



MicroBlaze Tutorial Creating a Simple Embedded System and Adding Custom Peripherals Using Xilinx EDK Software Tools

Rod Jesman Fernando Martinez Vallina Jafar Saniie

1





INTRODUCTION

This tutorial guides you through the process of using Xilinx Embedded Development Kit (EDK) software tools, in which this tutorial will use the Xilinx Platform Studio (XPS) tool to create a simple processor system and the process of adding a custom OPB peripheral (an 32-bit adder circuit) to that processor system by using the Import Peripheral Wizard.

OBJECTIVES

After completing this tutorial, you will be able to:

- Create an XPS Project by using Base System Builder (BSB)
- Create a simple hardware design by using Xilinx IPs available in the Embedded Design Kit
- Add a custom IP to your design
- Modify a Xilinx generated software application to access an IP peripheral
- Implement the design
- Generate and Download the bit file to verify in hardware

In order to download the completed processor system, you must have the following hardware:

- Xilinx Spartan-3 Evaluation Board (3S200 FT256 –4)
- Xilinx Parallel -4 Cable used to program and debug the device
- Serial Cable

PROCEDURE

The purpose of the tutorial is to walk you through a complete hardware and software processor system design. In this tutorial, you will use the BSB of the XPS system to automatically create a processor system and then add a custom OPB peripheral (adder circuit) to that processor system which will consist of the following items:

- MicroBlaze Processor
- Local Memory Bus (LMB) Bus
 - LMB BRAM controllers for BRAM
 - BRAM BLOCK (On-chip memory)
- On-chip Peripheral Bus (OPB) BUS
 - Debug Module (OPB_MDM)
 - UART (OPB_UARTLITE)
 - 2 General Purpose Input/Output pheriphals (OPB_GPIOs)
 - Push Buttons
 - Dip Switches
 - Custom peripheral (32-bit adder circuit)

2





BACKGROUND

First, before designing the embedded processor system, some background information needs to be provided to inform you about the processor to be used and some items about the Xilinx Embedded Development Kit (EDK) software tools. The microprocessors available for use in Xilinx Field Programmable Gate Arrays (FPGAs) with Xilinx EDK software tools can be broken down into two broad categories. There are soft-core microprocessors (MicroBlaze) and the hard-core embedded microprocessor (PowerPC). This tutorial will only focus on the soft-core MicroBlaze microprocessor, which can be used in most of the Spartan-II, Spartan-3 and Virtex FPGA families. The hard-core embedded microprocessor mentioned is an IBM PowerPC 405 processor, which is only available in the Virtex-II Pro and Virtex-4 FX FPGA's. You don't have to use the PowerPC 405 processors but you also can't remove them from the Virtex-II Pro and Virtex-4 FX FPGA's because they are in the fabric of the chip. This section will now go into more details about the MicroBlaze microprocessor and Xilinx Embedded Development Kit (EDK) software tools.

The MicroBlaze is a virtual microprocessor that is built by combining blocks of code called cores inside a Xilinx Field Programmable Gate Array (FPGA). The beauty to this approach is that you only end up with as much microprocessor as you need. You can also tailor the project to your specific needs (i.e.: Flash, UART, General Purpose Input/Output pheriphals and etc.).

The MicroBlaze processor is a 32-bit Harvard Reduced Instruction Set Computer (RISC) architecture optimized for implementation in Xilinx FPGAs with separate 32-bit instruction and data buses running at full speed to execute programs and access data from both on-chip and external memory at the same time. The backbone of the architecture is a single-issue, 3-stage pipeline with 32 general-purpose registers (does not have any address registers like the Motorola 68000 Processor), an Arithmetic Logic Unit (ALU), a shift unit, and two levels of interrupt. This basic design can then be configured with more advanced features to tailor to the exact needs of the target embedded application such as: barrel shifter, divider, multiplier, single precision floating-point unit (FPU), instruction and data caches, exception handling, debug logic, Fast Simplex Link (FSL) interfaces and others. This flexibility allows the user to balance the required performance of the target application against the logic area cost of the soft processor. Figure 1 shows a view of a MicroBlaze system. The items in white are the backbone of the MicroBlaze architecture while the items shaded gray are optional features available depending on the exact needs of the target embedded application. Because MicroBlaze is a soft-core microprocessor, any optional features not used will not be implemented and will not take up any of the FPGAs resources.

The MicroBlaze pipeline is a parallel pipeline, divided into three stages: Fetch, Decode, and Execute. In general, each stage takes one clock cycle to complete. Consequently, it takes three clock cycles (ignoring delays or stalls) for the instruction to complete. Each stage is active on each clock cycle so three instructions can be executed simultaneously, one at each of the three pipeline stages. MicroBlaze implements an Instruction Prefetch Buffer that reduces the impact of multi-cycle instruction memory latency. While the pipeline is stalled by a multi-cycle instructions. Once the pipeline resumes execution the fetch stage can load new instructions directly from the Instruction Prefetch Buffer rather than having to wait for the instruction memory access to complete. The Instruction Prefetch Buffer is part of the backbone of the MicroBlaze architecture and is not the same thing as the optional instruction cache.







Figure 1-1. A view of a MicroBlaze system

The MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data accesses and instruction accesses. MicroBlaze does not separate between data accesses to I/O and memory (i.e. it uses memory mapped I/O). The processor has up to three interfaces for memory accesses: Local Memory Bus (LMB), IBM's On-chip Peripheral Bus (OPB), and Xilinx CacheLink (XCL). The LMB provides single-cycle access to on-chip dual-port block RAM (BRAM). The OPB interface provides a connection to both on-chip and off-chip peripherals and memory. The CacheLink interface is intended for use with specialized external memory controllers. MicroBlaze also supports up to 8 Fast Simplex Link (FSL) ports, each with one master and one slave FSL interface. The FSL is a simple, yet powerful, point-to-point interface that connects user-developed custom hardware accelerators (co-processors) to the MicroBlaze processor pipeline to accelerate time-critical algorithms.

All MicroBlaze instructions are 32 bits wide and are defined as either Type A or Type B. Type A instructions have up to two source register operands and one destination register operand. Type B instructions have one source register and a 16-bit immediate operand. Type B instructions have a single destination register operand. Instructions are provided in the following functional categories: arithmetic, logical, branch, load/store, and special. MicroBlaze is a load/store type of processor meaning that it can only load/store data from/to memory. It cannot do any operations on data in memory directly; instead the data in memory must be brought inside the MicroBlaze





processor and placed into the general-purpose registers to do any operations. Both instruction and data interfaces of MicroBlaze are 32 bit wide and uses Big-Endian, bit-reversed format to represent data (Order of Bits: Bit 0 Bit 1 Bit 30 Bit 31 with Bit 0 the MSB and Bit 31 the LSB). MicroBlaze supports word (32 bits), half-word (16 bits), and byte accesses to data memory. Data accesses must be aligned (i.e. word accesses must be on word boundaries, half-word on half-word boundaries), unless the processor is configured to support unaligned exceptions. All instruction accesses must be word aligned.

The stack convention used in MicroBlaze starts from a higher memory location and grows downward to lower memory locations when items are pushed onto a stack with a function call. Items are popped off the stack the reverse order they were put on; item at the lowest memory location of the stack goes first and etc.

The MicroBlaze processor also has special purpose registers such as: Program Counter (PC) can read it but cannot write to it, Machine Status Register (MSR) to indicate the status of the processor such as indicating arithmetic carry, divide by zero error, a Fast Simplex Link (FSL) error and enabling/disabling interrupts to name a few. An Exception Address Register (EAR) that stores the full load/store address that caused the exception. An Exception Status register (ESR) that indicates what kind of exception occurred. A Floating Point Status Register (FSR) to indicate things such as invalid operation, divide by zero error, overflow, underflow and denormalized operand error of a floating point operation.

MicroBlaze also supports reset, interrupt, user exception, break and hardware exceptions. For interrupts, MicroBlaze supports only one external interrupt source (connecting to the Interrupt input port). If multiple interrupts are needed, an interrupt controller must be used to handle multiple interrupt requests to MicroBlaze. An interrupt controller is available for use with the Xilinx Embedded Development Kit (EDK) software tools. The processor will only react to interrupts if the Interrupt Enable (IE) bit in the Machine Status Register (MSR) is set to 1. On an interrupt the instruction in the execution stage will complete, while the instruction in the decode stage is replaced by a branch to the interrupt vector (address 0x10). The interrupt return address (the PC associated with the instruction in the decode stage at the time of the interrupt) is automatically loaded into general-purpose register R14. In addition, the processor also disables future interrupts by clearing the IE bit in the MSR. The IE bit is automatically set again when executing the RTID instruction.

Writing software to control the MicroBlaze processor must be done in C/C++ language. Using C/C++ is the preferred method by most people and is the format that the Xilinx Embedded Development Kit (EDK) software tools expect. The EDK tools have built in C/C++ compilers to generate the necessary machine code for the MicroBlaze processor.

The MicroBlaze processor is useless by itself without some type of peripheral devices to connect to and EDK comes with a large number of commonly used peripherals. Many different kinds of systems can be created with these peripherals, but it is likely that you may have to create your own custom peripheral to implement functionality not available in the EDK peripheral libraries and use it in your processor system.

To maximize the automation that EDK tools provide with you, when creating your own custom peripheral you must take into account the following considerations:





The processor system by EDK is connected by On-chip Peripheral Bus (OPB) and/or Processor Local Bus (PLB), so your custom peripheral must be OPB or PLB compliant (see *note*). Meaning the top-level module of your custom peripheral must contain a set of bus ports that is compliant to OPB or PLB protocol, so that it can be attached to the system OPB or PLB bus. See figure 1-2.



Figure 1-2. OPB bus protocol example used in a MicroBlaze system

Note: You may also create peripherals attached other bus interfaces that Xilinx supports as well, such as FSL bus interface. They are not covered in this guide.

EDK uses Intellectual-Property Interface (IPIF) library to implement common functionality among various processor peripherals. It is verified, optimized and highly parameterizeable. It also gives you a set of simplified bus protocol called IP Interconnect (IPIC), which is much easier to use rather than operate on OPB or PLB bus protocol directly. Using the IPIF module with parameterization that suits your needs will greatly reduce your design and test effort because you don't have to re-invent the wheel. See figure 1-3. This is done in EDK with a wizard that walks you through the entire process.



Figure 1-3. Using IPIF module in your peripheral





Considering all the above, you should use the following design flow when creating custom peripherals in EDK:

- **Determine Interface:** Identify the bus interface (OPB or PLB) your custom peripheral should implement, so that it can be attached to that bus in your processor system.
- **Implement and Verify Functionality:** Implement your custom functionality, reuse the common functionality already available from EDK peripheral libraries as much as possible, and verify your peripheral as a stand-alone core.
- **Import to EDK:** Copy your peripheral to an EDK recognizable directory structure and create the PSF interface files (.mpd/.pao) so that other EDK tools can access your peripheral.
- Add to System: Add your peripheral to the processor system in EDK.

This background section gave only a very small introduction about some things to know about MicroBlaze and EDK for this tutorial. For even more information about MicroBlaze or EDK, please refer to the MicroBlaze Reference Guide at <u>http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf</u> and the EDK Reference Documents at <u>http://www.xilinx.com/ise/embedded/edk_docs.htm</u>.



CREATING THE PROJECT IN XPS

The first step in this tutorial is using the Xilinx Platform Studio (XPS) of the EDK software tools to create a project file. XPS allows you to control the hardware and software development of the MicroBlaze system, and includes the following:

- An editor and a project management interface for creating and editing source code
- Software tool flow configuration options

You can use XPS to create the following files:

- Project Navigator project file that allows you to control the hardware implementation flow
- Microprocessor Hardware Specification (MHS) file
 - The MHS file is a textual schematic of the embedded system used by the tools
- Microprocessor Software Specification (MSS) file
 - The MSS file describes all the drivers (software) for all components in the system
 - XPS supports the software tool flows associated with these software specifications. Additionally, you can use XPS to customize software libraries, drivers, and interrupt handlers, and to compile your programs.

Note: For more information on the MHS file, refer to the "Microprocessor Hardware Specification (MHS)" chapter in the Embedded System Tools Guide and for more information on the MSS file, refer to the "Microprocessor Software Specification (MSS)" chapter in the Embedded System Tools Guide.

STARTING XPS:

- To open XPS, select the following: Start → Programs → Xilinx Platform Studio 7.1i → Xilinx Platform Studio
- 2. If Figure 2-1 appears, select Base System Builder Wizard (BSB) and Click *Ok* to open the Create New Project Using BSB Wizard dialog box shown in Figure 2-3.

Xilinx Pla	atform Studio 🔹 💽 🔀		
	 new XPS project using Base System Builder Wizard C Blank XPS Project C Blank XPS Project 		
	Projects Dpen A Recent Project Browse for more Projects		
Browse EDK Examples (Installed Projects) Do not show this dialog again OK Carcel			

Figure 2-1. Xilinx Platform Studio Dialog





If Figure 2-1 does not appear, then from the XPS main menu, Click **File** \rightarrow **New Project** \rightarrow **Base System Builder** ... which is shown in Figure 2-2 to open the Create New Project Using BSB Wizard dialog box shown in Figure 2-3.

📕 Xilinx Platform Studio				
File Edit View I	Help			
New Project		Base System Builder		
Open Project		Platform Studio	坺	
Recent Projects	•	尙 : 二 《 > = 네	INI	
Exit		<u> </u>	h	

Figure 2-2. New Base System Builder Based Project Creation using XPS main menu

This will open the Create New Project Using Base System Builder Wizard dialog.

Create New Project Using Base System B	uilder Wizard 🛛 ? 🔀			
New Project The project file will be created in the current direct Project File C:\ece597\test\system.xmp	tory if a path is not specified.			
Peripheral Repository Directory (Advanced Option) User Peripheral Repository search path for IP, driver and library files. Can be a semicolon separated list of directories.				
	Browse			
	OK Cancel			

Figure 2-3. Create New Project Using Base System Builder Wizard Dialog

Use the Project File **Browse** button to browse to the folder you want as your project directory and Click **Open** when done. Keep the **Peripheral Repository Directory** check box **unchecked** and Click **Ok** to create the system.xmp file. It may take a while (up to 1-2 minutes sometimes) for Base System Builder wizard to load and get started.

Note: XPS does not support directory or project names that include spaces.

3. In the **Base System Builder - Welcome** screen, select **I would like to create a new design** and click **Next** to get the **Base System Builder - Select Board** dialog box.

This tutorial uses the Digilent Spartan-3 board, which is supported in Base System Builder.

Verify that **I would like to create a system for the following development board** option is selected and select the following options:

Select Board Vender: Xilinx





Select **Board Name: Spartan-3 Starter Board** Select **Board Revision: E** Click **Next** and the **Select Processor** dialog will be displayed (Figure 2-5)

Base System Builde	r - Select Board	×
Select a target devek	apment board: aate a system for the following development board	
Board Vendor	Xins 💌	
Board Name	Spartan-3 Starter Board	
Board Revision	E	
Vendor's Websk Download Third C I would like to ore Board Description – Spartan 3 Starter K The board includes push buttons, four o Push buttons 1 is us data but.	e Contact Info Party Board Definition Files aate a system for a custom board It Board utileses Xilnx Sportan-3 XC3S20D-4FT256 device. 1 RS232 serial port, 2 256kx16 fast SPAM,8 DIP switches, 4 tigtal 7 segment LEDs. 8 LEDs. 1 VGA port, 1 PS/2 port, ed as system reset, 2 SRAMs are combined to form a 32 bit	
<u>M</u> are Infa	< <u>B</u> ack <u>N</u> ext> <u>C</u> ance	1

Figure 2-4. Base System Builder - Select Board Dialog





 In the Base System Builder – Select Processor panel (as shown in figure 2-5): MicroBlaze is the only processor option available for use on the Spartan3 Started Board, so click Next.

The board you telects Architecture: spartan3	ad has the following FPGA device: Device Size: Package: Speedgrade: Y NC35200 Y FT256 Y -4 Y	
Select the processory	ou would like to use in this design:	
C <u>MicroBlazs</u> C <u>BowerPD</u> Not supported by this device.	Constant Serial Port	
Processor Descri The MicroBlaze(32 register by 32 instructions for de BlackRAM and/o the FPBA fabric a	ption (M) 32-bit soft processor is a RISC-based engine with a bit LUT RAM-based Register File, with separate its and memory access. It supports both on-oftip in external memory. All peripherals are implemented on and operate off the on-oftip peripheral bus (OPB).	

Figure 2-5. Base System Builder – Select Processor Dialog





- 5. The **Base System Builder Configure Processor** dialog will be displayed. Select settings to match the following:
 - Processor Clock Frequency: 50 MHz
 - Processor Bus Clock Frequency: 50 MHz
 - Debug Interface: On-chip H/W debug module
 - Local Data and Instruction Memory: 16 KB
 - Cache Enabled: unchecked

Micro	der - Configure Processor 🛛 🕑
⊏ Sustem Wide S	lings
Reference Cl Frequency:	ick Processor-Bus Dock Frequency.
E crue lbetur	u band is configured for the specified here even
Reset Polarity	Activo High 🚽
C XMD	ja Havon ja HAV debug modula with SAV debug itub ibug
Cache	MicroElaze
C No Ca C Enstit	he Cathelink
·····	The Real Hards Council

Figure 2-6. Base System Builder – Configure Processor Dialog

The following is an explanation of the settings specified in Figure 2-6:

- System Wide Setting:
 - Processor Clock Frequency: This is the frequency of the clock driving the processor system.
- Processor Configuration:
 - Debug I/F:
 - XMD with S/W Debug stub: Selecting this mode of debugging interface introduces a software intrusive debugging. There is a 1200-byte stub that is located at 0x00000000. This stub communicates with the debugger on the host through the JTAG interface of the OPB MDM module.
 - On-Chip H/W Debug module: When the H/W debug module is selected; an OPB MDM module is included in the hardware system. This introduces hardware intrusive debugging with no software stub required. This is the recommended way of debugging for MicroBlaze system.
 - No Debug: No debug is turned on.
- Users can also specify the size of the local instruction and data memory (BRAM).
- You can also specify the use of a cache.





6. Click **Next** and the **Base System Builder – Configure IO Interfaces** dialog will be displayed. Uncheck the **LED_Bit** and **LED_7SEGMENT**boxes, leaving the remaining devices with the default settings

Note: Depending on your screen resolution settings, the **Base System Builder – Configure IO Interfaces** dialog may show more or less devices in the initial screen than the one shown below.

Xilinx Spartan-3 Starter Board Revision E	
lease select the IO interfaces or ports which you would like to us	e:
IO Devices	
RS232	Data Sheet
Peripheral: OPB UARTLITE	
Baudrate (Bits per 9600	
Data Bits: 8	
Parity: NONE	
Use Interrupt	
	Data Sheet
Peripheral:	
	Data Sheet
- enpireitai,	
Push_Buttons_3Bit	
Peripheral: OPB GPI0	Data Sheet
✓ DIP_Switches_8Bit	Data Sheet
Peripheral: OPB GPI0	
Use Interrupt	

Figure 2-7. Base System Builder – Configure IO Interfaces Dialog





 Uncheck the SRAM_256Kx32 box in the Base System Builder – Configure Additional IO Interfaces dialog screen (Figure 2-8) and click the Next button and the Base System Builder – Add Internal Peripherals dialog will appear (Figure 2-9).

ase System Builder - Configure Additional IO Interfaces		
The following IO interfaces were found on your target board: Xilinx Spartan-3 Starter Board Revision E Please select the IO interfaces or ports which you would like to use:		
IO Devices SRAM_256Kx32 Peripheral:	Data Sheet	

Figure 2-8. Base System Builder – Configure Additional IO Interfaces Dialog

8. Click the **Next** button in the **Base System Builder – Add Internal Peripherals** dialog screen (Figure 2-9) and the **Base System Builder – Software Setup** dialog box (Figure 2-10) will appear

Base System Builder - Add Internal Peripherals		×
Add other peripherals that do not interact with off-chip compo Peripheral" button to select from the list of available peripheral If you do not wish to add any non-IO peripherals, click the "N	onents. Use the "Add als. Next" button.	
Peripherals	Add Peripheral	

Figure 2-9. Base System Builder – Add Internal Peripherals Dialog





9. Click the **Next** button in the **Base System Builder – Software Setup** dialog screen (Figure 2-10) and the **Base System Builder – Configure Memory Test Application** (Figure 2-10) dialog box will appear

Base System Builder - Software Setup	Base System Builder - Configure Memory Test Application
Devices to use as Standard Input and Standard Output STDIN RS232 STDOUT RS232	The simple Memory Test application will illustrate system alveness and perform a basic read/write test to your memory devices. MemoryTest Select the memory devices which will be used to hold the following program sections:
Sample Application Selection Select the sample C application that you would like to have generated.	Instructions: Inductivit
Remay Test	Stack/Heap. dmb_ontr
Hushete system aliveness and perform a basic read/write test to each memory in your system	WAFNING
 Perspectision con There is no selflest code available for the peripherals in your system. 	If you have placed the instruction or Dela section of this program in an entering memory, you mark use a debugget, bookbadet, or ADE He to initialize memory before you can run this program!
More Info	More Info

Figure 2-10. Base System Builder – Software Setup and Configure Memory Test Application Dialog Screens

By accepting all the defaults in the software setup and configuration panels, we will let BSB to generate a memory test application main program (including link script) for you. Later, you will edit this main program to operate on your custom peripheral via a software driver.





10. Click the Next button and the Base System Builder – System Created dialog (figure 2-11) will appear showing a summary of the system being created. Click the Generate button and the A Base System Builder – Finished screen will appear congratulating you on that Base System Builder successful generated your embedded system, which indicates the files the BSB has created. Click the Finish button to finish generating the project.

s. Otherwise return to	the previous page to m	nake corrections.	
Processor: Microbi System clock frequ Debug interface: O On Chip Memory :	laze iency: 50.000000 MHz n-Chip HW Debug Mod 16 KB	ule	
The address maps	below have been autor	matically assigned.	You can modify them using
the editing features	of XPS.		<u></u>
Coro Nomo	vzu inst name. mb	_opb Attached C	Jomponents.
oph mdm	debug, modulo	0x41400000	0x4140EEEE
opb_mam	DS332	0x41400000	0x4140FFFF
opb_uarine	Rush Buttone 2Pit	0x4000000	0x4000FFFF
opb_gpio	DIR Switchos SBit	0x40000000	0x4003EEEE
Jobp_Gbio	DIP_Switches_ODit	0,40020000	0,40021111
LMB Bus : LMB_	V10 Inst. name: ilmb	Attached Comp	oonents:
Core Name	Instance Name	Base Addr	High Addr
Imb_bram_if_cntlr	ilmb_cntlr	0x00000000	0x00003FFF
LMB Bus : LMB_	V10 Inst. name: dlm	b Attached Com	ponents:
Core Name	Instance Name	Base Addr	High Addr
Imb_bram_if_cntlr	dlmb_cntlr	0x0000000	0x00003FFF

Figure 2-11. Base System Builder – System Created Dialog





11. Once the Base System Builder wizard is closed, you will go back to the Xilinx Platform Studio IDE with the newly created Test project opening up for you. Depending on your setup, you may encounter the following popup (figure 2-12) to inform Platform Studio what you want to do next. Click OK to start using Platform Studio as the default (you may want to check off the checkbox to stop this popup next time).



Figure 2-12. The Next Step Screen

12. The Base System Builder Wizard has created the hardware and software specification files that define the processor system. In the XPS System tab, look at the hardware processor system (defined in the system.mhs file and system.pbd file), that BSB created for you, as well as the UCF constraints (data/system.ucf file) by double-clicking any of those items under Project Files to open it. The system.mhs file is a text file describing the embedded system whereas the system.pbd file shows a schematic view of it. Finally the data/system.ucf file shows the FPGA pin assignments for the devices used in the system. Also when you look at the project directory, shown in Figure 2-13, you also see the system.mhs and system.ms files.

There are also some directories created.

- data contains the UCF (user constraints file) for the target board.
- etc contains system settings for JTAG configuration on the board that is used when downloading the bit file and the default parameters that are passed to the ISE tools.
- pcores is empty right now, but is utilized for custom peripherals.
- TestApp_Memory contains a user application in C code source, for testing the memory in the system.

Name	Size	Туре	
Today			
🛅 data		File Folder	
etc		File Folder	
Decores		File Folder	
C TestApp_Memory		File Folder	
🚾 system.bsb	3 KB	BSB File	
🗐 system.log	1 KB	Text Document	
👼 system.mhs	7 KB	MHS File	
🔤 system.mss	2 KB	MSS File	
🗾 system.xmp	1 KB	XMP File	
<			>

Figure 2-13. Project Directory after Base System Builder completes

17





CREATING THE CUSTOM OPB PERIPHERAL USING WIZARD

1. In XPS menu, select **Tools** → **Create/Import Peripheral...** to start the wizard as shown in figure 3-1.



Figure 3-1: Create and Import Peripheral Wizard

This wizard is able to create 4 types of CoreConnect compliant peripherals using the predefined IPIF libraries to reduce development effort and time to market, it may also create FSL peripherals, which is not covered in this guide. The types of custom peripherals are:

- OPB slave-only peripheral
- OPB master-slave combo peripheral
- PLB slave-only peripheral
- PLB master-slave combo peripheral
- FSL master/slave peripheral

Click on the hyperlinks to open up corresponding data sheets for detail information on what features are supported, or the **More Info** button for quick overview.





2. Click **Next** to continue and the **Create and Import Peripheral Wizard's** flow selection will appear (Figure 3-2). This wizard will help you create templates for a new EDK compliant peripheral or help you import an existing peripheral into an XPS project or EDK repository. For this project we will create an EDK-compliant peripheral.

Create and Import Periphera	l Wizard	X
Create/Import User Peripher Indicate if you want to create a	r al a new peripheral or import an existing peripheral.	a star
This tool will help you create temp existing peripheral into an XPS pro- required by EDK will be generated Create Templates	Iates for a new EDK compliant peripheral, or help you import an oject or EDK repository. The interface files and directory structures d. Select Flow Create templates for a new peripheral Import existing peripheral Flow Description This tool will create HDL templates that have the EDK compliant port/parameter interface. You will need to implement the body of the peripheral.	
More Info	<back next=""> Finish Can</back>	cel

Figure 3-2. Create/Import User Peripheral Screen

3. The default selection is **Create template for a new peripheral**. Ensure the radio button is on for this selection, click **Next** and the **Create and Import Peripheral Wizard's** target selection will appear (Figure 3-3).

Create and Impo	ort Peripheral Wizard	X
Repository or Indicate where	Project re you want to store the new peripheral.	No. of
A new peripheral c repository the perip	can be stored in an EDK repository, or in an XPS project. When stored in an EDK ipheral can be accessed by multiple XPS projects.	
O To an EDK us	ser repository (Any directory outside of your EDK installation path)	
Repository	Browse	
To an XPS pro	oject	
Project	C:\ece597\test\ Browse	
Peripheral will be C:\ece597\tes	e placed under: st\pcores	
More Info	< <u>B</u> ack <u>N</u> ext> Finish Cancel	

Figure 3-3. Repository or Project Screen





4. Make sure the radio button for To an XPS project is selected, and navigate to c:\ece597\Test\system.xmp. Click Next and the Create Peripheral – Step 1 dialogue will appear for you to indicate the name of the peripheral (Figure 3-4).

Enter custom_logic_adder in the name field, as shown in Figure 3-4 and click Next.

Create Peripheral - Step 1	X
Name and Version Indicate the name and version of your peripheral.	a and a second
Enter the name of your peripheral. This name will be used as the top HDL design entity. Name: custom_logic_adder Version: 1.00.a Major Revision Minor Revision 1 0 • a • Logical library name: custom_logic_adder_v1_00_a All HDL files (either created by you or generated by this tool) used to implement this peripheral must be compiled into the logical library named above. Any other logical libraries referred to in your HDL are assumed to be available in the XPS project where this peripheral is used, or in EDK repositories indicated in the XPS project settings.	
More Info	cel

Figure 3-4. Provide Core Name and Version Number

5. The Create Peripheral – Step 2 dialogue will appear for you to indicate the type of bus interface to attach to the peripheral (Figure 3-5). This is an OPB peripheral so leave the default settings and click Next.

Create Peripheral - Step 2	
Bus Interfaces	Sec. 1
To which bus will this peripheral be attached?	
On-chip Peripheral Bus (OPB)	
O Processor Local Bus (PLB)	
NOTE: Other bus interfaces are not supported in this EDK release.	
More Info Can	cel

Figure 3-5. Select the Bus Interface

20





6. In the **IPIF Services** window (Figure 3-6), you select the IPIF features you want to support in your peripheral. Table 1 gives descriptions of all the IPIF Service features available. For the custom_logic_adder, we only need registers for the digit values. Select **User Logic S/W Register Support** as shown below and click **Next**



Figure 3-6. Select IPIF Services

IPIF Feature	Description
Include Software Reset & Module Information registers	The peripheral has a special write only address. When a specific word is written to this address, the IPIF generates a reset signal for the peripheral. The peripheral should reset itself using this signal. This allows individual peripherals to be reset from the software application.
	The peripheral has a read-only register that identifies the revision level of the peripheral.
Include Burst Cacheline Transaction Support	Burst and cacheline transactions allow the bus master to issue a single request that results in multiple data values being transferred. Support of these transactions requires significant hardware resources. Presently, the 'fast' burst mode is used. Cacheline is available for the PLB peripherals only.
Include DMA	The IPIF part of the peripheral has a built in DMA service.
Include FIFO	The IPIF part of the peripheral has a built in FIFO service.
User-logic interrupt support	The peripheral has an interrupt collection mechanism that manages the interrupts generated by the user-logic and the IPIF services and generate a single interrupt output line out of the peripheral.
Include Software Addressable Registers in user-logic	The user-logic part of the peripheral has registers addressable through software.
Include master support in user-logic	This includes the IPIC master interface signals for user logic master operations. It also includes example HDL for a simple master operation model. This HDL indicates how the user logic master model operates.
Include Address Range Support in user-logic	This generates enable signals for each address range. This feature is useful for peripherals that need to support multiple address ranges, <i>e.g.</i> multiple memory banks. The distinction between this and other cases is that the enable signals are generated for each address range of the address space supported by the peripheral, rather than for each addressable register in the user-logic module.

Table 1. IPIF Service Features Descriptions





7. We need three registers (one for the Augend, one for the Addend and one for the Sum) and each register will be 32-bits wide. Select the values as shown in Figure 3-7 and click **Next.**

Create Peripheral - Step 4	
User S/W Register Software accessible registers in your peripheral.	and a second
The software accessible registers will be implemented in the user-logic module of your peripheral. These registers are addressable on the byte, half-word or word boundaries. The following fields determine the characteristics of the registers.	
Number of software accessible registers: 3	
Data width of each register: 32	
More Info Sack Next > Finish Cance	!

Figure 3-7. Configure Registers

8. The IP Interconnect (IPIC) lets you customize the signals in the interface between the custom logic and the IPIF (Figure 3-8). The wizard already selected all the IPIC ports that are necessary to complete the services/supports you choose in previous steps, and typically you don't need to change anything here. You're free to add any extra IPIC ports that you want to use but for this peripheral the default connections are all that are needed, therefore click **Next**.

Create Peripheral - St	ер б			×
IP Interconnect () Select the interface the IPIF.	IPIC) between the logic to be	implemented in	your peripheral and	a con
Your peripheral is connect interfaces to the IPIF throu of the ports are always pra- functionality required by po	ed to the bus through a : ugh a set of signals called sent. You can choose to our peripheral.	suitable IPIFm d the IP interco a include the of	cdule. Your periphen nnect (IPIC) interfac hers based on the	al e. Some
	Note: all IPIC ports are	active high	art Discovishion	
OPB or PLB bus	P2Bus_Ck ♥ Bus2IP_Ck ♥ Bus2IP_Reset ■ us2IP_Reset ■ Us2IP_Reset ■ us2IP_Addr ♥ Bus2IP_Addr ♥ Bus2IP_Bt ■ us2IP_Bt		un descripcion	X
More Inio	< <u>B</u> ack	لخ <u>N</u> ext>	Finish	Cancel

Figure 3-8. IP Interconnect (IPIC)

• IPIC stands for IP Interconnect, it's a simplified interface (protocol) that allow you to hook up your custom function (user logic) to the corresponding IPIF module and let IPIF worry about the master/slave attachment and other common functionality (FIFO, DMA). Using IPIC, it's possible that your custom function (user logic) can be easily attached to either OPB or PLB bus, and you only need to take care of a small set of ports, which is easy to understand and manage.





9. EDK gives you the option to generate **Bus Functional Models (BFM)** to help you simulate your peripherals (Figure 3-9). Click **Next** to skip the Create Peripheral – Step 6 dialogue since we will not perform a simulation on the peripheral.

Create Peripheral - Step 6				
(OPTIONAL) Peripheral Simulation Support Generate optional files for simulation using Bus Functional Models (BFM).				
The EDK provides a BFM simulation platform to help you simulate your peripheral. Indicate if you want this tool to generate the appropriate HDL and Bus Functional Language (BFL) stimulus file for the target bus.				
More Info				

Figure 3-9. Generate Bus Functional Models (BFM)

10. For Create Peripheral – Step 7, Check Generate ISE and XST project files to help you implement the peripheral using XST flow. Check Generate template driver files to help you implement software interface and click Next (Figure 3-10).

Create Peripheral - Step	7	X			
(OPTIONAL) Peripheral Generate optional files f	(OPTIONAL) Peripheral Implementation Support Generate optional files for hardware/software implementation				
Upon completion, this tool will requested. A stub 'user_logic this module using standard H (mpd/pao) for the synthesizal processor system. Peripheral (VHDL) IPIF (VHDL) User Logic (VHDL)	I create synthesizable HDL files that implement the IPIF services you 'module will be created. You will need to complete the implementation of IDL design flows. The tool will also generate EDK interface files ble templates, so that you can hook up the generated peripheral to a NOTE: Should the peripheral interface (ports/parameters) or file list change, you will need to regenerate the EDK interface files using the import functionality of this tool. Generate stub 'user_logic' template in Verilog instead of VHDL. Generate ISE and XST project files to help you implement the peripheral using XST flow. Generate template driver files to help you implement software interface.				
More Info	< <u>B</u> ack <u>N</u> ext > Finish Canc	el			

Figure 3-10. Select Design Flow





- It is recommended that you always generate these ISE/XST project files, as they will greatly reduce your effort when you use Xilinx tools to implement your peripheral. This will be demonstrated in a later step.
- This wizard will create a driver template (including the directory structure and interface files, as well as a self test function) under your project if this feature is selected as indicated above, it will help you to get started on your software interface implementation of your custom peripheral hardware.
- It is possible to generate the user-logic template in Verilog instead of VHDL, which makes your peripheral a mixed language design since the peripheral top template is always in VHDL. EDK tools support mixed language designs by using the default-binding rule.
- 11. This completes the Peripheral template generation (Figure 3-11) and click Finish.

Create and Import Periph	eral Wizard	X
	Congratulations! When you click Finish, HDL files representing your peripheral will be generated. You will have to implement the functionality of your peripheral in the stub 'user_logic' template file. IMPORTANT: If you make any changes to the generated port and parameter interfaces, or add new files you will need to regenerate the EDK interface files by using this tool in the Import mode. Thank you for using Create and Import Peripheral Wizard! Peripheral summary top name : custom_logic_adder version : 1.00.a	
More Info	< <u>Back</u> Next> Finish Cano	cel

Figure 3-11. Template Complete





12. Use Windows Explorer to browse to your project directory and ensure that the following structure has been created by the Importing Peripheral Wizard (Figure 3-12).



Figure 3-12. Structure Created by the Importing Peripheral Wizard

The following is a description of the files located in each directory:

• HDL source file(s)

c:\ece597\Test\pcores\custom_logic_adder _v1_00_a\hdl

• vhdl/custom_logic_adder.vhd

This is the template file for your peripheral's top design entity. It configures and instantiates the corresponding IPIF unit in the way you indicated in the wizard GUI and hooks it up to the stub user logic where the actual functionalities should get implemented. You are not expected to modify this template file except certain marked places for adding user specific generics and ports.

• vhdl/user_logic.vhd

This is the template file for the stub user logic design entity, either in VHDL or Verilog, where the actual functionalities should get implemented. Some sample code snippet may be provided for demonstration purpose.





• <u>XPS interface file(s)</u>

 $c:\ece597\Test\pcores\custom_logic_adder_v1_00_a\data$

• custom_ip_v2_1_0.mpd

This Microprocessor Peripheral Description file contains information of the interface of your peripheral, so that other EDK tools can recognize your peripheral.

• custom_ip_v2_1_0.pao

This Peripheral Analysis Order file defines the analysis order of all the HDL source files that are used to compile your peripheral.

Note: The ipwiz.log is the importing peripheral wizard log file

• Driver source file(s)

c:\ece597\Test\drivers\custom_logic_adder_v1_00_a\src:

• custom_logic_adder.h

This is the software driver header template file, which contains address offset of software addressable registers in your peripheral, as well as some common masks and simple register access macros or function declaration.

• custom_logic_adder.c

This is the software driver source template file, to define all applicable driver functions.

• custom_logic_adder_selftest.c

This is the software driver self test example file, which contain self-test example code to test various hardware features of your peripheral.

• Makefile

This is the software driver makefile to compile drivers.

13. Next the new custom peripheral must be added to your embedded system, which will be discussed in the next section.





Adding and Customizing the Peripheral to the System

1. In XPS, click **Project** → **Add/Edit Cores.** (*dialog*). This will open the **Add/Edit Hardware Platform Specifications** dialog box, as shown in Figure 4-1, used to modify an existing design.

Add/Edit Hardware	Platform S	pecifications		
Peripherals Bus Con	nections Add	resses Ports Parameters		
Cells with w	hite backgroun	ds can be edited. To delete	Show All Component Filter	C Datur
Peripheral	HW Ver	Instance	C MicroBlaze Only C Analog	C Interrupt
microblaze	4.00.a 💌	microblaze_0	C Either Processor	C 10
opb_mdm	2.00.a 💌	debug_module	C Clocking	C Logic
Imb_bram_if_cntlr	1.00.b	dimb_cntir	C DCR C OPB	C Memory
lmb_bram_if_cntlr	1.00.b	imb_cntir	C FSL C PLB	C Serial
bram_block	1.00.a	lmb_bram	C LMB C Trans	C Timers
opb_uartite	1.00.b	RS232	C OCM	
opb_gpio	3.01.b 💌	Push_Buttons_3Bit	bram block	
opb_gpio	3.01.b 💌	DIP_Switches_8Bit	chipscope_icon	
custom_logic_a	1.00.a	custom_logic_adder_0	chipscope_ula chipscope_opb_iba	
			Construction of the second	2
			OK Cancel Apply	Help

Figure 4-1. Add/Edit Hardware Platform Specifications Dialog

2. In the **Peripherals** tab, select **custom_logic_adder** from the available IPs list, and click **ADD** to add the peripheral to the system. See Figure 4-1.

Click << *Add* to add the **custom_logic_adder** peripheral to the project. The instance name for **custom_logic_adder** peripheral can be changed by clicking in the Instance name field.

3. Select the **Bus Connections** tab., click in the box next to **custom_logic_adder**_sopb, as shown in Figure 4-2, to connect the **custom_logic_adder** peripheral to the OPB Bus. A connection is made once the "s" is displayed in the box. "s" indicates a slave device, "m" indicates a master device, "bm" indicates a bus monitor. ChipScope Pro cores are an example of a bus monitor.





Add/Edit Hardware Platform Specifications			
Peripherals Bus Connections Addresses Ports Parameters			
Click on squares to make master, slave or master-slave (M, S, MS) connections. Right click on any bus instance (column header) for a context menu.	Choose the BRAM port to Give a name to the conner	Choose one or more der, v29, v1, 00, a decem, v10, v1, 00, b decem, v10, v2, 00, a fsi v20, v1, 00, b fsi v20, v2, 00, a isocm, v10, v1, 00, b connect to the controller pot cion.	suses (use Shift or Ctrl). Click Add.
debug_module sfsI0	Cntlr Port	BRAM Port	Connector
debug_module mfsl0	dlmb_cntlr bram_port	Imb_bram PORTB	✓ dlmb_port
dimb_cntir simb 5	ilmb_cntlr bram_port	Imb_bram PORTA	imb_port
ilmb_cntir simb S			
RS232 sopb 5			
Push_Buttons_3Bit sopb 5			
DIP_Switches_8Bit sopb 5	<		>
custom_logic_adder_0 sopb 5			
	Other Transparent bus (p	ooint to point) connections	
	Source Destina	ation Connec	
		,	
	1		
		01/ 0	Analy Usia
		Cancel	

Figure 4-2. Assign the custom_logic_adder_0 Instance

4. Select the **Addresses** tab to define an address for the newly added **custom_logic_adder** peripheral. The address can be assigned by entering the Base Address or the tool can assign an address. For the purpose of this tutorial, the tool will be used to assign an address (Figure 4-3).

Click *Generate Addresses*. Click Yes in the Information dialog. Once it successfully generated the address map click Ok.

PRODUCTION PURCE CONDUCTS	ons Addre	sses Ports Paran	neters							
ssign Address ranges fo	ar all periphr	erals and bus arbiters	i interna l		Pres	as F1 for more information				
Instance	Prefix	Base Address	High Address	Size	Min Size	Info/Error	L	I C D	Cache	
mb_opb				UNSPECIFIED	• 0x200					
debug_module		0x41400000	0x4140ffff	64 KB	• 0x100					
limb_cntir		0x00000000	0x00003fff	16 KB	• 0x800					
nb_cntir		0x00000000	0x00003fff	16 KB	• 0x800					
5232		0x40600000	0x4060ffff	64 KB	• 0x100		-			
ush_Buttons_3Bit		0x40000000	0x4000ffff	64 KB	• 0x200		-			
IP_Switches_8Bit		0x40020000	0x4002ffff	64 KB	• 0x200		-			
ustom_logic_adder_0		0x73000000	0x7300ffff	64 KB	▼ 0x100					
1										
٩		Generate addres	ses for peripherals t	hat do not have lock c	heckbox checked	Generate Addresses				

Figure 4-3. Generate Addresses for the New custom_logic_adder Peripheral





dd/Edit Hardware Platform Specifications												
Peripherals Bus Connections Addresses Ports Parameters												
External Ports Co	onnections:	,										Show ports with default connections
Port Name		Net Name	Pol	Range	Class	Sen						custom 👻
fpga_0_RS232_RX_pin		fpga_0_RS23	IN				1					List of Ports Click Add to add ports
fpga_0_RS232_TX_pin		fpga_0_RS23	OUT									
fpga_0_Push_Buttons_3Bit_GPIO_in_pin fpga_0_Push		. IN [0:2]								microblaze_0		
fpga_0_DIP_Switches_	fpga_0_DIP_Switches_8Bit_GPIO_in_pin fpga_0_DIP		IN [0:7]								debug module	
sys_clk_pin		sys_clk_s	IN CLK								~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~	
sys_rst_pin		sys_rst_s	IN							Add Dart	dimb_cntir	
											Add Folt	ilmb_cntlr
												Imb_bram
												PS232
Internal Ports Connection:	5:											~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Instance	Port Name Pola Range Class						^		Push_Buttons_3Bit			
RS232	TX	TX fpga 0 RS232 TX V V							DIP_Switches_8Bit			
Push_Buttons_3Bit	OPB_Clk sys_clk_s I					Clk			custom logic adder 0			
Push_Buttons_3Bit	GPIO_in		fpga	_0_Push_Buttons_3Bit_	_GPIO_i	n .	· 1	[0:(C_GPI				OPB_CIk
DIP_Switches_8Bit	OPB_Clk		sys_	clk_s		-	·I		Clk			mb_opb
DIP_Switches_8Bit	GPIO_in		fpga_0_DIP_Switches_8Bit_GPIO_in			1	·I	[0:(C_GPI			Make	limb
mb_opb	SYS_Rst			sys_rst_s			· 1				External	
mb_opb	OPB_Ck			sys_clk_s			·I		CLK		<< Add	dimp
imb	SYS_Rst sys_rst_s			rst_s		-	·I					
imb	LMB_Clk sys_clk_s				· I		CLK		Delete			
dlmb	SYS_Rst	'S_Rst sys_rst_s I										
dimb	LMB_Clk	sys_clk_s		-	· I		CLK		Connect			
custom_logic_adder_0	OPB_Clk		sys_	clk_s			· I		Ck			
										014		
										UK	Car	.cei <u>A</u> ppiy Help

Figure 4-4. Declaring New External/Internal Ports in the System

5. In the **Ports** tab, type **custom_logic_adder** (or enough of its name to filter it out) in the Ports Filter located on the right side of the window. Select the following port **OPB_CLK** under **custom_logic_adder_0** and click the << Add button. See Figure 4-4.

The ports for the **custom_logic_adder_0** instance need to be adjusted by changing the net name connected to the **OPB_CLK** port to **sys_clk_s** and leave it as **internal**. The net name for the main clock is changed to **sys_clk_s** because this is the net name the XPS tool gave it.

Click the **OK** button to accept the settings and close the dialog window.

Note: No additional ports from the new custom peripheral will be going external of the FPGA, so nothing else will be made external. But notice the push buttons and dip switches were declared external from you made the system initial.





6. Now that the template has been created by XPS, the user_logic.vhd file must be modified to incorporate the custom IP functionality. Open user_logic.vhd by double clicking on it after expanding custom_logic_adder_0 fully (showing all levels) in the System screen. See Figure 4-5 (Notice that custom_logic_adder_0 is fully expanded which gives access to the vhdl code).



Figure 4-5. Expanding custom_logic_adder_0 to Open user_logic.vhd

Currently the code provides an example of reading and writing to three 32-bit registers. For the purpose of this tutorial, this code will be modified slightly to create the custom 32-bit adder peripheral.

Add the following vhdl code to **user_logic.vhd** file.

First, place the code in Table 2 after the last signal used for the "Signals for user logic slave model s/w accessible register example" which should be at approximately line 144. This is declaring signals to be used in the custom 32-bit adder circuit.

signal sum	: std_logic_vector(0 to C_DWIDTH-1);	
signal c	: std_logic_vector(0 to 32);	

Table 2. First vhdl Code to Add to user_logic.vhd file

Second, add the signal **sum** to the SLAVE_REG_READ_PROC : process list () and change the signal **slv_reg2** to **sum**, in which both are shown in Table 3 (Notice them blinking). This will the user to read the sum by reading the third register located at the base address of custom_logic_adder + $0x^2$ (hex value). This should start at approximately line 211.





Table 3. Second vhdl Code to Add to user_logic.vhd file

Finally, all the vhdl code in Table 4 is needed to create the custom adder peripheral, a 32-bit ripple carry adder circuit that needs to be placed after the end process **SLAVE_REG_READ_PROC**; code in Table 3.. This is implemented in a process that will execute every time that the value of slv_reg0 or slv_reg1 or if both of them change. The 32-bit ripple carry adder circuit is implemented using a for loop instead of a vhdl port map method because of the fact of not being able to get the vhdl port map method to work with EDK.

adder_PROC : process(slv_reg0, slv_reg1) is
begin
 c(32) <= '0'; -- Let c(32) = cin of the first LSB full adder
 -- Create 32 full adder using a for loop
 for i in 31 downto 0 loop
 sum(i) <= slv_reg0(i) xor slv_reg1(i) xor c(i+1);
 c(i) <= (slv_reg0(i) and slv_reg1(i)) or (slv_reg0(i) and c(i+1)) or (slv_reg1(i) and c(i+1));
 end loop;
end process adder_PROC;</pre>

Table 4. Last vhdl Code to Add to user_logic.vhd file

Save the changes and close **user_logic.vhd.** Nothing has to be done to **custom_logic_adder.vhd** because no signals are coming in internally from outside the FPGA and no internal signals are going external of the FPGA. If any of these conditions were true, it must be declared in **custom_logic_adder.vhd**.





MODIFY APPLICATION CODE, XFLOW AND PROGRAMING HARDWARE

 In XPS, click the Applications tab and expand Sources. This show a file called TestApp_Memory.c, which is the sample test application code written in C language by XPS. See figure 5.1.



Figure 5-1: Opening Sample Test Application Code File - TestApp_Memory.c

2. Double click on **TestApp_Memory.c** to open it. Modify the main program as shown in Table 5. Save the changes after done.



ł



```
int main (void) {
    Xuint32 *customLogicPtr;
    Xuint32 data,data1,data2,result;
    int a = 0xFFFFFFFF, b = 0x00000001, c = 0x12345678, i=0;
    print("-- Entering main() --\r\n\n");
    // do MicroBlaze addition and display on RS232 terminal
    print("Software\r\n");
    c=a+b;
    xil printf("%d + %d = %d \r\n", a,b,c);
    xil_printf("%x + %x = %x \r\n\n", a,b,c);
    customLogicPtr = (Xuint32 *)XPAR_CUSTOM_LOGIC_ADDER_0_BASEADDR;
    // write values to software addressable register 0 and register 1
    *(customLogicPtr) = a;
                                      // send 0xFFFFFFF to register 0 of custom logic
    *(customLogicPtr + 0x1) = b;
                                      // send 0x0000001 to register 1 of custom logic
    c = a; // assign new value to c (c = 0xFFFFFFF) to make sure custom logic produces a new result
    // read values from software addressable register 2 (the result of custom logic adder) and
    //display on RS232 terminal
    c = *(customLogicPtr + 0x2);
    print("Custom Logic\r\n");
    xil_printf("\%d + \%d = \%d \r\n", a,b,c);
    xil_printf("\%x + \%x = \%x \ (r\n, a, b, c);
    do{
        // read a new 8-bit values from dip switches and signal to accept the new number for addition with the 3 push buttons
        do{
            data = ReadFromGPInput(XPAR PUSH BUTTONS 3BIT BASEADDR);
             data1 = ReadFromGPInput(XPAR DIP SWITCHES 8BIT BASEADDR);
        while(data == 0);
        xil_printf("Data read from Push_Buttons_3Bit: 0x%x\r\n", data);
        xil_printf("Data read from DIP_Switches_8Bit: 0x%x\r\n", data1);
        // loop for delay for time to release switch and debounce
        for(i=0; i <= 0x1FFFFFF; i++) { ;
                                              }
        // read another new 8-bit values from dip switches and signal to accept the new number for addition with the 3 push buttons
        do{
            data = ReadFromGPInput(XPAR_PUSH_BUTTONS_3BIT_BASEADDR);
            data2 = ReadFromGPInput(XPAR_DIP_SWITCHES_8BIT_BASEADDR);
        while(data == 0);
        xil_printf("Data read from Push_Buttons_3Bit: 0x%x\r\n", data);
        xil printf("Data read from DIP Switches 8Bit: 0x%x\r\n", data2);
        // use custom logic to add the two numbers and display on RS232 terminal
        result = data1 + data2;
        print("Custom Logic with User Input\r\n");
        xil_printf("%d + %d = %d \r\n", data1, data2, result);
        xil printf("\%x + \%x = \%x \r\n\n", data1, data2, result);
        // loop for delay for time to release switch and debounce
        for(i=0; i <= 0x1FFFFFF; i++) { ;</pre>
    }while(1); // continues for ever to get user input
    print("-- Exiting main() --\r\n\n\n");
    return 0;
```

Table 5. Main Program to Modify to Test Custom Logic

33





Program Notes:

The main program does an addition with MicroBlaze's adder circuit first and displays the result (in decimal and hex) to the RS232 terminal via the UART placed in the embedded system. Next, the custom logic (32-bit adder) is tested by using the declared pointer, *customLogicPtr. The pointer, customLogicPtr gets the base address of the custom logic with this command:

customLogicPtr = (Xuint32 *)XPAR_CUSTOM_LOGIC_ADDER_0_BASEADDR;

Then both numbers to be added are sent over the OPB bus to register 0 and register 1 of the custom logic circuit by the following commands:

*(customLogicPtr) = a;	// send 0xFFFFFFFF	to register 0 of custom logic
*(customLogicPtr + $0x1$) = b;	// send 0x0000001	to register 1 of custom logic

The result is read from register 2 of the custom logic circuit by the command:

c = *(customLogicPtr + 0x2);

The result (in decimal and hex) obtained from the custom logic is then displayed on the RS232 terminal. The program then waits for a user to input two 8-bit numbers via the 8 dip switches (only one 8-bit number is entered at a time) and the user will signal via the three push-buttons (any value except zero) to get the new 8-bit number. When two numbers are inputted by the user, the custom logic adder adds the user inputted numbers and displaying the results. It repeats this process forever until the power is turned off or until the FPGA is forced into reset.

Additional Notes:

• The header file, "**xparameters.h**" holds important parameters for each device in your embedded system. These parameters are needed by your application code software, written in C/C++ language, to be able to communicate with the devices in your embedded system. For example, the following command used in the main program gets the parameter of the base address of custom logic from this header file.

customLogicPtr = (Xuint32 *)XPAR_CUSTOM_LOGIC_ADDER_0_BASEADDR;

- The header file, "**xgpio_l.h**" contains identifiers and low-level driver function that are used to access the devices in your embedded system. It is used inside the **ReadFromGPInput**() function to get the values from the 8 dip switches and 3 push buttons by using this Xilinx built-in general purpose input/output function of **XGpio_mGetDataReg(Base Address of Device, Channel 1**). The channel can be either channel 1 or 2 (possible to have two 32-bit OPB busses).
- "Xuint32" is used to declare variable types that are a 32-bit unsigned integers.
- The I/O function commands of "**print**()" and "**xil_printf**()" are used in the program instead of "**printf**()" because they have been optimized for embedded systems with limited memory by being much smaller in code size. The only exception is that the "**print**()" function only outputs a string with no interpretation on the string passed. For example, a "\n" passed is interpreted as a new line character and not as a carriage return and a new line as in the case of the ANSI C "**printf**()" function. The "**xil_printf**()" function does not support printing floating point numbers and printing 64-bit numbers.





3. Connect the included programming cable to the PC parallel port and the Spartan3 Starter Board (Figure 5-2). Connect a serial cable between the PC and the DB-9 connector on the board. Attach the included power supply to the board.



Figure 5-2: Spartan3 Starter Board

4. Open a HyperTerminal session by clicking Start → Programs → Accessories →
 Communications → HyperTerminal from the Main Windows Start Menu of your computer.

Within HyperTerminal, select the **COM** port that you connected the serial cable to and set it with the **baud rate** set to **9600** with **flow control** set to **none**.

5. In XPS, click the **Options** \rightarrow **Project Options** \rightarrow **Hierarchy and Flow** tab.

Select **XPS**(**XFLOW**) flow and click **OK** to accept the settings.

6. Click **Tools** → **Download** implement the design. This will start XFLOW which will compile the software, run synthesis and verification on the vhdl code, generate netlists, merge the Hardware and Software images, any other necessary operations for implementation and finally download the merged bit file to hardware.

Observe the implementation process in the console window as it progresses. Once its done, (assuming no errors found – ignore the warnings) it will program the hardware.

- Note: This will take several minutes (~ 5 to 30 minutes) depending on the speed of your computer, amount of ram memory available on the computer, and if any additional items are included in your embedded system (multiple MicroBlaze processors, additional peripherals and etc..) that needed to processed by XFLOW.
- 7. After the board is programmed, you will see a message on the terminal window showing the results of the two adder circuits used in the C program application code (Figure 5-3).





Figure 5-3: Output on HyperTerminal Screen

The terminal window shows an example of the results obtained (in decimal and hex) using MicroBlaze's adder circuit and the custom logic adder peripheral created. The terminal window shows an example of two 8-bit numbers entered via the 8 dip switches and what the user pressed to signal to get the new 8-bit numbers via the three push-buttons (any value except zero). Finally the terminal window shows the result of adding the user-inputted numbers with the custom logic adder. This process repeats forever until the power is turned off or until the FPGA is forced into reset.

Note: The program does not output any messages asking the user for any numbers to input.

8. Turn off the power when done

Conclusion

The Base System Builder (BSB) can be used in XPS to create an embedded microprocessor project with the EDK tools. Several files, including an MHS file representing the processor system and a PBD file representing the schematic view, are created. The Import Peripheral Wizard can be used to integrate your user peripheral into an existing processor system. The wizard creates the necessary directory structure and adds the necessary files (MPD, PAO) to the project directory. After the peripheral is imported, you can use the peripheral in the design by using the XPS flows process. The Xilinx generated a simple software application can be modified to access your custom IP peripheral as needed and verified by generating and downloading the bit file into actual hardware. One thing to note about this tutorial is that it only shows one method of many possible ways to build a system and to add a custom peripheral using the EDK tools.