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

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:

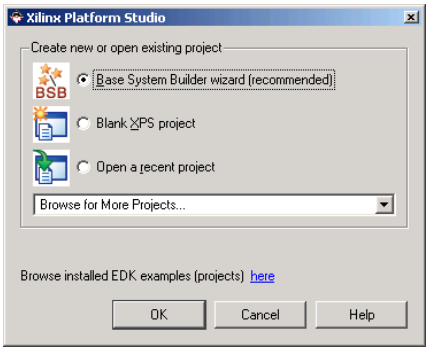
```
module mu_add(
        clk, rst, en_in,
        a, b, c,
        d,
        en_out//,
        );
        input clk, rst, en_in;
        input [0:31] a,b,c;
        output [0:31] d;
        output en_out;

        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.

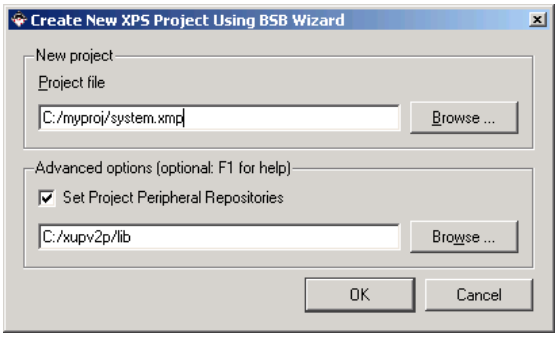
Downloading the board files and creating the XPS project

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

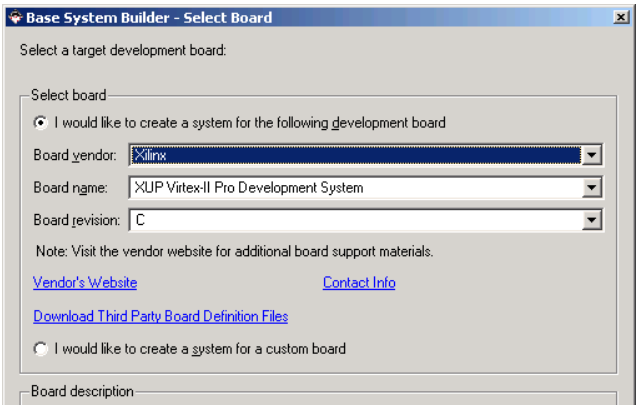
1. Download the *XUP-V2Pro Pack* from Digilent (http://www.digilentinc.com/Data/Products/XUPV2P/EDK-XUP-V2ProPack.zip)
2. Extract it in *C:\xupv2p* – you should see a folder named *lib*
3. Open Xilinx XPS and create a new project with the Base System Builder wizard



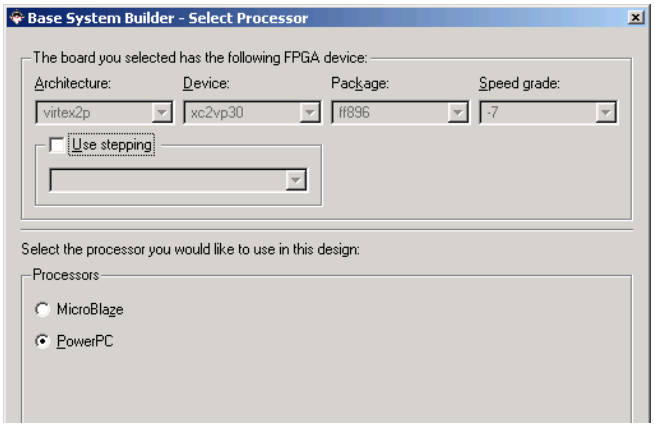
4. Create a folder in C:\ called *myproj* and then select that as your project directory. Also select the previous *lib* directory in the project peripheral repository



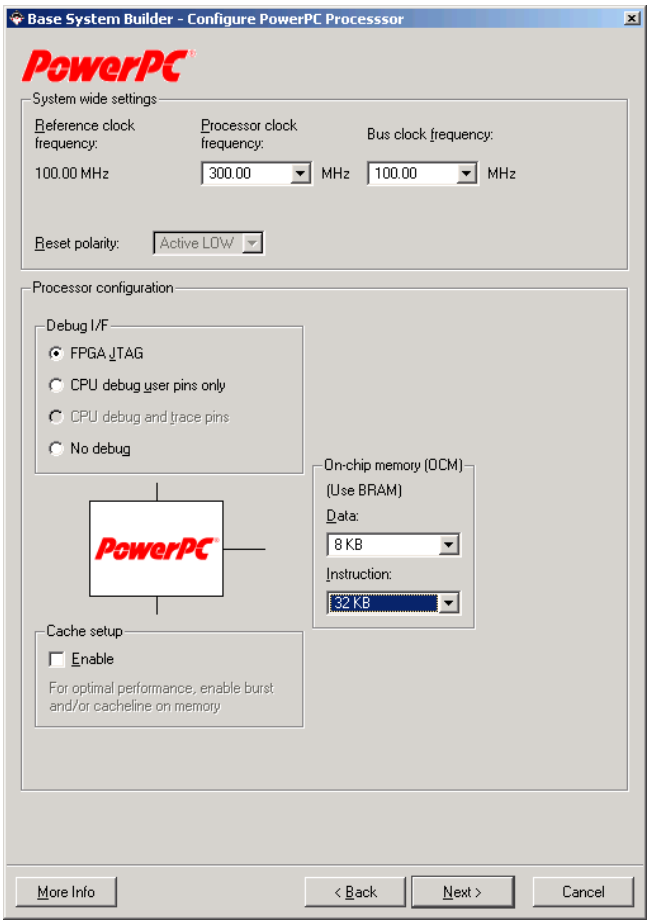
5. Create a new design
6. Select the target development board



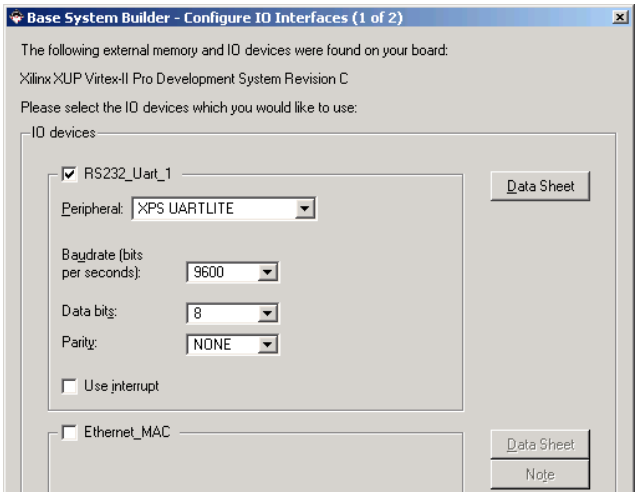
7. Select the PowerPC



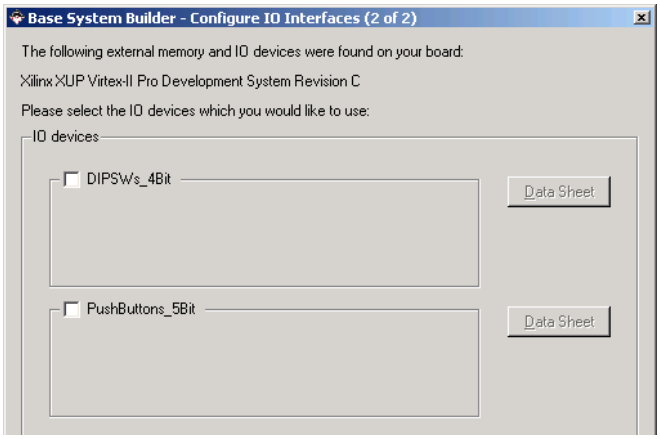
8. Set the PPC specs:



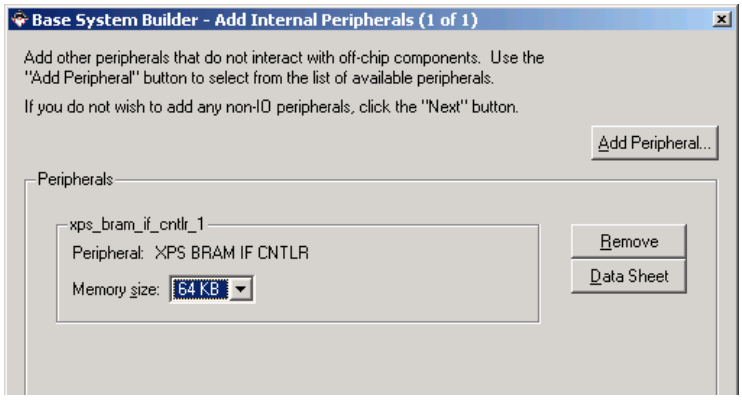
9. Select the RS232_Uart_1, leaving the baudrate/bit/parity default and unselect the other components



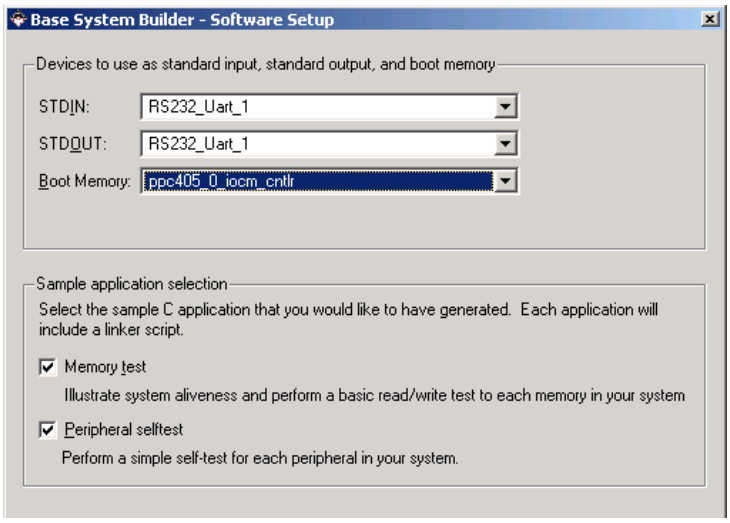
10. Unselect the DIP switches and pushbuttons



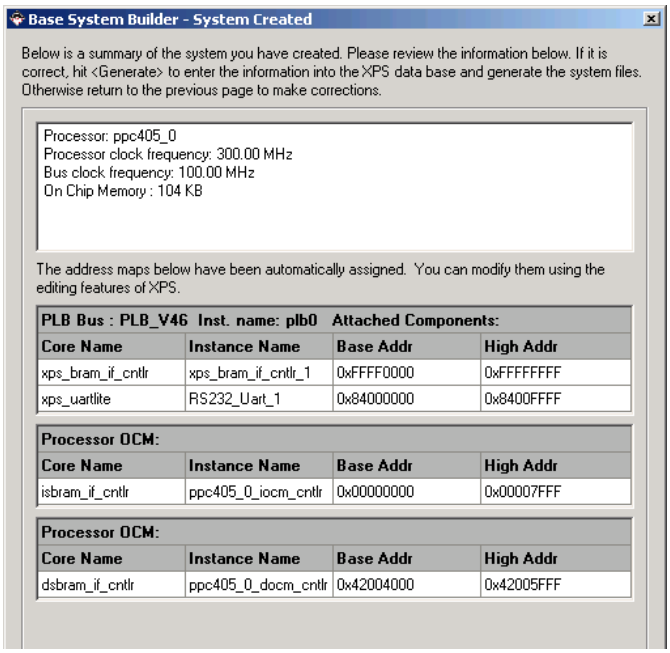
11. Increase the block ram interface controller size to 64k



12. Leave the Software Setup to default



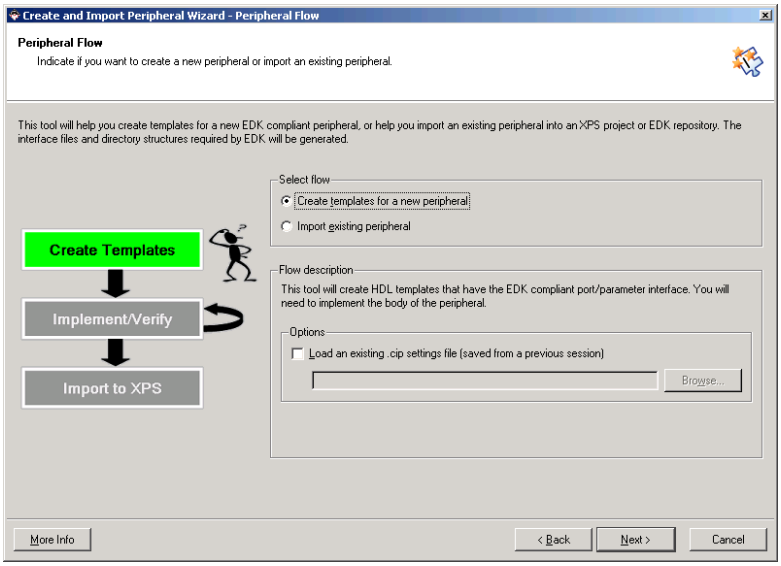
13. Leave the memory and peripheral test applications to default, they can be changed later to generate a new linker script
14. Here is the summary of the system to be created:



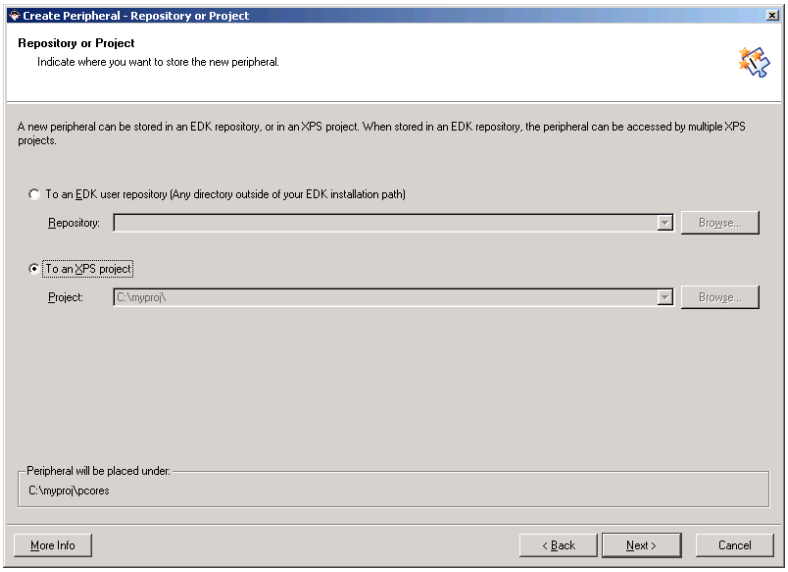
15. Generate and finish!

Creating the IP

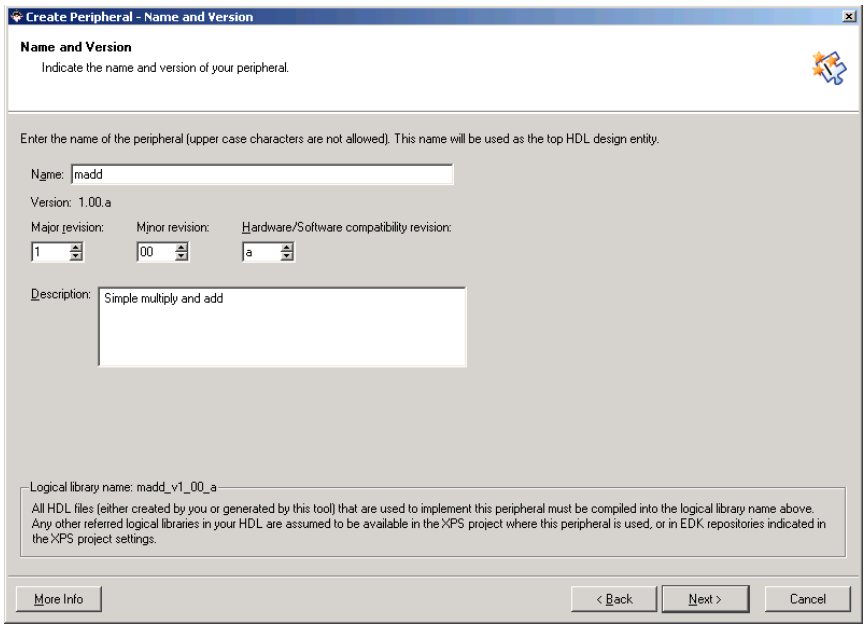
1. In XPS click create a new peripheral (Hardware->Create or Import Peripheral)
2. Create template for new peripheral



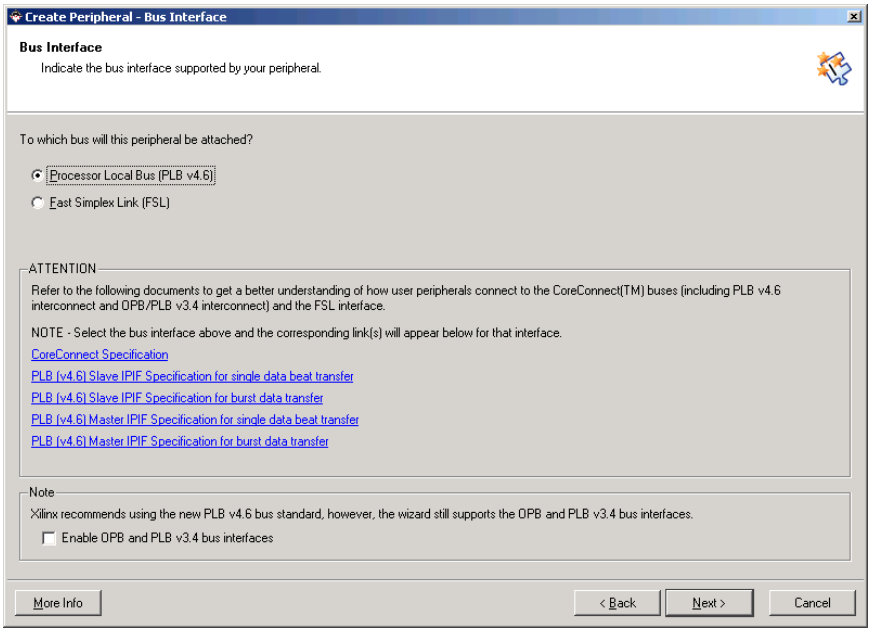
3. Add it to the XPS project



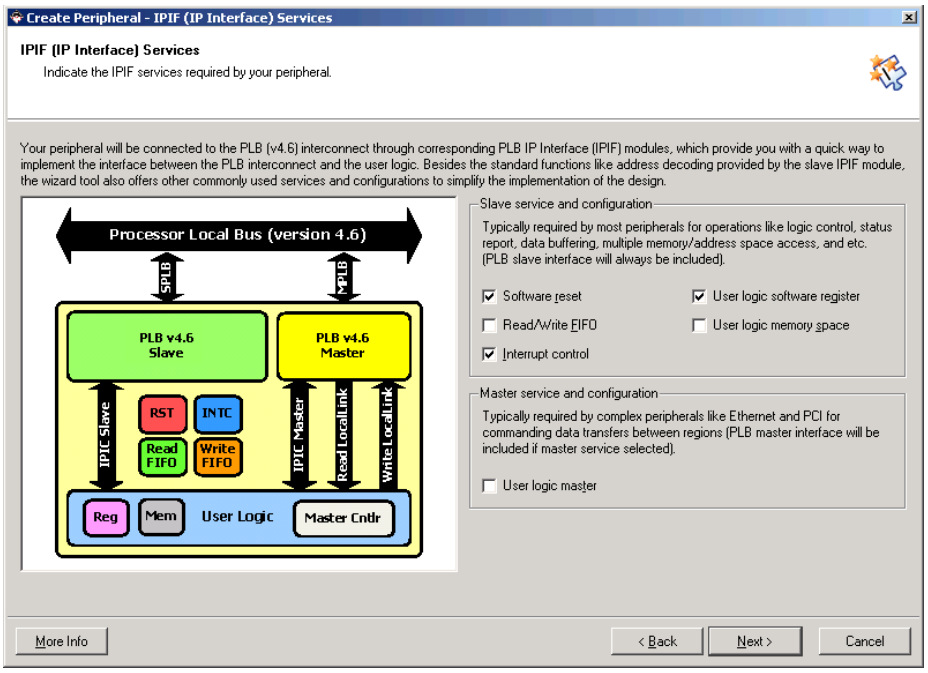
4. Name it *madd*



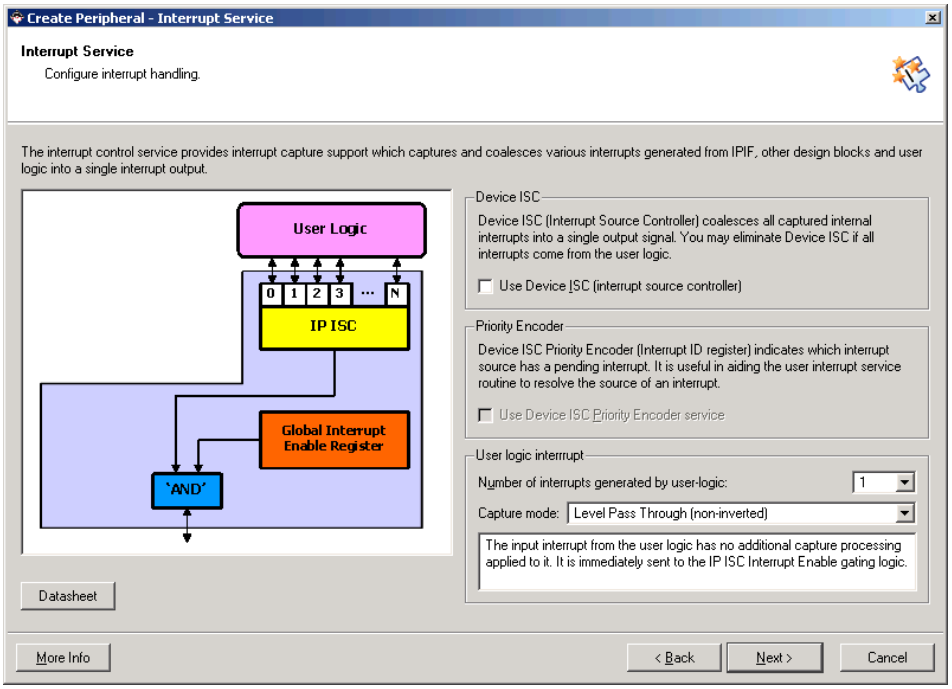
5. Choose the PLB v4.6 bus



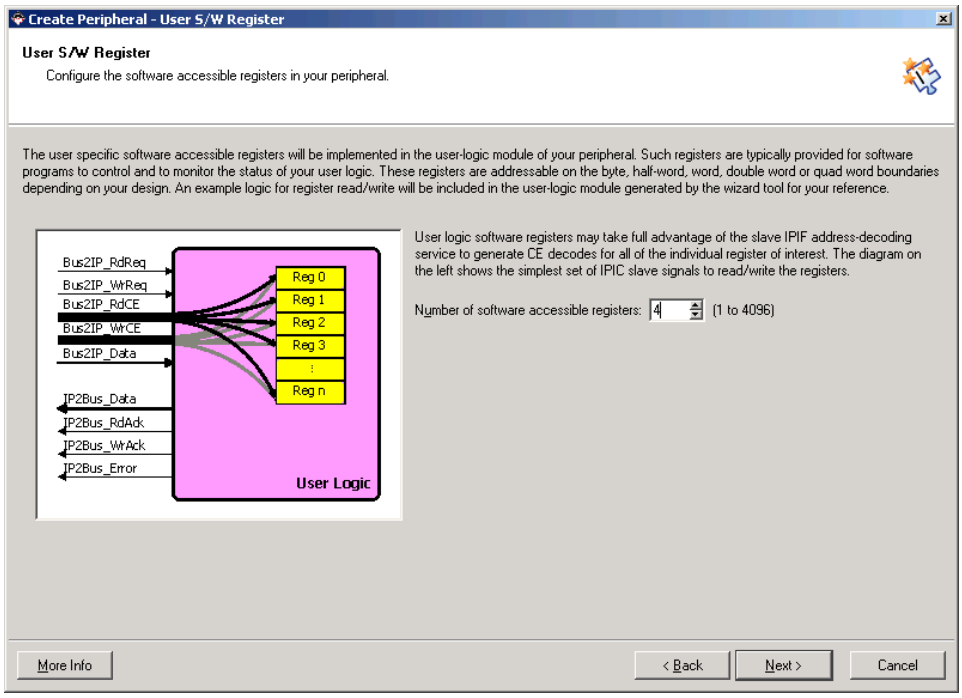
6. Select Software reset, registers and interrupt control



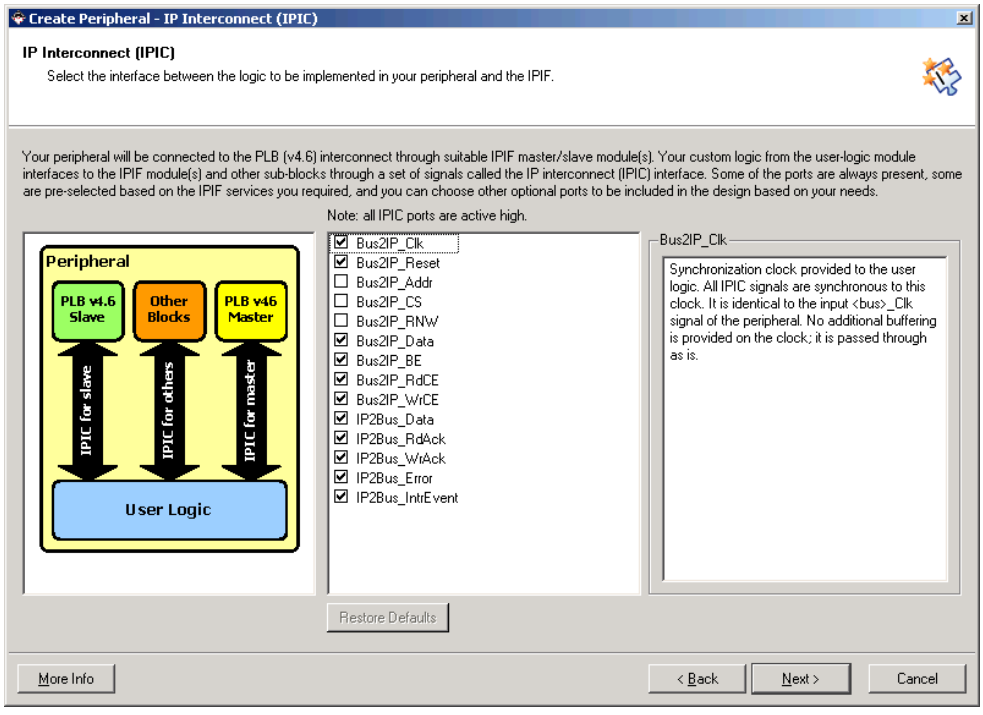
7. Disable the Device Interrupt Source controller and choose 1 logic interrupt



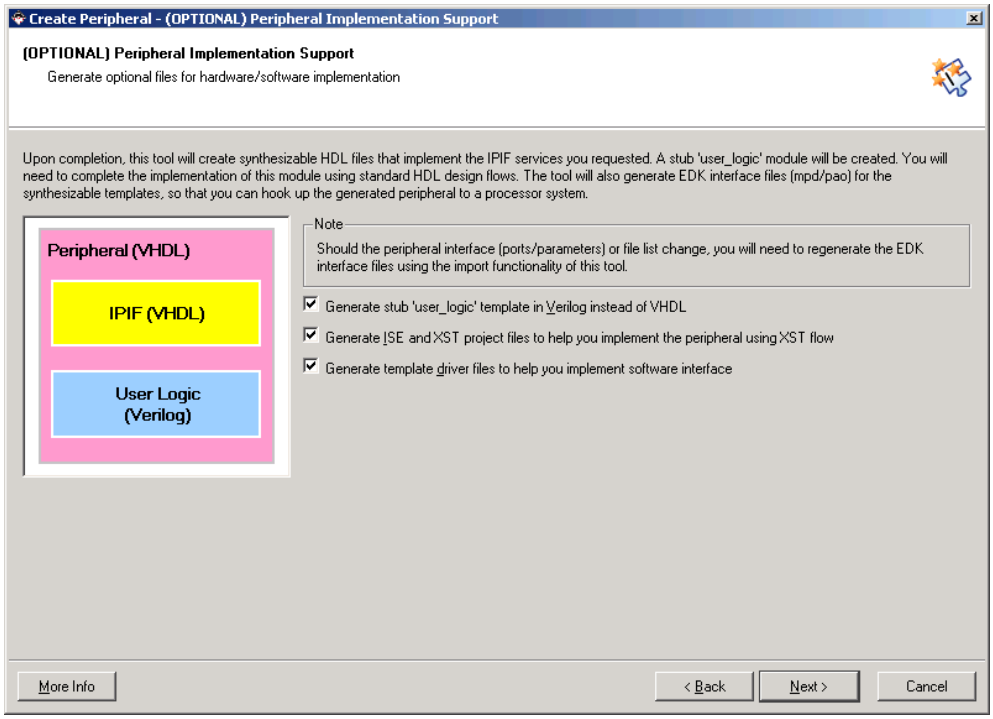
8. Change the number of software accessible registers to 4



9. Leave the IP interconnect settings to the default



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) Peripheral Implementation Support
(OPTIONAL) Peripheral Implementation Support
Generate optional files for hardware/software implementation
Upon completion, this tool will create synthesizable HDL files that implement the IPIF services you requested. A stub 'user_logic' module will be created. You will need to complete the implementation of this module using standard HDL design flows. The tool will also generate EDK interface files (mpd/pao) for the synthesizable templates, so that you can hook up the generated peripheral to a processor system.
Peripheral (VHDL)
IPIF (VHDL)
User Logic (Verilog)
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
< Back
Next >
Cancel

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_Clk,                     // 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
  IP2Bus_Data,                    // IP to Bus data bus
  IP2Bus_RdAck,                   // IP to Bus read transfer acknowledgement
  IP2Bus_WrAck,                   // IP to Bus write transfer acknowledgement
  IP2Bus_Error,                   // IP to Bus error response
  IP2Bus_IntrEvent                // IP to Bus interrupt event
  // -- 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_Clk;
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;
input      [0 : C_NUM_REG-1]              Bus2IP_WrCE;
output     [0 : C_SLV_DWIDTH-1]           IP2Bus_Data;
output                                    IP2Bus_RdAck;
output                                    IP2Bus_WrAck;
output                                    IP2Bus_Error;
output     [0 : C_NUM_INTR-1]             IP2Bus_IntrEvent;
```

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

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

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_reg0 <= 0;
        slv_reg1 <= 0;
        slv_reg2 <= 0;
        slv_reg3 <= 0;
      end
    else
      case ( slv_reg_write_sel )
        4'b1000 :
          for ( byte_index = 0; byte_index <= (C_SLV_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 )
                slv_reg0[bit_index] <= Bus2IP_Data[bit_index];
        4'b0100 :
          for ( byte_index = 0; byte_index <= (C_SLV_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 )
                slv_reg1[bit_index] <= Bus2IP_Data[bit_index];
        4'b0010 :
          for ( byte_index = 0; byte_index <= (C_SLV_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 )
                slv_reg2[bit_index] <= Bus2IP_Data[bit_index];
        4'b0001 :
          for ( byte_index = 0; byte_index <= (C_SLV_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 )
                slv_reg3[bit_index] <= Bus2IP_Data[bit_index];
        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;
     endcase

   end // SLAVE_REG_READ_PROC
```

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_RdAck   = slv_read_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];
          4'b0100 :
            for ( byte_index = 0; byte_index <= (C_SLV_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 )
                  slv_reg1[bit_index] <= Bus2IP_Data[bit_index];
          4'b0010 :
            for ( byte_index = 0; byte_index <= (C_SLV_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 )
                  slv_reg2[bit_index] <= Bus2IP_Data[bit_index];
//          4'b0001 :
//            for ( byte_index = 0; byte_index <= (C_SLV_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 )
//                  slv_reg3[bit_index] <= Bus2IP_Data[bit_index];
          default : ;
        endcase

 if ( Bus2IP_Reset == 1 )
   begin
     slv_reg0 <= 0;
     slv_reg1 <= 0;
     slv_reg2 <= 0;
     //slv_reg3 <= 0;
     slv_regs_wr <= 0;
   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_PROC

    if ( Bus2IP_Reset == 1 )
      begin
        slv_reg0 <= 0;
        slv_reg1 <= 0;
        slv_reg2 <= 0;
       // slv_reg3 <= 0;
       slv_regs_wr<=0;
      end
    else
      begin
      if(slv_regs_ok)
              slv_regs_wr<=slv_reg_write_sel;
      else
              slv_regs_wr<=slv_regs_wr|slv_reg_write_sel;

      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   = slv_write_ack;
 assign IP2Bus_RdAck   = slv_read_ack;
 assign IP2Bus_Error   = 0;
 assign IP2Bus_IntrEvent       = slv_reg3_ok;
```

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

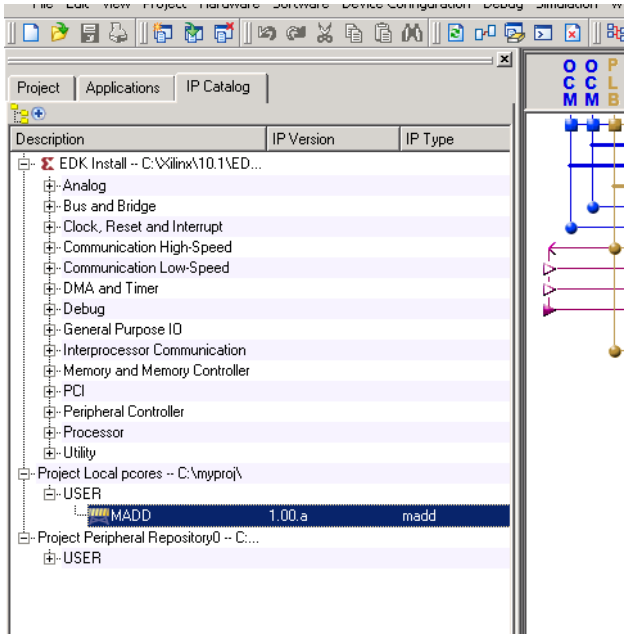
We now have to copy our *mu_dd* project properly.

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*

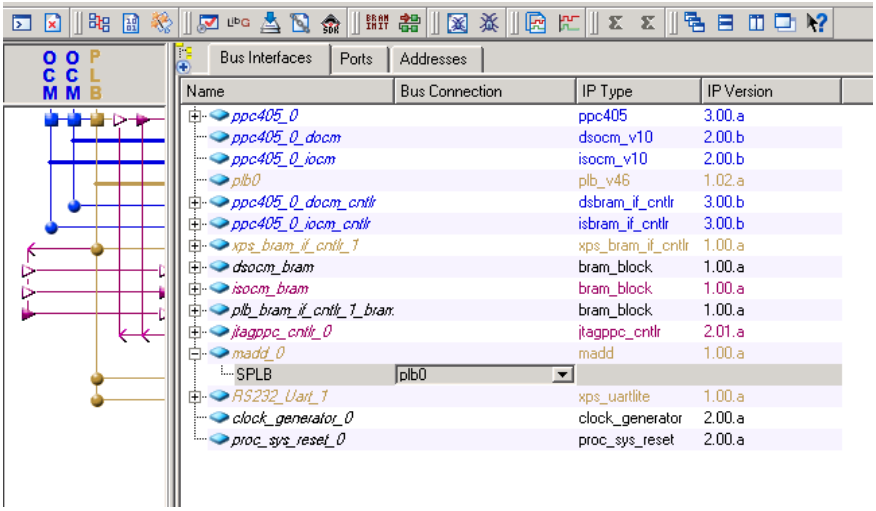
Including the Customized IP

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

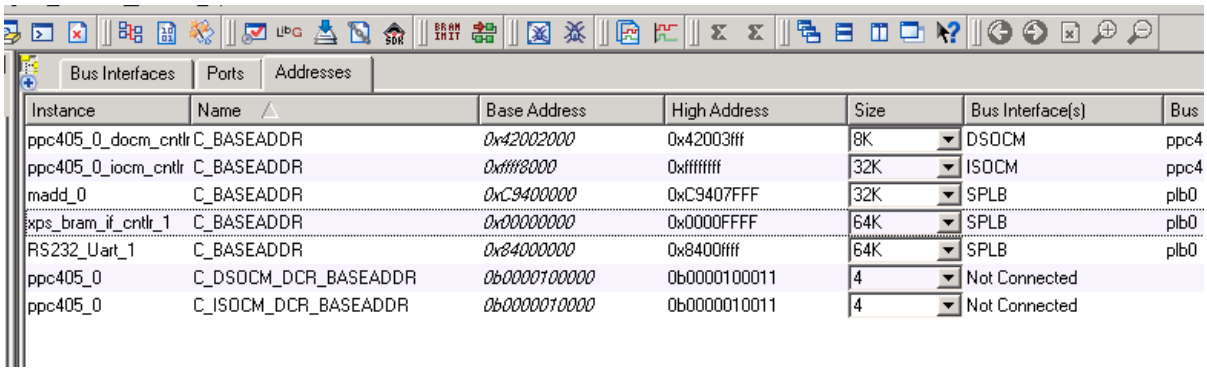
1. In the IP Catalog right-click MADD and click on 'Add IP'



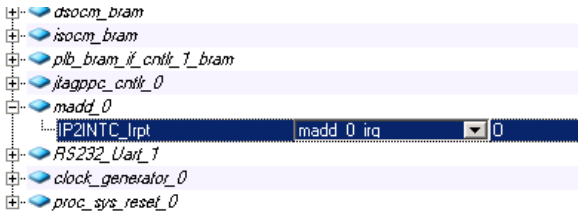
2. In the 'Bus Interfaces' tab expand 'madd_0' and choose the SPLB connection to be plb0



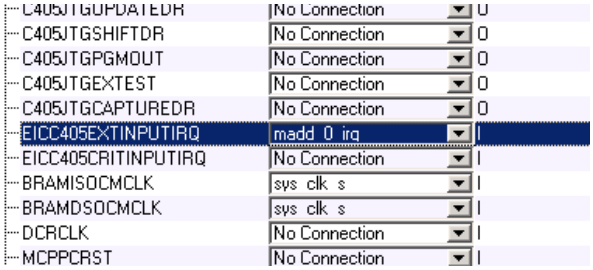
3. In the 'Addresses' tab select 'madd_0' change the size to 32K and click Generate Addresses



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



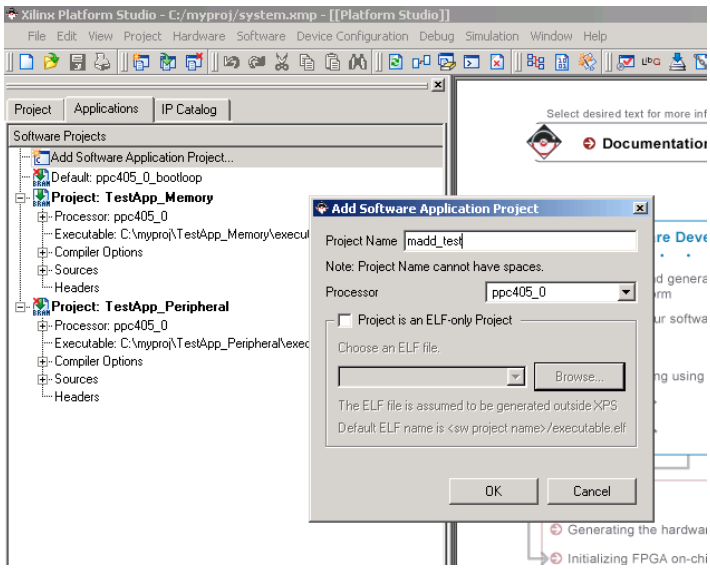
5. Expand ‘ppc405_0’ and find ‘EIC405EXTINPUTIRQ’ and choose ‘mad_0_irq’ instead of ‘No Connection’



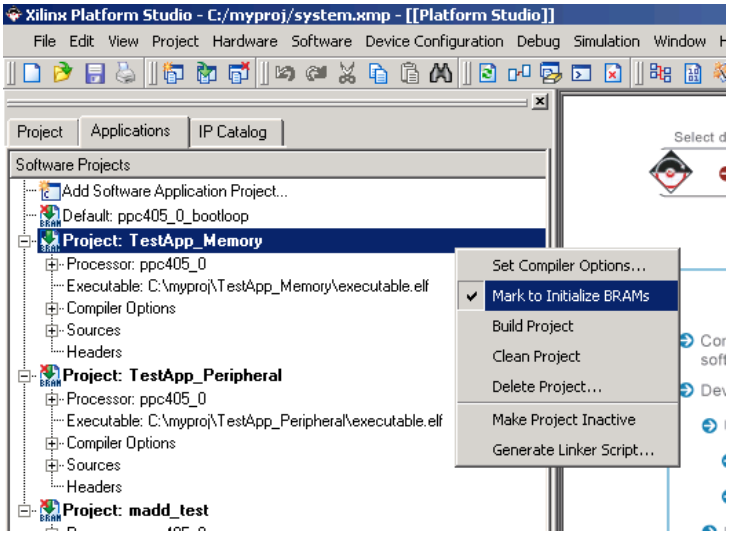
The core is now connected to the bus and the interrupt is connected to the PowerPC, let’s modify the software now.

Modifying the software

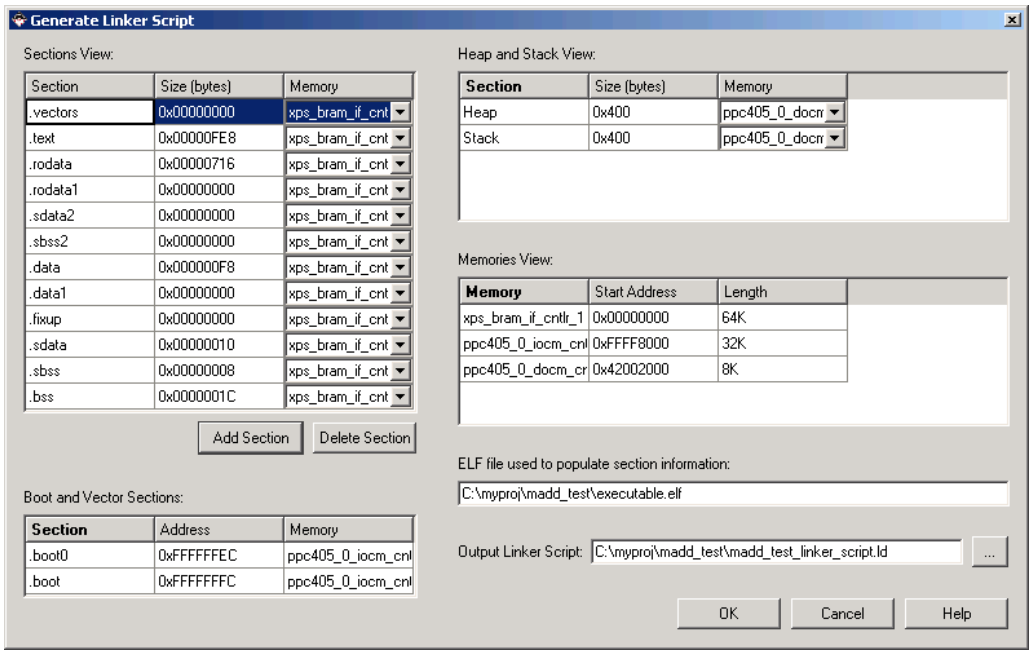
1. Let's first add the drivers created by the wizard. In the Applications tab click on 'Add Software Application Project' and name it 'madd_test'



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



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:

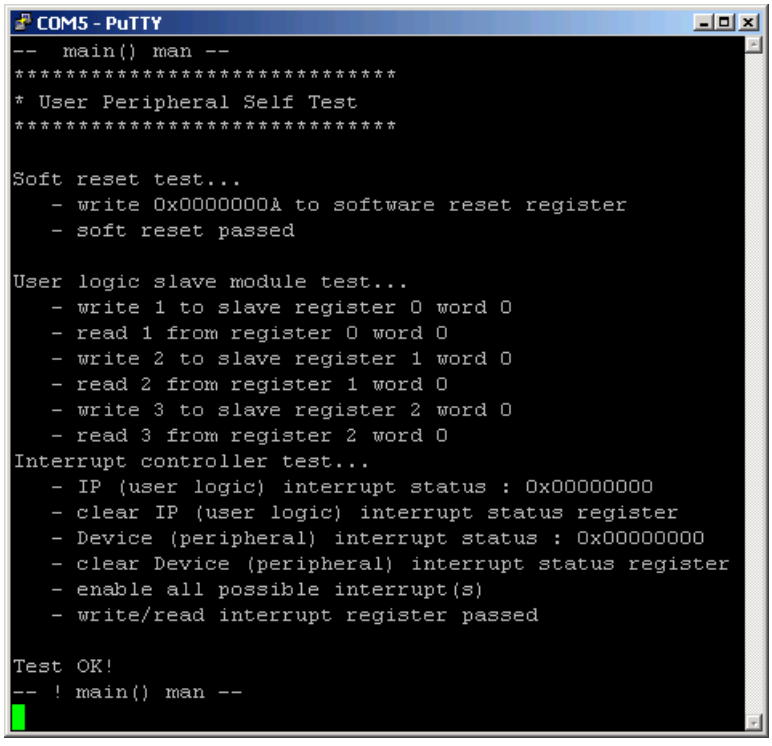
```
 93    {
 94      xil_printf("   - slave register 2 word 0 write/read failed\n\r");
 95      return XST_FAILURE;
 96    }
 97  //  xil_printf("   - write 4 to slave register 3 word 0\n\r");
 98  //  MADD_mWriteSlaveReg3(baseaddr, 0, 4);
 99  //  Reg32Value = MADD_mReadSlaveReg3(baseaddr, 0);
100  //  xil_printf("   - read %d from register 3 word 0\n\r", Reg32Value);
101  //  if ( Reg32Value != (Xuint32) 4 )
102  //  {
103  //    xil_printf("   - slave register 3 word 0 write/read failed\n\r");
104  //    return XST_FAILURE;
105  //  }
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);
113    xil_printf("   - IP (user logic) interrupt status : 0x%08x\n\r", Reg32Value);
```

7. Add a new source file named ‘madd_test.c’ to *C:\myproj\madd_test.c:*

```c
#include "xparameters.h"
#include "xbasic_types.h"
#include "stdio.h"
#include "madd.h"



int main(void) {
   Xuint32 IpStatus;
   XStatus stat;
   print("--  main() man --\r\n");

   if((stat=MADD_SelfTest(((void *)XPAR_MADD_0_BASEADDR))==XST_SUCCESS) {
      print("Test OK!\n\r");
   } else {
      print("Test FAIL!\n\r");
   }

   print("-- ! main() man --\r\n");
   return stat;
}
```

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:



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:

```
82  int main(void) {
83     Xuint32 IpStatus;
84     XStatus stat;
85     print("--  main() man --\r\n");
86  #ifdef SELFTEST
87     if((stat=MADD_SelfTest(((void *)XPAR_MADD_0_BASEADDR)))==XST_SUCCESS) {
88  #else
89     xil_printf("Initializing interrupt vector table\r\n");
90     XExc_Init();
91     XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
92                          (XExceptionHandler)MADD_Intr_Handler,
93                          (void *)XPAR_MADD_0_BASEADDR);
94
95     if((stat=MADD_Test(((void *)XPAR_MADD_0_BASEADDR)))==XST_SUCCESS) {
96  #endif
97        print("Test OK!\n\r");
98     } else {
99        print("Test FAIL!\n\r");
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;
```

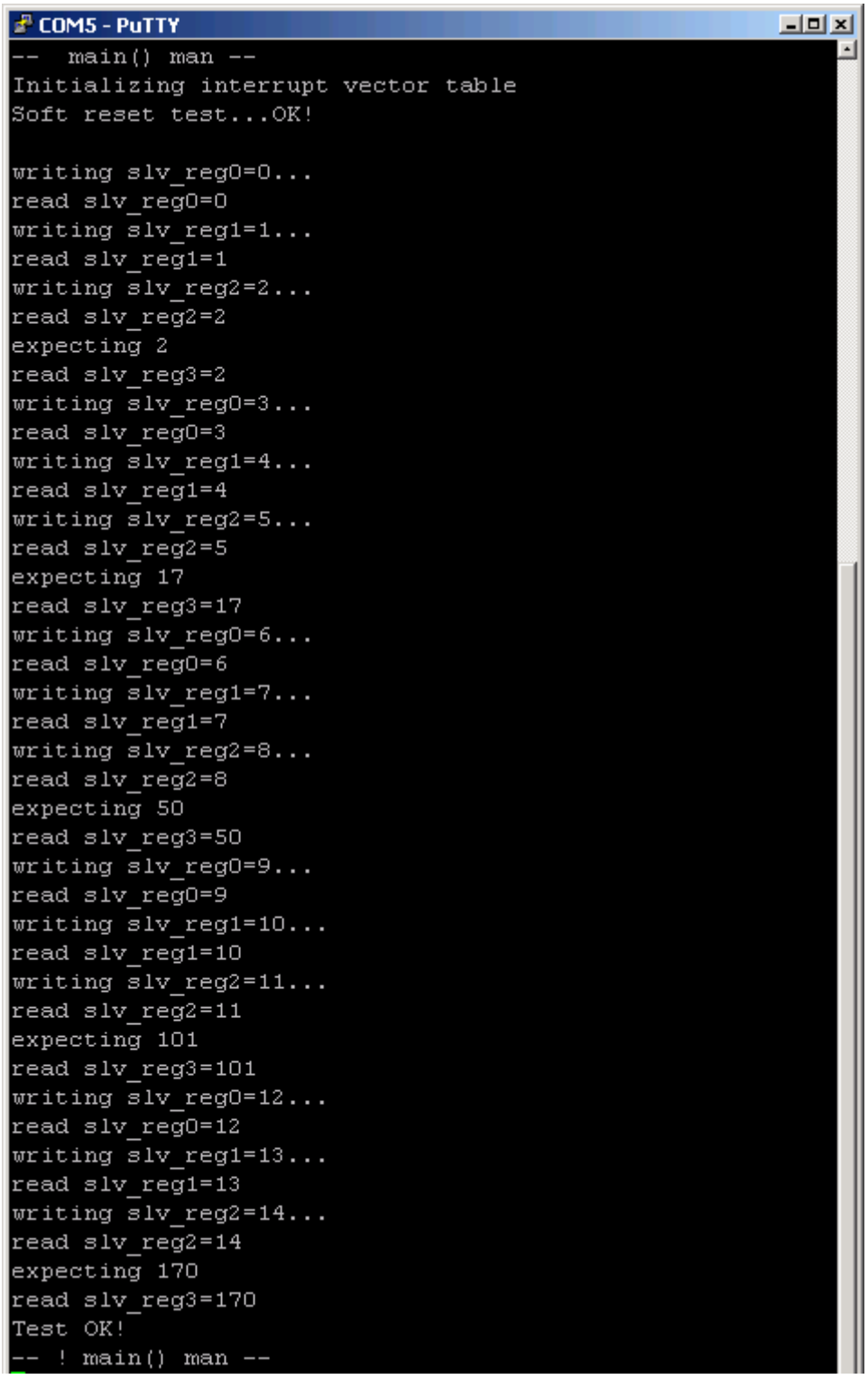
16. And create an interrupt handler which just sets the flag:

```
11  void MADD_Intr_Handler(void * baseaddr_p)
12  {
13     Xuint32 baseaddr;
14     Xuint32 IpStatus;
15     Xuint32 Reg32Value=0xdeadbeef;
16
17     XASSERT_NONVOID(baseaddr_p != XNULL);
18     baseaddr = (Xuint32) baseaddr_p;
19
20     //should be disabling interrupts in a real example
21     IpStatus = MADD_mReadReg(baseaddr, MADD_INTR_IPISR_OFFSET);
22     intr_flag=1;
23     MADD_mWriteReg(baseaddr, MADD_INTR_IPISR_OFFSET, IpStatus);
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;
35     xil_printf("Soft reset test...");
36     MADD_mReset(baseaddr);
37     xil_printf("OK!\n\n\r");
38
39     MADD_EnableInterrupt(baseaddr_p);
40     XExc_mEnableExceptions(XEXC_NON_CRITICAL);
41
42     for(i=0;i<15;i+=3) { //loop a bit
43        intr_flag=0;
44
45        xil_printf("writing slv_reg0=%d...\n\r",i);
46        MADD_mWriteSlaveReg0(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);
56        MADD_mWriteSlaveReg2(baseaddr, 0, i+2);
57        Reg32Value = MADD_mReadSlaveReg2(baseaddr, 0);
58        xil_printf("read slv_reg2=%d \n\r", Reg32Value);
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:



That's it!