HUNT ENGINEERING
Chestnut Court, Burton Row,
Brent Knoll, Somerset, TA9 4BP, UK
Tel: (+44) (0)1278 760188,
Fax: (+44) (0)1278 760199,
Email: sales@hunteng.co.uk
http://www.hunteng.co.uk
http://www.hunt-dsp.com

Getting Started with the Embedded PowerPC – PowerPC Example A

Rev 1.0 J. Thie 13-10-04

Rev 2.0 K.Chircop 10-04-06

Introduction

HUNT ENGINEERING modules such as the HERON-FPGA9 and HERON-FPGA12 feature the Virtex II Pro and Virtex 4 devices from Xilinx. These devices do not consist only of an FPGA but also of one or more embedded PowerPC processors.

The embedded PowerPC processor in the Virtex II Pro/Virtex 4 device is actual silicon embedded within the FPGA. On its own, the embedded PowerPC can't do much. It needs a bus and memory attached to it so that it can run programs. Such resources have to be created out of FPGA gates and connected to the embedded PowerPC processor.

Similarly, the embedded PowerPC can access a variety of other resources, such as UARTs, timers, JTAG interface, and external memory (and external memory controllers). The resources (or the access to them) are created out of FPGA gates, and you will have to write the VHDL to implement the resources to be used by the embedded PowerPC.

With having an embedded PowerPC so tightly integrated with the FPGA, you can also think of adding less traditional resources, such as an FFT engine, filters, or convolution, to the embedded PowerPC. You can do anything you like really, but you will have to write the VHDL to create the resource, or to write VHDL to implement access to off-chip resources (as in the case of external memory).

EDK

To assist you in creating a basic hardware design using the embedded PowerPC, Xilinx supplies the EDK (Embedded Development Kit). The most important constituent of the EDK is XPS (Xilinx Platform Studio). XPS is a Windows/Linux program with which you can graphically add or remove hardware resources, build the hardware, and do the software development.

With XPS you can write and build embedded PowerPC C code, and create/maintain PowerPC software projects. Integrated within XPS are debug tools for the embedded PowerPC, called XMD and Software Debugger. The Software Debugger is a version of GDB (the GNU debugger), used for C code level debugging. XMD is a low-level debugger that is used as an intermediary by the Software Debugger.

Tool Flow

With XPS you can build only a basic hardware design. Typically one or two buses are added to the embedded Power PC, some BlockRAM (to run the embedded PowerPC program out of), a clock module (to clock the hardware accessed by the embedded PowerPC), a system reset (to enable resetting the embedded PowerPC and the hardware it accesses) and a JTAG module (used for debugging embedded PowerPC programs).

Often you also want to use hardware resources that are not available in XPS. One example of such

hardware resources is the HERON/HEART and module resources, such as FIFOs, HSB, and the LEDs. Another example is CoreGen FFTs, filters, and convolutions. In almost all designs you will want access to resources that are not available via XPS.

XPS allows you to export a project to ISE's Project Navigator. In the exporting process, the Project Navigator design is changed so that it interfaces with the embedded PowerPC hardware design. Within Project Navigator you can access hardware resources such as the HIL (Hardware Interface Layer, which you use to access HERON/HEART and module features), CoreGen blocks, and so on.

Once you have completed the Project Navigator design, and built it successfully, import it back into XPS. In the importing process, hardware details of the Project Navigator design are added to the XPS project.

The final step is to add the embedded PowerPC software executable to the bit-stream. This step initialises the BlockRAM with code and data of the executable. Upon loading the bit-stream, the embedded Power PC will then start executing code from the BlockRAM. When you are still in a development stage, however, you can also use XMD and the Software Debugger to initialise the BlockRAM (with an embedded PowerPC executable program).

Using PowerPC Example A

On the HUNT CD you will find a directory for each HERON-FPGA and -IO module in the 'cd:\fpga' directory. Each directory contains the HIL (Hardware Interface Library) and examples for that module. For HERON modules with an embedded PowerPC you will find a sub-directory 'PPC_Ex_A' which contains an example of how you could make use of the embedded PowerPC processor that is within the Virtex II Pro/Virtex 4 device.

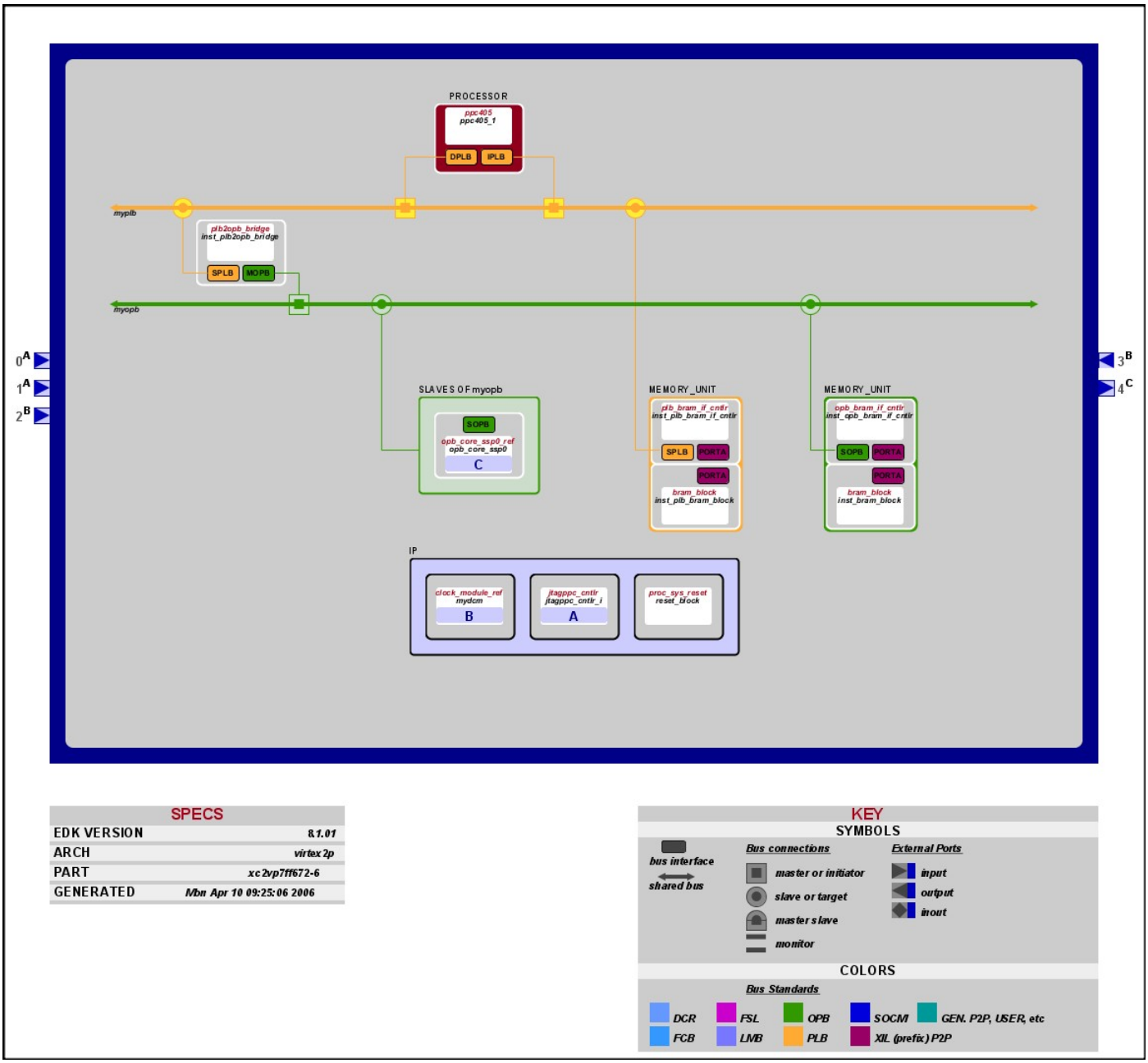
The 'PPC_Ex_A' directory contains an example project for XPS (Xilinx Platform Studio), which is used as a starting point to create a bit-stream. In this example, our aim is to get an embedded PowerPC program up and running, and to flash some LEDs that are on the Virtex II Pro/Virtex 4 HERON module. We will use some BlockRAM that can be accessed by the embedded PowerPC. The BlockRAM is used to store and execute the embedded PowerPC program.

First, copy the whole of the module's example directory from the HUNT ENGINEERING CD to your hard disk. For example, for a HERON-FPGA12 you would copy 'cd:\fpga\fpga12v1'. All files will have the 'read-only' attribute set because you copy from a CD. Change all file attributes in the directory you just copied (except for the files in the 'Common' sub-directory). Alternatively, use the zip file in the module directory and unzip into a directory on your hard disk. The zip file will contain the whole module directory, but all file-attributes will be set properly.

When you have a proper copy on your hard disk, open the example project with XPS (Start → Programs → Xilinx Platform Studio x.x → Xilinx Platform Studio, where x.x is the version number, 8.1i in my case). Open the example project, 'system.xmp', in the 'PPC_Ex_A' sub-directory of the directory you copied to hard disk.

Block Diagram (memory)

Let's first have a look at what components we have in the example project, and how they are connected up to the embedded PowerPC Processor. You can get a Block Diagram by selecting 'Project → Generate and View Block Diagram'.



First, notice two sets of BlockRAM memory. One piece of memory is attached to the OPB (On-chip Processor local Bus), and the other piece of memory to the PLB (the IBM Processor Local Bus). Please refer to Xilinx EDK documentation for more information on the OPB and PLB. For this discussion it is sufficient to know that PLB and OPB are buses, each with their own characteristics, from the Xilinx core library. The address bus width of the PLB is 32 bits; the data bus width is 64 bits. The address bus width of the OPB is also 32 bits and the data bus width is 32 bits. The PLB is faster than the OPB, but consumes more FPGA resources. Typically you use the PLB for fast resources.

The two pieces of memory are the two 'bram_block' components. Two controllers control access from the OPB to its BlockRAM ('opb_bram_if_cntlr') and PLB to its BlockRAM ('plb_bram_if_cntlr'). For readers who are not hardware engineers, you can view the controllers as an interface between the bus and the BlockRAM.

The properties most interesting about the BlockRAM components is at what (bus) address they are and what size they are. If you select select 'Project → Generate and View Design Report' and clicking on 'Memory Controllers → inst_plb_bram_if_cntrl', the properties of the PLB BlockRAM controller will appear.

General	
IP Core	plb_bram_if_cntlr
Version	1.00.a
Driver	API

Parameters

These are parameters set for this module. Refer to the IP documentation for complete information about module parameters.
Parameters marked with yellow indicate parameters set by the user.
Parameters marked with blue indicate parameters set by the system.

Name	Value
c_baseaddr	0xFFFFC000
c_highaddr	0xFFFFFFFF
c_include_burst_cacheln_support	0
c_num_masters	2
c_plb_awidth	32
c_plb_clk_period_ps	20000
c_plb_dwidth	64
c_plb_mid_width	1

Post Synthesis Device Utilization

Resource Type	Used	Available	Percent
Slices	193	4928	3
Slice Flip Flops	293	9856	2
4 input LUTs	185	9856	1

We can see that the PLB BlockRAM is addressed from 0xFFFFC0000 until 0xFFFFFFFF, a 16 Kb size of memory.

Clicking on the BlockRAM component itself 'Memory → inst_bram_block', you can see the block's ports' address width ('C_PORT_AWIDTH') and data width ('C_PORT_DWITH').

General			
IP Core	bram_block		
Version	1.00.a		
Parameters			
These are parameters set for this module. Refer to the IP documentation for complete information about module parameters. Parameters marked with yellow indicate parameters set by the user. Parameters marked with blue indicate parameters set by the system.			
Name	Value		
C_FAMILY	virtex2p		
C_MEMSIZE	0x8000		
C_NUM_WE	4		
C_PORT_AWIDTH	32		
C_PORT_DWIDTH	32		
Post Synthesis Device Utilization			
Resource Type	Used	Available	Percent
BRAMs	16	44	36

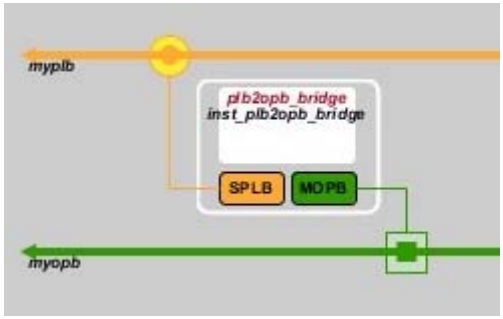
It is absolutely essential that a block of memory is present at the upper end of the embedded PowerPC's 4 Gb address space. Upon reset, the embedded PowerPC will jump to address 0xFFFFFFFC. Here it will read one 32-bit instruction, typically a jump to an adjacent address. The jump can only be made to an adjacent address because the address is encoded in the instruction, and they can only jump to a sub-section of the 4Gb embedded PowerPC address space. The Xilinx EDK tools by default implement a jump to an address not far from 0xFFFFFFFC, say 0xFFFFFFC0.

At address 0xFFFFFFC0 there are 4 instructions. A target address is constructed, the address can be anywhere in the embedded PowerPC's 4 Gb address range. The target address is stored in a 32-bit register. Next, a jump is made to whatever address the register points to. This way any address in the 4 Gb address space can be reached. The target address is usually the starting point of your program ('_crt0').

It is the Xilinx tools which, when creating an executable, insert those instructions at addresses 0xFFFFFFFC and 0xFFFFFFC0. When you use the Xilinx kernel, the procedure is the same, but the second jump will be made to the start of the Xilinx kernel rather than your program.

Because the embedded PowerPC always starts reading instructions at address 0xFFFFFFFC, you must always have some memory present at this address (and later on, when creating programs, ensure that there's a jump instruction there, but we'll discuss that later). In this example, we have 16 Kb, and this covers the 0xFFFFFFFC address, and also the 0xFFFFFFC0 address to which the Xilinx tools will have the program jump to.

Block Diagram (plb to opb bridge)



The OPB to PLB bridge is used to map a range of PLB addresses onto OPB addresses. If in the 'System Design Report' you click 'Bridges → inst_plb2opb_bridge', you can see that in the example we map the bottom 2 Gb of PLB addresses onto the OPB.

General	
IP Core	plb2opb_bridge
Version	1.01.a
Driver	API

Parameters

These are parameters set for this module. Refer to the IP documentation for complete information about module parameters.
Parameters marked with yellow indicate parameters set by the user.
Parameters marked with blue indicate parameters set by the system.

Name	Value	Name	Value
C_BGI_TRANSABORT_CNT	31	C_OPB_DWIDTH	32
C_CLK_ASYNC	0	C_PLB_AWIDTH	32
C_DCR_AWIDTH	10	C_PLB_DWIDTH	64
C_DCR_BASEADDR	0b1111111111	C_PLB_MID_WIDTH	1
C_DCR_DWIDTH	32	C_PLB_NUM_MASTERS	2
C_DCR_HIGHADDR	0b0000000000	C_RNG0_BASEADDR	0x00000000
C_DCR_INTFCE	1	C_RNG0_HIGHADDR	0x7FFFFFFF
C_FAMILY	virtex2p	C_RNG1_BASEADDR	0xFFFFFFFF
C_HIGH_SPEED	1	C_RNG1_HIGHADDR	0x00000000
C_INCLUDE_BGI_TRANSABORT	1	C_RNG2_BASEADDR	0xFFFFFFFF
C_IRQ_ACTIVE	1	C_RNG2_HIGHADDR	0x00000000
C_NO_PLB_BURST	0	C_RNG3_BASEADDR	0xFFFFFFFF
C_NUM_ADDR_RNG	1	C_RNG3_HIGHADDR	0x00000000
C_OPB_AWIDTH	32		

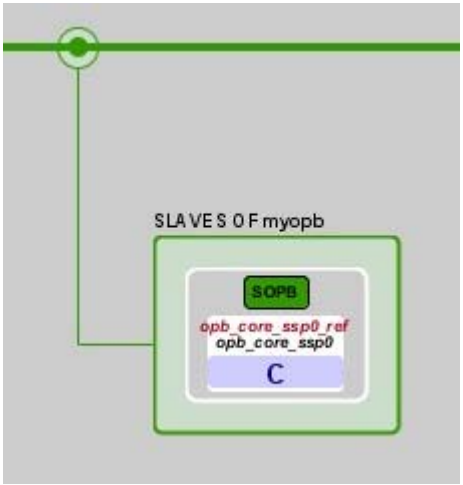
Post Synthesis Device Utilization

Resource Type	Used	Available	Percent
Slices	590	4928	11
Slice Flip Flops	742	9856	7
4 input LUTs	753	9856	7

In itself, it is not absolutely necessary to have an OPB in the project, we can get the embedded PowerPC up and running without it. However, we want to access the LEDs, and the component that is capable of accessing them is an OPB component. So for our example we do need the OPB.

Block Diagram (LEDs)

To access the LEDs, we add an 'interface' that maps some addresses to access the LEDs. We cannot access the LEDs just using the EDK. The LEDs are a feature of the HERON module, and there is no standard Xilinx core component that will access the LEDs for us. Therefore, we place just the 'interface', and in a later stage will connect the loose 'wires' of the 'interface' to the LEDs. The 'interface' is the Xilinx OPB Core Slave Service Package 0 ('opb_core_ssp0_ref') component.



If you look at the component's properties, you can see that it is mapped onto OPB address range 0x30000000 to 0x300000FF. If you would like to know more about the component, in 'System Assembly View', if you double-click 'opb_core_ssp0', a window will open with its properties, if you click on 'Datasheet' the component's datasheet will be opened. In particular, notice that the core implements 4

registers at addresses 0x30000000, 0x30000004, 0x30000008, and 0x3000000C. Each of these addresses accesses just 1 LED, but each 8-byte value stored at an address controls the brightness of its LED! This should make you understand the LEDs behaviour once the example bit-stream is up-and-running.

General	
IP Core	opb_core_ssp0_ref
Version	1.00.c
Driver	API

Parameters	
These are parameters set for this module. Refer to the IP documentation for complete information about module parameters. Parameters marked with yellow indicate parameters set by the user. Parameters marked with blue indicate parameters set by the system.	
Name	**Value**
c_baseaddr	0x30000000
c_family	virtex2p
c_highaddr	0x300000FF
c_mir_baseaddr	0x30000100
c_mir_highaddr	0x300001FF
c_opb_awidth	32
c_opb_dwidth	32
c_user_id_code	10

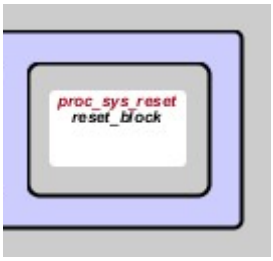
Post Synthesis Device Utilization			
Resource Type	**Used**	**Available**	**Percent**
Slices	128	4928	2
Slice Flip Flops	117	9856	1
4 input LUTs	73	9856	0

Block Diagram (JTAG Controller)

The JTAG controller is absolutely essential. Having the JTAG controller means that we can connect a Xilinx Parallel Cable III/IV or Xilinx Platform Cable USB, and, after the bit-stream has been programmed into the FPGA, we can access the embedded PowerPC and BlockRAM via the JTAG. This allows utilities such as the XMD and GDB debuggers to run.

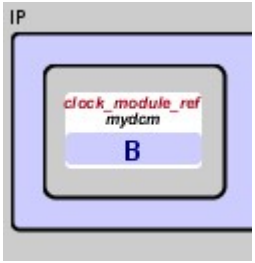
Block Diagram (reset)

Most likely you will want to be able to reset the embedded PowerPC processor, and also hardware components related to the embedded PowerPC, such as PLB and OPB, BlockRAM controllers and the JTAG controller. Xilinx has a core block component that does just that: Processor System Reset Module, instantiated as 'proc_sys_reset'.



Block Diagram (clock)

Finally, we will also need a clock that drives the embedded PowerPC and all the components that we have created in the design as shown in the Block Diagram. Xilinx has a core block component that does just that: Clock Module Reference Core, instantiated as 'clock_module_ref'.



Building the Example Project

Before we start building anything, we must verify that we have all components. The necessary components are there: the system clock, system reset and JTAG controller. We have hooked up 16 Kb BlockRAM at the top of the embedded PowerPC 4 Gb address space so that we have memory at address 0xFFFFFFFC and near addresses. We have added an OPB and PLB-to-OPB bridge, and added a component that will interface to the LEDs. The LEDs are not used yet; we will have to wire the LEDs up later. We have verified that the PLB-to-OPB bridge maps a sensible range of addresses. There's an additional 32 Kb BlockRAM on the OPB.

To build the example project, select the menu item 'Hardware → Generate Netlist'. If you look in the 'PPC_Ex_A' directory, you will see that a lot of files and several new directories have been created.

Build Results

Amongst the directories and files that have been created by the 'Generate Netlist' step are 'implementation', 'hdl', and 'synthesis'. The first thing that 'Generate Netlist' does is to run the 'platgen' utility. It is 'platgen' that has created the 'implementation', 'hdl', and 'synthesis' directories.

The 'platgen' utility operates on the 'system.mhs' file (mhs = microprocessor hardware specification), which is a textual description of the hardware design created with XPS. The 'system.mhs' file is maintained by XPS and is changed when you add, delete, or edit a component.

The 'hdl' directory contains the VHDL wrappers for the individual IP components in the embedded

PowerPC system. It also contains top-level VHDL for the embedded PowerPC system ('system.vhd') and its instantiation ('system_stub.vhd').

The 'implementation' directory contains implementation netlists of the peripherals ('*_wrapper.ngc'), and a netlist for the system design ('system.ngc'). Note also the 'system.bmm' and 'system_stub.bmm' files. The bmm files (bmm = BlockRAM Memory Map) describe how individual BlockRAMs used to create a continuous address space. Later on we will refer again to these two *.bmm files.

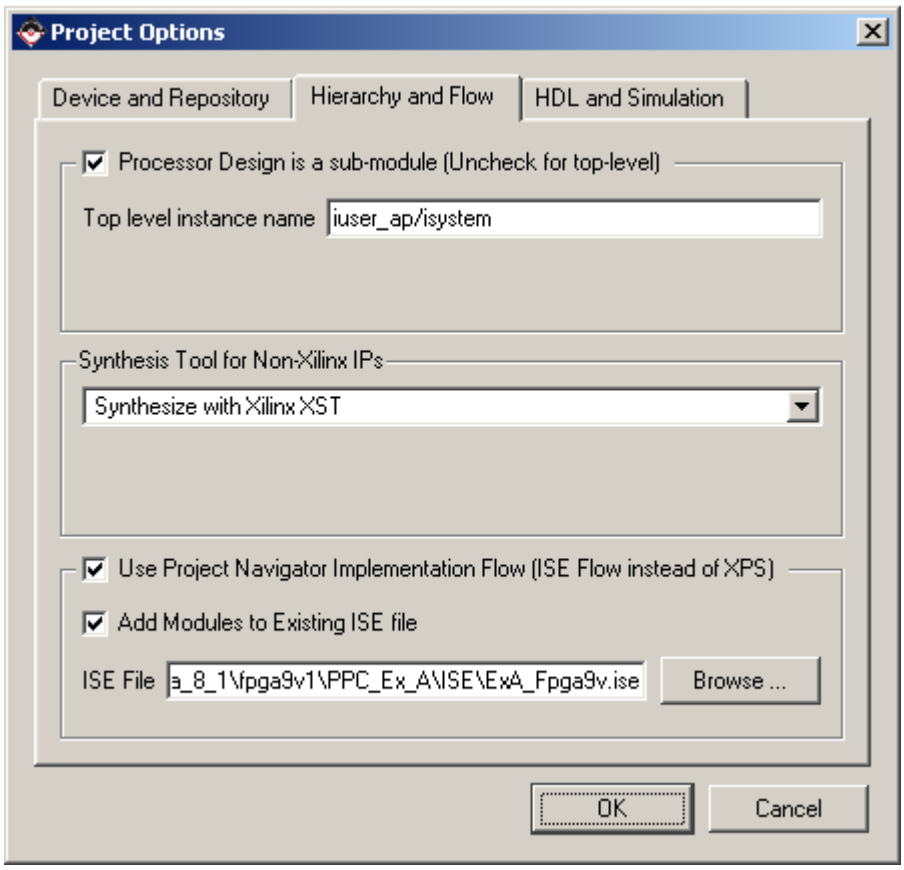
The 'synthesis' directory contains the synthesis project files for all individual IP components as well as the synthesis project file for the overall embedded PowerPC system.

Exporting to Project Navigator

To hook up the LEDs into the project, we need to export the XPS system design into Project Navigator. However, XPS assumes that there's an existing Project Navigator project that it can export to. In the case of this embedded PowerPC example, we simply use the 'example1' project of the same module for this purpose. We have copied the 'ISE' and 'Src' sub-directories of the 'Example1' sub-directory into the 'PPC_Ex_A' sub-directory.

Example1 is a Project Navigator example that only uses the FPGA, that reads data from a FIFO, and then writes the same data back onto another FIFO. We will not use any functionality of example1; we just use example1 to create a starting project so that we have a project we can export to. Instead of example1, we could just as well have used a different Project Navigator example, such as 'Memory_Test(ex2)' or 'Sdram_Fifo(ex3)' (in the case of a HERON-FPGA9 or HERON-FPGA12). In itself, we can export to any Project Navigator project, but as we want to use a module feature (the LEDs) it is useful to use a Project Navigator project that already incorporates and uses the HIL (the Hardware Interface Layer, whose VHDL files are in the 'Common' directory).

First we have to tell XPS that we want to export, and then we have to specify to what Project Navigator project the system information should be exported. Select 'Project → Project Options', then click the 'Hierarchy and Flow' tab, and make sure that 'Use Project Navigator Implementation Flow' is checked. Also make sure that 'Add modules to existing ISE file' is ticked, and that the Project Navigator '.ise' file in the 'PPC_Ex_A\ISE' sub-directory is selected. The project in the 'PPC_Ex_A\ISE' sub-directory is called 'ExA_Fpga9v.ise'. (In example1 the Project Navigator project was called 'Ex1_Fpga9v.ise', but we renamed the copied file in 'PPC_Ex_A\ISE' to make clear we use a local copy of example1 and not example1 itself.)



Note also that we have denoted the Project Navigator project as the top level, and that the XPS design is set to be a sub-module. The top of the whole design must be the file 'top.vhd' in the Project Navigator (ISE) project, because this file handles important I/O interfacing issues. The top-level design created for the PowerPC in XPS must be placed as a sheet in User_Ap ('User_ap1.vhd' in this example) of the Project Navigator (ISE) project.

There are key components in the Hardware Interface Layer (HIL) that take care of interfacing to HERON FIFOs and DDR memory. These components must be what is used when accessing those particular hardware resources.

Other functional elements, such as Flash Memory interfaces and UARTs, can be placed as blocks of peripheral IP in the embedded PowerPC design. The appropriate hardware interface signals of those components must then be connected to ports in the User_Ap ('User_ap1.vhd' in this example) entity.

The next task is to export the embedded PowerPC design created in XPS. Unfortunately, the XPS 'Tools → Export to ProjNav' does not work properly. Therefore HUNT ENGINEERING has created a batch file that will correctly export the XPS project, called 'Export_to_ProjNav.bat'.

Export to ProjNav.bat Batchfile

Before running the batch file, let's first have a look at it. The 'Export_to_ProjNav.bat' batch file is located in the 'PPC_Ex_A' directory. The batch file's contents are:

```
copy implementation\system_stub.bmm ISE\system.bmm
pjcli -v -f hunt_npl_cmdfile
```

The 'system_stub.bmm' was generated in the previous step by the 'platgen' tool, when we did the 'Tools → Generate Netlist'. A .bmm file is a text file that is used to track where each separate piece of BlockRAM is placed within the FPGA. We use 'system_stub.bmm' and not 'system.bmm' because we use the embedded PowerPC project as a sub-module, as you can see in the 'Project Options' box

above. We use the top level of the Project Navigator project.

The 'pjcli' tool will do the actual 'export'. It changes the Project Navigator project by adding some files to it, adding some directories, as specified by the 'hunt_npl_cmdfile' file. Let's have a look the file's contents:

```
OpenProject(ISE\ExA_Fpga9v.ise)
AddSource(..\hdl\system.vhd, VHDL Design File)
SetProperty( Macro Search Path,..\implementation\,..\hdl\system.vhd,
Implement Design)
SetPreference(UserLevel, Advance)
SetProperty( Hierarchy Separator,/,..\hdl\system.vhd,Synthesize - XST)
AddSource(system.bmm, top)
CloseProject()
```

First we can see how the Project Navigator project is opened ('OpenProject(ISE\ExA_Fpga9v.ise)'). Note how the project name is hard-coded here, so if you want to use this batch file for other embedded PowerPC projects, you may have to edit the 'hunt_npl_cmdfile' file.

Next ('AddSource(..\hdl\system.vhd, VHDL Design File)'), we see how the system level VHDL file of the XPS embedded PowerPC project is added to the Project Navigator project.

In the next step ('SetProperty(…)'), we see how the Project Navigator project is made aware of the netlists in the XPS embedded PowerPC's project (the netlists are of the individual components and of the system level design).

Finally, the 'system.bmm' file is added to the Project Navigator project. This was the 'system_stub.bmm' file that we copied out of the XPS embedded PowerPC project before we ran 'pjcli'.

If you were to run 'Project → Export to ProjNav' (don't, because it won't work), you would find that XPS creates a 'npl_cmdfile' of its own, and then runs 'pjcli' on that file. Our batch file does a similar thing, but with the EDK that we have (7.1) 'Tools → Export to ProjNav' doesn't seem to perform the correct steps.

Running the Batch file

With the Project Navigator project in place, and with the 'Export_to_ProjNav.bat' batch file using the correct Project navigator project name, we are now ready to run the batch file.

Open a DOS-box and go to the 'PPC_Ex_A' directory. Then run 'Export_to_ProjNav.bat'.

Project Navigator

When the batch file has completed, start Project Navigator and open the ISE project in the 'PPC_Ex_A\ISE' directory. What we need to do is 'hook up' the Xilinx interface core component we use to access the LEDs with the actual LEDs of the HERON module.

In Project Navigator, open 'system.vhd'. This file is actually in the '..\hdl' directory, the export process should have added this file to the Project Navigator project. Here you should see how a 'system' entity is defined. The 'system' entity defines the interface between our embedded PowerPC XPS system and the Project Navigator design.

```
entity system is
  port (
    sys_rst_n : in  std_logic;
    sys_clk   : in  std_logic;
    GPIO      : out std_logic_vector(0 to 3);
    JTAG_TRST : in  std_logic;
    JTAG_HALT : in  std_logic
  );
end system;
```

In the rest of the 'system.vhd' file you will no doubt recognize the components that we defined in the XPS embedded PowerPC project. You can also view how XPS has generated VHDL to connect up all those components.

Next, open 'User_Ap1.vhd'. You will find a 'component system' declared, matching the entity in 'system.vhd'. In this example this has been prepared for you, but usually you would copy and paste the entity definition out of 'system.vhd' into 'User_Ap1.vhd', and change the 'entity' into a 'component'.

The 'system' component is instantiated towards the bottom of 'User_Ap1.vhd'. Again, this is done for you in this example, but usually you would instantiate a 'system' component yourself.

```
isystem : system
  port map (
    sys_rst_n => PPC_RESET,
    sys_clk   => OSC3,
    GPIO      => PPC_LEDS,
    JTAG_TRST => VCC,
    JTAG_HALT => VCC );
```

Again, we have already added the necessary signals to the VHDL: PPC_RESET, PPC_LEDS and VCC. OSC3 was already part of the original 'example1' VHDL. PPC_RESET is a local signal, the inverse of RESET, and relates to the reset component we added to the embedded PowerPC design in XDS. OSC3 is the local oscillator, and relates to the clock component we added to the embedded PowerPC design in XDS. The VCC signal is simply wired to '1', as you would expect. Finally, PPC_LEDS is a 4-wire signal.

So, compared to the 'Example1' project, we have added a 'system' component and its instantiation, and 3 signals PPC_RESET, VCC, and PPC_LEDS. Now we are finally ready to connect the LEDs to the embedded PowerPC LED interface. On line 642 in the User_Ap1.vhd file this happens:

```
-- Connect LEDs to the GPIO outputs from the PowerPC design
LED(3 downto 0) <= PPC_LEDS(3) & PPC_LEDS(2) & PPC_LEDS(1) & PPC_LEDS(0);
```

Now we are ready to build the design. Make sure that 'top-rtl' is selected (in the 'Sources in Project' window), then right-click 'Generate Programming File' (in the 'Processes for Source' window) and select 'Run'.

At this point you have a working bit-stream, and you could load it onto your device. But you won't see the LEDs flash (yet). That is because the BlockRAMs haven't been initialized (yet), and the embedded PowerPC would be executing whatever data happened to be in the BlockRAMs. Using XMD and the software debugger you could at this point load an embedded PowerPC executable onto the device (how to do this is explained later in this document), using the Xilinx Parallel/USB Cable and XPS. Loading the embedded PowerPC with an executable (via XMD/Software Debugger) will store the executable's code and data in the BlockRAMs. If you then 'run' the code, in the Software Debugger, you would see the LEDs flash. But for now, let's continue and complete the process.

Importing back to XPS

We still want to create and add a C program to the project. To work with the embedded PowerPC, we need to get back to XPS, but we need the additions created in the Project Navigator project. To get those, we import the Project Navigator project back into our XPS project.

Unfortunately 'Project → Import from ProjNav' does not seem to work and therefore we need to import the required files manually. Go to the 'ISE' directory and copy the files 'top.bit' and 'system_bd.bmm'. Now go to the 'implementation' directory and paste the files. Finally right-click on 'top.bit' and click rename, change the name to 'system.bit'.

As you can see not much work was required. Note that in the setup of this example we would always use Project Navigator to create a bit-stream.

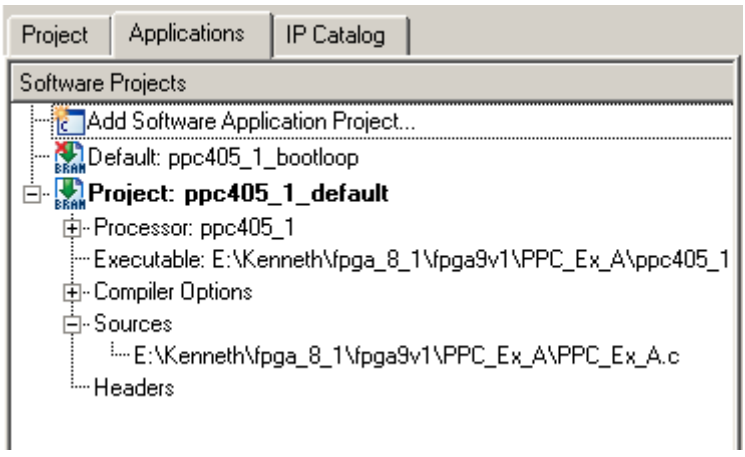
Also notice the 'system_bd.bmm' file, and how it differs from the original 'system_stub.bmm' that we exported earlier out of the XPS embedded PowerPC project. If you compare them, you will see that the BlockRAMs have been assigned to actual resources ('PLACED = …' encoding at the end of each line) in the 'system_bd.bmm' file.

Building the embedded PowerPC Program

We have already prepared a working example C program for the embedded PowerPC. At this point we just want to build the software project, and use it to initialize the BlockRAMs, so that we can create a bit-stream where the embedded PowerPC can run proper code out of the BlockRAMs from the moment the bit-stream is loaded. We will look later in more detail how to set up a software project, how to use the debugger, and so on.

The first step in building the embedded PowerPC program is to create header files that describe the embedded PowerPC system that we have created. This is simply done by 'Software → Generate Libraries and BSPs' within XPS. If you look in the 'PPC_Ex_A\ppc405_1' you will see that header files describing the system are created into the 'inc' directory, and libraries in the 'lib' directory. In particular, review the 'xparameters.h' file, where you will recognize the addresses of the LED and BlockRAM components that we defined in our XPS embedded PowerPC hardware design.

Let's have a look at how the software project is setup in this example. Click on the 'Application' tab in the left window in XPS. You will find here an embedded PowerPC software project with an example C file ('PPC_Ex_A.c') already in the project. This is for this example only; usually you would create and add a project yourself, then create and add C source files to the software project.



Let's have a quick look at the C code of 'PPC_Ex_A.c'. First, you will find pointers that are initialized to where the 4 LEDs are:

```
volatile int *led0 = (int *)0x30000000;
volatile int *led1 = (int *)0x30000004;
volatile int *led2 = (int *)0x30000008;
volatile int *led3 = (int *)0x3000000C;
```

You will recognize the address (0x30000000) from the LED component's ('opb_core_ssp0_ref') properties in the XPS embedded PowerPC hardware design (page 7 of this document). As you may recall from this component's PDF file, it implements a number of registers, 4 of which address an LED each. The value written to an LED relates to the brightness that the LED will glow with.

So, to switch an LED off, you write 0xFF (255) to a LED's address, to switch it fully on, you write 0 to an LED's address. Any other value represents brightness in between off and fully on.

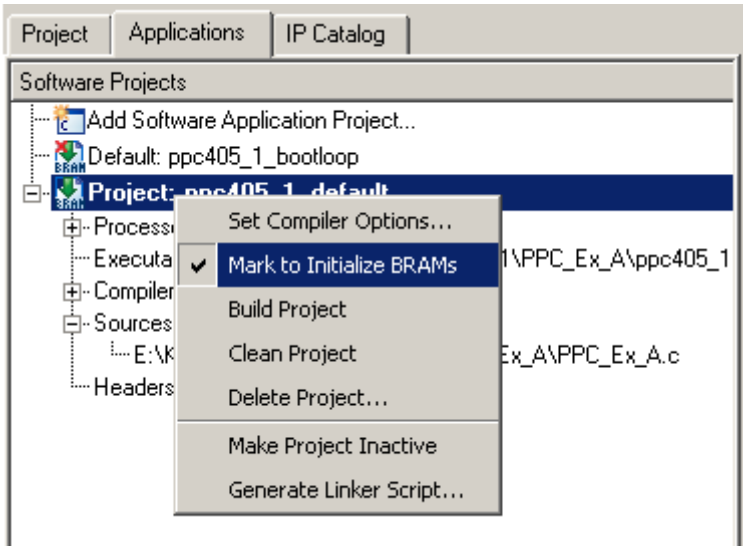
To create an embedded PowerPC ELF file, do a 'Software → Build All User Applications'. The embedded PowerPC executable is called 'executable.elf' and is deposited into the 'PPC_Ex_A\ppc405_1\code' directory.

```
powerpc-eabi-size ppc405_1/code/executable.elf
   text    data     bss     dec     hex filename
   1170     836    8220   10226    27f2 ppc405_1/code/executable.elf

GNU ld version 2.15

Done!
```

Adding the ELF Executable to the Bitstream

Now that we have created an executable (embedded PowerPC ELF format), we want to initialize the BlockRAMs with it. In the embedded PowerPC system design there was a piece of memory attached to the PLB, at address 0xFFFFC000. In the bit-stream we have now ('system.bit') this memory (BlockRAMs) is as yet un-initialized.

First, make sure that is 'Mark to Initialise BRAMs' is ticked (right click on 'Project:…' as show below).



Now run 'Device Configuration → Update Bitstream' from within XPS. This process will take the bit-stream, 'system.bit' and use this and the embedded PowerPC ELF executable, to create a new bit-stream, 'download.bit'. In this new bit-stream, the BlockRAMs (of the memory at 0xFFFFC000) will be initialized with the PowerPC executable's code and data.

Note how first 'bitinit' is used, and then the 'data2mem' utility, to create the 'download.bit' bit-stream, using the 'system.bit' bit-stream (copied from the Project Navigator directory earlier on), the embedded PowerPC ELF executable, and 'system_bd.bmm' as input files.

```
Analyzing file ppc405_1/code/executable.elf...
Running Data2Mem with the following command:
data2mem -bm implementation/system_bd -bt implementation/system.bit  -bd
ppc405_1/code/executable.elf tag inst_bram_block inst_plb_bram_block  -o b
implementation/download.bit

Memory Initialization completed successfully.


Done!
```

Running the Bitstream

There are two ways to run the bit-stream that we just generated. The first is to use the HUNT ENGINEERING Confidence Checks program. Start this program (Start → Programs → HUNT ENGINEERING → Confidence Checks), then select 'FPGA → Program FPGA'. Click 'Detect' or fill in the 'Board', 'Nu.' and 'Slot' fields yourself. Browse to the 'PPC_Ex_A\implementation' directory and open 'download.bit'. Finally, click 'Program FPGA'. After the programming completes, you should see the LEDs light up, running from left-to-right, then from right-to-left, continuously.

Instead of using the Confidence Checks, you can also use 'iMPACT'. You need to use a Xilinx Parallel Cable IV or Xilinx Platform Cable USB to program the FPGA. There is a separate document that describes how to use 'iMPACT' to program the FPGA with a bit-stream. (On the HUNT ENGINEERING CD: Application Notes → HUNT ENGINEERING papers and apps notes → Using Impact and HERON modules, and read the 'Downloading Bit-streams via JTAG' chapter towards the end of this document). Where you get to 'Assign New Configuration File', select the 'download.bit' bit-stream in the 'PPC_Ex_A\implementation' directory. After programming the FPGA, you should see the LEDs light up, running from left-to-right, then from right-to-left, continuously.

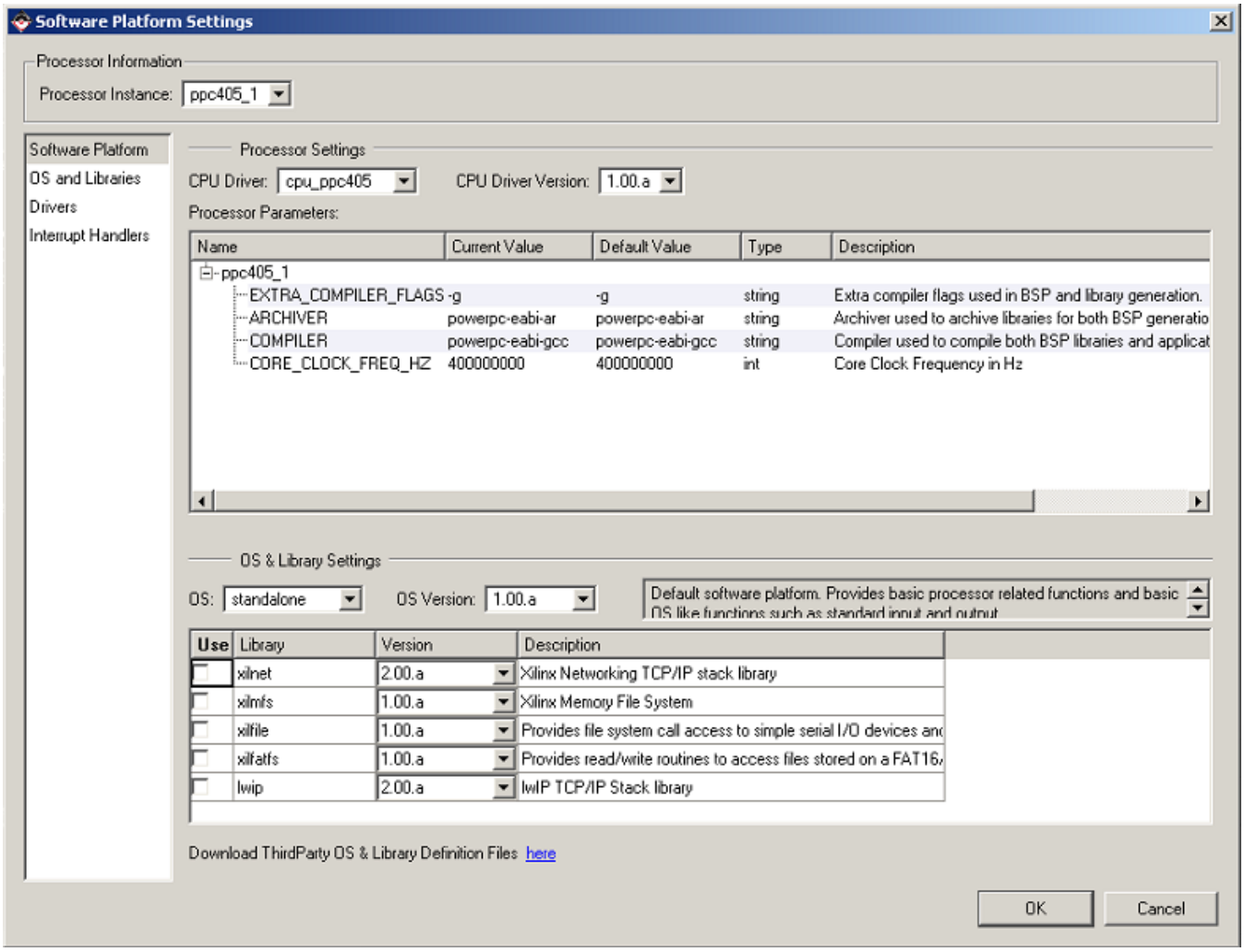
The embedded PowerPC project in more detail

We quickly brushed over the embedded PowerPC project, and built it without looking how the software project was setup. We will now look in detail at the software project within this example XPS project, and show how to use the debugger (XMD and the Software Debugger (GDB)) that come with the EDK.

First we will look at the settings used in the embedded PowerPC software project. Note that several different Kernels and Operating Systems can be used with the embedded PowerPC (with the 8.1 EDK, used when writing this document, these are the Xilinx Kernel, VxWorks, and Linux), but we will use Stand-Alone mode.

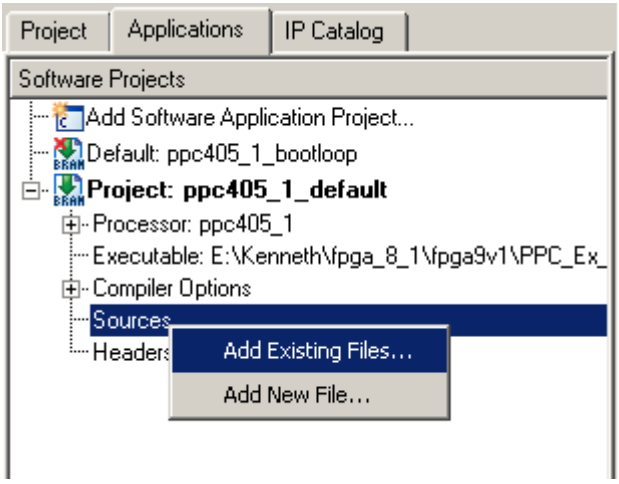
(1) Verify Kernel/Operating System

First, verify that the software environment is configured to be in 'standalone' mode. Click 'Software → Software Platform Settings . . .'. A window will appear from which you can verify the OS setting.



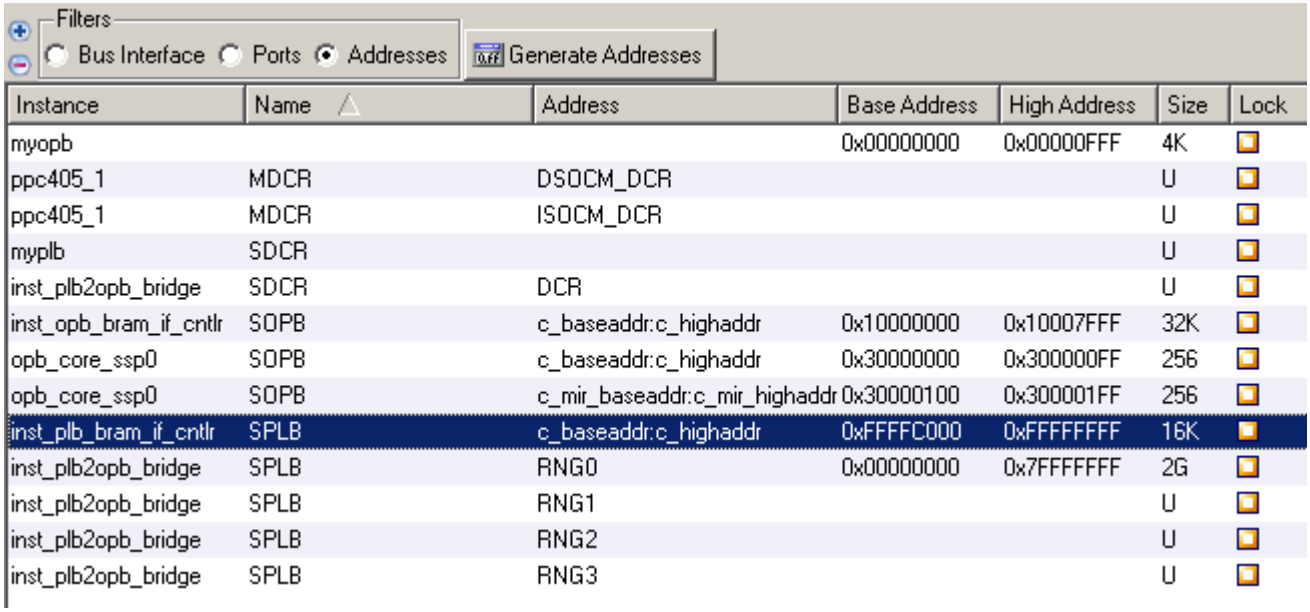
(2) Add Sources

In this example, the 'PPC_Ex_A.c' file has already been added to the project. But if you were to start and create a new project, you would have to add the example C file, 'PPC_Ex_A.c', to the project. To add a C file to your embedded PowerPC project: right-click 'Sources' then click 'Add Existing Files . . .' and select 'PPC_Ex_A.c'.



(3) Verify BlockRAM addresses

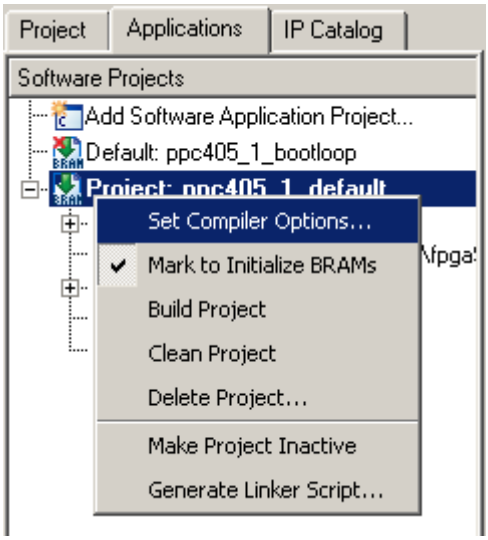
We must make sure that the software project knows at what address memory starts. First we must make sure that we verify in the XPS hardware project what that address is. We have seen earlier how this could be done, but a quick way is to go to the 'System Assembly View', select the filter to be 'Addresses', and find the BlockRAM controller component. The start and end address of the BlockRAM should be displayed. With this example, the start address is 0xFFFFC000, but this may be different if you changed the embedded PowerPC hardware design.



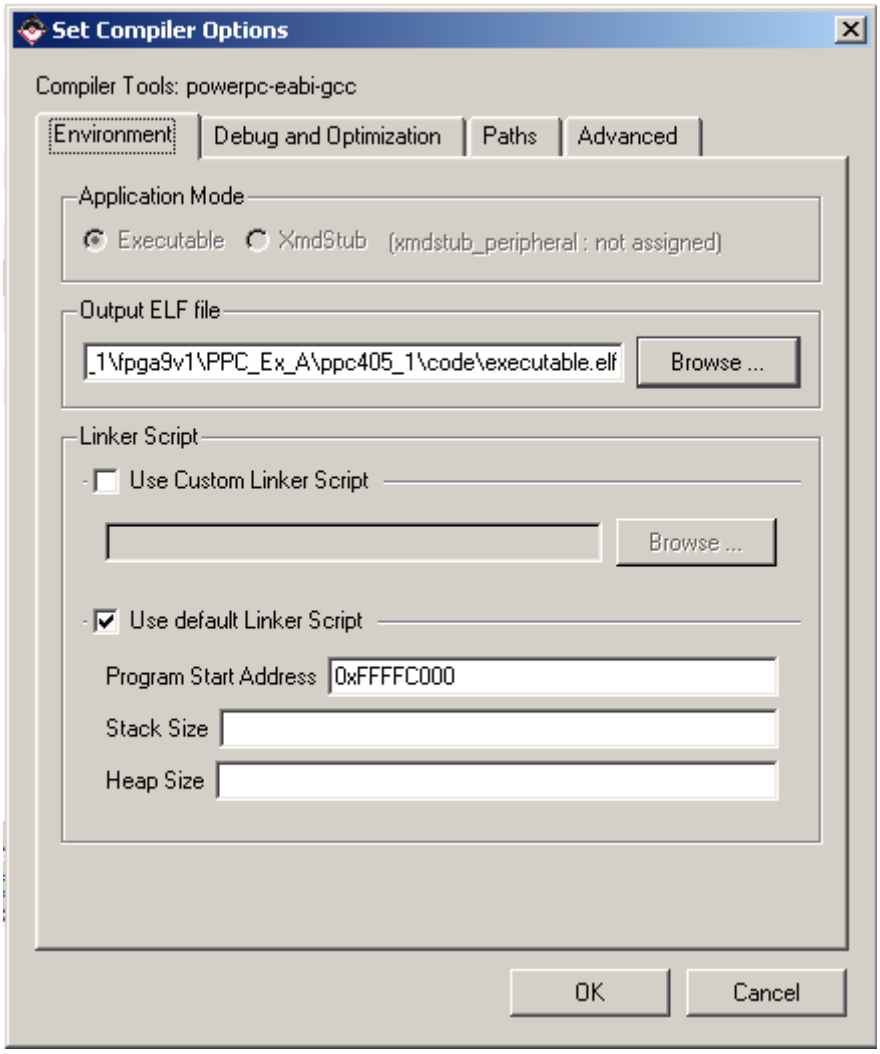
Instance	Name	Address	Base Address	High Address	Size	Lock
myopb			0x00000000	0x00000FFF	4K	
ppc405_1	MDCR	DSOCM_DCR			U	
ppc405_1	MDCR	ISOCM_DCR			U	
myplb	SDCR				U	
inst_plb2opb_bridge	SDCR	DCR			U	
inst_opb_bram_if_cntlr	SOPB	c_baseaddr:c_highaddr	0x10000000	0x10007FFF	32K	
opb_core_ssp0	SOPB	c_baseaddr:c_highaddr	0x30000000	0x300000FF	256	
opb_core_ssp0	SOPB	c_mir_baseaddr:c_mir_highaddr	0x30000100	0x300001FF	256	
inst_plb_bram_if_cntlr	SPLB	c_baseaddr:c_highaddr	0xFFFFC000	0xFFFFFFFF	16K	
inst_plb2opb_bridge	SPLB	RNG0	0x00000000	0x7FFFFFFF	2G	
inst_plb2opb_bridge	SPLB	RNG1			U	
inst_plb2opb_bridge	SPLB	RNG2			U	
inst_plb2opb_bridge	SPLB	RNG3			U	

(4) Edit Compiler Options

Next, verify that your project settings are correct. Right-click 'Project', then select 'Set Compiler Options'.



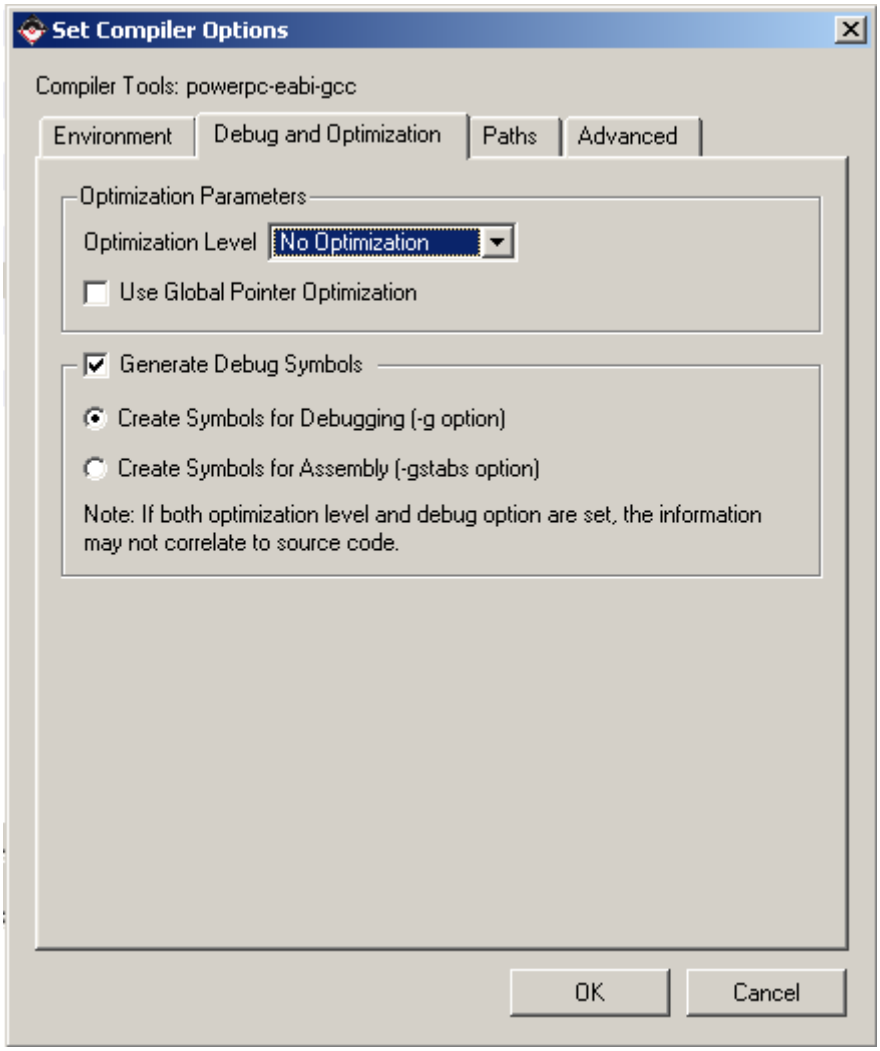
A window will appear, entitled 'Set Compiler Options', where 'ppc405_1_default' is the name of the embedded PowerPC software project.



At the 'Environment' tab, make sure that the 'Program Start Address' is set correctly. It must match the start address of the BlockRAM, which you verified in the previous step, 3. In this example, if you look back at step 3, where you verified the BlockRAM start address, the correct address is 0xFFFFC000. If you leave the 'Program Start Address' field empty, the software tools will use the default start address, 0xFFFF0000.

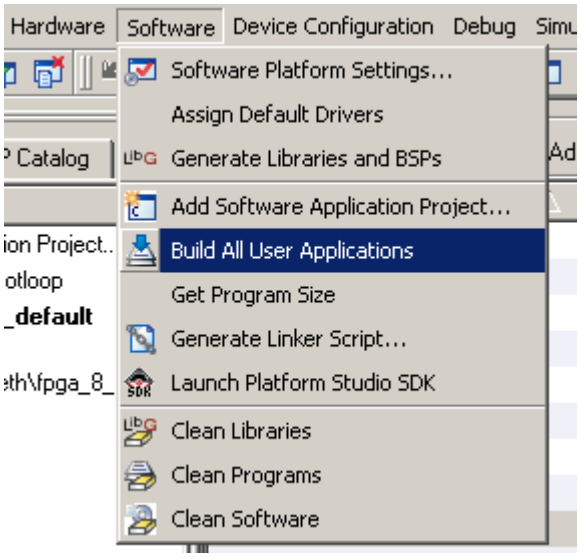
At this point you can also initialise the stack and heap size. If these fields are left empty, the software tools will use default stack and heap sizes of 4 Kb. In our case we have 16 Kb BlockRAM and only a very small C program so we take the default values (i.e. we leave the fields empty).

When done, click on the 'Debug and Optimization' tab. We want to enable debugging, as we intend to use the GDB debugger later on, so we need symbol information to be generated. Tick 'Generate Debug Symbols', and select 'Create symbols for debugging (-g option)', and at 'Optimization Level' select 'No Optimization'. When done, click 'OK'.



(5) Build Project

Now the project is ready to be build. This can be done via 'Software → Build All User Applications'.



Make a note of the executable file that is produced ('executable.elf') and its path.

```
collect2 version 3.4.1 ( Xilinx EDK 8.1.1 Build EDK_I.19.3 190106 ) (PowerPC Em
/cygdrive/c/EDK_8_1/gnu/powerpc-eabi/nt/bin/../lib/gcc/powerpc-eabi/3.4.1/../..
GNU ld version 2.15

   text    data     bss     dec     hex filename
   2002     836    8220   11058    2b32 ppc405_1/code/executable.elf

Done!
```

Download the Bitstream

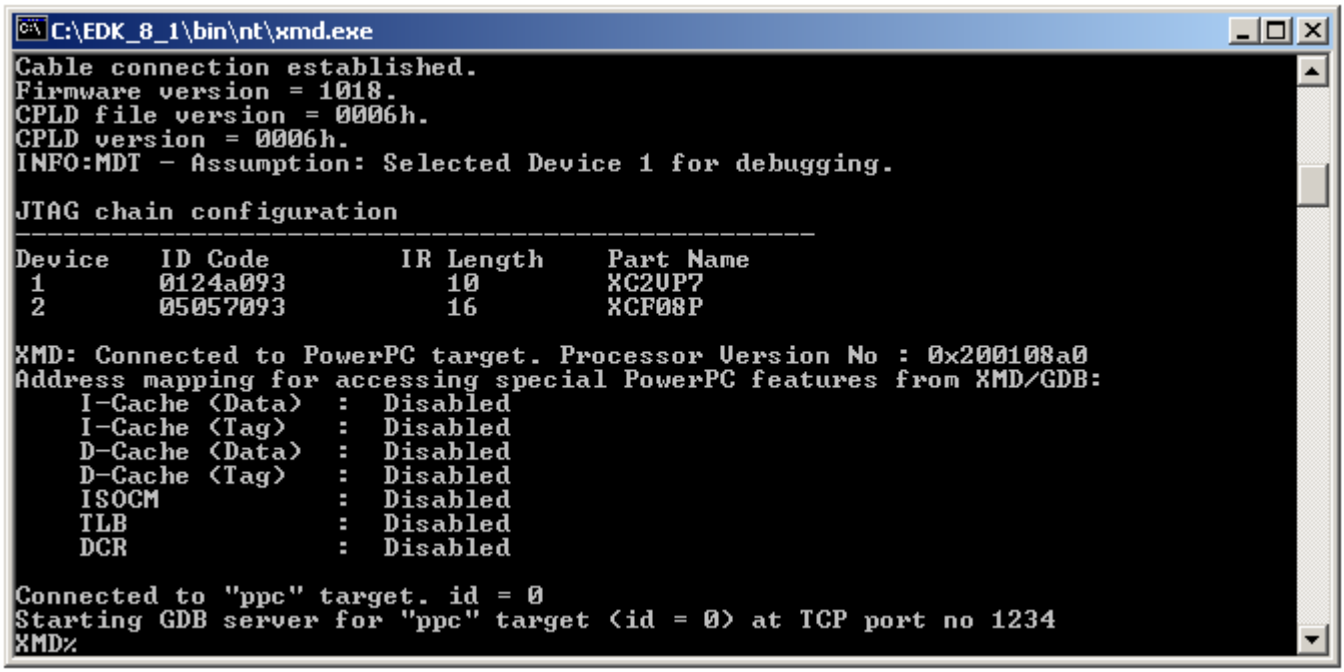
Before you can load and debug your embedded PowerPC program, you must first load the bit-stream. You can use either the 'download.bit' or 'system.bit' in the 'PPC_Ex_A\implementation' directory. Or you can use the 'top.bit' or 'top.rbt' in the 'PPC_Ex_A\ISE' directory.

You can use Impact to download a *.bit file, but you could also use the HUNT ENGINEERING tools to program the bit-stream. If you use the 'download.bit' bit-stream, the LEDs will start flashing as soon as the Virtex II Pro or Virtex 4 has been programmed. When you use any of the other bit-streams mentioned ('system.bit', 'top.bit', or 'top.rbt') the LEDs will not flash after programming. But you might prefer these bit-streams, as it will clearly demonstrate that XMD and the software debugger use BlockRAM by loading and running code out of it.

Start XMD

XMD is the Xilinx embedded PowerPC debugger, which gives you low-level debug access to the embedded PowerPC. You need to have a Xilinx Parallel Cable III/IV or Xilinx Platform Cable USB attached to the HERON module at this point. The program loading and debugging session will proceed via this JTAG cable.

In Xilinx Platform Studio, start XMD. With version 8.1 of the tools, that is 'Debug → Launch XMD. . .' A DOS box will now appear.



If XMD does not connect to the embedded PowerPC automatically as shown above, then you need to connect it manually.

If you have a type III parallel cable, type:

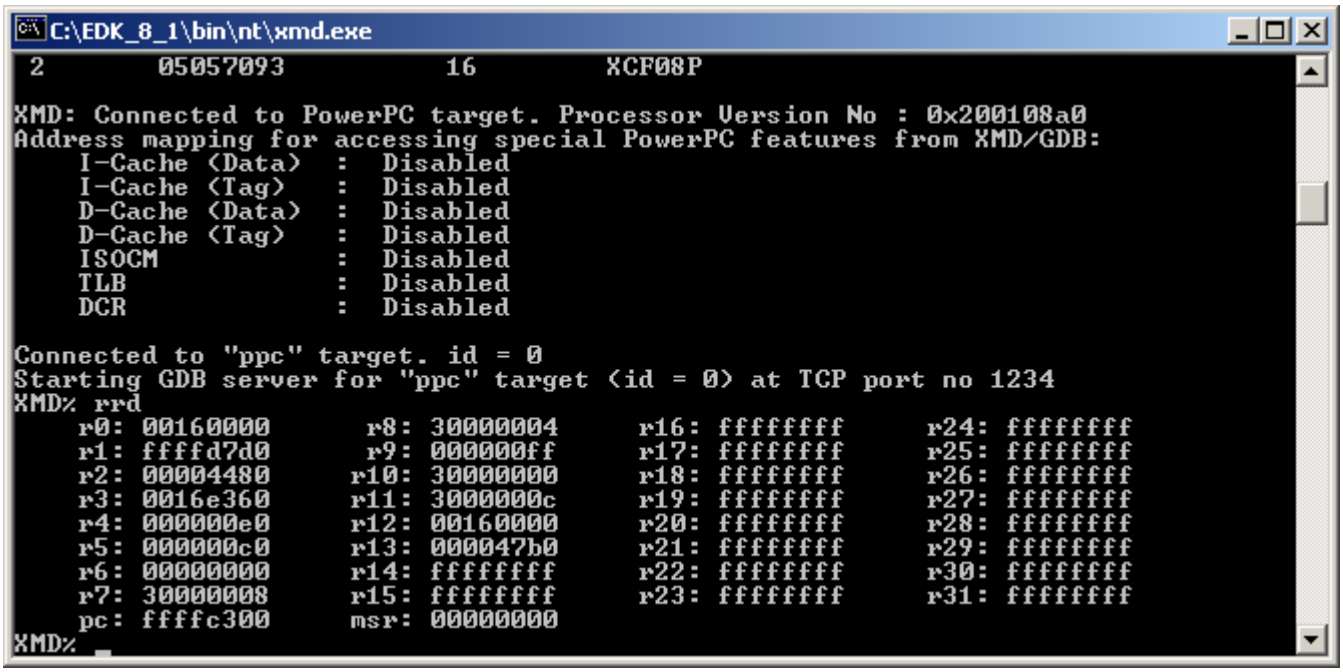
```
connect ppc hw -cable type xilinx_parallel3
```

If you have a USB cable, type:

```
connect ppc hw -cable type xilinx_platformusb
```

To verify that you have a proper connection, type:

```
rrd
```



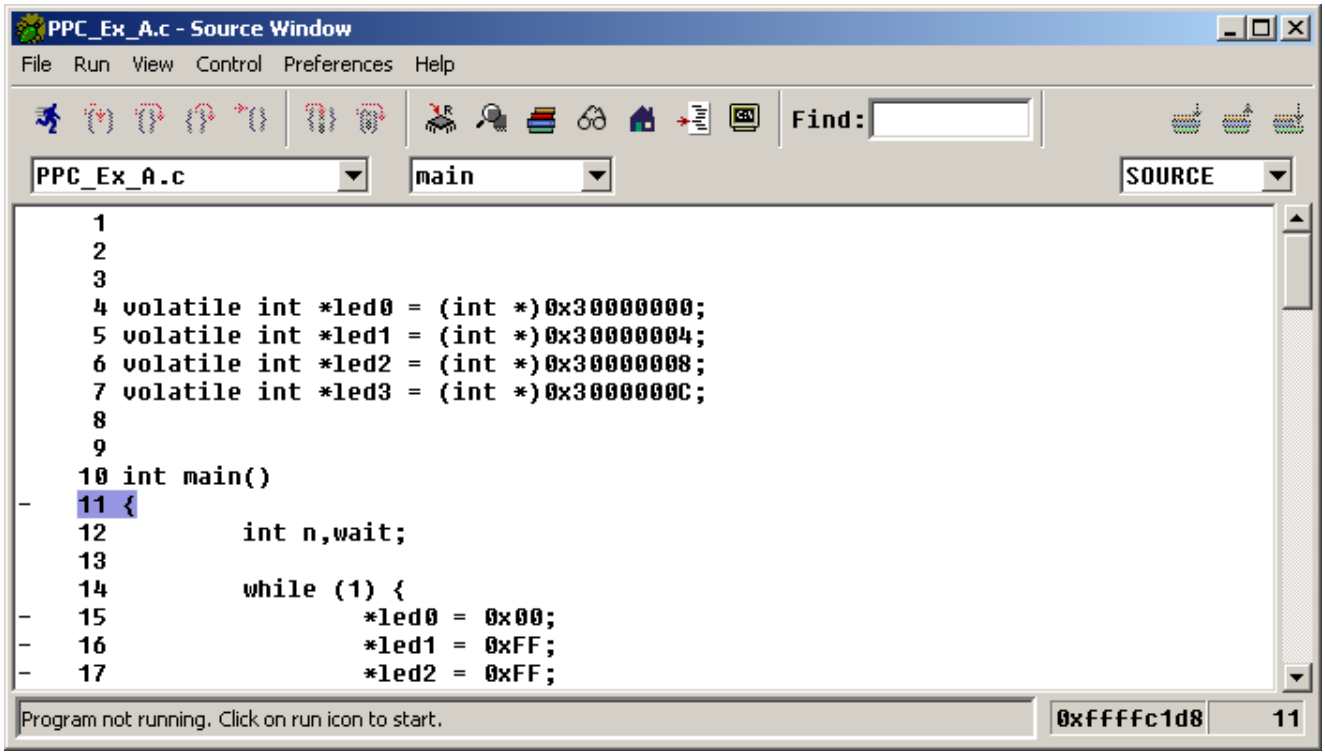
The 'rrd' command reads the embedded PowerPC registers. If you see their contents printed, the connection is working. If not, then try again making a connection with `ppcconnect`. If you use printer port LPT2 instead of LPT1, then:

```
ppcconnect -cable port lpt2
```

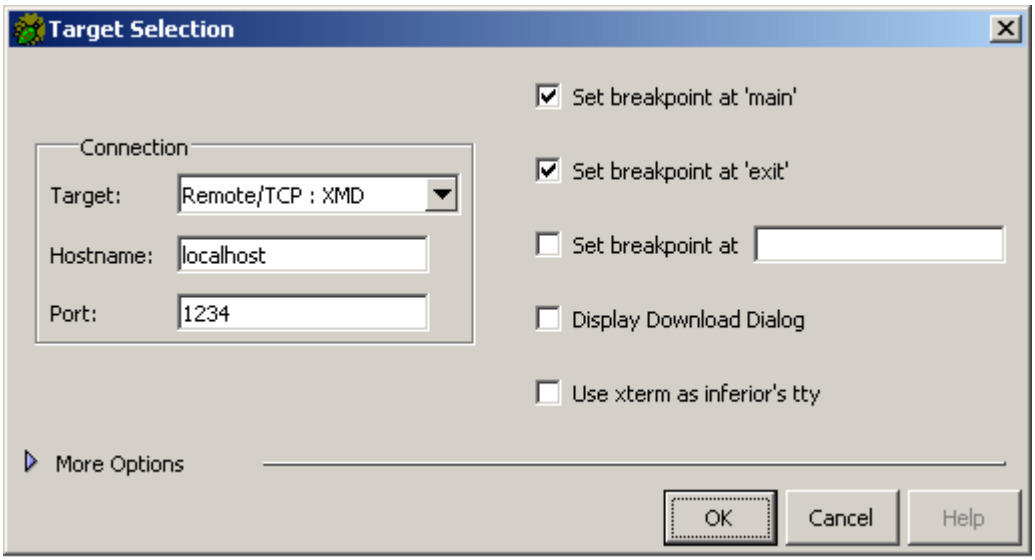
As said earlier, you must have a bit-stream programmed into the Virtex II Pro/Virtex 4 device for XMD to be able to connect to the PowerPC. XMD has a help page that displays if you type 'help' in the DOS box. When you do this, you can see that XMD is an elementary debugger that allows you to read and change register values, memory, run and set breakpoints, single step, and lots of other things, but all on a processor level.

Software Debugger

In Xilinx Platform Studio, now start the Software Debugger. In version 8.1 this is done with 'Debug → Launch Software Debugger . . .'. First a DOS box will appear, but ignore this. Five to ten seconds later a proper window will appear. This is GDB, the GNU debugger, but it is called 'Software Debugger' by Xilinx. On the top bar it will say 'PPC_Ex_A.c' - Source Window'.

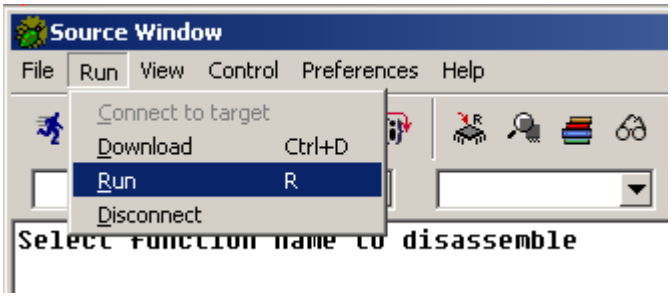


Although the Software Debugger shows a blue bar on the '{' line, the executable has not been loaded yet and we are not yet debugging. First we need to 'connect' GDB to the XMD. XMD listens on port 1234 to requests from GDB. In the GDB window, do a 'Run → Connect to Target'. A 'Target Selection' window appears. In the 'Connection' section, for 'Target' choose 'Remote/TCP : XMD', for 'Hostname' type 'localhost' (if not already there), and for 'Port' type '1234' (if not already there). Then click 'OK'.

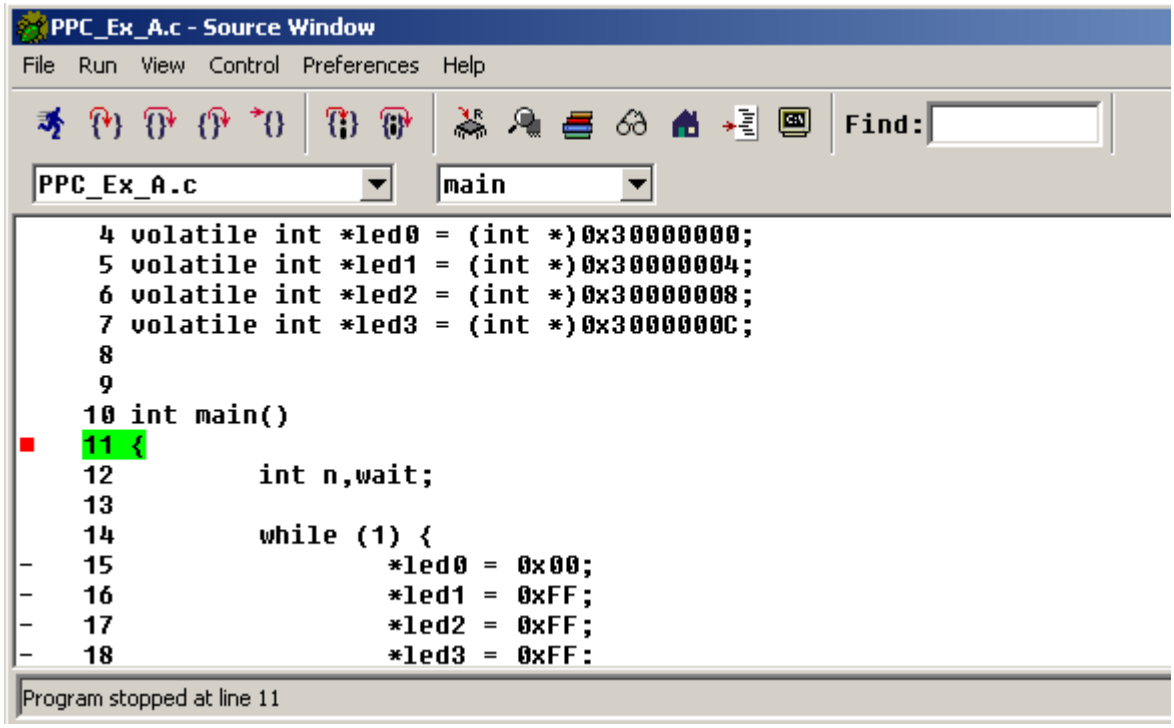


GDB should reply with showing a message box saying 'Successfully connected'.

At this point, the Software Debugger (GDB) is connected to the embedded PowerPC, but we still have no program in BlockRAM. That is, if you used the 'system.bit', 'top.bit' or 'top.rbt' bit-streams. With the 'download.bit' bitstream the BlockRAM would have been initialised already with executable code (but not necessarily with the code of the executable we want to debug now). Do a 'Run → Run', this will load our PowerPC ELF executable and run to main.



GDB should now show our 'PPC_Ex_A.c' file with a green bar on the first line in 'main'.



Now the system is ready for debug: the executable is loaded (into PLB BlockRAM), and the Software Debugger has run successfully until the first line in 'main()'. You can single step by pressing 's' (or via Control → Step). If you Single Step a few times, you will see the LEDs associated with 'led1', 'led2' and 'led3' switch off.

You can place a breakpoint by clicking on the '-' before a line number; a red square dot should appear in its place. To remove a breakpoint, click once on the red square dot before a line number. To run to a breakpoint, press 'c' (or do a Control → Continue), but don't mistakenly do a 'Run → Run' (shortcut: 'r') (as this will restart the debug session).

If you find that the GDB debug window doesn't show 'PPC_Ex_A.c' but some other C file, GDB might be confused and has loaded another program. Use GDB's 'File → Open' to select the program you want to debug. The File menu also shows shortcuts to programs you used or opened before.

If you find that you can only see disassembly but no C code, then you have forgotten to switch on debugging information (-g option) in the Compiler Settings.

Booting Process and BlockRAM

The embedded PowerPC 405 will at reset start executing at address 0xFFFFFFFC. When your project was built as a standalone project, the build tools will have added code (a jump instruction) at 0xFFFFFFFC. However, a 32-bit jump instruction can only jump to a 24-bit address. Therefore, the build tools have added another bit of code, 4 instructions, not too far away from address 0xFFFFFFFC. The instruction at 0xFFFFFFFC will branch to these 4 lines of code. It is here, in these 4 lines, that a full 32-bit jump is possible, and from here a jump will be made to the user program, where ever it happens to be located in the 4 Gb embedded PowerPC address space.

You can see this behaviour by resetting and loading your program, and not do a run to 'main'. In the GDB debug window, do a 'Run → Run', then a 'Run → Download'. In the XMD window you should now see

```
XMD% PC reset to 0xfffffffc, Clearing MSR register
```

In the XMD window, type

```
rrd
```

and then type return. The registers display will show the PC set at 0xFFFFFFFC.

Do another XMD single step, type:

```
stp
```

and then type return, in the XMD DOS box.

In my case the PC now becomes 0xFFFFE9B0, but the precise value may be different in your case. Do another single step, and the PC becomes 0xFFFFE9B4. A third single step, and the PC becomes 0xFFFFE9B8, and after a fourth single step the PC becomes 0xFFFFE9BC. If you do another single step, program execution will now jump to your application. In my case the PC is now at address 0xFFFFC520. But this address may well be very different in your case, as every application will likely have a different start address. (Note that the start address is the '_crt0' code start address, not the address of main. The code in crt0 sets up the stack, initialises the dynamic memory system (malloc etc) and does other initialisations and then calls your 'main()' function. The code for crt0 is in 'ppc405_1\libsrc\standalone_v1_00_a\src' directory, named 'crt0.S', in assembly code).

Because of this behaviour of the embedded PowerPC at reset, your hardware design must have a piece of BlockRAM that extends up to address 0xFFFFFFFF. Typically you would place 4 Kb to 32 Kb of BlockRAM at the top of the 4 Gb address range. For example, with 16 Kb BlockRAM placed at 0xFFFFC000 you would cover the top 16 Kb.

The Boot Process and XPS Software Tools

If you look in the 'ppc405_1\libsrc\standalone_v1_00_a\src' directory we can find the assembly code for the start-up code in the file 'boot.S'. Here we can see:

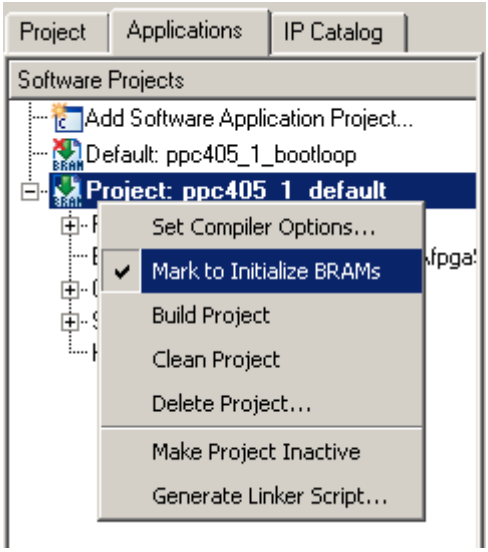
```
                .file "boot.S"
                .section .boot0,"ax"
                .global _boot0
_boot0:
                lis 0, _start@h
                ori 0, 0, _start@l
                mtlr 0
                blr
                .section .boot,"ax"
                .global _boot
_boot:
                b _boot0
```

The '_boot' label would be mapped onto address 0xFFFFFFFC. At the '_boot0' label we can see how a register is initialised with a '_start' address, the entry point of the embedded PowerPC executable. The code then jumps to the address that the register points to.

Adding your embedded PowerPC Executable Code to the Bitstream

Once you have completed development of the software, you probably want the software to start at the same time that the bit-stream is loaded, without using XMD or the Software Debugger (GDB). This can be done with 'Device Configuration → Update Bitstream'. What will happen is that the BlockRAM's contents (in the bit-stream) are initialised with the embedded PowerPC executable's code and data. Then, as soon as the bit-stream is loaded, whether via Impact or via HUNT ENGINEERING's hrn_fpga, the embedded PowerPC code will start to run, as the code and data already exist in memory.

First, make sure that is 'Mark to Initialize BRAMs' is ticked (right click on 'Project:…' as show below).



Now run 'Device Configuration → Update Bitstream' from within XPS. This process will take the bit-stream, 'system.bit' and use this and the embedded PowerPC ELF executable, to create a new bit-stream,

'download.bit'. In this new bit-stream, the BlockRAMs (of the memory at 0xFFFFC000) will be initialized with the embedded PowerPC executable's code and data.

Use Impact to program the bit-stream, but be sure to use the file 'download.bit' instead of the file 'system.bit'.

The above will work when your embedded PowerPC executable fits within the BlockRAM that you placed at the top of the 4 Gb PowerPC address space. If your application is too big to fit in BlockRAM, enlarge the BlockRAM sufficiently so that your program will fit. BlockRAM is a finite resource and your application may be too big even if you use all available BlockRAM. In such cases you may have to use external memory to store your application.

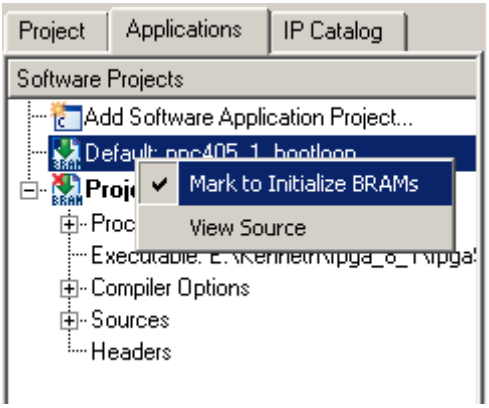
The Xilinx Bootloop

You may wonder why there is such a thing as 'Mark to Initialise BRAMs'. This setting allows you to select whether to update the bitstream with your application or with the 'bootloop'. But what is a bootloop? This is what Xilinx says in the 'Initialise Bitstreams with Bootloops' section of the 'Platform Studio User Guide'.

Once the FPGA has been configured with a bitstream, the processor is brought out of reset and starts executing. If the system has not yet been initialized with the software application, the processor may execute code that puts it into a state that it cannot be brought out of with a soft reset. The processor must therefore be kept in a known good state until the system can be completely initialized.

A bootloop is a software application that keeps the processor in a defined state until the actual application can be downloaded and run. It consists of a simple branch instruction, and is located at the processor's boot location. XPS contains a predefined bootloop application. To use a bootloop, the software application is created as usual. The linker script used is no different from the case in which no bootloop is used, i.e., the software application should contain instructions at the processor's boot location.

The software application should not be used to initialize the system BRAMs. Right click on the project name in the tree view, and ensure that 'Mark to Initialize BRAMs' is not selected. If it is, deselect it. To initialize BRAMs with the bootloop, right click on the default bootloop project ('ppc405_1_bootloop') in the tree view and click on 'Mark to Initialize BRAMs'.



Update the bitstream with the bootloop by selecting 'Device Configuration → Update Bitstream' in the main XPS window.

This bitstream may then be downloaded to the FPGA. The software application may then be downloaded using either XMD or System ACE.

If you view the source code of the bootloop, it is simply a continuous jump to itself, as you would expect:

```
.section ".boot","ax"
_boot: b _boot
```