space (*FSL\_M\_Full*), and the master requests to write data (*FSL\_M\_Write*).

XPS can create the template files for coprocessors through the same wizard used for peripherals. Launch the wizard by clicking the menu *Hardware* → *Create or Import Peripheral*:

- (1) Choose create templates in the XPS directory
- (2) Set the name to *fsl\_mixcolumns* and the version to *v1.00.a*
- (3) Select FSL attachment and go ahead with the default settings
- (4) Select to implement the driver template and finish the wizard

The coprocessor does not require additional parameters, ports or VHDL files, therefore, the MPD and PAO files are not modified. The template file *fsl\_mixcolumns.vhd* is modified to design the coprocessor. It implements a 4x4 matrix of 8-bit registers (*reg\_state*) to compute and store the AES *state*. Additionally, a 1-bit register (*reg\_mode*) configures the computation mode as *MixColumns* or *InvMixColumns*. MicroBlaze will execute a control-type write instruction to the coprocessor's slave FSL to set the mode register. Next, it follows four data-type write instructions to set the registers of the AES *state*. Matrices are stored in rows in the microprocessor's memory, and each 32-bit data sets the 4 registers of a row in the coprocessor. The coprocessor starts the computation when the registers *reg\_state* and *reg\_mode* have been written. The coprocessor must acknowledge the read of data to the slave-FSL in order to delete it from the FIFO.

The coprocessor computes and stores the result in the registers *reg\_state*. The coprocessor writes the resulting data to its master-FSL in a quite similar way. It writes a new row on the FSL when the computation is completed and the FIFO is not full. MicroBlaze executes four read instructions to the FSL in order to retrieve the resulting AES *state*.

A key idea to accelerate a computation is in parallel processing. A full-parallel implementation could compute the entire array of the AES *state* in a single clock cycle, although it may occupy a large area. However, MicroBlaze would not completely take profit of this architecture since it takes several more clock cycles to execute the FSL instructions to set and retrieve the AES *state*. A semi-parallel architecture offers a good trade-off between speed and area. The coprocessor computes the left-side column of the *reg\_state* in a clock cycle. The state register shifts one column and the computation is repeated for the next 3 columns. Therefore, it takes 4 clock cycles to complete.

```

SUBTYPE t_byte IS STD_LOGIC_VECTOR(7 DOWNTO 0);
TYPE t_state IS ARRAY(0 TO 3,0 TO 3) OF t_byte;
TYPE t_col IS ARRAY(0 TO 3) OF t_byte;
TYPE t_row IS ARRAY(0 TO 3) OF t_byte;
...
SIGNAL reg_mode: STD_LOGIC;
SIGNAL reg_state, state: t_state;
SIGNAL row,resrow: t_row;
...
p_read_sfsl: PROCESS(...) --Slave-FSL to write registers
VARIABLE cnt_mode: INTEGER RANGE 0 TO 1;
VARIABLE cnt_row: INTEGER RANGE 0 TO 4;
BEGIN
  IF RISING_EDGE(fsl_clk) THEN --Counts number of FSL writes
    IF fsl_rst='1' OR start='1' THEN
      cnt_row:=0; cnt_mode:=0;
    ELSE
      IF wr_mode='1' THEN cnt_mode:=cnt_mode+1; END IF;
      IF wr_row='1' THEN cnt_row:=cnt_row+1; END IF;
    END IF;
  END IF;

  IF fsl_s_exists='1' AND fsl_s_control='0' AND
    NOT(cnt_row=4) THEN wr_row<=ready; --enables write
  ELSE wr_row<='0'; END IF; --to state register
  IF fsl_s_exists='1' AND fsl_s_control='1' AND
    NOT(cnt_mode=1) THEN wr_mode<=ready; --enables write
  ELSE wr_mode<='0'; END IF; --to mode register

  fsl_s_read<=wr_mode OR wr_row; --Acknowledge slave-FSL
  mode<=fsl_s_data(fsl_s_data'RIGHT); --The LSB sets the mode
  row<=(fsl_s_data(0 TO 7), fsl_s_data(8 TO 15),
    fsl_s_data(16 TO 23), fsl_s_data(24 TO 31));
  idx_row<=CONV_UNSIGNED(cnt_row, idx_row'LENGTH);--row index

  IF cnt_mode=1 AND cnt_row+1=4 AND wr_row='1' --starts the
    THEN start<='1'; ELSE start<='0'; END IF; --coprocessor
END PROCESS;

p_state: PROCESS
...
BEGIN
  WAIT UNTIL RISING_EDGE(fsl_clk);
  IF fsl_rst='1' THEN
    ready<='1';
  ELSIF ready='1' THEN
    IF wr_mode='1' THEN --Writes the mode register
      reg_mode<=mode;
    END IF;
    IF wr_row='1' THEN --Writes a row in state register
      reg_state<=f_set_row(reg_state, idx_row, row);
    END IF;
  ...
END PROCESS;

```

```

col0<=f_get_col(reg_state,0);           --Get column#0
ncol0<=f_mix_col(col0) WHEN reg_mode='0' ELSE
    f_invmix_col(col0);                --Computes column#0
state<=f_set_col(reg_state,0,ncol0);   --Stores column#0

p_state: PROCESS
    VARIABLE cnt_col: INTEGER RANGE 0 TO 4;
    VARIABLE ncol: t_col;
    BEGIN
        WAIT UNTIL RISING_EDGE(fsl_clk);
        IF fsl_rst='1' THEN
            ready<='1';
        ELSIF ready='1' THEN
            ...
            IF start='1' THEN
                ready<='0'; cnt_col:=0;      --Starts computation
            END IF;
        ELSE
            reg_state<=f_lshift_col(state);--Shifts state register
            cnt_col:=cnt_col+1;
            IF cnt_col=4 THEN
                ready<='1';                  --Computation completes
            END IF;
        END IF;
    END PROCESS;

```

### 15.3.4.2 Driver Design

The XPS wizard generates the H and C templates for the coprocessor's driver. The driver provides a single function which sends the AES *state* and gets the result through FSLs. It executes a control-type write instruction followed by four data-type instructions to set the computation mode and the AES *state*. The *state* is transmitted in rows to avoid rearrangement of data from memory to the FSL. The last instructions read the resulting *state*.

The function uses macros to set/get data to/from FSLs. During the BSP generation, the compiler replaces them by the MicroBlaze instructions that read or write an FSL slot. MicroBlaze provides 16 master-FSLs and 16 slave-FSLs. The input slot is the FSL index from which the coprocessor reads the input data (coprocessor's slave-FSL). The output slot is the FSL index which connects to the coprocessor's master-FSL. The slot names must match with the name of the coprocessor's instance (*fsl\_mixcolumns\_0*) which will be declared in the MHS file, otherwise the BSP will fail.

```

#include <xparameters.h>
    //parameters XPAR_FSL_INSTANCENAME_INPUT/OUTPUT_SLOT_ID
    //INSTANCE_NAME must match with the MHS file
#include <fsl.h>    // cputfsl, putfsl, getfsl macros

#define FSL_INPUT_SLOT
    XPAR_FSL_FSL_MIXCOLUMNS_0_INPUT_SLOT_ID
#define FSL_OUTPUT_SLOT
    XPAR_FSL_FSL_MIXCOLUMNS_0_OUTPUT_SLOT_ID
void fsl_mixcolumns (unsigned char mode,
                    unsigned char state[4][4])
{
    unsigned int *p1=(unsigned int*)state[0];
    unsigned int *p2=(unsigned int*)state[0];
    cputfsl(mode,FSL_INPUT_SLOT);    //writes mode register
    putfsl(*p1++,FSL_INPUT_SLOT);    //writes row#0 to FSL
    putfsl(*p1++,FSL_INPUT_SLOT);    //        row#1
    putfsl(*p1++,FSL_INPUT_SLOT);    //        row#2
    putfsl(*p1 ,FSL_INPUT_SLOT);    //        row#3
    getfsl(*p2++,FSL_OUTPUT_SLOT);    //reads row#0 from FSL
    getfsl(*p2++,FSL_OUTPUT_SLOT);    //        row#1
    getfsl(*p2++,FSL_OUTPUT_SLOT);    //        row#2
    getfsl(*p2 ,FSL_OUTPUT_SLOT);    //        row#3
}

```

### 15.3.5 Modification of the Embedded System

The hardware must be modified to attach MicroBlaze to the coprocessor through FSLs. The class *CAESI28* must accelerate the computation of the steps *MixColumns* and *InvMixColumns* using the coprocessor's driver.

#### 15.3.5.1 Hardware Modification

The XPS offers a wizard (see Fig. 15.37) to connect a coprocessor when clicking the menu *Hardware* → *Configure Coprocessor*. It modifies the MHS file to attach the coprocessor's instance (*fsl\_mixcolumns\_0*) to MicroBlaze through two FSLs.

```

BEGIN microblaze
  PARAMETER INSTANCE = microblaze_0
  ...
  PARAMETER C_FSL_LINKS = 1
  BUS_INTERFACE SFSL0 = fsl_mixcolumns_0_to_microblaze_0
  BUS_INTERFACE MFSL0 = microblaze_0_to_fsl_mixcolumns_0
END

BEGIN fsl_mixcolumns
  PARAMETER INSTANCE = fsl_mixcolumns_0
  ...
  BUS_INTERFACE MFSL = fsl_mixcolumns_0_to_microblaze_0
  BUS_INTERFACE SFSL = microblaze_0_to_fsl_mixcolumns_0
END

BEGIN fsl_v20
  PARAMETER INSTANCE = fsl_mixcolumns_0_to_microblaze_0
  ...
END

BEGIN fsl_v20
  PARAMETER INSTANCE = microblaze_0_to_fsl_mixcolumns_0
  ...
END

```

### 15.3.5.2 Software Modification

Export the new hardware to SDK and set the workspace to the path *c:\edk13\ncoprocessor\_ip\SDK\workspace* before continuing. By default, the MSS associates a generic driver to the coprocessor. The BSP generation with the coprocessor's driver is similar to the case of the peripheral. First, edit the file *libgen.options* to add the local repository which contains the driver's source files. Then, edit the MSS file to change the driver associated to the coprocessor.

```

BEGIN DRIVER
  PARAMETER DRIVER_NAME = fsl_mixcolumns
  PARAMETER DRIVER_VER = 1.00.a
  PARAMETER HW_INSTANCE = fsl_mixcolumns_0
END

```

Clean and build the BSP project to generate the new BSP from scratch. The software applications use the C++ class *CAESI28* which implements the block



cipher. The member methods *MixColumns* and *InvMixColumns* are modified in order that the coprocessor computes these steps. The class also provides conditional compilation to permit the computation by software for testing purposes.

```
#include "caes128.h"           //declares CAES128 class
#ifdef ENABLE_COPROCESSOR_MIXCOLUMNS
    #include <fsl_mixcolumns.h> //coprocessor's driver
#endif
...
#ifdef ENABLE_COPROCESSOR_MIXCOLUMNS
    //Hardware computation on the coprocessor
    void CAES128::MixColumns()
        {fsl_mixcolumns(MIXCOLUMNS,State);}
    void CAES128::InvMixColumns()
        {fsl_mixcolumns(INVMIXCOLUMNS,State);}
#endif

#ifdef ENABLE_COPROCESSOR_MIXCOLUMNS
    //Software computation on the microprocessor
    ...
#endif
```

The rest of the source files are not modified, since the class *CAES128* carries out the encryption/decryption of blocks. Build the C++ projects to generate the new executables.

### 15.3.6 Simulation

The coprocessor can be simulated in a very similar way as described for the peripheral. The application *profiling* is used to build the simulation model since it continuously encrypts/decrypts blocks using the coprocessor.

Search when the coprocessor asserts the signal *start* which launches the computation, as shown in Fig. 15.38. The waveform previously shows the execution of five FSL write instructions by MicroBlaze. The coprocessor reads the data from its slave-FSL. The first FSL instruction is a control-type access which writes the mode register (*reg\_mode*). The next four FSL instructions are data-type accesses to write the rows of the state register (*reg\_state*). The coprocessor starts the computation after the completion of the five write accesses.

Figure 15.39 shows that the coprocessor takes 4 clock cycles to compute the state register. Then, it writes the four rows of the resulting *reg\_state* to its master-FSL, in order the MicroBlaze can read them. The coprocessor does not have to wait for the MicroBlaze to read the resulting data, since the FIFO of the FSL is not full.

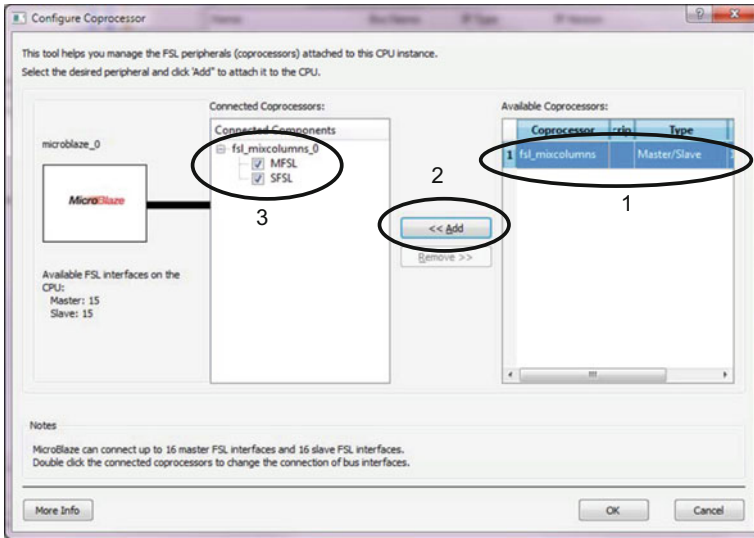


Fig. 15.37 EDK wizard to connect a coprocessor

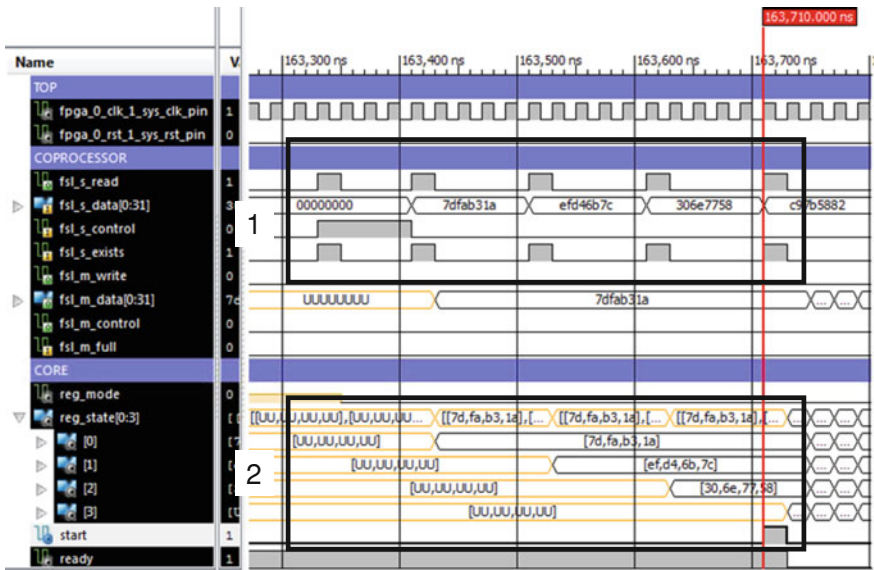


Fig. 15.38 Simulation of the coprocessor's slave FSL

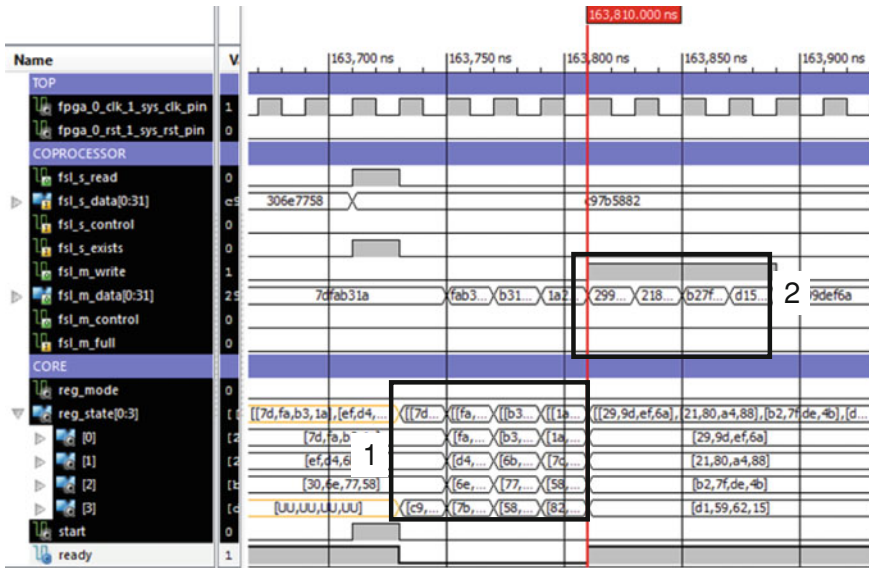


Fig. 15.39 Simulation of the coprocessor computation and the master FSL

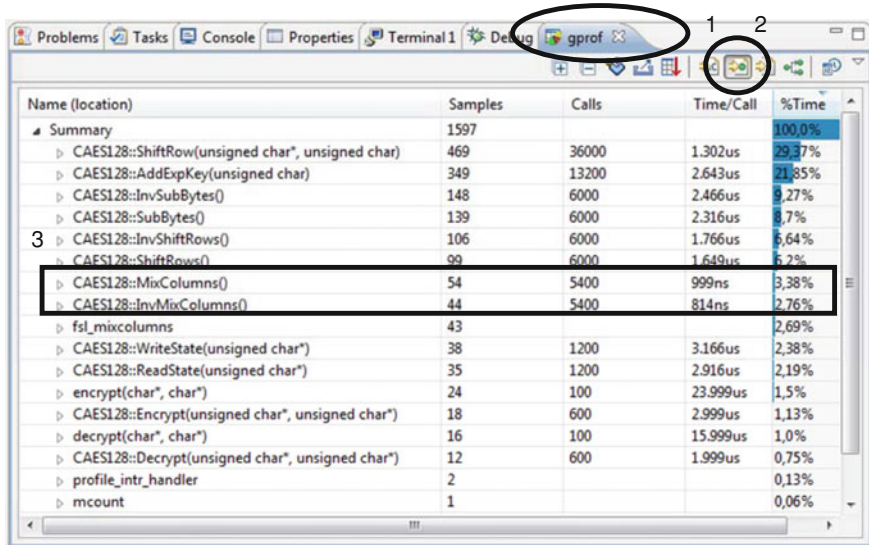


Fig. 15.40 Profiling data with the coprocessor

### 15.3.7 Experimental Results

Follow the steps previously described to collect the new profiling data. Before programming the FPGA, check the directories used to get the BIT, BMM and ELF files are in the current SDK workspace. Figure 15.40 shows the functions *MixColumns* and *InvMixColumns* currently represent only the 6.14% of the execution time. The time in these functions is mainly devoted to the transmitting and receiving of the *state* variable through the FSL instructions.

The application *server* can be tested as described before. The measured number of clock cycles to encrypt and decrypt a block are now 5,189 and 5,150, respectively. Therefore the decryption time is greatly improved from 1.76 ms to 103  $\mu$ s (50 MHz clock frequency), which represents about 17 times faster.

## References

1. Nist (2002) NIST Advanced Encryption Standard (AES) FIPS PUB 197
2. Xilinx (2005) Spartan-3 Starter Kit Board User Guide (UG130)
3. Xilinx (2010a) XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices (UG627)
4. Xilinx (2010b) LogiCORE IP Processor Local Bus PLB v4.6 (DS531)
5. Xilinx (2010c) XPS General Purpose Input/Output (GPIO) v2.00.a (DS569)
6. Xilinx (2010d) LogiCORE IP XPS Timer/Counter v1.02.a (DS573)
7. Xilinx (2010e) LogiCORE UART Lite v1.01.a (DS571)
8. Xilinx (2010f), LogiCORE IP XPS Interrupt Controller v2.01.a (DS572)
9. Xilinx (2010g) Xilinx Processor IP Library. Software Drivers. uartlite v2.00.a
10. Xilinx (2010h) Xilinx Processor IP Library. Software Drivers. tmrctr v2.03.a
11. Xilinx (2010i) Xilinx Processor IP Library. Software Drivers. intc v2.02.a
12. Xilinx (2010j) EDK Profiling Use Guide (UG448)
13. Xilinx (2011a) Embedded System Tools Reference Manual (UG111)
14. Xilinx (2011b) Platform Specification Format Reference Manual (UG642)
15. Xilinx (2011c) Command Line Tools User Guide (UG628)
16. Xilinx (2011d) MicroBlaze Processor Reference Guide (UG081)
17. Xilinx (2011e) LogicCORE Local Memory Bus LMB v10 (DS445)
18. Xilinx (2011f) Standalone (v.3.0.1.a) (UG647)
19. Xilinx (2011g) ISim User Guide v13.1 (UG660)
20. Xilinx (2011h) Data2MEM User Guide (UG658)
21. Xilinx (2011i) LogiCORE IP Fast Simplex Link (FSL) V20 Bus v2.11d (DS449)
22. Menezes AJ, Oorschot PC, Vanstone SA (1996) Handbook of applied cryptography. CRC Press, Boca Raton