S*ProCom²

Adding a custom IP to PowerPC using Xilinx XPS

A very short tutorial

Deian Stefan 1/8/2009

Table of Contents

The Custom Core	3
Downloading the board files and creating the XPS project	4
Creating the IP	8
Adding custom IP to default user_logic generated code	.13
Including the Customized IP	. 17
Modifying the software	. 19

The Custom Core

The custom core is very simple. It takes three inputs a,b,c and returns an output d=a*b+c with a 4 cycle latency. The verilog code is shown below:

```
module mu add(
        clk, rst, en_in,
        a, b, c,
        d,
        en_out//,
        );
        input clk, rst, en_in;
        input [0:31] a,b,c;
        output [0:31] d;
        output en_out;
        wire [0:31] m,c_d4;
        d4 bit en out delay (
                .clk(clk),
                .rst(rst),
                .i(en_in),
                .o(en_out)
        );
        d4 c_delay (
                .clk(clk),
                .rst(rst),
                .i(c),
                .o(c_d4)
        );
        //4 pipeline stages
        mult multiplier (
                .clk(clk),
                .sclr(rst),
                .a(a),
                .b(b),
                .p(m)
        );
assign d=m+c_d4;
endmodule
```

The multiplier *mult* was generated using CORE gen, with 4 pipeline stages and a synchronous reset. The *d4* and *d4_bit* are 4cycle delay clocks for 32bit inputs and 1bit input, respectively.

Downloading the board files and creating the XPS project

We're using the Virtex-2 Pro XC2VP30 on the XUPV2P Digilent board.

- Download the XUP-V2Pro Pack from Digilent
 (<u>http://www.digilentinc.com/Data/Products/XUPV2P/EDK-XUP-V2ProPack.zip</u>)
- 2. Extract it in C:\xupv2p you should see a folder named *lib*
- 3. Open Xilinx XPS and create a new project with the Base System Builder wizard

🖗 Xilinx Platform Studio	×
Create new or open existing project	
BSB (C Base System Builder wizard (recommended)	
C Blank ≚PS project	
C Open a recent project	
Browse for More Projects	
Browse installed EDK examples (projects) <u>here</u>	

4. Create a folder in C:\ called *myproj* and then select that as your project directory. Also select the previous *lib* directory in the project peripheral repository

🏶 Create New XPS Project Using BSB Wizard	×
New project	
Project file	
C:/myproj/system.xmp	Browse
Advanced options (optional: F1 for help)	
C:/xupv2p/lib	Bro <u>w</u> se
OK	Cancel

- 5. Create a new design
- 6. Select the target development board

🏶 Base System Builder - Select Board 🗾 🗾
Select a target development board:
Select board
I would like to create a system for the following development board
Board vendor: Xilinx
Board name: XUP Virtex-II Pro Development System
Board revision: C
Note: Visit the vendor website for additional board support materials.
Vendor's Website Contact Info
Download Third Party Board Definition Files
C I would like to create a system for a custom board
Board description

7. Select the PowerPC

Architecture:	<u>D</u> evice:	Pac <u>k</u> age:	<u>Speed grade:</u>	
virtex2p	💌 хс2vp30	✓ ff896	-7	Ψ.
- <mark>∏ ∐</mark> se steppin	g			
		-		
,		_		
lect the processor	you would like to use	in this design:		
Processors				
C MicroBlage				

8. Set the PPC specs:

🏶 Base System Builder - Configure Power	rPC Processsor	×
System wide settings Beference clock frequency: 100.00.MHz 300.00	Bus clock frequency:	
Beset polarity: Active LOW -		
FPGA JTAG CPU debug user pins only CPU debug and trace pins No debug	_On-chip memory (DCM) _	
PowerPC	(Use BRAM) Data: 8 KB Instruction: 32 KB	
Cache setup Enable For optimal performance, enable burst and/or cacheline on memory		
More Info	< <u>B</u> ack <u>N</u> ext > Cance	

9. Select the RS232_Uart_1, leaving the baudrate/bit/parity default and unselect the other components

🖗 Base System Builder - Configure IO Interfaces (1 of 2)		×
The following external memory and IO devices were found on your board:		
Xilinx XUP Virtex-II Pro Development System Revision C		
Please select the IO devices which you would like to use:		
10 devices		ן ר
_ 🔽 RS232_Uart_1		
Peripheral: XPS UABTLITE	Data Sheet	
Baudrate (bits per seconds): 9600		
Data bit <u>s</u> : 8		
Parity: NONE		
Use interrupt		
	J	
Ethernet_MAC	<u>D</u> ata Sheet	
	Note	

10. Unselect the DIP switches and pushbuttons

🌳 Base System Builder - Configure IO Interfaces (2 of 2)		×
The following external memory and IO devices were found on your board:		
Xilinx XUP Virtex-II Pro Development System Revision C		
Please select the IO devices which you would like to use:		
-IO devices		
DIPSWs_4Bit	Data Sheet	
PushButtons_5Bit	Data Sheet	

11. Increase the block ram interface controller size to 64k

W Base System Builder - Add Internal Peripherals (1 of 1)	×
Add other peripherals that do not interact with off-chip components. Use the "Add Peripheral" button to select from the list of available peripherals.	
If you do not wish to add any non-IO peripherals, click the "Next" button.	
	Add Peripheral
Peripherals	
-xps_bram_if_cntlr_1	Daman
Peripheral: XPS BRAM IF CNTLR	<u>H</u> emove
Memory size: 54 KB	<u>D</u> ata Sheet

12. Leave the Software Setup to default

Base System	Builder - Software Setup	×
-Devices to use	as standard input, standard output, and boot memory	
STD <u>I</u> N:	RS232_Uart_1	
STD <u>O</u> UT:	RS232_Uart_1	
Boot Memory:	ppc405_0_iocm_cnttr	
-Sample applica	ation selection	
Select the sam include a linke	ple C application that you would like to have generated. Each application will r script.	
Memory te	st	
Illustrate sy	stem aliveness and perform a basic read/write test to each memory in your system	
Peripheral	selftest	
Perform a s	imple self-test for each peripheral in your system.	

- 13. Leave the memory and peripheral test applications to default, they can be changed later to generate a new linker script
- 14. Here is the summary of the system to be created:

low is a summary of the rrect, hit <generate> to herwise return to the pr</generate>	e system you have create e enter the information into evious page to make con	d. Please review the in the XPS data base ar rections.	formation below. If it is id generate the system file:
Processor: ppo//05_0			
Processor clock frequency: On Chip Memory : 104	ency: 300.00 MHz 100.00 MHz 4 KB		
The address maps below have been automatically assigned. You can modify them using the editing features of XPS.			
Core Name	Instance Name	Base Addr	High Addr
xps_bram_if_cntlr	xps_bram_if_cntlr_1	0xFFFF0000	0xFFFFFFFF
xps_uartlite	RS232_Uart_1	0x84000000	0x8400FFFF
Processor OCM:			
Core Name	Instance Name	Base Addr	High Addr
isbram_if_cntlr	ppc405_0_iocm_cntlr	0x00000000	0x00007FFF
Processor OCM:			
Core Name	Instance Name	Base Addr	High Addr
dsbram_if_cntlr	ppc405_0_docm_cntlr	0x42004000	0x42005FFF

15. Generate and finish!

Creating the IP

- 1. In XPS click create a new peripheral (Hardware->Create or Import Peripheral)
- 2. Create template for new peripheral

🖗 Create and Import Peripheral Wizard - Periph	ieral Flow
Peripheral Flow Indicate if you want to create a new peripheral or in	nport an existing peripheral.
This tool will help you create templates for a new EDK interface files and directory structures required by EDK	compliant peripheral, or help you import an existing peripheral into an XPS project or EDK repository. The will be generated.
Create Templates	Select flow © Create templates for a new peripheral © Import gxisting 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. Options Load an existing, cip settings file (saved from a previous session)
More Info	< <u>B</u> ack Cancel

3. Add it to the XPS project

	ninat						
Indicate where y	you want to store I	he new peripheral.					
new peripheral ca rojects.	n be stored in an f	DK repository, or in	an XPS project. Wł	nen stored in an EDK	repository, the peripheral ca	an be accessed by	y multiple XPS
○ To an <u>E</u> DK us	er repository (Any	directory outside of y	our EDK installation	ı path)			
<u>R</u> epository:						<u>*</u>	Bro <u>w</u> se
To an ≚PS pro	oject						
Project:	C:\myproj\					Ŧ	Brow <u>s</u> e
Peripheral will be p	blaced under:						
C.1							

4. Name it madd

🗣 Create Peripheral - Name and Version	×
Name and Version Indicate the name and version of your peripheral.	1
Enter the name of the peripheral (upper case characters are not allowed). This name will be used as the top HDL design entity.	
N <u>a</u> me: madd	
Version: 1.00.a	
Major revision: Minor revision: Hardware/Software compatibility revision: 1 ∰ 00 ∰ a ∰	
Description: Simple multiply and add	
-Logical library name: madd v1 00 a-	
All HDL files (either created by you or generated by this tool) that are used to implement this peripheral must be compiled into the logical library name abov Any other referred logical libraries in your HDL are assumed to be available in the XPS project where this peripheral is used, or in EDK repositories indicate the XPS project settings.	re. ed in
More Info	incel

5. Choose the PLB v4.6 bus

🖗 Create Peripheral - Bus Interface	×
Bus Interface Indicate the bus interface supported by your peripheral.	× Co
To which bus will this peripheral be attached?	
(Processor Local Bus (PLB v4.6))	
C East Simplex Link (FSL)	
ATTENTION	
Refer to the following documents to get a better understanding of how user peripherals connect to the CoreConnect[TM] buses [including PLB v4.6 interconnect and DPB/PLB v3.4 interconnect] and the FSL interface.	
NOTE - Select the bus interface above and the corresponding link(s) will appear below for that interface.	
CoreConnect Specification	
PLB (v4.6) Slave IPIF Specification for single data beat transfer PLB (v4.6) Slave IPIF Specification for burdt data transfer	
PLB (v4.6) Master IPIE Specification for single data heat transfer	
PLB (v4.6) Master IPIF Specification for burst data transfer	
Note	
Xilinx recommends using the new PLB v4.6 bus standard, however, the wizard still supports the OPB and PLB v3.4 bus interfaces.	
Enable OPB and PLB v3.4 bus interfaces	
MoreInfo	Cancel

6. Select Software reset, registers and interrupt control

🗇 Create Peripheral - IPIF (IP Interface) Services	×
IPIF (IP Interface) Services Indicate the IPIF services required by your peripheral.	
Your peripheral will be connected to the PLB (v4.6) interconnect through correspondence in the interface between the PLB interconnect and the user logic. Beside the wizard tool also offers other commonly used services and configurations to sime the interface between the PLB v4.6 (PLB v	anding PLB IP Interface (IPIF) modules, which provide you with a quick way to s the standard functions like address decoding provided by the slave IPIF module, plify the implementation of the design. Slave service and configuration Typically required by most peripherals for operations like logic control, status report, data buffering, multiple memory/address space access, and etc. [PLB slave interface will always be included]. Software geset Read/Write EIFO I User logic memory space Interrupt control Master service and configuration Typically required by complex peripherals like Ethernet and PCI for commanding data transfers between regions [PLB master interface will be included if master service selected]. User logic master
More Info	< <u>B</u> ack <u>N</u> ext > Cancel

7. Disable the Device Interrupt Source controller and choose 1 logic interrupt

🗢 Create Peripheral - Interrupt Service	×
Interrupt Service Configure interrupt handling.	
The interrupt control service provides interrupt capture support which captures logic into a single interrupt output.	and coalesces various interrupts generated from IPIF, other design blocks and user Device ISC Device ISC (Interrupt Source Controller) coalesces all captured internal interrupts into a single output signal. You may eliminate Device ISC if all interrupts come from the user logic. Use Device ISC (interrupt source controller) Priority Encoder Device ISC Priority Encoder (Interrupt ID register) indicates which interrupt source has a pending interrupt. It is useful in aiding the user interrupt service routine to resolve the source of an interrupt. Use Device ISC Priority Encoder service Use Device ISC Priority Encoder service
Datasheet	User logic interrupt Number of interrupts generated by user-logic: 1 Capture mode: Level Pass Through (non-inverted) The input interrupt from the user logic has no additional capture processing applied to it. It is immediately sent to the IP ISC Interrupt Enable gating logic.
More Info	< Back Next > Cancel

8. Change the number of software accessible registers to 4

🖗 Create Peripheral - User S/W Register	X
User S/W Register Configure the software accessible registers in your peripheral.	
The user specific software accessible registers will be implemented in programs to control and to monitor the status of your user logic. Thes depending on your design. An example logic for register read/write with the status of your user logic. The status of your user logic to register read/write with the status of your user logic. The status of your user logic to register read/write with the status of your user logic. The status of your user logic to register read/write with the status of your user logic. The status of your user logic to register read/write with the status of your user logic. The status of your user logic to register read/write with the status of your user logic to register read/write with the status of your user logic. The status of your user logic to register read/write with the status of your user logic to register read/write with the status of your user logic. The status of your user logic to register to register read/write with the status of your user logic to register to re	the user-logic module of your peripheral. Such registers are typically provided for software e registers are addressable on the byte, half-word, word, double word or quad word boundaries ill be included in the user-logic module generated by the wizard tool for your reference. User logic software registers may take full advantage of the slave IPIF address-decoding service to generate CE decodes for all of the individual register of interest. The diagram on the left shows the simplest set of IPIC slave signals to read/write the registers. Number of software accessible registers: 1 (1 to 4096)
More Info	< <u>B</u> ack <u>N</u> ext > Cancel

9. Leave the IP interconnect settings to the default

Create Peripheral - IP Interconnect (IPIC)		x				
Select the interface between the logic to be implemented in your peripheral and the IPIF.						
Your peripheral will be connected to the PLB (v4.6) interconnect through suitable IPIF master/slave module(s). Your custom logic from the user-logic module interfaces to the IPIF module(s) and other sub-blocks through a set of signals called the IPI interconnect (IPIC) interface. Some of the ports are always present, some are pre-selected based on the IPIF services you required, and you can choose other optional ports to be included in the design based on your needs. Note: all IPIC ports are active high. <u>Bus2IP_CIk</u> <u>Bus2IP_Add</u> <u>Bus2IP_Add</u> <u>Bus2IP_Data</u> <u>Bus2IP_BE</u> <u>Bus2IP_Reget</u> <u>Bus2IP_Reget</u> <u>Bus2IP_Reget</u> <u>Bus2IP_Reget</u> <u>Bus2IP_RMW</u> <u>Bus2IP_RMW</u> <u>Bus2IP_RMW</u> <u>Bus2IP_Reget</u> <u>Bus2IP_Reg</u>						
More info		< <u>B</u> ack <u>N</u> ext > Cancel				

- 10. For this tutorial we'll leave the BFM simulation unselected
- 11. Select the peripheral implementation support It's helpful to have some initial code written which you can just modify

🏶 Create Peripheral - (OPTIONAL) Per	ipheral Implementation Support	×			
(OPTIONAL) Peripheral Implementation Support					
Generate optional files for hardware/software implementation					
Upon completion, this tool will create synthe need to complete the implementation of this synthesizable templates, so that you can ho	esizable HDL files that implement the IPIF services you requested. A stub 'user_logic' module will be created. You w module using standard HDL design flows. The tool will also generate EDK interface files (mpd/pao) for the bok up the generated peripheral to a processor system.	ăll			
	-Note				
Peripheral (VHDL)	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 VHDI				
	Generate ISE and VST project files to belo you implement the peripheral using VST flow				
	Constants to project mest to help you implement on functional damp to innov				
User Logic (Verilog)	Denerate template ginver mes to nelp you implement soltware interface				
(
More Info	< <u>B</u> ack Next> Canc	el			

12. Finish!

Adding custom IP to default user_logic generated code

The previous wizard created some code for your core – the wrappers for the PLB bus and a simple core which allows you to read and write to the 4 registers.

Here is the input/output portmap:

```
module user logic
 // -- ADD USER PORTS BELOW THIS LINE ------
 // --USER ports added here
 // -- ADD USER PORTS ABOVE THIS LINE ------
 // -- DO NOT EDIT BELOW THIS LINE ------
  // -- Bus protocol ports, do not add to or delete
 Bus2IP_C1k,
                                // Bus to IP clock
 Bus2IP_Reset,
                                // Bus to IP reset
 Bus2IP_Data,
                                // Bus to IP data bus
 Bus2IP_BE,
                                // Bus to IP byte enables
 Bus2IP_RdCE,
                                // Bus to IP read chip enable
 Bus2IP_WrCE,
                                // Bus to IP write chip enable
                                // IP to Bus data bus
 IP2Bus_Data,
 IP2Bus RdAck,
                                // IP to Bus read transfer acknowledgement
 IP2Bus_WrAck,
                                // IP to Bus write transfer acknowledgement
                                // IP to Bus error response
 IP2Bus Error,
                                // IP to Bus interrupt event
 IP2Bus IntrEvent
 // -- DO NOT EDIT ABOVE THIS LINE ---
); // user_logic
// -- ADD USER PARAMETERS BELOW THIS LINE ------
// --USER parameters added here
// -- ADD USER PARAMETERS ABOVE THIS LINE ------
// -- DO NOT EDIT BELOW THIS LINE ------
// -- Bus protocol parameters, do not add to or delete
parameter C_SLV_DWIDTH
                                      = 32;
parameter C NUM REG
                                       = 4;
parameter C_NUM_INTR
                                       = 1;
// -- DO NOT EDIT ABOVE THIS LINE ------
// -- ADD USER PORTS BELOW THIS LINE ------
// --USER ports added here
// -- ADD USER PORTS ABOVE THIS LINE ------
// -- DO NOT EDIT BELOW THIS LINE ------
// -- Bus protocol ports, do not add to or delete
input
                                        Bus2IP_C1k;
input
                                        Bus2IP_Reset;
input
          [0 : C_SLV_DWIDTH-1]
                                        Bus2IP_Data;
input
          [0 : C_SLV_DWIDTH/8-1]
                                        Bus2IP_BE;
input
          [0 : C_NUM_REG-1]
                                        Bus2IP_RdCE;
          [0 : C NUM REG-1]
                                        Bus2IP_WrCE;
input
          [0 : C_SLU_DWIDTH-1]
output
                                        IP2Bus_Data;
                                        IP2Bus_RdAck;
output
output
                                        IP2Bus WrAck
                                        IP2Bus Error:
output
          [0 : C_NUM_INTR-1]
                                        IP2Bus_IntrEvent;
output
```

The 4 registers we requested and some control regs/wires:

```
// Nets for user logic slave model s/w accessible register example
           [0 : C_SLU_DWIDTH-1]
[0 : C_SLU_DWIDTH-1]
reg
                                            slv_reg0;
reg
                                            slv_reg1;
reg
           [0 : C
                  SLU_DWIDTH-1
                                            slv_reg2;
reg
           [0 : C_SLV_DWIDTH-1]
                                            slv_reg3;
wire
           [0:3]
                                            slv_reg_write_sel;
           [0:3]
                                            slv_reg_read_sel;
wire
           [0 : C_SLV_DWIDTH-1]
                                            slv_ip2bus_data;
req
wire
                                            slv_read_ack;
                                            slv write ack;
wire
                                            byte_index, bit_index;
integer
```

The bus to IP (Bus2IP_Data) limits you to one write per cycle so we can access only one of the registers for writing. The Bus2IP_BE is used for make sure that the bytes in the word are good:

```
always @( posedge Bus2IP_Clk )
  begin: SLAVE REG WRITE PROC
    if ( Bus2IP_Reset == 1 )
      begin
         slv_reg1 <= 0;
         slv_reg2 <= 0;
         slv_reg3 <= 8;
      end
    else
      case ( slv_reg_write_sel )
         4'b1000
           for ( byte index = 0; byte index <= (C SLU DWIDTH/8)-1; byte index = byte index+1 )</pre>
             if ( Bus2IP_BE[byte_index] == 1 )
    for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
    slv_reg0[bit_index] <= Bus2IP_Data[bit_index];</pre>
         4'50100
           for ( byte_index = 0; byte_index <= (C_SLU_DWIDTH/8)-1; byte_index = byte_index+1 )</pre>
             if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )</pre>
                  slv_reg1[bit_index] <= Bus2IP_Data[bit_index];</pre>
         4'60010
           for ( byte_index = 0; byte_index <= (C_SLU_DWIDTH/8)-1; byte_index = byte_index+1 )
             if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index*1 )</pre>
                  slv_reg2[bit_index] <= Bus2IP_Data[bit_index];</pre>
         4160001
           for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )</pre>
             if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
slv_reg3[bit_index] <= Bus2IP_Data[bit_index];</pre>
         default : ;
      endcase
```

```
end // SLAVE_REG_WRITE_PROC
```

Writing from the IP (IP2Bus Data):

```
// implement slave model register read mux
always @( slv_reg_read_sel or slv_reg0 or slv_reg1 or slv_reg2 or slv_reg3 )
begin: SLAVE_REG_READ_PROC
case ( slv_reg_read_sel )
    4'b1000 : slv_ip2bus_data <= slv_reg0;
    4'b0100 : slv_ip2bus_data <= slv_reg1;
    4'b0010 : slv_ip2bus_data <= slv_reg2;
    4'b0001 : slv_ip2bus_data <= slv_reg3;
    default : slv_ip2bus_data <= 0;
end // SLAVE_REG_READ_PROC
</pre>
```

Finally the read/write acknowledgements and selects are quite straight forward defined as:

```
assign
slv_reg_write_sel = Bus2IP_WrCE[0:3],
slv_reg_read_sel = Bus2IP_RdCE[0:3],
slv_write_ack = Bus2IP_WrCE[0] || Bus2IP_WrCE[1] || Bus2IP_WrCE[2] || Bus2IP_WrCE[3],
slv_read_ack = Bus2IP_RdCE[0] || Bus2IP_RdCE[1] || Bus2IP_RdCE[2] || Bus2IP_RdCE[3];
assign IP2Bus_Data = slv_ip2bus_data;
assign IP2Bus_WrAck = slv_write_ack;
assign IP2Bus_Error = 0;
```

See the attached *original_user_logic.v* file for the full file.

Now, let's actually modify the file!

For this core we want 3 write registers and a read register (the result). So let's choose *slv_reg[0-2]* as the input and *slv_reg3* as the output, so lets prevent direct writing to slv_reg3 by modifying *write_ack* bit:

```
assign
    slv_reg_write_sel = Bus2IP_WrCE[0:3],
    slv_reg_read_sel = Bus2IP_RdCE[0:3],
    slv_write_ack = Bus2IP_WrCE[0] || Bus2IP_WrCE[1] || Bus2IP_WrCE[2],// || Bus2IP_WrCE[3],
    slv_read_ack = Bus2IP_RdCE[0] || Bus2IP_RdCE[1] || Bus2IP_RdCE[2] || Bus2IP_RdCE[3];
```

and disable direct writing to it:

```
slv_reg0[bit_index] <= Bus2IP_Data[bit_index];</pre>
           4'50100
              for ( byte_index = 0; byte_index <= (C_SLU_DWIDTH/8)-1; byte_index = byte_index+1 )</pre>
                if ( Bus2IP_BE[byte_index] == 1 )
for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )</pre>
                    slv_reg1[bit_index] <= Bus2IP_Data[bit_index];</pre>
           4'50010
              for ( byte_index = 0; byte_index <= (C_SLU_DWIDTH/8)-1; byte_index = byte_index+1 )</pre>
                if ( Bus2IP_BE[byte_index] == 1 )
for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )</pre>
                    slv_reg2[bit_index] <= Bus2IP_Data[bit_index];</pre>
              4'b0001 :
11111
                for ( byte_index = 0; byte_index <= (C_SLU_DWIDTH/8)-1; byte_index = byte_index+1 )</pre>
                  if ( Bus2IP_BE[byte_index] == 1 )
for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index*1 )</pre>
                       slv_reg3[bit_index] <= Bus2IP_Data[bit_index];</pre>
           default : :
         endcase
if ( Bus2IP_Reset == 1 )
    beqin
      slv_reg0 <= 0;</pre>
      slv req1 <= 0;
      slv_reg2 <= 0;
      //slv_reg3 <= 0;
      slv_regs_wr <= 0;</pre>
   end
```

and finally, change it from a *reg* to a *wire*, since we'll connect it to our core as the output *d*.

A further step would be to set *IP2Bus_Error* if an attempt to writing is made, but since we're also writing the device drivers we don't have to.

The custom core takes in a flag *en_in* and delays it by 4 cycles to output *en_out*, so as to avoid needing to halt the multiplier and under-utilize the pipelining. The *en_in* is simply a flag that says "the inputs are valid" and 4-cycles later the output of the core also has the flag *en_out* which says "this output is valid." So to integrate this in *user_logic* we want to keep track of which of the three registers were written to already and we can do this by creating a 4-bit flag, *slv_regs_wr* which is added as before the *case(slv_reg_write_sel)*:

```
always @( posedge Bus2IP_Clk )
  begin: SLAVE_REG_WRITE_P
   if ( Bus2IP_Reset == 1 )
      begin
       slv_reg1 <= 0;
       slv_reg2 <= 0;
       // slv_reg3 <= 0;
      slv_regs_wr<=0;</pre>
      end
    else
      begin
     if(slv_regs_ok)
             slv_regs_wr<=slv_reg_write_sel;</pre>
      else
             slv_regs_wr<=slv_regs_wr|slv_reg_write_sel;</pre>
      case ( slv_reg_write_sel )
        4'b1000
```

When all 3 registers were written we can consider this as valid inputs so let's create a flag *slv_regs_*ok for the *en_in* and one for the valid output, which we'll name *slv_reg3_ok*) and add our module:

```
assign slv_regs_ok = (&slv_regs_wr[0:2]);
    mu_add multadd(
        .clk(Bus2IP_Clk),
        .rst(Bus2IP_Rst),
        .en_in(slv_regs_ok),
        .a(slv_reg0),
        .b(slv_reg1),
        .c(slv_reg2),
        .d(slv_reg3), //slv_reg3=slv_reg0*slv_reg1+slv_reg2;
        .en_out(slv_reg3_ok)
);
```

Finally, let's add the interrupt event which will be high whenever the output is valid, so if our module is modified to take 100 cycles instead of 4 we don't have to write drivers which will constantly poll the *slv_reg3* register for changes:

assign	IP2Bus_Data	=	slv_ip2bus_data;
assign	IP2Bus_WrAck	=	<pre>slv_write_ack;</pre>
assign	IP2Bus_RdAck	=	slv_read_ack;
assign	IP2Bus_Error	=	0;
assign	IP2Bus_IntrEven	t	= slv_reg3_ok;

See the attached *user_logic.v* file for the full file.

We now have to copy our *mu_dd* project properly.

- 1. Copy d4,v,d4_bit.v,mu_add.v to C:\myproj\pcores\madd_v1_00_a\hdl\verilog
- 2. Copy the coregent multiplier *mult* files *mult*.* to C:\myproj\pcores\madd_v1_00_a\hdl\vhdl
- 3. Edit the madd PAO file (C:\myproj\pcores\madd_v1_00_a\data\ madd_v2_1_0.pao) to include these files. Append the following lines:

lib madd_v1_00_a mu_add verilog

- lib madd_v1_00_a d4 verilog
- lib madd_v1_00_a d4_bit verilog
- lib madd_v1_00_a mult vhdl
- 4. Copy the multiplier netlist *mult.ngc* to *C*:\myproj\implementation

Including the Customized IP

Now let's add our modified madd project to the XPS project.

1. In the IP Catalog right-click MADD and click on 'Add IP'



2. In the 'Bus Interfaces' tab expand 'madd_0' and choose the SPLB connection to be plb0

🔽 😧 🛛 🍀 🔡 🎇	🛛 🜌 ७ 📥 🛐 🏤 🗍 🖬 🖞	🖀 🛛 🐹 🄉 🖉 🕅	[]ΣΣ[]Έ	5 8 0 🗗 😽	
	Bus Interfaces Ports	Addresses			
й й в	Name	Bus Connection	IP Type	IP Version	
			ррс405	3.00.a	
			dsocm_v10	2.00.b	
╵╴┝┿┿┿┿┿┙║			isocm_v10	2.00.b	
			plb_v46	1.02.a	
	🗄 🗢 ppc405_0_docm_cntlr		dsbram_if_ontlr	3.00.Б	
	🗄 🗢 ppc405_0_iocm_cntlr		isbram_if_ontlr	3.00.Б	
	🗈 🗢 xps_bram_ii_cntil_1		xps_bram_if_ontlr	1.00.a	
	🗄 🗢 dsocm_bram		bram_block	1.00.a	
⊳ ∎	🗄 🗢 isocm_bram		bram_block	1.00.a	
L 🕨 🕂 L	🛛 🗄 🗢 plb_bram_if_cntir_1_bran	bram_block	1.00.a		
	itagppc_cntlr_0		jtagppc_ontlr	2.01.a	
	🛱 🗢 madd_0		madd	1.00.a	
	SPLB	plb0 💌			
<u> </u>	🗄 🗢 RS232_Uart_1		xps_uartlite	1.00.a	
			clock_generator	2.00.a	
	proc_sys_reset_0		proc_sys_reset	2.00.a	
1 11	11				

3. In the 'Addresses' tab select 'madd_0' change the size to 32K and click Generate Addresses

3	∑ 🛛 🗍 🌬 🔝	🎨 🛛 🔽 🗠 📥 🐚 🏫 🗍 👬 🕯	:::::::::::::::::::::::::::::::::::::	ະ 🛛 ະ ະ 🗍 🔁 🗉	I 🗆 🗖 😽		
l l	Bus Interfaces	Ports Addresses					
	nstance	Name 🛆	Base Address	High Address	Size	Bus Interface(s)	Bus
	pc405_0_docm_cntlr	C_BASEADDR	0x42002000	0x42003fff	8K 💌	DSOCM	ppc4
	pc405_0_iocm_cntlr	C_BASEADDR	0xffff8000	Oxfffffff	32K 🔽	ISOCM	ppc4
	nadd_0	C_BASEADDR	0xC9400000	0xC9407FFF	32K 💌	SPLB	plb0
	ps_bram_if_cntlr_1	C_BASEADDR	0x00000000	0x0000FFFF	64K 🗾	SPLB	рlbO
I F	RS232_Uart_1	C_BASEADDR	0x84000000	0x8400ffff	64K 💌	SPLB	plb0
	pc405_0	C_DSOCM_DCR_BASEADDR	060000100000	0Ь0000100011	4 💌	Not Connected	
	pc405_0	C_ISOCM_DCR_BASEADDR	06000010000	0Ь0000010011	4 💌	Not Connected	

4. In the 'Ports' tab expand 'madd_0' and for the 'Net' create a new connection and name it 'madd_0_irq'

🖽 😎 dsocm_bram		
🕂 🗢 isocm_bram		
🗄 🗢 plb_bram_i/_cntlr_1_bram		
⊕· ∽ jtagppc_cntlr_0		
🖕 🗢 madd_0		
IP2INTC_Irpt	madd 0 irg	
⊕- <i>◇RS232_Uart_</i> 1		
🗄 🗢 clock_generator_0		
🗄 🗢 proc_sys_reset_0		

5. Expand 'ppc405_0' and find 'EIC405EXTINPUTIRQ' and choose 'mad_0_irq' instead of 'No Connection'

- C405JTGUPDATEDR	No Connection	I U
C405JTGSHIFTDR	No Connection	V 0
- C405JTGPGMOUT	No Connection	V 0
C405JTGEXTEST	No Connection	V 0
C405JTGCAPTUREDR	No Connection	V 0
EICC405EXTINPUTIRQ	madd 0 irg	
EICC405CRITINPUTIRQ	No Connection	
BRAMISOCMCLK	sys clk s	
BRAMDSOCMCLK	sys clk s	•
DCRCLK	No Connection	V
MCPPCRST	No Connection	

The core is now connected to the bus and the interrupt is connected to the PowerPC, let's modify the software now.

Modifying the software

1. Let's first add the drivers created by the wizard. In the Applications tab click on 'Add Software Application Project' and name it 'madd_test'



2. Right click on 'TestApp_memory' and deselect it from begin initialized on the BRAMs



- 3. Select 'madd_test' to be initialized instead.
- 4. Right-click on it again and click go to generate the linker script. Modify the heap and stack

Sections View:			Heap and Stack View:			
Section	Size (bytes)	Memory	Section	Size (bytes)	Memory	
vectors.	0x00000000	xps_bram_if_cnt 💌	Heap	0x400	ppc405_0_docrr 💌	
.text	0x00000FE8	xps_bram_if_cnt 💌	Stack	0x400	ppc405_0_docrr 💌	
rodata	0x00000716	xps_bram_if_cnt 💌				
rodata1	0x00000000	xps_bram_if_cnt 💌				
sdata2	0x00000000	xps_bram_if_cnt 💌				
.sbss2	0x00000000	xps_bram_if_cnt 💌				
.data	0x000000F8	xps_bram_if_cnt 💌	Memories View:			
data1	0x00000000	xps_bram_if_cnt 💌	Memory	Start Address	Length	
fixup	0x00000000	xps_bram_if_cnt 💌	xps_bram_if_cntlr_1	0x0000000	64K	
sdata	0x00000010	xps_bram_if_cnt 💌	ppc405_0_iocm_cn	0xFFFF8000	32K	
.sbss	0x0000008	xps_bram_if_cnt 💌	ppc405_0_docm_cr	r 0x42002000	8K	
.bss	0x0000001C	xps_bram_if_cnt 💌				
	Add Sec	ction Delete Section	l ELF file used to popu	late section inform	ation:	
loot and Vecto	r Sections:		C:\myproj\madd_tes	t\executable.elf		
Section	Address	Memory				
.boot0	0xFFFFFFEC	ppc405_0_iocm_cnl	Output Linker Script:	C:\myproj\madd	_test\madd_test_linker_s	cript.ld
.boot	0xFFFFFFFC	ppc405 0 jocm cnl				

- 5. Now add the sources (*madd*.[*ch*],*madd_selftest.c*) from *C*:*myproj**drivers**madd_v1_00_a**src*
- 6. Modify *madd_selftest.c* by commenting out the lines related to register 3:

```
93
       {
94
         xil_printf(" - slave register 2 word 0 write/read failed\n\r");
95
         return XST FAILURE;
96
       }
97
     // xil_printf(" - write 4 to slave register 3 word 0\n\r");
    // MADD_mWriteSlaveReg3(baseaddr, 0, 4);
// Reg32Value = MADD_mReadSlaveReg3(baseaddr, 0);
98
99
100 // xil printf(" - read %d from register 3 word 0\n\r", Reg32Value);
101 // if ( Reg32Value != (Xuint32) 4 )
    11 (
102
103
     17
           xil printf("
                         - slave register 3 word 0 write/read failed\n\r");
          return XST_FAILURE;
    11
104
105
    11 \rightarrow
106 // xil printf(" - slave register write/read passed\n\n\r");
107
108
       /*
109
        * Enable all possible interrupts and clear interrupt status register(s)
110
        */
111
       xil printf("Interrupt controller test...\n\r");
112
       Reg32Value = MADD_mReadReg(baseaddr, MADD_INTR_IPISR_OFFSET);
       xil_printf(" - IP (user logic) interrupt status : Ox%08x\n\r", Reg32Value);
113
```

7. Add a new source file named 'madd_test.c' to C:\myproj\madd_test.c:

```
1 #include "xparameters.h"
2 #include "xbasic_types.h"
3
   #include "stdio.h"
4 #include "madd.h"
5
6
7
8
   int main(void) {
9
     Xuint32 IpStatus;
10
     XStatus stat;
     - print("-- main() man -- \r \n");
11
12
     if((stat=MADD_SelfTest(((void *)XPAR_MADD_0_BASEADDR)))==XST_SUCCESS) {
13
        print("Test OK!\n\r");
14
15
     } else {
        print("Test FAIL!\n\r");
16
17
      - }
18
19
     print("-- ! main() man --\r\n");
20
       return stat;
21
    }
```

- 8. Right-click on 'madd_test' and click on 'Build Project'
- 9. Click on Hardware->Generate Netlist and then Hardware->Generate Bitstream
- 10. Open your favorite terminal client (Putty and connect to the COM port to which you connected the dev board)
- 11. In XPS click on Device Configuration->Download Bitstream
 - a. If you get an error copy the mult.ncg again and retry remember that mult was created using COREGEN

The output should be as shown below:

🖻 COM5 - PuTTY
main() man
* * * * * * * * * * * * * * * * * * * *
* User Peripheral Self Test
* * * * * * * * * * * * * * * * * * * *
Soft reset test
 write 0x0000000Å to software reset register
- soft reset passed
User logic slave module test
- write 1 to slave register 0 word 0
- read 1 from register 0 word 0
- write 2 to slave register 1 word 0
- read 2 from register 1 word 0
- write 3 to slave register 2 word U
- read 3 from register 2 word U
Interrupt controller test
- IP (user logic) interrupt status : 0x00000000
- Clear IP (user logic) interrupt status register
- bevice (peripheral) interrupt status : 0x00000000
- clear bevice (peripheral) interrupt status register
- enable all possible interrupt(s)
- write/read interrupt register passed
Test OK!
' main() man

- 12. Now let's write our own test code, modifying the original self test
- 13. We are going to use interrupts to read *slv_reg3* so we need to include some additional header files:

```
1 #include "xparameters.h"
2 #include "xbasic_types.h"
3 #include "time.h"
4 #include "stdio.h"
5 #include "xexception_l.h"
6 #include "madd.h"
```

14. Let's modify main to setup the interrupts and register the interrupt handler:

```
int main(void) {
82
83
        Xuint32 IpStatus;
84
        XStatus stat;
        print("-- main() man --\r\n");
85
86 #ifdef SELFTEST
        if((stat=MADD SelfTest(((void *)XPAR MADD 0 BASEADDR)))==XST SUCCESS) (
87
88 #else
        xil_printf("Initializing interrupt vector table\r\n");
89
90
        XExc Init();
        XExc RegisterHandler(XEXC ID NON CRITICAL INT,
91
92
                             (XExceptionHandler) MADD Intr Handler,
                             (void *) XPAR MADD O BASEADDR);
93
94
95
        if((stat=MADD_Test(((void *)XPAR_MADD_O_BASEADDR)))==XST_SUCCESS) (
96 #endif
          print("Test OK!\n\r");
97
98
        } else {
           print("Test FAIL!\n\r");
99
100
        }
101
102
        print("-- ! main() man --\r\n");
103
        return stat;
104
    }
105
```

15. In a real example we should use conditional sleep instead on a flag, but for this example we will use a simple flag:

```
8 //flag which is set by the interrupt handler
9 volatile unsigned intr flag=0;
```

16. And create an interrupt handler which just sets the flag:

```
void MADD Intr Handler(void * baseaddr p)
11
12
    {
13
       Xuint32 baseaddr;
14
       Xuint32 IpStatus;
15
       Xuint32 Reg32Value=Oxdeadbeef;
16
17
       XASSERT NONVOID (baseaddr p != XNULL);
18
       baseaddr = (Xuint32) baseaddr p;
19
20
       //should be disabling interrupts in a real example
       IpStatus = MADD mReadReg(baseaddr, MADD INTR IPISR OFFSET);
21
22
       intr flag=1;
       MADD mWriteReg(baseaddr, MADD INTR IPISR OFFSET, IpStatus);
23
24
25
```

17. Finally let's look at the test code:

```
26
    XStatus MADD Test(void * baseaddr p)
27
    {
28
       int i;
29
       Xuint32 baseaddr;
30
       Xuint32 Reg32Value;
31
32
33
       XASSERT NONVOID(baseaddr p != XNULL);
34
       baseaddr = (Xuint32) baseaddr p;
       xil printf("Soft reset test...");
35
36
       MADD_mReset (baseaddr);
37
       xil_printf("OK!\n\n\r");
38
39
       MADD EnableInterrupt(baseaddr p);
       XExc mEnableExceptions(XEXC NON CRITICAL);
40
41
42
       for(i=0;i<15;i+=3) { //loop a bit</pre>
43
          intr flag=0;
44
45
          xil printf("writing slv reg0=%d...\n\r",i);
46
          MADD_mWriteSlaveRegO(baseaddr, 0, i);
47
          Reg32Value = MADD mReadSlaveReg0(baseaddr, 0);
48
          xil_printf("read slv_reg0=%d \n\r", Reg32Value);
49
50
          xil_printf("writing slv_reg1=%d...\n\r",i+1);
51
          MADD mWriteSlaveReg1(baseaddr, 0, i+1);
52
          Reg32Value = MADD mReadSlaveReg1(baseaddr, 0);
53
          xil_printf("read slv_reg1=%d \n\r", Reg32Value);
54
55
          xil printf("writing slv reg2=%d...\n\r",i+2);
           MADD mWriteSlaveReg2(baseaddr, 0, i+2);
56
57
          Reg32Value = MADD mReadSlaveReg2(baseaddr, 0);
          xil_printf("read slv_reg2=%d \n\r", Reg32Value);
58
59
60
          xil printf("expecting %d\n\r",i*(i+1)+(i+2));
61
62
          //in a real example a waitque should be used
63
          //sleeping until an interrupt arrive, not as in this example
64
          sleep(2);
65
          Reg32Value = MADD_mReadSlaveReg3(baseaddr, 0);
66
          xil printf("read slv reg3=%d\n\r", Reg32Value);
67
          if(!intr_flag || (i*(i+1)+(i+2))!=Reg32Value) {
68
             return XST FAILURE;
69
          }
70
       }
71
72
      return XST SUCCESS;
73
```

The output should be as shown below:

🖉 COM5 - PuTTY	<u>- </u>
main() man	<u> </u>
Initializing interrupt vector table	
Soft reset testOK!	
writing slv_reg0=0	
read slv_reg0=0	
writing slv_reg1=1	
read slv_reg1=1	
writing slv_reg2=2	
read slv_reg2=2	
expecting 2	
read slv_reg3=2	
writing slv_regO=3	
read slv_regO=3	
writing slv_reg1=4	
read slv_reg1=4	
writing slv_reg2=5	
read slv_reg2=5	-
expecting 17	
read slv_reg3=17	
writing slv_regO=6	
read slv_reg0=6	
writing slv_reg1=7	
read slv_reg1=7	
writing slv_reg2=8	
read slv_reg2=8	
expecting SU	
read siv_reg3=50	
writing siv_rego-9	
riting alw reg1=10	
read alv_reg1=10	
writing sly reg2=11	
read slv reg2=11	
expecting 101	
read slv reg3=101	
writing slv reg0=12	
read slv reg0=12	
writing slv reg1=13	
read slv_reg1=13	
writing slv_reg2=14	
read slv_reg2=14	
expecting 170	
read slv_reg3=170	
Test OK!	
<u> ! main() man</u>	

That's it!