Running Linux on a Xilinx XUP Board

John H. Kelm

June 23, 2006

Abstract

A tutorial for booting a fully functional operating system based on the Linux 2.4 kernel on a Xilinx University Program Virtex II-Pro based development board is presented. Furthermore, we describe a reconfigurable hardware accelerator that can be accessed directly by applications or via a character device driver.

1 Introduction

The Xilinx University Program (XUP) development board provides a rich environment in which students can gain an understanding of system on a chip (SoC) design, software-hardware codesign, computer architecture and digital logic design and synthesis. The board is based on the Xilinx Virtex-II Pro field programmable gate array (FPGA), which has two embedded PowerPC405 cores in addition to nearly 31,000 logic cells in which to synthesize the necessary system components and implement the students' own hardware designs.

Although it is possible to build interesting SoC designs on the XUP board and run them with free-standing applications, doing so fails to exercise the full capabilities of the board. It is not only possible to build a fully functioning system on the Virtex-II Pro using one of the embedded PowerPC cores, but to also run a full-fledged operating system on top of the reconfigurable hardware. This tutorial describes: synthesizing the hardware necessary to boot Linux 2.4.26 on the XUP board; how to obtain, configure and compile the kernel to run on the board; how to boot a fully functioning operating system using the hardware and kernel developed in the tutorial; and methods for connecting hardware accelerators to the the XUP board.

To exploit the potential of the XUP board with Linux we have incorporated a motion estimation hardware accelerator—used for fast encoding of digital movies—to demonstrate the potential of such a reconfigurable platform. This tutorial steps through the basic procedure required to build a Block RAM (BRAM) based hardware accelerator that can be connected to the On-Chip Memory (OCM) bus of the embedded PowerPC 405 core on the XUP board. Interfacing with the hardware accelerator synthesized in the reconfigurable logic of the FPGA is covered briefly for direct methods using the `mmap()` system call. Furthermore, interfacing with the accelerator using a character device driver implementation under the Linux 2.4 kernel is described.

2 Hardware Configuration

Booting Linux on the XUP board requires some familiarity with the Xilinx tool chain. This tutorial assumes that the reader has already installed the Xilinx Embedded Developers Kit (EDK), Platform Studio and ISE. The entirety of our work was completed using EDK Version 7.1.01i; the methods and advice may or may not apply for other versions.

2.1 Base System Builder

The first step uses the Xilinx wizard—Base System Builder (BSB), a wizard provided in the EDK that is used to generate a basic hardware configuration on the XUP board—to build the files and integrate the components necessary to boot Linux on the XUP board. After the wizard is complete, the system is synthesizable, however unable to boot the Linux kernel; modifications to the environment created by the BSB must be made. The tutorial covers the necessary augmentations of the Machine System Settings (MSS) file which contains high-level configuration data, the Microprocessor Hardware Specification (MHS) file which provides the non-default mappings for the system components, and the User Constraints File (UCF) which maps internal wire names to external pins on the chip.

Step 1

Go to the *File* menu, select the *New Project* submenu and select *Base System Builder ...* menu item. The BSB wizard allows the selection of hardware components to add to the design. Select a location to place all the files, noting that there must be **no spaces** in the file names referenced from within the Xilinx tools. Go to the next menu and select *I would like to create a new design.* and continue. Select the board vendor as *Xilinx*, the board name as *XUP Virtex-II Pro Development System* and the board revision as *C*. The board definition files may need to be downloaded from Xilinx if not already installed.

Step 2

In the next menu select PowerPC cores (this tutorial does not make use of the MicroBlaze soft cores) and 100 MHz bus and processor clock frequencies. Disable all external caches by selecting *NONE* for both On–Chip Memory (OCM) drop–down boxes. Enable the JTAG interface to the FPGA. The checkbox for *Cache* has no effect; it enables the cache from within the stand–alone application template created by the BSB which will not be used by this tutorial. Proceed to the next window.

Our current designs run the Processor Local Bus (PLB) at 100 MHz and the processor at 300 MHz, but to keep the tutorial consistent with our original process and to ensure accuracy 100 MHz clocks are recommended. All external caches are disabled in the initial design as well. Data caches are not possible if a hardware accelerator is connected via the Data Side OCM (DSOCM) as described in Section 4.

Step 3

The following list covers what peripherals are chosen in the next three windows of the BSB—all should be interrupt driven except for the DDR memory controller:

- RS232_UART_1 – *OPB UART16550, Configure as UART 16550*
- Ethernet_MAC – *PLB ETHERNET, No DMA*

- SysACE_CompactFlash – *OPB SYSACE*
- DDR_512MB_64Mx64_rank1_row13_col10_cl2_5 – *PLB DDR*
- PS2_Ports – *OPB PS2 DUAL REF*
- VGA_FrameBuffer – *PLB TFT CNTLR REF*

All other peripherals are to be disabled.

Step 4 In the next window select 16KB of BRAM attached to the PLB BRAM controller. At least one BRAM must be present in the design for the hardware build system to function properly. The framebuffer may cause some systems to crash while generating BSPs and Libraries and may need to be added manually.

Step 5 The next menus relate to software and do not pertain to the system being built here. Disable any related options. Select *None* as default values for STDIN and STDOUT. Do not select any of the sample applications. The wizard will now build the system and the necessary files are generated.

2.2 Configuration

The BSB generates all the necessary files needed to build a functioning SoC on the Virtex-II Pro. However, the files generated by the BSB are incomplete and the following alterations must be made.

Step 6 **DDR Clocks** Go to the *Projects* menu, select *Add/Edit Cores* then select the **Parameters** tab. Under the *dcm_1* entry confirm the following parameters are present. Add any absent parameter/value pairs listed below:

- `C_PHASE_SHIFT` and set its value to "60"
- `C_CLKOUT_PHASE_SHIFT` and set its value to "FIXED"

Step 7 **Frame Buffer Setup** Switch back to the **Peripherals** tab, add the Digital Control Register (DCR) to On-chip Peripheral Bus (OPB) bridge if it is not already present by selecting *opb2dcr_bridge* and adding it to the design. Select the **Bus Connections** tab and add the *dcr_v29_v1_00_a* IP core to the design; a new column should appear in the left panel. By clicking on the intersection of each of the following, connect the *VGA_FrameBuffer sdcr* as the slave on the DCR bus (An "S" should appear at the intersection), the *opb2dcr_bridge_0 mdcr* as master on the DCR, and the *opb2dcr_bridge_0 sopb* as slave on the OPB.

Step 8 Select the **Ports** tab, under **Internal Port Connections** add the *SYS_dcrClk* port from the right menu under *VGA_FrameBuffer*. Set the Net Name to *sys_clk_s*. Select the **Addresses** tab. Generate new addresses, ignoring any warnings referencing *ppc405_1*. Conflicts in the automatically generated addresses may arise. Therefore, manually verify that the base addresses for the the OPB and DCR are unique from other devices listed. Lock all the addresses to prevent future address generation steps from altering the current configuration.

Step 9

System MSS The following lines (one line per item) must appear in the `OS` section of the *system.mss* for the design. Make any alterations necessary such that the configuration section for *ppc405_0* matches the values provided here. The file is accessible by double–clicking on the *MSS FILE: system.mss* entry in the left–side pane of the EDK.

- `OS_NAME = linux`
- `OS_VER = 2.00.b` – The version has no correlation to the kernel number and may be irrelevant.
- `TARGET_DIR = C:/BSP_Files` – Where the BSP files is created. **Note:** There must be no spaces in the name and forward slashes replace backslashes on Windows platforms.
- `MEM_SIZE = 0x10000000` – The number of bytes of memory attached to the DDR controller. Shown as 256MB.
- `connected_periphs = (RS232_UART_1, Ethernet_MAC, SysACE_CompactFlash, PS2_Ports, opb_intc_0, VGA_FrameBuffer)` – The names of all connected peripherals to include when building the BSPs (see Section 3.2.2).
- `PLB_CLOCK_FREQ_HZ = 100000000` – The bus frequency in Hz. 100 MHz is used in all of our systems.

Under the section for the PS/2 Ports:

- `DRIVER_NAME = ps2_ref` – This replaces `generic` as the driver.

Step 10

Hardware capable of booting the Linux kernel is now configured. Before proceeding to get the kernel source and attempt to compile a kernel image, synthesize the hardware by going to the *Tools* menu and selecting the *Update Bitstream* menu item. Innocuous warnings may occur, but any errors will immediately stop the build.

The *ps2_ref* driver must be up to date. Its absence will lead to the following error message:

```
Driver ps2_ref1.00a does not support peripheral
opb_ps2_dual_ref.  Download the updated cores.
```

Any errors that stop the build unrelated to the above steps are outside the scope of this tutorial and answers can be found by consulting the Xilinx documentation (see Appendix A). Once the bitfile has been created, it can be downloaded to the XUP board. At this point the bitfile is much like any piece of software—the developer was able to convince it to compile, but no one knows what bugs lurk in the output. Be warned that even though it builds, the bitfile may not be correct and debugging the problem may be nontrivial. With any SoC design, the problem could lay with faulty hardware, a missing or incorrect parameter, an incomplete design or any of the myriad problems that can manifest in hardware or software or both.

3 Compiling Linux

3.1 The Development Environment

Before a kernel can be generated for the XUP board, a PowerPC development environment must be created. If a native PowerPC Linux machine is accessible, this section can possibly be skipped as kernels should be able to build under such an environment without alteration. For PCs running Linux, a cross-compiling environment is the method used to generate kernel binaries. Our test environment consists of a VirtualPC 5.3.582.27 installation of Slackware Linux 10.1 running the 2.6.13 kernel, but any Linux distribution that meets the requirements of Crosstool should suffice.

Step 11

Crosstool (see Appendix A) is a software package created by Dan Kegel that allows x86 Linux machines to target the PowerPC405 core of the XUP board. Details on how Crosstool should be installed are available from the Crosstool website. The script used in our environment was *demo_ppc405.sh* and the configuration lines used are as follows:

```
'cat powerpc-405.dat gcc-3.4.1-glibc-2.3.3.dat'
sh all.sh --notest
```

3.2 The Linux Kernel

At this point in the tutorial the cross–compiler must be set up to target the PowerPC 405. After obtaining the kernel, the Board Support Packages (BSP) for the XUP board must be created and integrated into the kernel source tree. A list of hand alterations that must be made in addition to integrating the BSP files is given. A bootable kernel image will be configured and built with the customized source tree.

3.2.1 Obtaining the Kernel Source

Step 12

The source tree used is the linuxppc_2_4_devel fork and is available via BitKeeper and can be downloaded from *http://ppc.bkbits.net*. Further instructions on how to download and install BitKeeper and download the kernel is beyond the scope of this tutorial.

3.2.2 Board Support Packages

Xilinx provides a tool integrated into the EDK that allows the user to generate configuration and header files necessary to compile the Linux kernel for the ML310 development board. Many of the options covered here reference the ML310, but work with the XUP board as well—deviations are noted.

Step 13

Generate the BSP for the XUP board by entering the *Tools* menu and selecting the menu item *Generate Libraries and BSPs*. The files are placed in the directory specified in the *system.mss* file. Once the BSPs have been generated, copy the BSP directory contents just created into the kernel source directory obtained above. As can be seen inside the generated directory, the BSP contains a `drivers/` directory with files needed to build Linux device drivers for the Virtex-II Pro synthesized peripherals and an `arch/` directory containing the necessary configuration files for the particular implementation of the PowerPC 405 core on

the Virtex-II Pro.

Step 14 The main file used by the Linux build system included in the BSP is:

arch/ppc/platforms/xilinx_ocp/xparameters_ml300.h

The file contains all the `#define`'s specific to the implementation described here (e.g., memory ranges and connected peripherals). To enable the PS/2 ports insert the following two lines into the file:

```
#define XPAR_PS2_PORTS_DEVICE_ID_0 0
#define XPAR_PS2_PORTS_DEVICE_ID_1 1
```

Step 15 Next edit the Makefile in the top–level directory. Add the following line to the Makefile:

```
EXTRA_CFLAGS = -I$(TOPDIR)/arch/ppc/platforms/xilinx_ocp
```

Step 16 Comment out line 708 of *arch/ppc/boot/simple/embed_config.c*. A note is present in the code to inform the curious reader. Next add the following line to *drivers/net/xilinx_enet/xemac.c*:

```
#include "xenv.h"
```

Step 17 In the file *arch/ppc/boot/simple/Makefile* at line 267 change:

```
mv zvmlinux ...
      to
cp zvmlinux ...
```

Step 18 In the file *arch/ppc/platforms/xilinx_ocp/Makefile* remove the references to fifos and replace it with the following two values noting that the character is a lower–case 'L' not a numeral one. These references must also be updated in *arch/ppc/platforms/xilinx_ocp/xilinx_syms.c*:

```
xpacket_fifo_v2_00_a.o
xpacket_fifo_l_v2_00_a.o
```

Step 19 In the file *drivers/video/xilinxfb.c*, add the following line:

```
#include "xparameters.h"
```

Step 20 In the main Makefile, set the following environment variables:

```
ARCH := ppc
CROSS_COMPILE := powerpc-405-linux_gnu
```

The kernel source tree should now build without errors.

3.2.3 Configuration

Step 21 Listed below is the configuration hierarchy for our build of the Linux 2.4 kernel. Other options can be enabled, but some may cause the system to not boot or not compile. Execute `make menuconfig` and select the following options:

```
Platform Support:
    Processor Type:  40X
    Machine Type:  Xilinx--ML310
    Math Emulation
    TTY0 device and default console:  UART0
General Setup:
    PC PS/2 Style Keyboard
    Networking Support
    Sysctl Support
    System V IPC
    Default Bootload Commandline:
        ‘‘console=ttyS0,9600 console=tty1 root=/dev/xsysace/disc0/part3 rw’’
Memory Technology Devices:
    MTD Partioning Support
    RedBoot partition table parsing
    Direct char device access to MTD devices
    Caching block device access to MTD devices
    RAM/ROM/Flash chip drivers:
        Detect flash chips by Common Flash Interface (CFI) probe
        Support for AMD/Fujitsu flash chips
Block Devices:
    Xilinx On-Chip System ACE
    Loopback device support
    RAM disk support
    Initial RAM disk (initrd) support
Networking:
    UNIX domain sockets
    TCP/IP Networking
Networking Device Support:
    Ethernet 10/100:
        Xilinx on-chip ethernet
Console Drivers:
    Framebuffer Support:
        Xilinx LCD Display support
        Select compiled-in fonts
        VGA 8x16 Font
Character Devices:
    Virtual terminal
    Support for console on virtual terminal
    Standard/Generic serial support
    Support for console serial port
    Mice:
        Mouse Support
        PS/2 Mouse
File systems:
```

```
Ext3 Journaling file system
DOS FAT fs support
MSDOS fs support
VFAT fs support
/proc file system support
/dev file system support
/dev/pts file system support
Second extended fs support
```

Step 22 Test the kernel build. The following command string will allow you to build the kernel and view any errors that may occur:

```
make dep
make zImage 1>build.stdout 2>build.stderr
```

A new kernel image *zImage.elf* will be created in *arch/ppc/boot/images*. This is the compressed kernel image can be used to boot the PowerPC 405 core of the XUP. To get the system to run, a root file system is still needed (see Section 3.2.4), but the remaining files from the kernel source tree are no longer required for booting.

For debugging, the uncompressed kernel image with its symbol table intact is in the root directory for the kernel image named *vmlinux*. It can be disassembled using the Crosstool (see Appendix A) version of *objdump* with the command:

```
powerpc-405-linux-gnu-objdump -d vmlinux | less
```

After making changes to the kernel source tree seemingly erroneous errors may occur upon building the kernel. The kernel build system related to the board support packages was found to not account for modified files and thus it would not rebuild necessary parts of the kernel source tree. Often it is necessary to rebuild the whole kernel by executing `make clean` and then proceeding with the steps outlined above. Furthermore, to ensure that the build system takes note of any changes to the BSP, the files located in the include directory should be given an updated time stamp with the following command line:

```
touch arch/ppc/platforms/xilinx_ocp/*
```

3.2.4 Root File System

Step 23 The BYU Linux on FPGA Project page listed in the Appendix provides a step–by–step tutorial for generating a root file system on a Compact Flash device. We have elected to use Yellow Dog Linux and not BusyBox—details regarding how we generated the root file system will be forthcoming, but are similar to the procedure detailed on the BYU page.

3.2.5 Booting Linux

There are three ways in which to boot the kernel created above on an XUP board: a) using the XMD debugger from within the EDK, b) by using a boot loader to bootstrap the kernel directly from a Compact Flash card or micro drive, or c) a network boot. This tutorial

covers only the first method, as it leads to expedited debugging and fewer complications. For information regarding booting from the Compact Flash directly see the BYU FPGA Project in Appendix A.

Step 24 To boot the kernel, move it to a location that can be accessed from the EDK. Insert the Compact Flash device with the root file system into the Compact Flash slot on the XUP board, turn on the board, and go to the *Tools* menu and select *XMD* to connect to the XUP board. If the JTAG chain is not correctly configured an error may occur. If XMD is unable to find a configuration file, place the following line in a file named *xmd_ppc405_0.opt* under the project directory `etc/`:

```
connect ppc hw -cable type xilinx_platformusb frequency 12000000
dow c:/YOUR/PATH/TO/zImage.elf
con
```

Step 25 Restart the XMD Debugger to connect via the USB cable. Once connected, the script places the kernel into memory with the command `dow` and boots the kernel with the command `con` after the kernel is loaded into memory. The XMD window can now be closed. The kernel boot process can be viewed via HyperTerminal by attaching a serial cable between the XUP board and to the host WinXP PC. The HyperTerminal connection should match the setting given at the command line to the kernel which are a 9600 baud connection, 8 data bits, no parity, 1 stop bit, and no flow control. After the kernel is uncompressed and the framebuffer driver is loaded, the remaining portion of the boot process should be viewable from a monitor connected to the Sub-D connector of the XUP board. At this point the reader should have a bootable Linux system running on the XUP board.

4 Attaching an Accelerator

The impetus for enabling Linux to boot on a Virtex-II Pro FPGA is to demonstrate reconfigurable hardware accelerators being accessed from within the context of a standard operating system to improve performance. Our test accelerator performs motion estimation for an H.264 video encoder to demonstrate the speed up of a hardware accelerator over a software–only approach. The method for implementing a hardware accelerator and accessing it from the Linux system created in this tutorial is described. Furthermore, we present two ways in which the accelerator can be interfaced: directly through software and via a character device driver.

4.1 Implementing a Hardware Accelerator

A complete discussion of implementing a hardware accelerator on the Virtex-II Pro FPGA running Linux is outside the scope of this tutorial. However, an overview is provided to motivate interest in designing such accelerators and to provide the information necessary to access such a device from within the context of the operating system running on the XUP board.

The motion estimation hardware accelerator used in our studies can be though of a memory–mapped device in a conventional computing environment. The implementation is realized

using Block RAMs (BRAM) in the FPGA fabric and memory mapping them into our design. The decision was made to attach the device to the Data Side On–Chip Memory (DS-OCM) bus, however the interface provided by our hardware accelerator should be amenable to being placed on any of the other buses within the SoC design (e.g., PLB or OPB). The DS–OCM provides a low latency bus that can have only one device present—in our case, the hardware accelerator. Details of the OCM bus can be found in the Xilinx reference documents.

The Virtex-II Pro provides an IP core that handles all bus transactions with the PowerPC 405 core and exports an interface that is meant to attach directly to a BRAM. Our hardware accelerator maps the signals provided by the BRAM bus connection to BRAM modules inside the hardware accelerator. The abstraction provided by the BRAM controllers allows the device to reside on any bus, but we have not yet attempted to attach the accelerator to other buses.

To connect a device to the design created in this tutorial, add a directory for the accelerator to the *pcores* directory at the top level of the project. The newly created directory must contain another directory `hdl/` with `vhdl/` and/or `verilog/` subdirectories, depending on the accelerator implementation language. In the `pcores/` directory there must be a `data/` directory containing an MPD file, as discussed below. From inside the EDK it is now possible to add the accelerator to the design by selecting *Project* followed by the menu item *Add/Edit Cores ...*. Select the accelerator design from the list at the right and add it to the design.

A MicroProcessor Definition (MPD) file must be generated for the device. Add the file to the `pcores/data/` directory under the top–level directory for the accelerator. An example file is located in Appendix B.

The DS–OCM BRAM controller must also be added to the design to interface with the newly added accelerator. Again, go to *Project* and then to *Add/Edit Cores ...* and add the `dsocm_if_bram` core. A memory range must be assigned to the DS-OCM BRAM controller that will be mapped to the accelerator. Set a memory range by going to the *Addresses* tab in the *Add/Edit Cores ...* dialog and enter an appropriate range for the device.

Note BRAMs are 2048 bytes in size. Unless the design is performing banking and requires multiple BRAMs, a 2048 byte range is appropriate.

The BRAM controller must be connected to the hardware accelerator. To do so, open the *system.mhs* file by double–clicking on the *Open the system.mhs file ...* item of the left–most pane of the EDK main window. The system should have added an entry based on the information provided in the MPD file. The DS-BRAM controller must have its `PORTA` attached to the accelerator's `PORTA`. The mapping provided in the MPD file will reference the bus interface exported by the DS-OCM BRAM controller and be mapped into the 2 KB address space specified earlier.

Note The Xilinx ISE development tool was used to develop and test the design used in our studies. ISE may be capable of generating the necessary files listed here, but is beyond the scope of this tutorial.

4.2 Direct Interface

The first method employed by our group to interface with the motion estimation hardware accelerator was by using the *mmap()* system call to map the address space of the accelerator into an application. The OCM bus presents a slight challenge to the developer in that it requires the physical and virtual addresses of the memory mapped region to be the same. From an application the memory mapping can be done with the following code sequence:

```
int fd;
unsigned int *bram;

fd = open("/dev/mem", O_RDWR);
bram = mmap(0x40000000, 2048, PROT_READ
   | PROT_WRITE, MAP_SHARED, fd, 0x40000000);
assert(bram == 0x40000000);
```

The pointer `bram` can now be accessed (i.e., indexed via the `[ ]` operator) like an array. Note that the code must be run as root user. The *mmap()* system call may return a different virtual address from the one specified (i.e., the virtual address that matches the physical address 0x40000000), and therefore an assertion is placed to ensure that an erroneous mapping does not occur.

4.3 Device Driver Interface

An in–depth description of the character device driver used to interface with the hardware accelerator is beyond the scope of this tutorial. However, a brief overview of the necessary components for generating a skeleton driver for a BRAM–based hardware accelerator will be presented. At a high level the driver can be thought to reimplement a reduced overhead version of the *mmap()* system call. The following is an outline of the steps to generate the driver:

1. Obtain the memory map semaphore for the current process (i.e., `current->mm->mmap_sem`) at `open()`. The driver must insert a new virtual memory area (VMA) for the memory mapped range.

2. Make a call to `get_unmapped_area` with the assigned address and range for the device as specified in the *Addresses* tab of the the *Add/Edit Cores ...* dialog with the flag `MAP_SHARED`.

3. Allocate and initialized the VMA for the area—for reference the reader can look at the `do_mmap()` kernel function. The pages must be marked `VM_SHARED` and `VM_RESERVED` along with any other flags relevant to the reader's design.

4. Now that an area of virtual memory with the proper characteristics for the device has been allocated to the driver instance, a call must be made to `remap_page_range()` to build the page tables.

   Note Both addresses and offsets must be equivalent for OCM–based BRAM hardware accelerators in the call to `remap_page_range()`.

5. Insert and enable access to the newly created vma using `insert_vma_struct()` and then call `make_pages_present()`.

At this point the user is now able to directly access the memory mapped into the accelerator from user space. It may be desirable to alter the permissions on the pages so that the user does not have access to them while the driver is functioning for safety and correctness. The semantics chosen for interacting with the device are application specific. For our model, we chose to create mutually exclusive accesses between `write()`'s to the device for placing blocks of video frames into the device's memory and `read()`'s to access the motion vector when completed. To signal completion, the device could use interrupts, but our initial model simply polls the device, which toggles a value in the BRAM when completed. Timeouts were also added to ensure liveness in the event of process failure.

APPENDIX

A Reference Websites

1. Crosstool
   http://www.kegel.com/crosstool
2. Xilinx PowerPC405 Block Reference Manual
   http://www.xilinx.com/bvdocs/userguides/ug018.pdf
3. BYU Linux on FPGA Project
   http://splish.ee.byu.edu/projects/LinuxFPGA/index.htm

B Example MPD File

```
BEGIN mpeg4_me
    OPTION IPTYPE = PERIPHERAL
    OPTION IMP_NETLIST = TRUE
    OPTION HDL = VHDL
    OPTION SIM_MODELS = BEHAVIORAL : STRUCTURAL
    OPTION CORE_STATE = ACTIVE
    OPTION IP_GROUP = LOGICORE
    OPTION ARCH_SUPPORT = virtex2p

    BUS_INTERFACE BUS = PORTA, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF

    PARAMETER C_MEMSIZE = 2048, DT = integer
    PARAMETER C_PORT_DWIDTH = 32, DT = integer, BUS = PORTA
    PARAMETER C_PORT_AWIDTH = 32, DT = integer, BUS = PORTA
    PARAMETER C_NUM_WE = 4, DT = integer
    PARAMETER C_FAMILY = virtex2, DT = string
    PARAMETER DWIDTH = 32, DT = integer
    PARAMETER AWIDTH = 9, DT = integer

    PORT MPG_ADDRA = BRAM_Addr, DIR = I, VEC = [31:0], ENDIAN = LITTLE, BUS = PORTA
    PORT MPG_CLK = plb_clk, DIR = IPORT MPG_CLKA = BRAM_Clk, DIR = I, BUS = PORTA
    PORT MPG_DIA = BRAM_Dout, DIR = I, VEC = [31:0], ENDIAN = LITTLE, BUS = PORTA
    PORT MPG_ENA = BRAM_EN, DIR = I, BUS = PORTA
    PORT MPG_RESET = sys_rst, DIR = IPORT MPG_SSRA = BRAM_Rst, DIR = I, BUS = PORTA
    PORT MPG_WEA = BRAM_WEN, DIR = I, VEC = [3:0], ENDIAN = LITTLE, BUS = PORTA
    PORT MPG_DOA = BRAM_Din, DIR = O, VEC = [31:0], ENDIAN = LITTLE, BUS = PORTA
END
```