GIMME2 - a getting started guide

Carl Ahlberg
Robotics Group, Intelligent Future Technologies (IFT)
School of Innovation, Desing and Engineering (IDT)
Mälardalen University
e-mail: carl.ahlberg@mdh.se

Fig. 1. GIMME2 with F1.2/6mm lenses from Edmund Optics

Abstract

GIMME2 is an embedded stereo vision system based on the Xilinx Zynq processing platform. The Zynq combines FPGA fabric with a dual-core ARM on a single chip. Hence, applications on GIMME2 may benefit from FPGA-based hardware acceleration complemented by conventional sequential processing on the ARM cores. This is a guide how to setup GIMME2 for operation and run simple example applications. This involves configuration of the Zynq with respect to peripheral hardware, basic interfacing between the different processing units, and booting Linux, using the Xilinx Vivado tool suite. By following this guide the user will get acquainted with part of the tools and the Vivado work-flow, concepts of the Zynq platform, and be able to create a base setup for GIMME2, which can serve as a reference for future application development.

Index Terms

GIMME2, Xilinx Zynq, FPGA, ARM, Linux, U-boot, device tree, ramdisk

I. Introduction

GIMME2 is designed with image processing and stereo vision in mind, to be versatile and adaptable, possible to implement in different applications and application areas. To achieve this a combination of the following properties is required:

- High frame-rate, not only from the camera but a holistic sensory system.
- Low latency, to be able to operate in real time with respect to human interaction or faster than human reaction.
- Physical characteristics of restricted dimensions and ruggedness to be implemented in small, mobile, autonomous systems. Ease, or even possibility, of integration in existing systems with limited effect on motion dynamics.
- Power efficient as GIMME2 relies on a host system, where energy may be a limited resource, as in battery powered devices, or need a power source of its own.
- Low cost with respect to performance and host system. Sensors are all-important for creating awareness of the surrounding environment, and the more complex and dynamic the surroundings and the less a priori knowledge gathered, the higher demands on the sensory system, and the higher the cost.

To achieve high processing power and power efficiency at a low cost the traditional sequential processing platform was discarded for a Xilinx Zynq-7020 All Programmable System on Chip (SoC). The Zynq combines FPGA fabric with a dual-core ARM CPU on a single chip. The FPGA-part, referred to the Programmable Logic (PL), can be used for hardware acceleration of preferably parallel and local algorithms. Global and non-deterministic approaches, such as iterative and dynamic, is better suited for the ARM, or the Processing System (PS). The penalty for crossing between processing platforms is minimal thanks the SoC architecture. Hence, FPGA-acceleration is not limited to the pre-processing stages. However, image processing applications and how to best implement them for GIMME2 is out of scope for this guide. Here the focus is getting acquainted with the hardware and the Xilinx tools - and of course getting GIMME2 up and running a first application.

Before continuing, consider the following prerequisites to complete this guide (the hardware part that is):

- Hardware:
  - GIMME2 (with power source 12-24V)
  - Xilinx Platform Cable USB II
  - USB cable
  - Ethernet cable
  - (For future use: Lenses and lens holders)
- Host computer with:
  - USB
  - Ethernet
  - Xilinx Vivado with license (2014.2 was used for this guide)
  - (Linux environment for advanced steps - U-Boot, device tree and ramdisk)

Before going into more GIMME2 specific section it is worth pointing out sources for more information on the Zynq:

- Xilinx resources, available via the home page http://www.xilinx.com.
  - Product information

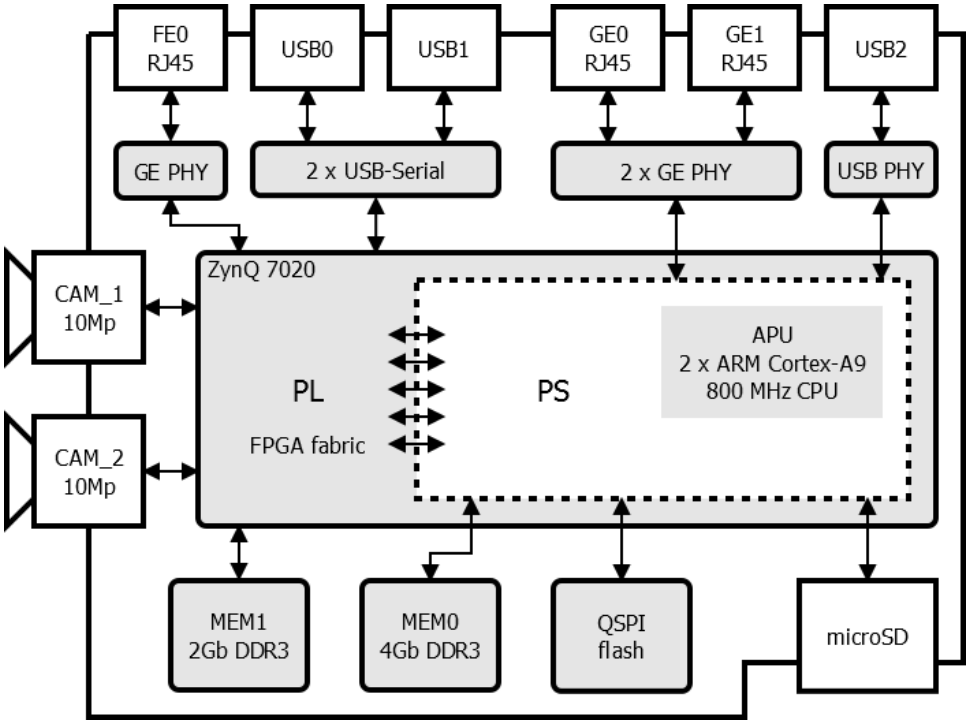


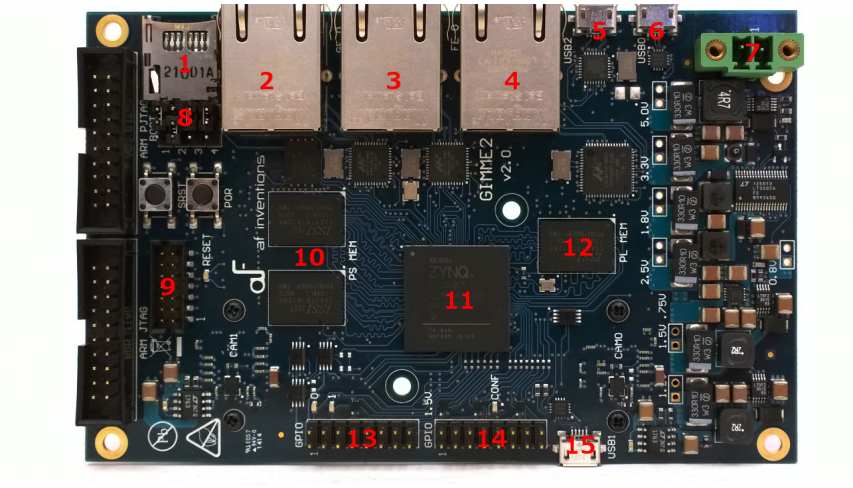
Fig. 2. GIMME2 block diagram

  - User guides and application notes - preferably accessed through DocNav, a tool included in the Vivado tool suite.
  - Training videos
  - Community forum
- The Zynq Book is a book covering all the aspects of the Zynq platform from user perspective. The book is complemented by a set of exercises on the ZedBoard [1].
- ZedBoard provide documentation, reference designs, and training material for several Zynq-based kits [2].

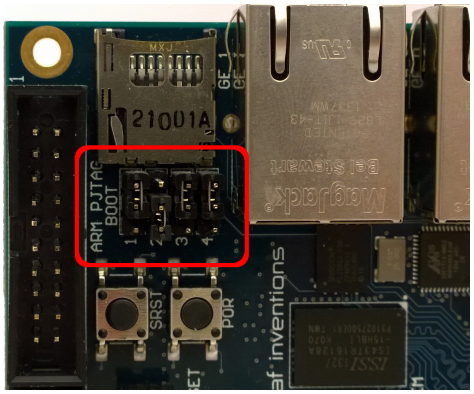
In the next section, the GIMME2 hardware will be described, that is the main components and interfaces/connectors. Following that a basic application will implemented, step-by-step, in accordance with the work-flow of the Vivado tool-suite, similar to the Xilinx UG873 [3] and the Wiki guide [4]. First the PL configuration will be addressed through Vivado. Then the PS application will be implemented and GIMME2 configured to boot Linux in SDK. By the end of the guide, there are sections on how to configure the U-Boot, the ramdisk and the device tree. The corresponding GIMME2 file package contains pre-configured/compiled versions of files and sources created/used throughout this guide [5].

II. Hardware

Figure 2 shows a function diagram of the GIMME2 hardware. The Zynq with the PL and PS being the central component to which the other peripherals, whether on-board components or connectors, are connected. Peripherals are pin-mapped to predefined ports on the Zynq. These may be configured for PS-access through the tool suite. The image sensors, the PL-memory (MEM1), Fast Ethernet (FE0) and the UARTs (USB0, USB1) are connected to the PL solely and need to be configured manually if to be accessed by the PS. Pin-mappings can be found in appendices A and B.



(a) GIMME2 backside. Main components and connectors have been numbered.



(b) Boot mode jumpers. QSPI boot setting.

Figure 3a identifies the main components and connectors on the backside of GIMME2. Below is a listing of the numbered components. The naming is more or less self-explanatory and will be used throughout this guide. The names are also printed on the PCB.

1) micro SD-card
2) Gigabit Ethernet 1 (GE1)
3) GE0
4) FE0
5) USB2
6) USB0
7) Power connector (12-24V)
8) Boot mode jumpers
9) JTAG programmer connector
10) PS-memory
11) Zynq 7020
12) PL-memory
13) 6 GPIO 2.5V (LED 0,1)
14) 8 GPIO 1.5V
15) USB1

The image sensors are mounted on the front side and are hence not listed above. In front of the sensors optic lenses are mounted on lens holders, as seen in figure 1.

With the boot mode jumpers the boot medium can be selected (number 8 in figure 3a and zoomed in figure 3b). In this guide GIMME2 will be setup for JTAG boot and QSPI boot. All jumpers set to 0 designates JTAG boot. To switch to QSPI boot the position of boot jumper 2 is changed to 1, as seen in figure 3b.

In the following sections a simple project will be created and run. This example will toggle the LEDs (pins 0 and 1 of GPIO 2.5V) from a software application running under Linux. To achieve this the PL and PL-PS interface need to be configured, booting of Linux needs to be setup, and a software application running on the PS needs to be written. All these steps, and more, will be covered in the following sections starting with the PL and Vivado. This guide is based on Vivado 2014.2.

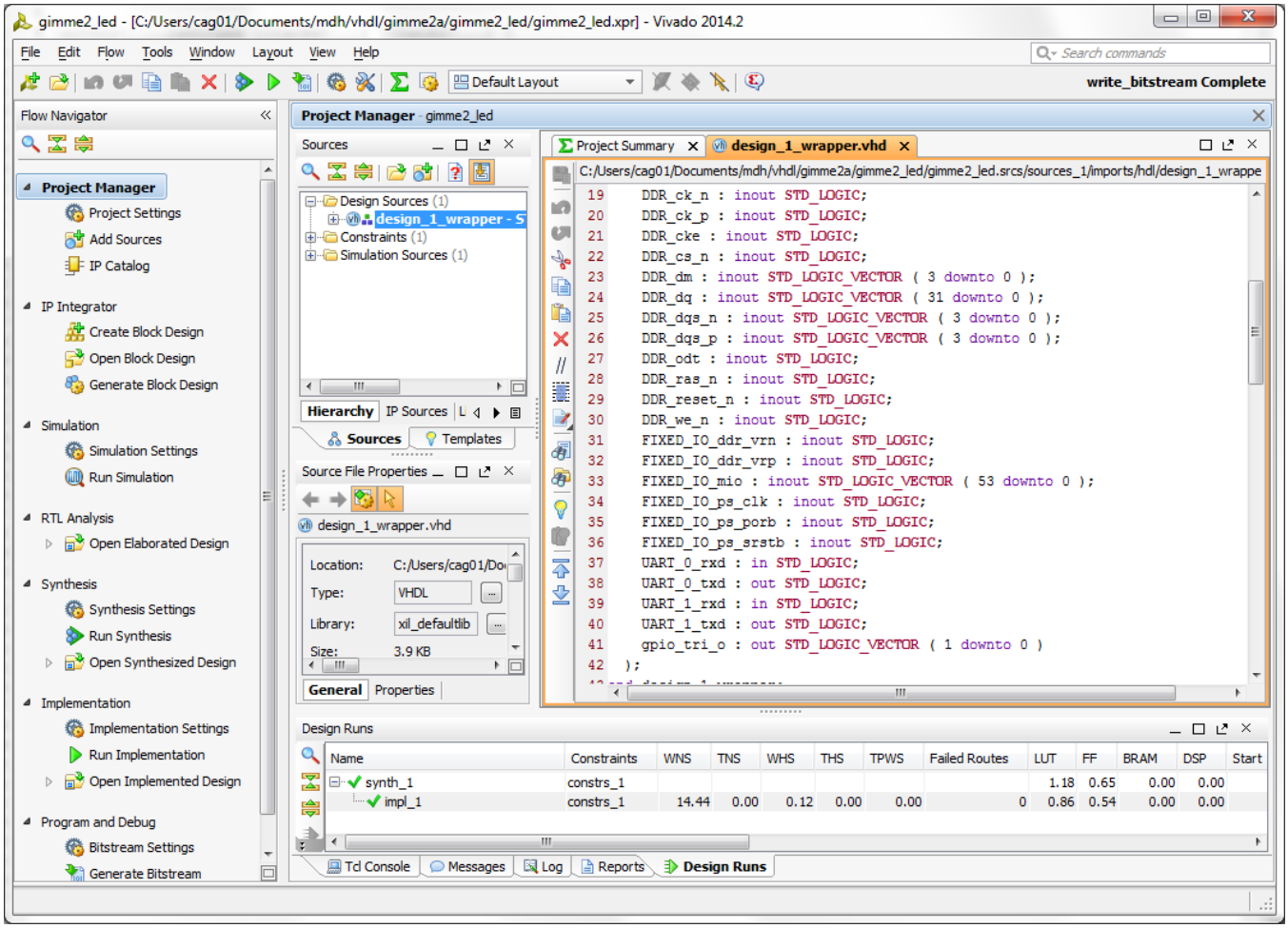


Fig. 3. Xilinx Vivado main view

III. Vivado

First off, start Vivado and select Create New Project. This will start a wizard that will help the user setup the project. Click next to come the project naming dialogue. Name the project (eg. `gimmme2_led`) and select location. Make sure create subdirectory is checked and click next. Project type ⇒ RTL project and check the box not to specify sources. Default part ⇒ search for xc7z020clg484-2 and select the hit. Summary screen ⇒ click finish. This will open the Vivado main view.

The main view of Vivado can be seen in figure 3. To the left is the Flow Navigator. This follows the FPGA design flow from top to bottom, i.e. from source files to generated configuration file (.bit-file). To the right of the Flow Navigator is the sources window showing the project included sources and the file/component hierarchy. Furthest to the right is the main window located, showing the source code, schematics, layout, simulation, debug cores, etc, depending on the stage of the design flow. At the bottom is a field showing project status and information messages during different stages of the design run.

Now it is time to start adding functionality to the empty project. First to be added is a so called block design.

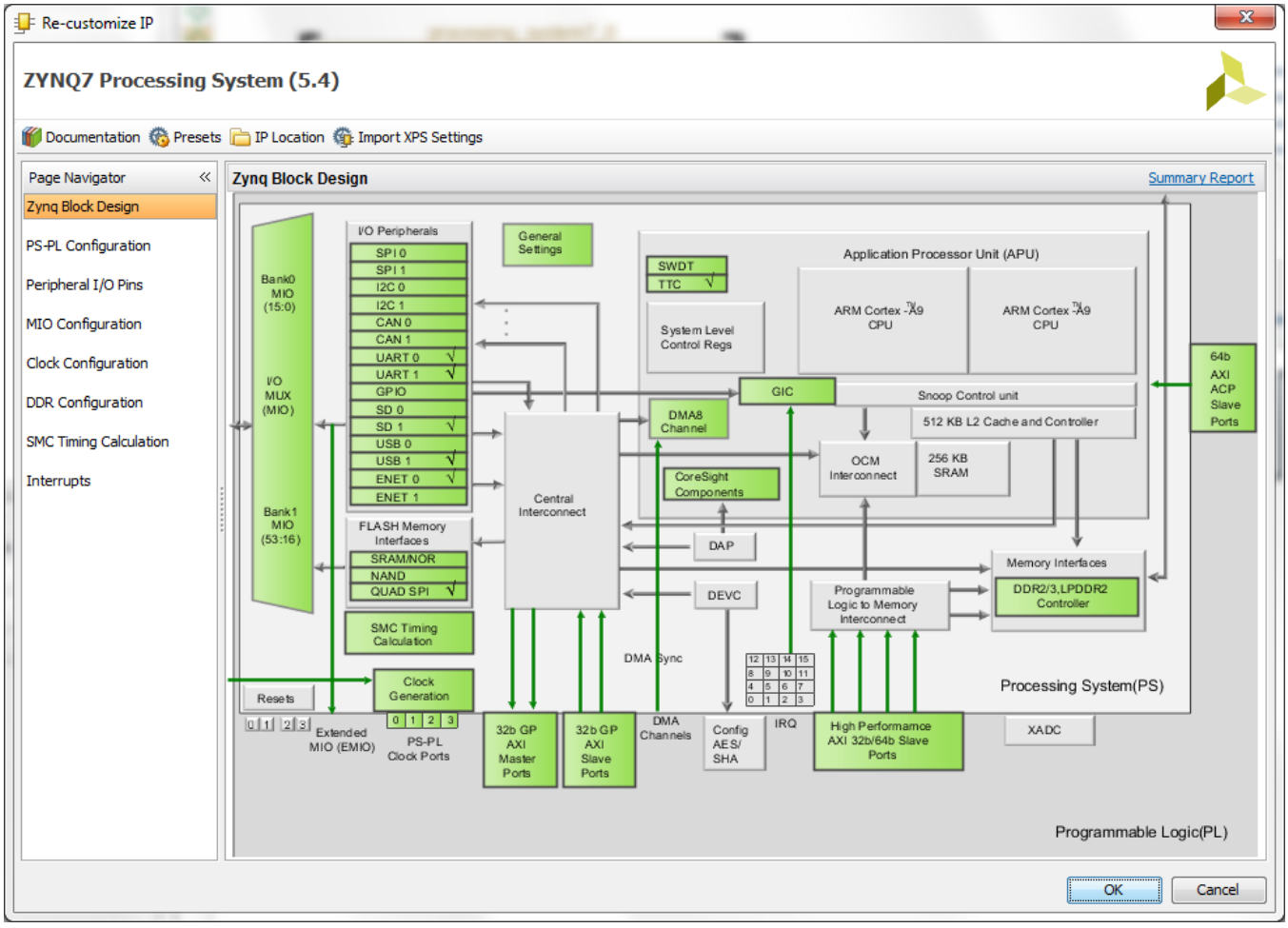


Fig. 4. Zynq processing system configuration window

A. *Block Design*

Block Design is a graphical programming interface where blocks of different functionality, i.e. IP-cores, are added, selected and connected in drag-and-drop fashion. In this tool users are restricted to the functionality of the IP-cores in available library/libraries. New cores can be generated from source but how to do this is out of scope for this guide. (There is a .tcl script to generate the Block Design and skip this part of the guide. Locate `gimme2_tech_files\hdl\design_1.tcl` in the file pack. To run the script type `source design_1.tcl` in the tcl-console, located in the Vivado bottom pane).

In the Vivado Flow Navigator, under IP Integrator, click Create Block Design. Name the design (`design_1`) and set the directory to be local to project. The main window will show the empty Diagram and the `design_1` will be added to the design hierarchy. It is now time to add IP-blocks, as suggested by the block design tool. There will be several suggestions from the program to facilitate the programming. Click Add IP, either in the tool tip or from the icon, search for 'zynq' and select ZYNQ7 Processing System, an a new block is added to the design. At the same time the design assistant will suggest to run an automatic block connection - do that and click OK. This will make the DDR and FIXED_IO external. Corresponding hardware is connected to pre-defined pins on the Zynq and will be mapped automatically.

For Zynq devices the ZYNQ7 Processing System-block is possibly the most central block as it defines the PL-to-PS-interface, PS-memory configuration, clocks, peripherals etc. As this example will control the

LEDs from the PS this block is mandatory. Double-click on the block in order to start the configuration. There will now be a window showing a block diagram of the Zynq, see figure 4. From here it is possible to double-click on the green boxes in order to change the corresponding setting. However, to keep it simple, the navigator pane to the left will be used as a reference in this guide. To get back to this view select Zynq Block Design from the navigator pane.

Select Peripheral I/O Pins. The appearing dialog allows for selection of signals/peripherals and pin-mapping. Apply the settings listed below.

- Set the I/O to LVCMOS 1.8V for Bank 0 and 1
- Quad SPI Flash, expand with the '+', select Single SS 4bit IO and check the Feedback Clock. The SPI-flash will contain the boot image with all the necessary information to boot Linux. More on that in sections IV-C and IV-D.
- ENET 0, expand and also check MDIO.
- UART 0 and from the possible pin-mappings chose EMIO farthest to the right. The UART on GIMME2 does not follow the any Zynq standard placement and needs to be mapped externally.
- UART 1 on EMIO (not used in this guide)
- SD 1 first location, pin 10-15 (not used in this guide but configured for future use)
- TTC0 - Timer 0 - Must be enabled in order for Linux to boot though the signals are not connected in the block design.

Click on MIO configuration. Scroll down to I/O Peripherals and set the IO Type for ENET 0 as shown in figure 5.

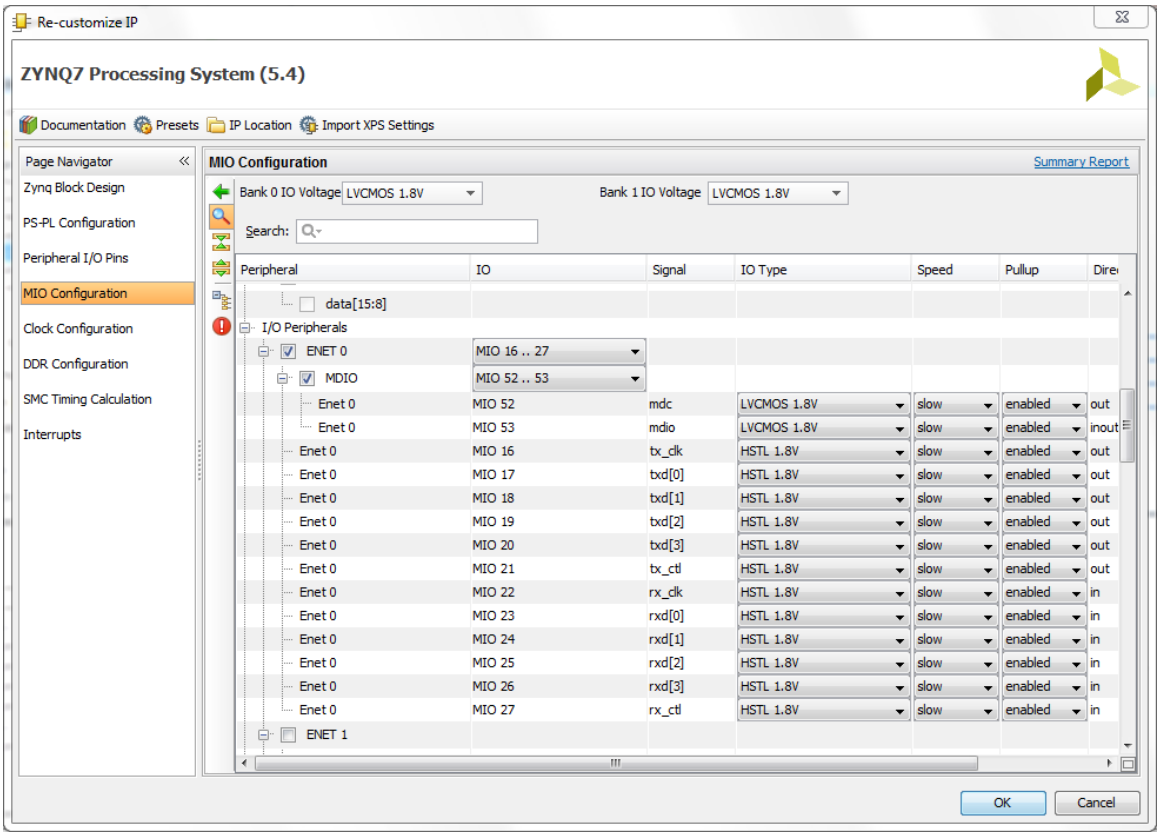


Fig. 5. Zynq I/O configuration for Ethernet 0

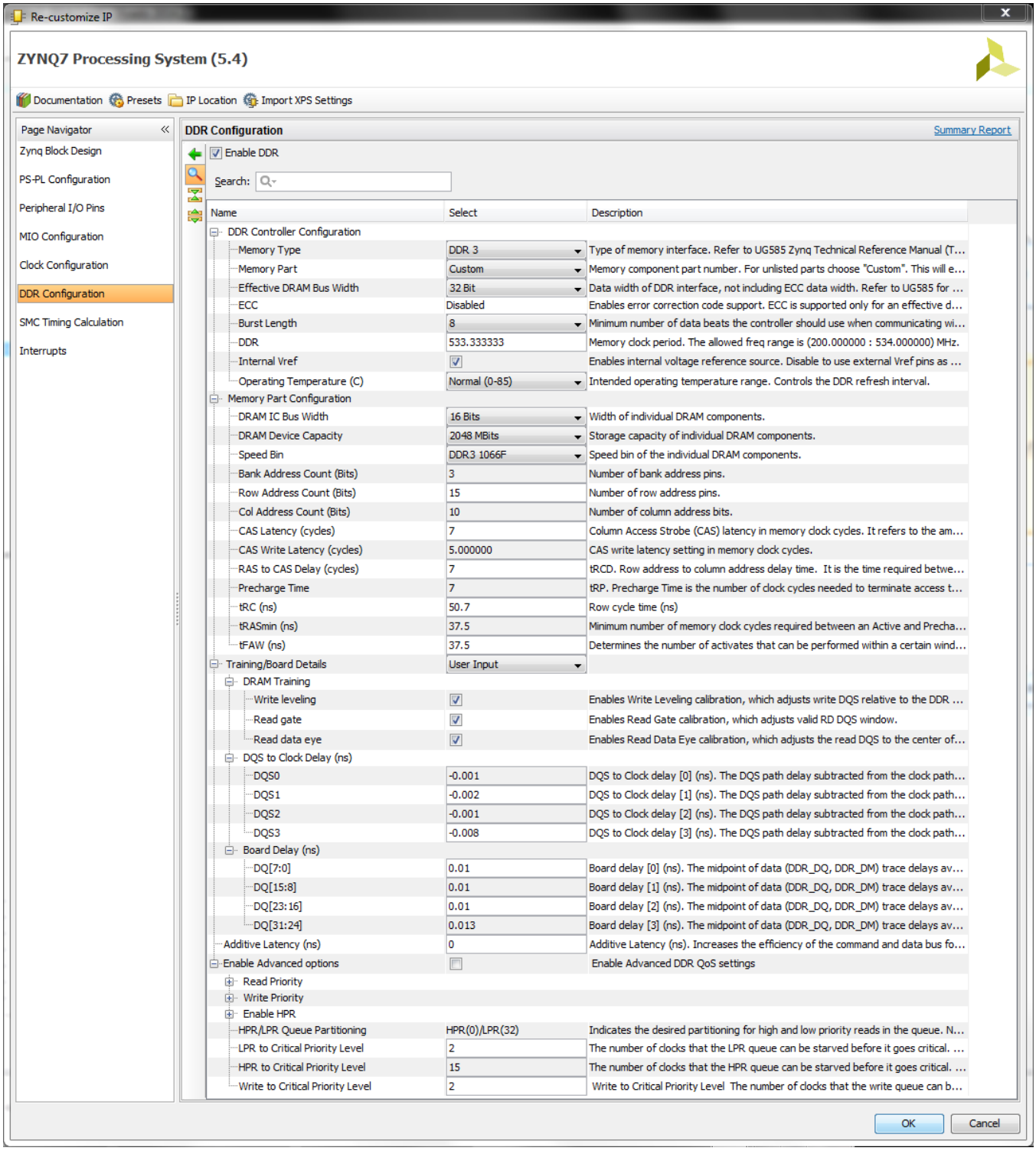


Fig. 6. PS memory configuration

Click OK and then re-open the block. Now the selected I/O peripherals should be checked as in figure 4. To enable more peripherals refer to appendix B for placement selection.

Select DDR configuration. Expand all options and set the values in accordance with figure 6. Click on OK to go back to the block design window.

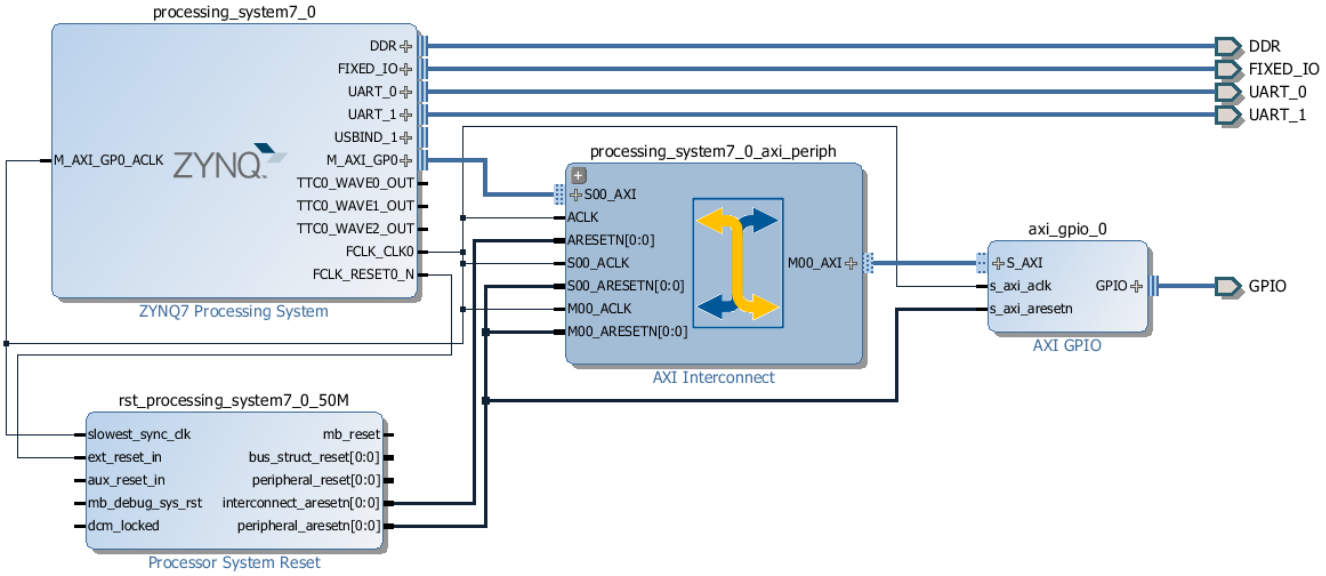


Fig. 7. The finished block design

Now there should be a ZYNQ7 Processing System-block, where DDR and FIXED_IO have external connections, i.e. connections that will become interface ports to the block design. For the Zynq-block there should also be not yet connected ports for the UARTs. Now, right-click on the UART_0 and select make external. This will create an external binding for UART_0 just as for the DDR and FIXED_IO. Also make UART_1 external to complete the UART configuration.

To complete the block design something is needed to carry data between the PS and the PL. Worth pointing out at this point is that in the ZYNQ7 Processing System-block the M_AXI_GP0 interface (under PS-PL Configuration $\Rightarrow$ GP Master AXI Interface) is enabled by default. This is a general purpose data bus and will be used for communication between the PL and PS. Click on Add IP (the icon market with a plus directly to the left of the diagram) $\Rightarrow$ search for axi_gpio $\Rightarrow$ select AXI_GPIO. This will add a new block to the design, a block for GPIO over AXI. Once again the designer assistant will suggest automatic connections. Click OK and ensure that the S_AXI of the AXI_GPIO-block is connected to the M_AXI_GP0 of the Zynq-block. Automatically a reset-block and a AXI Interconnect-block are added to the design. Double-click on the AXI_GPIO-block to configure it. Check All Outputs (the LEDs will be controlled not read) and set the width to 2 (as there are 2 LEDs). Click OK. To finish off, right click on the GPIO-connection in the diagram, and make it external.

The resulting block design should look something like figure 7. There is always the possibility to move blocks around to make the design easier to follow. Before closing the block design, press the Validate Design button (or hit F6) and ensure that there are no errors. Also take a look at the Address Editor tab. There should be a field for the AXI_GPIO-instance defining the offset memory address. This address can be used to access the registers of the core, in this case the control and data register, to setup the core (eg. input/output) and read/write data, from the PS (software application).

With the DDR and FIXED_IO defined it is now time to map the UART and GPIO-signals. This is done in VHDL. Save and close the block design to return to the Vivado main view.

B. VHDL-wrapper and Pin mapping

Back in Vivado the hierarchy now shows the new design_1 design source. Verify that the target language is set to VHDL (or verilog if that is preferred) by clicking on Project Settings ⇒ Target language. Now, right-click on design_1 and select Create HDL Wrapper... Check Copy generated wrapper to allow user edits. This will add a new design source, design_1_wrapper, as top-module. In the wrapper the block-design is declared and instantiated. More HDL-components can be added by editing the file. This example, however, does not require added logic but the external ports need to be pin-mapped.

Looking at the entity-declaration of the wrapper, the top-level ports are DDR, FIXED_IO, UART and GPIO, out of which the later two do not follow the Zynq standard, see figure 8. These need to be directed to the pins connected to corresponding hardware circuit. Vivado provides a graphical interface for this, but in this case it is easier to manually create and edit a constraint file. Click Add Sources ⇒ Add or Create Constraints ⇒ Create File... Enter the file name (`gimme2_pins`) and click OK. Click Finish. In the sources list, under the constraints folder, in the constraints set constrs_1, the created file can be found. Double-click to edit. Add the following text (or copy the file from the file package - `gimme2_tech_files/hdl/gimme2_leds.xdc`):

```
set_property IOSTANDARD LVCMOS25 [get_ports UART_1_rxd]
set_property IOSTANDARD LVCMOS25 [get_ports UART_1_txd]
set_property IOSTANDARD LVCMOS25 [get_ports UART_0_rxd]
set_property IOSTANDARD LVCMOS25 [get_ports UART_0_txd]

set_property PACKAGE_PIN V14 [get_ports UART_1_rxd]
set_property PACKAGE_PIN U7 [get_ports UART_1_txd]
set_property PACKAGE_PIN W20 [get_ports UART_0_rxd]
set_property PACKAGE_PIN AA22 [get_ports UART_0_txd]

set_property IOSTANDARD LVCMOS25 [get_ports {gpio_tri_o[1]}]
set_property IOSTANDARD LVCMOS25 [get_ports {gpio_tri_o[0]}]
set_property PACKAGE_PIN AB15 [get_ports {gpio_tri_o[1]}]
set_property PACKAGE_PIN AB14 [get_ports {gpio_tri_o[0]}]
```

This will specify the pin location and the interface I/O type. It is imperative that the naming between the wrapper ports, figure 8, and the constraint file agree.

Now all is setup in order to run synthesis, implementation and bit-file generation. After each step the design can be examined/verified on a schematic level for the synthesized design and layout level for the implemented design. Logic utilization and timing can also be investigated (and addressed by updating the code). When developing components it is recommended to precede synthesis with a simulation of a well written test bench in order to verify the logic circuit at a software level. Later, on-the-chip is available

```
34     FIXED_IO_ps_clk : inout STD_LOGIC;
35     FIXED_IO_ps_porb : inout STD_LOGIC;
36     FIXED_IO_ps_srstb : inout STD_LOGIC;
37     UART_0_rxd : in STD_LOGIC;
38     UART_0_txd : out STD_LOGIC;
39     UART_1_rxd : in STD_LOGIC;
40     UART_1_txd : out STD_LOGIC;
41     gpio_tri_o : out STD_LOGIC_VECTOR ( 1 downto 0 )
42   );
43 end design_1_wrapper;
```

Fig. 8. The unmapped external wrapper signals

through Hardware Manager, but this requires debug logic to be added to the design, to examine selected signal for a specified time/number of samples.

When the bit-file generation has finished the design can be exported to the Software Development Kit (SDK). The hardware definition will set the environment for the software development. Select File ⇒ Export ⇒ Export Hardware. Ensure that Include bitfile is checked and select Local to Project. Now launch SDK by clicking File ⇒ Launch SDK and select Local to Project. SDK will automatically detect the exported hardware definition.

IV. SDK

Xilinx SDK is, as the name implies, an environment for developing and debugging software applications. There are also tools for low-level configuration for the Zynq platform such as boot setup, both standalone and Linux, from JTAG or SPI-flash. In this section SDK will be used for a simple LED blink application, to generate a boot image and for programming GIMME2 to boot Linux from the SPI-flash. This guide follows selected parts of UG873 [3] and the Xilinx Wiki [4]. Following this projects are added to run OpenCV in context of C and C++.

If not already started, launch SDK and navigate to the workspace to where the hardware definition has been exported, for example: `<path>\gimme2_led\gimme2_led.sdk`. This should start SDK, opening the main view, see figure 9. To the left is the Project Navigator, here showing the hardware platform. More projects will be added to the workspace later. In the centre is the main window, i.e. the programming editor etc. Looking at the bottom there are several tabs; problems, console, terminal, etc. Information about the exported hardware can be found in `system.hdf` and `ps7_init.html` and the `ps7_init.tcl` script will be used later for the low-level start-up of the Zynq.

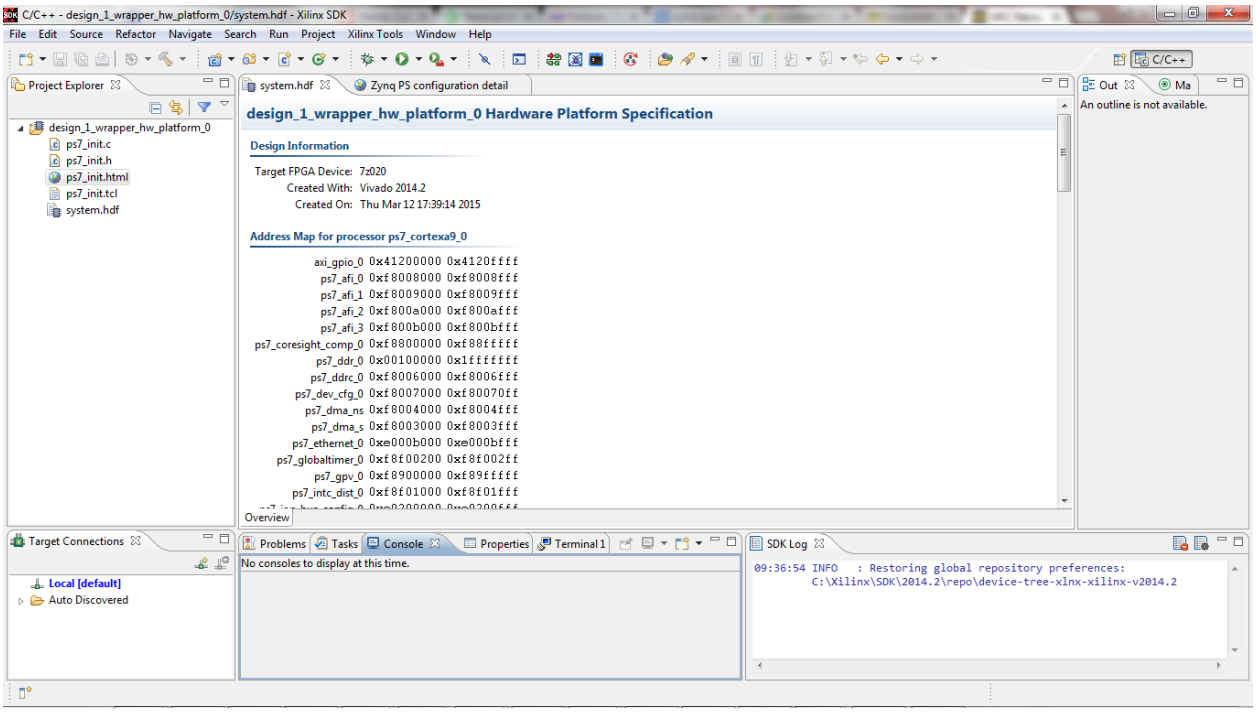


Fig. 9. Xilinx SDK main view

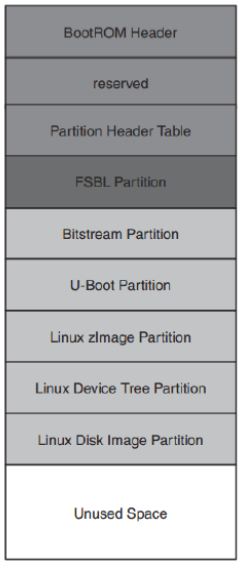


Fig. 10. The boot image file structure

Either Linux can be loaded over JTAG or from the SPI memory. JTAG is useful for debugging, but the process needs to be repeated every time the system is restarted. In this guide, as we have a known working configuration, GIMME2 will be configured to boot Linux from the SPI-flash memory and hence retain data over power-cycles. This involves the following steps - start GIMME2 in JTAG-mode, program the flash memory, restart GIMME2 in SPI-mode. Both JTAG and SPI methods are covered in UG873 [3]. However, before programming the flash, all data required to boot, need to be packed into a single file - the boot image.

A. Boot Image

Figure 10 shows the structure and contents of the boot image file. In SDK, there is a tool for creating a boot image, taking care of the partitioning, but not the content. The following files are required:

- FSBL - First stage bootloader. This will be generated next, see section IV-B.
- Bitstream - The configuration file for the FPGA. This war generated in Vivado and exported along with the hardware definition to SDK earlier in this guide. It should be located in `<sdk_path>\design_1_wrapper_hw_platform_0\design_1_wrapper.bit`, where the SDK workspace location, `<sdk_path>`, is typically `<path>\gimme2_led\gimme2_led.sdk`.
- U-Boot - Universal bootloader, which is launch by the FSBL. A compiled version, `u-boot.elf`, is provided in the file package. To configure and build U-boot from scratch refer to section V.
- Linux zImage - The Linux kernel, launched by U-boot, 2014.2 (`uImage.bin`). This and newer releases are available from Xilinx [6].
- Device tree - The device tree is a listing of the hardware for the OS. A pre-compiled device tree, `devicetree.dtb`, is provided in the file package. How to generate a device tree from scratch is described in section VI.
- Linux Disk Image - Ramdisk - The linux file system. A pre-configured version, `uramdisk.image.gz` is provided in the file package. For more on the ramdisk refer to VII. The ramdisk is also part of the Linux release [6].

To summarize, the bitstream has been generated, preconfigured versions of the U-boot, zImage, device tree and ramdisk are available. The only item missing from the list is the FSBL. This can be generated in SDK. A complete boot image is also available in the file package.

B. FSBL and BSP

This is the SDK explanation of the FSBL - *"First Stage Bootloader (FSBL) for Zynq. The FSBL configures the FPGA with HW bit stream (if it exists) and loads the Operating System (OS) Image or Standalone (SA) Image or 2nd Stage Boot Loader image from the non-volatile memory (NAND/NOR/QSPI) to RAM (DDR) and starts executing it. It supports multiple partitions, and each partition can be a code image or a bit stream."*

A board support package (BSP) is a collection of libraries and drivers which form the base for higher level software applications, and is required by the FSBL.

To generate a FSBL, click File ⇒ New ⇒ Application Project. Name the project (`fsbl`). If the default location is chosen the hardware platform `design_1_wrapper_hw_platform_0` is set by default, processor `ps7_cortexa9_0`, language C, standalone and create new BSP (`fsbl_bsp`). Click Next and select Zynq FSBL from the list. Click finish to generate the FSBL.

Now two new projects appear in the project explorer, the fsbl and fsbl_bsp, as seen in figure 11. If the project folders are expanded the executable `fsbl.elf` and information and documentation for the BSP can be found. Note also that the fsbl reference the bsp.

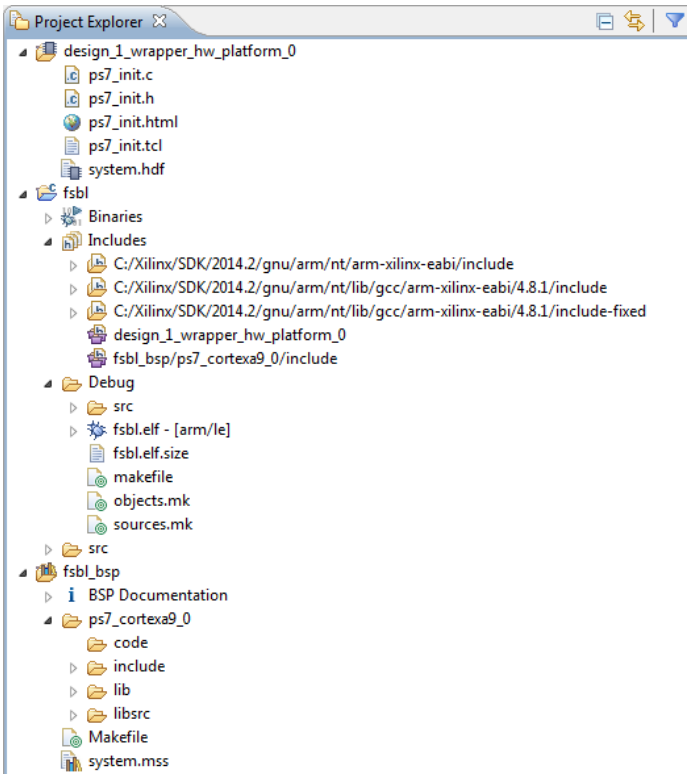


Fig. 11. The SDK Project Explorer with the hardware platform, the first stage bootloader and the board support package projects.

Now that all the required files for the boot image are available the boot image can be created.

C. Boot Image Creation

First, download and copy the GIMME2 files from the file package to a suitable location, for example: `<sdk_path>\bootimage`

To open the create a boot image dialogue, click Xilinx Tools ⇒ Create Zynq Boot Image. Create new BIF file and browse to a suitable location (the created folder). The BIF-file is the file listing for the boot image and can later be used if reconfiguration of the boot image if needed. Now it is time to add files to the boot image in accordance to figure 10.

First, add the boot loader. Click Add. Browse to `<sdk_path>\fsbl\Debug\fsbl.elf` and select it. Ensure that the partition type is set to bootloader and press OK.

The bitstream, `<sdk_path>\design_1_wrapper_hw_platform_0\design_1_wrapper.bit`, is next. Select it and check that the partition type is set to datafile (default).

The U-Boot, `<sdk_path>\bootimage\u-boot.elf`, is added in the same way.

Next up is the Linux kernel, `<sdk_path>\bootimage\uImage.bin`. Set Offset: 0x00450000. This offset is required for U-boot to be able to locate the file. This offset is set in the U-boot configuration, see section V.

Add the device tree, `<sdk_path>\bootimage\devicetree.dtb`, with offset 0x00780000.

Finally add the ramdisk, `<sdk_path>\bootimage\uramdisk.image.gz`, offset 0x00785000.

Specify the output directory (`<sdk_path>\bootimage`) and hit Create Image. This will generate the file BOOT.bin.

The resulting `output.bif` should look like this:

```
the_ROM_image:
{
  [bootloader]<sdk_path>\fsbl\Debug\fsbl.elf
  <sdk_path>\design_1_wrapper_hw_platform_0\design_1_wrapper.bit
  <sdk_path>\bootimage\u-boot.elf
  [offset = 0x00450000]<sdk_path>\bootimage\uImage.bin
  [offset = 0x00780000]<sdk_path>\device_tree_bsp_0\devicetree.dtb
  [offset = 0x00785000]<sdk_path>\bootimage\uramdisk.image.gz
}
```

A pre-configured `output.bif` is included in the file package. However, the paths need to be updated according to the current project.

Now that the boot image has been created it is time to program the flash memory.

D. QSPI-flash programming and boot

It is time to start up GIMME2. But before connecting the power, set jumper 2 to select JTAG boot mode, i.e. all jumpers are set to 0, not as in figure 3b. Connect a USB cable to the USB0 connector (for serial communication). Connect the Xilinx Platform Cable USB II to the JTAG programmer connector. Power up GIMME2 (12-24V).

Program the FPGA with the bitstream file in order to map the UART-signals, as serial communication is required later. Programming is initiated either by pressing the programming icon or by selecting Program FPGA from the Xilinx Tools menu. SDK will find/select the exported bitstream by default. Press program. When done the LED marked CONF on GIMME2 will be lit.

In SDK, make a serial connection from the terminal window (or use an external terminal such as TeraTerm) with the following settings:

- Baud rate: 115200
- Data: 8 bit
- Parity: none
- Stop: 1 bit
- Flow control: none

Open the XMD Console (via icon or menu) and type the commands below. Before executing the final command `con` ensure to have a serial terminal ready as U-Boot will boot within seconds unless stopped. The commands initializes the ARM, downloads the U-Boot executable and the boot image to the PS-memory.

```
connect arm hw
source <sdk_path>/design_1_wrapper_hw_platform_0/ps7_init.tcl
ps7_init
ps7_post_config
dow <sdk_path>/bootimage/u-boot.elf
dow -data <path>/bootimage/BOOT.bin 0x08000000
con
```

In the terminal window, first stop U-boot from auto-booting (hit any key), then type:

```
sf probe 0 0 0
sf erase 0 0x01000000
sf write 0x08000000 0 0xFFFFFF
```

This stores the boot image in the QSPI memory. Now GIMME2 can be powered off. Move jumper 2 to enable QSPI-boot and power up the system. This should boot Linux. The boot sequence can be seen in the terminal window. Use u:root p:root to login.

Appendix C shows the output from the XMD Console and the terminal for the programming procedure.

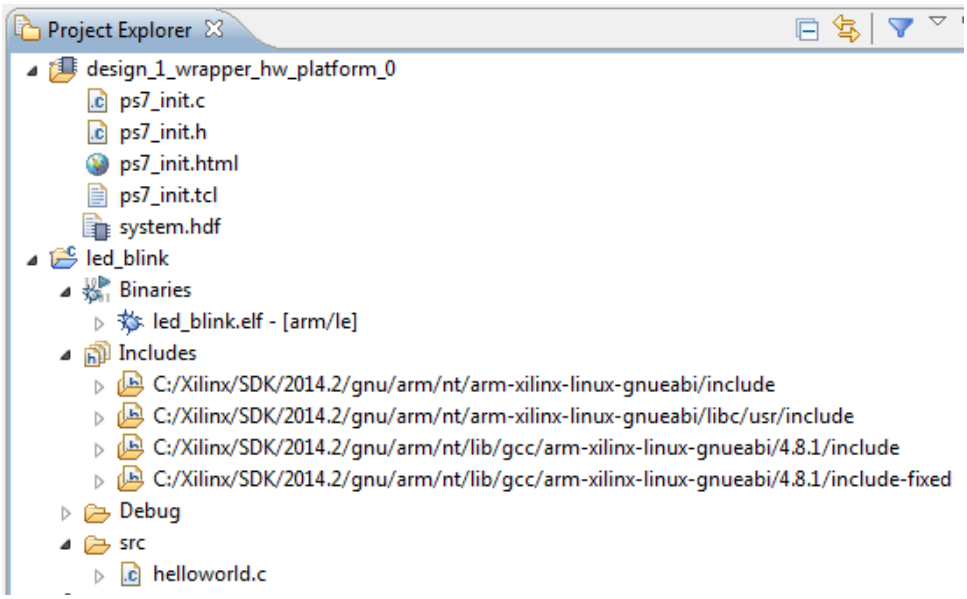


Fig. 12. SDK Project Explorer, hardware platform and application project structure

E. Linux Application

To start off, in order to verify that everything works, the first application will be ”hello world”. This will later be modified to blink the LEDs. Click File ⇒ New ⇒ Application Project. In the dialogue name the project (`led_blink`), select OS platform Linux, leave the rest as default. Click Next and select Linux Hello World. Click Finish to create the project, which should appear in the Project Explorer.

Expand the led_blink application project. There are folders for binaries, includes, debug and src, see figure 12. Under binaries there should already exist an .elf file, an arm executable file, as SDK is setup to auto-build with source changes.

F. Download and run application

In order to run the application it first has to be downloaded to GIMME2. First, remember that the ENET 0 was enabled/configured in the Zynq block. Secondly, it is worth pointing out that the ramdisk, now part of the boot image, has a number of start-up scripts, settings and applications. Linux is setup with a static ip for eth0, 192.168.1.10, and a SSH-server (port 22). Hence, downloading/running is a simple job using any SSH-client (eg. Bitvise under windows). SDK offers debug-features so it can make sense to setup for more complex applications (Run ⇒ Debug/Run Configuration). This is covered in UG873 [3].

- Connect an Ethernet cable between GE0 on GIMME2 and host computer
- Setup host computer IP - 192.168.1.100 (for example)
- Connect to GIMME2 with SSH-client (IP: 192.168.1.10, port: 22, u: root, p: root)
- Transfer `led_blink.elf` to GIMME2

On GIMME2, change the file permissions for the program in order to make it executable and run it:

```
root@zynq:~# chmod +x led_blink.elf
```

```
root@zynq:~# ./led_blink.elf
Hello World
```

G. LED blink application

1) Memory Manipulation: Now the application will be updated to control the LEDs on GIMME2. The AXI_GPIO-core connects the PS with the PL through the PS-memory. In SDK the address map for the hardware can be found in the system definition file (`system.hdf`). The address range for axi_gpio_0 is 0x41200000-0x4120FFFF. Next, a register map for the memory space is needed. As this is core specific refer to the documentation (possibly through Xilinx DocNav) for the core in question - in the case of axi_gpio PG144 [7]. Here two registers and offsets can be noted, GPIO_DATA 0x00 and GPIO_TRI 0x04. The data register contains the current value for the GPIO-pins. The other dictates the direction, 0 means output/write and 1 input/read. Remember, the AXI_GPIO-block was configured with two pins, i.e. 0 and 1, set as output. So, to light a LED set the corresponding bit of the data register. As the core was already configured as output in Block design there is no need to write to the GPIO_TRI register.

To be able to manipulate the memory, first it has to be opened (with system call `open`) and the specific range mapped (`mmap`). Perform the reads/writes. Close the file descriptor.

Below is the example source code, also available in the file package `gimme2_tech_files/src/led_blink.c`:

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>


#define GPIO_BASE_ADDR 0x41200000
#define GPIO_HIGH_ADDR 0x4120FFFF
#define GPIO_DATA_OFF 0x00
#define GPIO_TRI_OFF 0x04

#define REG_WRITE_T(addr, off, val, type) (*(volatile type*)(addr+off)=(val))
#define REG_READ_T(addr,off,type) (*(volatile type*)(addr+off))



int main()
{
  int fd = open( "/dev/mem", O_RDWR | O_SYNC);
  if (fd < 0)
  {
    perror("Failed to open /dev/mem!\n");
    return -1;
  }

  int map_len = GPIO_HIGH_ADDR-GPIO_BASE_ADDR;

  unsigned char* gpio_base = (unsigned char*)mmap(NULL, map_len, PROT_READ | PROT_WRITE,
    MAP_SHARED, fd, (off_t)GPIO_BASE_ADDR);

  if(gpio_base == MAP_FAILED)
  {
    perror("Mapping memory for absolute memory access failed.\n");
    close(fd);
    return -1;
  }
```

```
36
37
38    int i;
39    for (i=0;i<10;i++)
40    {
41      REG_WRITE_T(gpio_base, GPIO_DATA_OFF,1,int);
42      sleep(1);
43
44      REG_WRITE_T(gpio_base, GPIO_DATA_OFF,2,int);
45      sleep(1);
46
47      REG_WRITE_T(gpio_base, GPIO_DATA_OFF,3,int);
48      sleep(1);
49
50      REG_WRITE_T(gpio_base, GPIO_DATA_OFF,0,int);
51      sleep(1);
52    }
53
54    munmap((void *)gpio_base, map_len);
55    close(fd);
56
57      return 0;
58  }
```

Update/replace the `helloworld.c` with `led_blink.c`, build the project, download the .elf file to GIMME2 and run. Watch the LEDs blink.

As an exercise, connect the LED1 and LED2 pins, configure one of the pins as input (set the corresponding bit in GPIO_TRI), write to the output pin and read back (REG_READ_T) the value from the input pin.

2) Driver: There is a second way to access the register space for PL-core - through kernel drivers instead of user space. This requires that the device tree, see section VI, provides a binding point for the Linux kernel to the hardware. If a device tree is generated after the hardware is defined, as with the accompanying `devicetree.dtb`, these bindings will be defined. Xilinx provides several Linux drivers [8].

Here is the same example for lighting LED0 from user space through the GPIO driver:

```
> echo 254 > /sys/class/gpio/export
> echo out > /sys/class/gpio/gpio254/direction
> echo 1 > /sys/class/gpio/gpio254/value
```

Write 0 to value to make it go dark again. More on this and how to make user space applications can be found in the Xilinx Wiki [9].

H. OpenCV Template

OpenCV (Open Source Computer Vision) [10] is included in SDK and ready to be used in application projects. Just as with hello world, there exists a basic OpenCV template. Select File ⇒ New Application Project. Name the project (`opencv_test`), select Linux as OS Platform. Click Next. Select ”OpenCV Example Application”. Note the text: *A simple, introductory OpenCV program. The program reads an image from a file, inverts it, and displays the result. Users need to include the path to openCV libraries in LD_LIBRARY_PATH on the target environment*. Click Finish and the project is generated.

As pointed out the path to the OpenCV library has to be set before the application can be run. However, first the OpenCV libraries need to be present on GIMME2. Also needed are C libraries referenced by OpenCV. Ideally these would be added to the rootfs . This can be done using the PetaLinux tools, which require Linux. Another concern is the growing size of the ramdisk, as it needs to fit in the QSPI flash memory. The alternative is to copy the libraries to an SD-card, where memory size is an non-issue. The init scripts on the ramdisk are supposed to mount an SD-card by default at start-up to `/media/card/`. If this for some reason fails mount the SD-card manually:

```
root@zynq:/# mount /dev/mmcblk0p1 /media/card
```

This assumes that the SD-card is formatted and mounts the first partition.

Copy the following libraries (depending on your Xilinx installation path) from the host computer (sublibraries can be excluded).

```
C:\Xilinx\SDK\2014.2\gnu\arm\nt\arm-xilinx-linux-gnueabi\libc\usr\lib
C:\Xilinx\SDK\2014.2\data\embeddedsw\ThirdParty\opencv\lib
```

to the mounted SD-card on GIMME2, eg:

```
/media/card/arm-xilinx-linux-gnueabi_lib
/media/card/opencv_lib
```

With the following command the library paths are exported:

```
export LD_LIBRARY_PATH=/media/card/opencv_lib:
/media/card/arm-xilinx-linux-gnueabi_lib
```

Download your .elf file to GIMME2. Change the file permissions to make it executable:

```
chmod +x opencv_test.elf
```

and run it

```
./opencv_test.elf img.bmp
```

where `img.bmp` is an arbitrary input image. Have a look at the project source file and the output, `result.bmp`.

I. OpenCV Face Detection and C++

The following C++ example performs face detection using the pre-trained Haar classifier. It reads an colour image `infile.jpg`. It also assumes that the file `haarcascade_frontalface_alt2.xml` [11] containing the pre-trained classifier for faces has been copied to `/media/card/`. The original source can be found at the OpenCV by Examples blog [12].

```
#include <opencv2\objdetect\objdetect.hpp>
#include <opencv2\highgui\highgui.hpp>
#include <opencv2\imgproc\imgproc.hpp>

#include <iostream>
#include <stdio.h>

using namespace std;
using namespace cv;
```

```
10
11 int main(int argc, char** argv)
12 {
13   Mat image;
14   image = imread("infile.jpg", IMREAD_COLOR);
15
16   // Load Face cascade (.xml file)
17   CascadeClassifier face_cascade;
18   face_cascade.load("/mnt/haarcascade_frontalface_alt2.xml");
19
20   // Detect faces
21   std::vector<Rect> faces;
22   face_cascade.detectMultiScale(image, faces, 1.1, 3, 0, Size(30, 30), Size(100, 100));
23
24   // Draw circles on the detected faces
25   for (int i = 0; i < faces.size(); i++)
26   {
27     Point center(faces[i].x + faces[i].width*0.5, faces[i].y + faces[i].height*0.5);
28     ellipse(image, center, Size(faces[i].width*0.5, faces[i].height*0.5), 0, 0, 360, Scalar
       (255, 0, 255), 4, 8, 0);
29   }
30
31   imwrite("output.jpg", image);
32
33   return 0;
34 }
```

However, this cannot simply be added to the previous application project as the target language was C. Either modify the the settings for the project or create a new one and select C++ as target. As there is no template for OpenCV the libraries need to be added in the project settings. Right click on the project and select C/C++ Build Settings.

Go to ARM Linux g++ compiler → Directories and add the following paths (or similar depending on installation directory and version):
`"C:\Xilinx\SDK\2014.2\data\embeddedsw\ThirdParty\opencv\include"`
`"C:\Xilinx\SDK\2014.2\data\embeddedsw\ThirdParty\opencv\include\opencv"`

Next, navigate to ARM Linux g++ linker → Libraries. Add the following libraries:

```
avformat
avcodec
avutil
swscale
opencv_core
opencv_imgproc
opencv_highgui
```

Also add the Library search paths:

```
"C:\Xilinx\SDK\2014.2\data\embeddedsw\ThirdParty\opencv\lib"
"C:\Xilinx\SDK\2014.2\gnu\arm\nt\arm-xilinx-linux-gnueabi\libc\lib"
```

Build the project, download the .elf file to GIMME2, change the file permissions, download a suitable test image, rename it to `infile.jpg`, run the program and examine the `output.jpg`.

The following sections on how to configure U-Boot and make changes to the ramdisk are optional and require a Linux environment.

V. U-Boot

Das U-Boot is a universal open-source boot loader primarily used for embedded systems [13]. In this section U-Boot will be configured and built specifically for GIMME2 to generate the `u-boot.elf` used to boot GIMME2. A compiled version of this file is available in the file package, but if further configurations are required, this guide shows how to generate it from source.

Firstly, in order to build U-Boot, a Linux environment is required. Either access a computer running Linux, setup a dual-boot or use a virtual machine eg. Ubuntu under Virtualbox. Secondly, a cross-compilation tool chain is needed to be able to build for an ARM architecture. For this guide the Mentor CodeBench GNU Toolchain was used, http://www.xilinx.com/member/mentor_codebench/xilinx-2011.09-50-arm-xilinx-linux-gnueabi.bin. However, cross-compilation may be included in the PetaLinux Tools.

Xilinx maintain their own U-Boot project [14] with a corresponding repository https://github.com/Xilinx/u-boot-xlnx.

U-Boot needs to be configured specifically for GIMME2. In the main README for U-Boot the following can be read:

```
if (a similar board exists) { /* hopefully... */
   cp -a board/<similar> board/<myboard>
   cp include/configs/<similar>.h include/configs/<myboard>.h
```

Luckily there exist configurations for similar boards, eg. ZC702 and Zed. With these as reference the following files were created/modified (the files can be seen in appendix D and are available in the file package).

- `<path to u-boot>/include/configs/zynq_gimme2.h` - the main configuration file. First are defines for the GIMME2 hardware. Then the Zynq extra environment settings were updated with new offsets for qspi boot to accomodate larger files (fsbl, bit-stream, u-boot). Note that no other boot method than JTAG and QSPI has been attempted.
- `<path to u-boot>/board/xilinx/dts/zynq-gimme2.dts` - a copy of reference system with name changes.
- `<path to u-boot>/boards.cfg` - copy rows from reference system and change name to `zynq_gimme2`

Now U-Boot can be built for GIMME2 using the following command:
`make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- zynq_gimme2`
Rename the resulting `u-boot` to `u-boot.elf`.

To clean up the project type:
`make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- clean`

or
`make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- distclean`

To use the new U-Boot configuration, re-generate the zynq boot image, section IV-C, and program the QSPI, section IV-D.

VI. Device tree

The device tree is a complete description the hardware according to a specified format. This information is passed on by the bootloader (U-Boot) to the operating system (Linux). Before the introduction of device trees the kernel needed to contain the entire description of the hardware [15].

With the system defined from a hardware point of view, i.e. the exported system configuration from Vivado with respect to the Zynq processing system block (clocks, peripherals, memory configuration) and system address map (added IP-cores with register space), a corresponding device tree can be generated. On the Xilinx Wiki there is a howto-guide and this guide follows the ”Creating a Device Tree Source (.dts/.dtsi) files (Vivado 2014.2 onwards)”-section [16]. The device tree was generated under SDK 2014.2.

First the Linux device tree generator for the Xilinx SDK needs to be downloaded from the Git Hub [17], specifically:
`https://github.com/Xilinx/device-tree-xlnx/tree/xilinx-v2014.2`
If following the Xilinx wiki ensure to download the correct version from the repository. For 2014.2 select branch ⇒ Tags ⇒ xilinx-v2014.2 before downloading. The downloaded folder should be named like this:
`<path-to-repo>\device-tree-xlnx-xilinx-v2014.2\`

In section IV it is described how to setup a workspace in SDK. Open SDK and load this workspace. To add the device tree generator to SDK, select Xilinx Tools ⇒ Repositories to open the repositories dialogue window. Select New... (either local or global), navigate to the device tree download folder listed above, select it and click OK.

Now SDK should be updated to be able to generate device trees from the hardware definition. To do this, select File ⇒ New ⇒ Board Support Package. Name the project `device_tree_bsp_0`. Use the existing hardware platform. At the bottom select the new option device_tree. In the next dialogue add the following bootargs:

```
console=ttyPS0,115200 root=/dev/ram rw
ip=192.168.1.10:255.255.255.0:eth0 earlyprintk
```

and click OK.

Under device_tree_bsp_0 device tree source-files are now generated, as seen in figure 13:

- `system.dts` - main file
- `ps.dtsi` - nodes associated with the processing system (Zynq-block)
- `pl.dtsi` - nodes associated with the programmable logic (cores with register space). This part enables bindings for Xilinx core-specific Linux drivers.

However, as the Ethernet PHY is not known by the system the following code needs to be appended to `system.dts` [18].

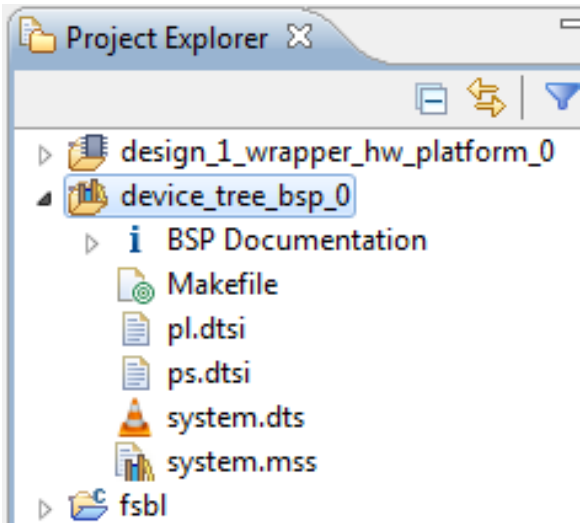


Fig. 13. Xilinx SDK Project Explorer - Device tree

```
&ps7_ethernet_0 {
   phy-handle = <&phy0>;
   mdio {
      #address-cells = <1>;
      #size-cells = <0>;
      phy0: phy@0 {
         compatible = "marvell,88e1510";
         device_type = "ethernet-phy";
         reg = <0>;
      };
   };
};
```

If enabled, Eth1 and USB2 has to be defined in a similar way. Worth pointing out is that both Ethernet devices are chained on the same MDIO.

The final step is to compile a device tree blob (.dtb) file from the sources. Just as with building U-boot this was done in a Linux environment. The device tree compiler (dtc) can be found in the linux-xlnx repository, which is available from the git hub, `linux-xlnx/scripts/dtc/`. . Alternatively, in Ubuntu, there is a device tree compiler in the standard repository. To install it type:
`aptitude install device-tree-compiler`.

To compile the device tree sources use the following command in the source folder:
`dtc -I dts -O dtb -o devicetree.dtb system.dts`
If using the Xilinx repository possibly the complete path to the dtc scripts needs to be specified. Also, change the mode of the output file:
`chmod 770 devicetree.dtb`

The `devicetree.dtb` is generated and the boot image can now be recreated, as described in section IV-C. However, instead of creating a new .bif-file, import the existing one and only update the device tree.

Note: It is also possible to generate the source from a blob.
`/dtc -I dtb -O dts -o system.dts system.dtb`

VII. RAMDISK

The ramdisk contains the Linux root file system. Xilinx provides a preconfigured version as a part of the Zynq release package [6]. This is a general setup and a couple of changes were made for GIMME2 (included in the ramdisk in the file package). There is a section on how to make changes to the rootfs on the Xilinx Wiki [19]. The procedure is described below.

Prerequisites are the ramdisk (`uramdisk.image.gz`) and a Linux environment with `mkimage`, which is located inside the tools folder of u-boot: `/u-boot-xlnx/tools/` or from the distribution (at least Ubuntu) repository.

Select/create a folder and copy the ramdisk-file to it. Open a terminal and navigate to the folder and follow the steps below.

```
sudo mkdir /tmp/rootfs
dd if=uramdisk.image.gz of=ramdisk.image.gz bs=64 skip=1
gunzip -c ramdisk.image.gz | sudo sh -c 'cd /tmp/rootfs/ && cpio -i'
```

Make changes in `/tmp/rootfs/`. Ensure to sudo all commands.

```
sh -c 'cd /tmp/rootfs/ && sudo find . | sudo cpio -H newc -o' |
   gzip -9 > ramdisk.image.gz
mkimage -A arm -T ramdisk -C gzip -d ramdisk.image.gz uramdisk.image.gz
rm ramdisk.image.gz
sudo rm -rf /tmp/rootfs
```

The following changes were made for GIMME2:

- `/etc/fstab` - uncomment the line below to mount sd-card.
  ```
  # uncomment this if your device has a SD/MMC/Transflash slot
  /dev/mmcblk0p1  /media/card   auto   defaults,sync,noauto  0  0
  ```
- `/etc/network/interfaces` - setup eth0 as static instead of dhcp
  ```
  # Wired or wireless interfaces
  auto eth0
  #iface eth0 inet dhcp
  iface eth0 inet static
          address 192.168.1.10
          netmask 255.255.255.0
          network 192.168.1.0
          gateway 192.168.1.1
  ```
- `/etc/default/rcS` - perhaps other settings

When finished remember to regenerate the Zynq boot image with the new ramdisk configuration, section IV-C, before reprogramming the QSPI, section IV-D.

VIII. What's next?

Now that GIMME2 is up and running new algorithms can be implemented either in Vivado, SDK or a combination thereof, depending on the application. A typical user scenario can be that image data is read, decoded and (pre-)processed by the PL and then passed on to the PS. The PS can be configured to have one core dedicated for high-level image-processing, possibly using OpenCV, and one core for Linux, control loop, communication, etc.

Some pointers on image processing for GIMME2:

- Take a look at [20] on how to design video systems on the Zynq. A possible exercise is to setup a dual channel VDMA. In the PL perform some simple operation on the data stream, eg. thresholding, color filtering, etc. This requires a PS-application that manipulates the VDMA register space (DocNav), reads/writes image data from/to the file system and PS-memory.
- Remove protective film from sensors
- Mount lens holders and optional converter ring CS-to-C-mount
- Mount lens(es)
- Interface image sensor(s) (serial communication + LVDS) - possibly a camera IP-core will be available in the future
- Verify focus to infinity or rectify (file down) lens-holder (requires sensor output)

APPENDIX A
GIMME2 PIN CONSTRAINTS

Pin mappings not following the Zynq standard, image sensors, FE0, PL-mem, UART, GPIO.

A. Image sensors, UART, GPIO, etc

The `gimme2_pins.xcd` from a stereo applicaton.

```
# CAMERAS GENERAL #################

set_property IOSTANDARD LVCMOS25 [get_ports CAM_RESET_B]
set_property PACKAGE_PIN V8 [get_ports CAM_RESET_B]


# CAM0 ##########################
set_property IOSTANDARD LVDS_25 [get_ports CAM0_CLK_P]
set_property DIFF_TERM TRUE [get_ports CAM0_CLK_P]
set_property IOSTANDARD LVDS_25 [get_ports CAM0_CLK_N]
set_property DIFF_TERM TRUE [get_ports CAM0_CLK_N]
set_property PACKAGE_PIN AA19 [get_ports CAM0_CLK_N]

set_property PACKAGE_PIN AA18 [get_ports CAM0_SIOD]
set_property PACKAGE_PIN Y18 [get_ports CAM0_SIOC]
set_property IOSTANDARD LVCMOS25 [get_ports CAM0_SIOD]
set_property IOSTANDARD LVCMOS25 [get_ports CAM0_SIOC]

set_property IOSTANDARD LVDS_25 [get_ports {CAM0_P[0]}]
set_property DIFF_TERM TRUE [get_ports {CAM0_P[0]}]
set_property IOSTANDARD LVDS_25 [get_ports {CAM0_N[0]}]
set_property DIFF_TERM TRUE [get_ports {CAM0_N[0]}]
set_property IOSTANDARD LVDS_25 [get_ports {CAM0_P[1]}]
set_property DIFF_TERM TRUE [get_ports {CAM0_P[1]}]
set_property IOSTANDARD LVDS_25 [get_ports {CAM0_N[1]}]
set_property DIFF_TERM TRUE [get_ports {CAM0_N[1]}]
set_property IOSTANDARD LVDS_25 [get_ports {CAM0_P[2]}]
set_property DIFF_TERM TRUE [get_ports {CAM0_P[2]}]
set_property IOSTANDARD LVDS_25 [get_ports {CAM0_N[2]}]
set_property DIFF_TERM TRUE [get_ports {CAM0_N[2]}]
set_property IOSTANDARD LVDS_25 [get_ports {CAM0_P[3]}]
set_property DIFF_TERM TRUE [get_ports {CAM0_P[3]}]
set_property IOSTANDARD LVDS_25 [get_ports {CAM0_N[3]}]
set_property DIFF_TERM TRUE [get_ports {CAM0_N[3]}]
set_property PACKAGE_PIN W15 [get_ports {CAM0_P[3]}]
set_property PACKAGE_PIN Y13 [get_ports {CAM0_P[2]}]
set_property PACKAGE_PIN AA17 [get_ports {CAM0_P[1]}]
set_property PACKAGE_PIN Y20 [get_ports {CAM0_P[0]}]

set_property IOSTANDARD LVCMOS25 [get_ports {CAM0_GPI[0]}]
set_property IOSTANDARD LVCMOS25 [get_ports {CAM0_GPI[1]}]
set_property IOSTANDARD LVCMOS25 [get_ports {CAM0_GPI[2]}]
set_property IOSTANDARD LVCMOS25 [get_ports {CAM0_GPI[3]}]
set_property PACKAGE_PIN Y11 [get_ports {CAM0_GPI[0]}]
set_property PACKAGE_PIN U10 [get_ports {CAM0_GPI[1]}]
set_property PACKAGE_PIN AB16 [get_ports {CAM0_GPI[2]}]
set_property PACKAGE_PIN AA16 [get_ports {CAM0_GPI[3]}]

set_property IOSTANDARD LVCMOS25 [get_ports CAM0_FLASH]
set_property IOSTANDARD LVCMOS25 [get_ports CAM0_SHUTTER]
set_property PACKAGE_PIN U16 [get_ports CAM0_FLASH]
set_property PACKAGE_PIN U15 [get_ports CAM0_SHUTTER]
```

```
53
54
55 ## CAM1 ############################
56 set_property IOSTANDARD LVDS_25 [get_ports CAM1_CLK_P]
57 set_property DIFF_TERM TRUE [get_ports CAM1_CLK_P]
58 set_property IOSTANDARD LVDS_25 [get_ports CAM1_CLK_N]
59 set_property DIFF_TERM TRUE [get_ports CAM1_CLK_N]
60 set_property PACKAGE_PIN AA6 [get_ports CAM1_CLK_N]
61
62 set_property PACKAGE_PIN AA8 [get_ports CAM1_SIOD]
63 set_property PACKAGE_PIN W7 [get_ports CAM1_SIOC]
64 set_property IOSTANDARD LVCMOS25 [get_ports CAM1_SIOD]
65 set_property IOSTANDARD LVCMOS25 [get_ports CAM1_SIOC]
66
67 set_property IOSTANDARD LVDS_25 [get_ports {CAM1_P[0]}]
68 set_property DIFF_TERM TRUE [get_ports {CAM1_P[0]}]
69 set_property IOSTANDARD LVDS_25 [get_ports {CAM1_N[0]}]
70 set_property DIFF_TERM TRUE [get_ports {CAM1_N[0]}]
71 set_property IOSTANDARD LVDS_25 [get_ports {CAM1_P[1]}]
72 set_property DIFF_TERM TRUE [get_ports {CAM1_P[1]}]
73 set_property IOSTANDARD LVDS_25 [get_ports {CAM1_N[1]}]
74 set_property DIFF_TERM TRUE [get_ports {CAM1_N[1]}]
75 set_property IOSTANDARD LVDS_25 [get_ports {CAM1_P[2]}]
76 set_property DIFF_TERM TRUE [get_ports {CAM1_P[2]}]
77 set_property IOSTANDARD LVDS_25 [get_ports {CAM1_N[2]}]
78 set_property DIFF_TERM TRUE [get_ports {CAM1_N[2]}]
79 set_property IOSTANDARD LVDS_25 [get_ports {CAM1_P[3]}]
80 set_property DIFF_TERM TRUE [get_ports {CAM1_P[3]}]
81 set_property IOSTANDARD LVDS_25 [get_ports {CAM1_N[3]}]
82 set_property DIFF_TERM TRUE [get_ports {CAM1_N[3]}]
83 set_property PACKAGE_PIN Y4 [get_ports {CAM1_P[3]}]
84 set_property PACKAGE_PIN V5 [get_ports {CAM1_P[2]}]
85 set_property PACKAGE_PIN U6 [get_ports {CAM1_P[1]}]
86 set_property PACKAGE_PIN W6 [get_ports {CAM1_P[0]}]
87
88 set_property IOSTANDARD LVCMOS25 [get_ports {CAM1_GPI[0]}]
89 set_property IOSTANDARD LVCMOS25 [get_ports {CAM1_GPI[1]}]
90 set_property IOSTANDARD LVCMOS25 [get_ports {CAM1_GPI[2]}]
91 set_property IOSTANDARD LVCMOS25 [get_ports {CAM1_GPI[3]}]
92 set_property PACKAGE_PIN AA12 [get_ports {CAM1_GPI[0]}]
93 set_property PACKAGE_PIN AB12 [get_ports {CAM1_GPI[1]}]
94 set_property PACKAGE_PIN AA11 [get_ports {CAM1_GPI[2]}]
95 set_property PACKAGE_PIN AB11 [get_ports {CAM1_GPI[3]}]
96
97 set_property IOSTANDARD LVCMOS25 [get_ports CAM1_FLASH]
98 set_property IOSTANDARD LVCMOS25 [get_ports CAM1_SHUTTER]
99 set_property PACKAGE_PIN AB1 [get_ports CAM1_FLASH]
100 set_property PACKAGE_PIN AB2 [get_ports CAM1_SHUTTER]
101
102
103 # UART ##################################
104
105 set_property IOSTANDARD LVCMOS25 [get_ports UART_1_rxd]
106 set_property IOSTANDARD LVCMOS25 [get_ports UART_1_txd]
107 set_property IOSTANDARD LVCMOS25 [get_ports UART_0_rxd]
108 set_property IOSTANDARD LVCMOS25 [get_ports UART_0_txd]
109
110 set_property PACKAGE_PIN V14 [get_ports UART_1_rxd]
111 set_property PACKAGE_PIN U7 [get_ports UART_1_txd]
112 set_property PACKAGE_PIN W20 [get_ports UART_0_rxd]
113 set_property PACKAGE_PIN AA22 [get_ports UART_0_txd]
114
115 # JTAG ##################################
116
117 set_property IOSTANDARD LVCMOS25 [get_ports PJTAG_tdo]
```

```
set_property IOSTANDARD LVCMOS25 [get_ports PJTAG_tdi]
set_property IOSTANDARD LVCMOS25 [get_ports PJTAG_tms]
set_property IOSTANDARD LVCMOS25 [get_ports PJTAG_tck]

set_property PACKAGE_PIN V7 [get_ports PJTAG_tdo]
set_property PACKAGE_PIN V9 [get_ports PJTAG_tdi]
set_property PACKAGE_PIN U9 [get_ports PJTAG_tms]
set_property PACKAGE_PIN Y9 [get_ports PJTAG_tck]



# GPIO ############################

set_property IOSTANDARD LVCMOS15 [get_ports {rsv_pins_1V5[7]}]
set_property IOSTANDARD LVCMOS15 [get_ports {rsv_pins_1V5[6]}]
set_property IOSTANDARD LVCMOS15 [get_ports {rsv_pins_1V5[5]}]
set_property IOSTANDARD LVCMOS15 [get_ports {rsv_pins_1V5[4]}]
set_property IOSTANDARD LVCMOS15 [get_ports {rsv_pins_1V5[3]}]
set_property IOSTANDARD LVCMOS15 [get_ports {rsv_pins_1V5[2]}]
set_property IOSTANDARD LVCMOS15 [get_ports {rsv_pins_1V5[1]}]
set_property IOSTANDARD LVCMOS15 [get_ports {rsv_pins_1V5[0]}]
set_property PACKAGE_PIN P22 [get_ports {rsv_pins_1V5[0]}]
set_property PACKAGE_PIN P21 [get_ports {rsv_pins_1V5[1]}]
set_property PACKAGE_PIN N22 [get_ports {rsv_pins_1V5[2]}]
set_property PACKAGE_PIN M21 [get_ports {rsv_pins_1V5[3]}]
set_property PACKAGE_PIN M22 [get_ports {rsv_pins_1V5[4]}]
set_property PACKAGE_PIN R16 [get_ports {rsv_pins_1V5[5]}]
set_property PACKAGE_PIN T17 [get_ports {rsv_pins_1V5[6]}]
set_property PACKAGE_PIN N19 [get_ports {rsv_pins_1V5[7]}]

set_property IOSTANDARD LVCMOS25 [get_ports {rsv_pins_2V5[5]}]
set_property IOSTANDARD LVCMOS25 [get_ports {rsv_pins_2V5[4]}]
set_property IOSTANDARD LVCMOS25 [get_ports {rsv_pins_2V5[3]}]
set_property IOSTANDARD LVCMOS25 [get_ports {rsv_pins_2V5[2]}]
set_property IOSTANDARD LVCMOS25 [get_ports {rsv_pins_2V5[1]}]
set_property IOSTANDARD LVCMOS25 [get_ports {rsv_pins_2V5[0]}]
set_property PACKAGE_PIN AB21 [get_ports {rsv_pins_2V5[5]}]
set_property PACKAGE_PIN AB20 [get_ports {rsv_pins_2V5[4]}]
set_property PACKAGE_PIN AB19 [get_ports {rsv_pins_2V5[3]}]
set_property PACKAGE_PIN Y16 [get_ports {rsv_pins_2V5[2]}]
set_property PACKAGE_PIN AB15 [get_ports {rsv_pins_2V5[1]}]
set_property PACKAGE_PIN AB14 [get_ports {rsv_pins_2V5[0]}]



#define timing constraints - can be done in Vivado after synthesis -> edit timing constraints

create_clock -period 4.167 -name CAM0_CLK_P -waveform {0.000 2.083} [get_ports CAM0_CLK_P]
create_clock -period 4.167 -name CAM1_CLK_P -waveform {0.000 2.083} [get_ports CAM1_CLK_P]
```

B. Fast Ethernet (FE)

Example from an old PlanAhead-project, hence the ucf-format. However, this can be converted to .xcd.

```
#######################################################
## Timings for FE ethernet ETH2 clocks
#######################################################
NET "ETH2_RX_CLK" TNM_NET = "TNM_ETH2_CLK";
NET "ETH2_TX_CLK" TNM_NET = "TNM_ETH2_CLK";
TIMESPEC TS_ETH2_CLK = PERIOD "TNM_ETH2_CLK" 40 ns;

```

```
8
9  NET "ETH2_RXD[0]" IOSTANDARD = LVCMOS25;
10 NET "ETH2_RXD[1]" IOSTANDARD = LVCMOS25;
11 NET "ETH2_RXD[2]" IOSTANDARD = LVCMOS25;
12 NET "ETH2_RXD[3]" IOSTANDARD = LVCMOS25;
13
14 NET "ETH2_TXD[0]" IOSTANDARD = LVCMOS25;
15 NET "ETH2_TXD[1]" IOSTANDARD = LVCMOS25;
16 NET "ETH2_TXD[2]" IOSTANDARD = LVCMOS25;
17 NET "ETH2_TXD[3]" IOSTANDARD = LVCMOS25;
18 NET "ETH2_RX_CLK" IOSTANDARD = LVCMOS25;
19 NET "ETH2_TX_CLK" IOSTANDARD = LVCMOS25;
20 NET "ETH2_RX_COL" IOSTANDARD = LVCMOS25;
21 NET "ETH2_RX_CRS" IOSTANDARD = LVCMOS25;
22 NET "ETH2_RX_DV" IOSTANDARD = LVCMOS25;
23 NET "ETH2_RX_ER" IOSTANDARD = LVCMOS25;
24 NET "ETH2_TX_EN" IOSTANDARD = LVCMOS25;
25
26
27 NET "ETH2_RXD[0]" LOC = U17;
28 NET "ETH2_RXD[1]" LOC = U19;
29 NET "ETH2_RXD[2]" LOC = U20;
30 NET "ETH2_RXD[3]" LOC = U22;
31 NET "ETH2_TXD[0]" LOC = W21;
32 NET "ETH2_TXD[1]" LOC = W22;
33 NET "ETH2_TXD[2]" LOC = V15;
34 NET "ETH2_TXD[3]" LOC = V17;
35 NET "ETH2_MDC" LOC = U21;
36 NET "ETH2_MDIO" LOC = T21;
37 NET "ETH2_RX_CLK" LOC = W17;
38 NET "ETH2_RX_COL" LOC = V22;
39 NET "ETH2_RX_CRS" LOC = V19;
40 NET "ETH2_RX_DV" LOC = T22;
41 NET "ETH2_RX_ER" LOC = V20;
42 NET "ETH2_TX_CLK" LOC = W16;
43 NET "ETH2_TX_EN" LOC = V18;
```

C. PL-memory

From an old PlanAhead example. The Memory Interface Generator (MIG) can be found in the IP-catalogue.

```
1  ##############################################################################################
2  ##
3  ##  Xilinx, Inc. 2010              www.xilinx.com
4  ##  Mi 27. Jun 16:14:59 2012
5  ##  Generated by MIG Version 1.5
6  ##
7  ##############################################################################################
8  ##  File name :       MIG.ucf
9  ##  Details :     Constraints file
10 ##                   FPGA Family:       ZYNQ
11 ##                   FPGA Part:         XC7Z020-CLG484
12 ##                   Speedgrade:        -2
13 ##                   Design Entry:      VHDL
14 ##                   Frequency:         400 MHz
15 ##                   Time Period:       2500 ps
16 ##############################################################################################
17
18 ##############################################################################################
```

```
## Controller 0
## Memory Device: DDR3_SDRAM->Components->MT41J256M16XX-187E
## Data Width: 16
## Time Period: 2500
## Data Mask: 1
##############################################################################################################


NET "MIG_sys_clk" TNM_NET = "TNM_sys_clk";
TIMESPEC TS_sys_clk = PERIOD "TNM_sys_clk" 20 ns;

############## NET - IOSTANDARD ##################

# Bank: 34 - Byte: T0
NET "MIG_dq[0]" IOSTANDARD = SSTL15;
NET "MIG_dq[0]" SLEW = FAST;
NET "MIG_dq[0]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[0]" LOC = N17;
# Bank: 34 - Byte: T0
NET "MIG_dq[1]" IOSTANDARD = SSTL15;
NET "MIG_dq[1]" SLEW = FAST;
NET "MIG_dq[1]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[1]" LOC = K15;
# Bank: 34 - Byte: T0
NET "MIG_dq[2]" IOSTANDARD = SSTL15;
NET "MIG_dq[2]" SLEW = FAST;
NET "MIG_dq[2]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[2]" LOC = M17;
# Bank: 34 - Byte: T0
NET "MIG_dq[3]" IOSTANDARD = SSTL15;
NET "MIG_dq[3]" SLEW = FAST;
NET "MIG_dq[3]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[3]" LOC = M15;
# Bank: 34 - Byte: T0
NET "MIG_dq[4]" IOSTANDARD = SSTL15;
NET "MIG_dq[4]" SLEW = FAST;
NET "MIG_dq[4]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[4]" LOC = L17;
# Bank: 34 - Byte: T0
NET "MIG_dq[5]" IOSTANDARD = SSTL15;
NET "MIG_dq[5]" SLEW = FAST;
NET "MIG_dq[5]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[5]" LOC = N18;
# Bank: 34 - Byte: T0
NET "MIG_dq[6]" IOSTANDARD = SSTL15;
NET "MIG_dq[6]" SLEW = FAST;
NET "MIG_dq[6]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[6]" LOC = J17;
# Bank: 34 - Byte: T0
NET "MIG_dq[7]" IOSTANDARD = SSTL15;
NET "MIG_dq[7]" SLEW = FAST;
NET "MIG_dq[7]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[7]" LOC = J16;
# Bank: 34 - Byte: T1
NET "MIG_dq[8]" IOSTANDARD = SSTL15;
NET "MIG_dq[8]" SLEW = FAST;
NET "MIG_dq[8]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[8]" LOC = K19;
# Bank: 34 - Byte: T1
NET "MIG_dq[9]" IOSTANDARD = SSTL15;
NET "MIG_dq[9]" SLEW = FAST;
NET "MIG_dq[9]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[9]" LOC = K20;
# Bank: 34 - Byte: T1
NET "MIG_dq[10]" IOSTANDARD = SSTL15;
```

```
NET "MIG_dq[10]" SLEW = FAST;
NET "MIG_dq[10]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[10]" LOC = J22;
# Bank: 34 - Byte: T1
NET "MIG_dq[11]" IOSTANDARD = SSTL15;
NET "MIG_dq[11]" SLEW = FAST;
NET "MIG_dq[11]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[11]" LOC = L18;
# Bank: 34 - Byte: T1
NET "MIG_dq[12]" IOSTANDARD = SSTL15;
NET "MIG_dq[12]" SLEW = FAST;
NET "MIG_dq[12]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[12]" LOC = K18;
# Bank: 34 - Byte: T1
NET "MIG_dq[13]" IOSTANDARD = SSTL15;
NET "MIG_dq[13]" SLEW = FAST;
NET "MIG_dq[13]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[13]" LOC = L21;
# Bank: 34 - Byte: T1
NET "MIG_dq[14]" IOSTANDARD = SSTL15;
NET "MIG_dq[14]" SLEW = FAST;
NET "MIG_dq[14]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[14]" LOC = J21;
# Bank: 34 - Byte: T1
NET "MIG_dq[15]" IOSTANDARD = SSTL15;
NET "MIG_dq[15]" SLEW = FAST;
NET "MIG_dq[15]" IN_TERM = UNTUNED_SPLIT_50;
NET "MIG_dq[15]" LOC = L22;
# Bank: 35 - Byte: T0
NET "MIG_addr[14]" IOSTANDARD = SSTL15;
NET "MIG_addr[14]" SLEW = FAST;
NET "MIG_addr[14]" LOC = F16;
# Bank: 35 - Byte: T0
NET "MIG_addr[13]" IOSTANDARD = SSTL15;
NET "MIG_addr[13]" SLEW = FAST;
NET "MIG_addr[13]" LOC = E16;
# Bank: 35 - Byte: T0
NET "MIG_addr[12]" IOSTANDARD = SSTL15;
NET "MIG_addr[12]" SLEW = FAST;
NET "MIG_addr[12]" LOC = D16;
# Bank: 35 - Byte: T0
NET "MIG_addr[11]" IOSTANDARD = SSTL15;
NET "MIG_addr[11]" SLEW = FAST;
NET "MIG_addr[11]" LOC = D17;
# Bank: 35 - Byte: T0
NET "MIG_addr[10]" IOSTANDARD = SSTL15;
NET "MIG_addr[10]" SLEW = FAST;
NET "MIG_addr[10]" LOC = G15;
# Bank: 35 - Byte: T0
NET "MIG_addr[9]" IOSTANDARD = SSTL15;
NET "MIG_addr[9]" SLEW = FAST;
NET "MIG_addr[9]" LOC = G16;
# Bank: 35 - Byte: T0
NET "MIG_addr[8]" IOSTANDARD = SSTL15;
NET "MIG_addr[8]" SLEW = FAST;
NET "MIG_addr[8]" LOC = F18;
# Bank: 35 - Byte: T0
NET "MIG_addr[7]" IOSTANDARD = SSTL15;
NET "MIG_addr[7]" SLEW = FAST;
NET "MIG_addr[7]" LOC = E18;
# Bank: 35 - Byte: T0
NET "MIG_addr[6]" IOSTANDARD = SSTL15;
NET "MIG_addr[6]" SLEW = FAST;
NET "MIG_addr[6]" LOC = G17;
# Bank: 35 - Byte: T0
```

```
148 NET "MIG_addr[5]" IOSTANDARD = SSTL15;
149 NET "MIG_addr[5]" SLEW = FAST;
150 NET "MIG_addr[5]" LOC = F17;
151 # Bank: 35 - Byte: T1
152 NET "MIG_addr[4]" IOSTANDARD = SSTL15;
153 NET "MIG_addr[4]" SLEW = FAST;
154 NET "MIG_addr[4]" LOC = C15;
155 # Bank: 35 - Byte: T1
156 NET "MIG_addr[3]" IOSTANDARD = SSTL15;
157 NET "MIG_addr[3]" SLEW = FAST;
158 NET "MIG_addr[3]" LOC = B15;
159 # Bank: 35 - Byte: T1
160 NET "MIG_addr[2]" IOSTANDARD = SSTL15;
161 NET "MIG_addr[2]" SLEW = FAST;
162 NET "MIG_addr[2]" LOC = B16;
163 # Bank: 35 - Byte: T1
164 NET "MIG_addr[1]" IOSTANDARD = SSTL15;
165 NET "MIG_addr[1]" SLEW = FAST;
166 NET "MIG_addr[1]" LOC = B17;
167 # Bank: 35 - Byte: T1
168 NET "MIG_addr[0]" IOSTANDARD = SSTL15;
169 NET "MIG_addr[0]" SLEW = FAST;
170 NET "MIG_addr[0]" LOC = A16;
171 # Bank: 35 - Byte: T1
172 NET "MIG_ba[2]" IOSTANDARD = SSTL15;
173 NET "MIG_ba[2]" SLEW = FAST;
174 NET "MIG_ba[2]" LOC = A17;
175 # Bank: 35 - Byte: T1
176 NET "MIG_ba[1]" IOSTANDARD = SSTL15;
177 NET "MIG_ba[1]" SLEW = FAST;
178 NET "MIG_ba[1]" LOC = A18;
179 # Bank: 35 - Byte: T1
180 NET "MIG_ba[0]" IOSTANDARD = SSTL15;
181 NET "MIG_ba[0]" SLEW = FAST;
182 NET "MIG_ba[0]" LOC = A19;
183 # Bank: 35 - Byte: T1
184 NET "MIG_ras_n" IOSTANDARD = SSTL15;
185 NET "MIG_ras_n" SLEW = FAST;
186 NET "MIG_ras_n" LOC = C17;
187 # Bank: 35 - Byte: T1
188 NET "MIG_cas_n" IOSTANDARD = SSTL15;
189 NET "MIG_cas_n" SLEW = FAST;
190 NET "MIG_cas_n" LOC = C18;
191 # Bank: 35 - Byte: T1
192 NET "MIG_we_n" IOSTANDARD = SSTL15;
193 NET "MIG_we_n" SLEW = FAST;
194 NET "MIG_we_n" LOC = D18;
195 # Bank: 34 - Byte: T1
196 NET "MIG_reset_n" IOSTANDARD = LVCMOS15;
197 NET "MIG_reset_n" SLEW = FAST;
198 NET "MIG_reset_n" LOC = L19;
199 # Bank: 35 - Byte: T2
200 NET "MIG_cke[0]" IOSTANDARD = SSTL15;
201 NET "MIG_cke[0]" SLEW = FAST;
202 NET "MIG_cke[0]" LOC = A21;
203 # Bank: 35 - Byte: T2
204 NET "MIG_odt[0]" IOSTANDARD = SSTL15;
205 NET "MIG_odt[0]" SLEW = FAST;
206 NET "MIG_odt[0]" LOC = A22;
207 # Bank: 35 - Byte: T1
208 NET "MIG_cs_n" IOSTANDARD = SSTL15;
209 NET "MIG_cs_n" SLEW = FAST;
210 NET "MIG_cs_n" LOC = C19;
211 # Bank: 34 - Byte: T0
212 NET "MIG_dm[0]" IOSTANDARD = SSTL15;
```

```
213 NET "MIG_dm[0]" SLEW = FAST;
214 NET "MIG_dm[0]" LOC = J15;
215 # Bank: 34 - Byte: T1
216 NET "MIG_dm[1]" IOSTANDARD = SSTL15;
217 NET "MIG_dm[1]" SLEW = FAST;
218 NET "MIG_dm[1]" LOC = J18;
219 # Bank: 35 - Byte:
220 NET "MIG_sys_clk" IOSTANDARD = LVCMOS15;
221 NET "MIG_sys_clk" LOC = D20;
222 # Bank: 34 - Byte: T0
223 NET "MIG_dqs_p[0]" IOSTANDARD = DIFF_SSTL15;
224 NET "MIG_dqs_p[0]" SLEW = FAST;
225 NET "MIG_dqs_p[0]" IN_TERM = UNTUNED_SPLIT_50;
226 NET "MIG_dqs_p[0]" LOC = K16;
227 # Bank: 34 - Byte: T0
228 NET "MIG_dqs_n[0]" IOSTANDARD = DIFF_SSTL15;
229 NET "MIG_dqs_n[0]" SLEW = FAST;
230 NET "MIG_dqs_n[0]" IN_TERM = UNTUNED_SPLIT_50;
231 NET "MIG_dqs_n[0]" LOC = L16;
232 # Bank: 34 - Byte: T1
233 NET "MIG_dqs_p[1]" IOSTANDARD = DIFF_SSTL15;
234 NET "MIG_dqs_p[1]" SLEW = FAST;
235 NET "MIG_dqs_p[1]" IN_TERM = UNTUNED_SPLIT_50;
236 NET "MIG_dqs_p[1]" LOC = J20;
237 # Bank: 34 - Byte: T1
238 NET "MIG_dqs_n[1]" IOSTANDARD = DIFF_SSTL15;
239 NET "MIG_dqs_n[1]" SLEW = FAST;
240 NET "MIG_dqs_n[1]" IN_TERM = UNTUNED_SPLIT_50;
241 NET "MIG_dqs_n[1]" LOC = K21;
242 # Bank: 35 - Byte: T0
243 NET "MIG_ck_p[0]" IOSTANDARD = DIFF_SSTL15;
244 NET "MIG_ck_p[0]" SLEW = FAST;
245 NET "MIG_ck_p[0]" LOC = E15;
246 # Bank: 35 - Byte: T0
247 NET "MIG_ck_n[0]" IOSTANDARD = DIFF_SSTL15;
248 NET "MIG_ck_n[0]" SLEW = FAST;
249 NET "MIG_ck_n[0]" LOC = D15;
250
251
252
253 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_D.ddr_byte_lane_D/phaser_out"
    LOC = PHASER_OUT_PHY_X1Y7;
254 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_C.ddr_byte_lane_C/phaser_out"
    LOC = PHASER_OUT_PHY_X1Y6;
255 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_0.ddr_phy_4lanes/ddr_byte_lane_D.ddr_byte_lane_D/phaser_out"
    LOC = PHASER_OUT_PHY_X1Y11;
256 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_0.ddr_phy_4lanes/ddr_byte_lane_C.ddr_byte_lane_C/phaser_out"
    LOC = PHASER_OUT_PHY_X1Y10;
257 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_0.ddr_phy_4lanes/ddr_byte_lane_B.ddr_byte_lane_B/phaser_out"
    LOC = PHASER_OUT_PHY_X1Y9;
258
259 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_D.ddr_byte_lane_D/phaser_in_gen.
    phaser_in" LOC = PHASER_IN_PHY_X1Y7;
260 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_C.ddr_byte_lane_C/phaser_in_gen.
    phaser_in" LOC = PHASER_IN_PHY_X1Y6;
261 ## INST "*/ddr_phy_4lanes_0.ddr_phy_4lanes/ddr_byte_lane_D.ddr_byte_lane_D/phaser_in_gen.
    phaser_in" LOC=PHASER_IN_PHY_X1Y11;
```

```
262 ## INST "*/ddr_phy_4lanes_0.ddr_phy_4lanes/ddr_byte_lane_C.ddr_byte_lane_C/phaser_in_gen.
    phaser_in" LOC=PHASER_IN_PHY_X1Y10;
263 ## INST "*/ddr_phy_4lanes_0.ddr_phy_4lanes/ddr_byte_lane_B.ddr_byte_lane_B/phaser_in_gen.
    phaser_in" LOC=PHASER_IN_PHY_X1Y9;
264
265
266 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_D.ddr_byte_lane_D/out_fifo" LOC
    = OUT_FIFO_X1Y7;
267 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_C.ddr_byte_lane_C/out_fifo" LOC
    = OUT_FIFO_X1Y6;
268 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_0.ddr_phy_4lanes/ddr_byte_lane_D.ddr_byte_lane_D/out_fifo" LOC
    = OUT_FIFO_X1Y11;
269 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_0.ddr_phy_4lanes/ddr_byte_lane_C.ddr_byte_lane_C/out_fifo" LOC
    = OUT_FIFO_X1Y10;
270 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_0.ddr_phy_4lanes/ddr_byte_lane_B.ddr_byte_lane_B/out_fifo" LOC
    = OUT_FIFO_X1Y9;
271
272 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_D.ddr_byte_lane_D/in_fifo_gen.
    in_fifo" LOC = IN_FIFO_X1Y7;
273 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_C.ddr_byte_lane_C/in_fifo_gen.
    in_fifo" LOC = IN_FIFO_X1Y6;
274
275 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/phy_control_i" LOC = PHY_CONTROL_X1Y1;
276 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_0.ddr_phy_4lanes/phy_control_i" LOC = PHY_CONTROL_X1Y2;
277
278 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/phaser_ref_i" LOC = PHASER_REF_X1Y1;
279 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_0.ddr_phy_4lanes/phaser_ref_i" LOC = PHASER_REF_X1Y2;
280
281
282 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_D.ddr_byte_lane_D/
    ddr_byte_group_io/slave_ts.oserdes_slave_ts" LOC = OLOGIC_X1Y93;
283 INST "MIG_example/u_MIG/u_memc_ui_top_std/mem_intfc0/ddr_phy_top0/u_ddr_mc_phy_wrapper/
    u_ddr_mc_phy/ddr_phy_4lanes_1.ddr_phy_4lanes/ddr_byte_lane_C.ddr_byte_lane_C/
    ddr_byte_group_io/slave_ts.oserdes_slave_ts" LOC = OLOGIC_X1Y81;
284
285 INST "MIG_example/u_MIG/u_ddr3_infrastructure/plle2_i" LOC = PLLE2_ADV_X1Y2;
286 INST "MIG_example/u_MIG/u_ddr3_infrastructure/mmcm_i" LOC = MMCME2_ADV_X1Y2;
287
288 CONFIG INTERNAL_VREF_BANK34= 0.750;
289
290 ##############################################################################
291 ## End of MIG.ucf
292 ##############################################################################
```

APPENDIX B
GIMME2 PERIPHERAL I/O PINS

Pin mappings following the Zynq placement standard populating the FIXED_IO-vector.

- QSPI - 1-6, 8
- microSD - 10-15 (SD1)
- GE0 - 16-27, 52-53
- GE1 - 28-39, (52-53 unless GE0 is configured)
- USB2 - 40-51 (USB1)

APPENDIX C
QSPI PROGRAMMING

A. XMD Console Input/Output

```
****** Xilinx Microprocessor Debugger (XMD) Engine
XMD%
XMD%
****** XMD v2014.2 (64-bit)
  **** SW Build 932637 on Wed Jun 11 13:31:38 MDT 2014
    ** Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.


Accepted a new TCLSock connection from 127.0.0.1 on port 63445

Accepted a new TCLSock connection from 127.0.0.1 on port 63445
Configuring Device 2 (xc7z020) with Bitstream -- C:/Users/cag01/Documents/mdh/vhdl/gimme2a/
    gimme2_led/gimme2_led.sdk/design_1_wrapper_hw_platform_0/design_1_wrapper.bit
.......10.......20.......30.......40.......50.......60.......70.......80.......90.......Done
connect arm hw

JTAG chain configuration
-------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       4ba00477           4        arm_dap
 2       23727093           6        xc7z020

-------------------------------------------------
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".

-------------------------------------------------

CortexA9 Processor Configuration
-------------------------------------
Version.............................0x00000003
User ID.............................0x00000000
No of PC Breakpoints................6
No of Addr/Data Watchpoints.........4

Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1236
XMD%  source C:/Users/cag01/Documents/mdh/vhdl/gimme2a/gimme2_led/gimme2_led.sdk/
    design_1_wrapper_hw_platform_0/ps7_init.tcl
XMD% ps7_init
XMD% ps7_post_config
XMD% dow C:/Users/cag01/Documents/mdh/vhdl/gimme2a/gimme2_led/gimme2_led.sdk/bootimage/u-boot.
    elf
Processor Reset .... DONE
Downloading Program -- C:/Users/cag01/Documents/mdh/vhdl/gimme2a/gimme2_led/gimme2_led.sdk/
    bootimage/u-boot.elf
        section, .text: 0x04000000-0x04039647
        section, .rodata: 0x04039648-0x040482e7
        section, .hash: 0x040482e8-0x04048313
        section, .data: 0x04048314-0x0404a3f7
        section, .got.plt: 0x0404a3f8-0x0404a403
        section, .u_boot_list: 0x0404a404-0x0404ac9b
        section, .rel.dyn: 0x0404ac9c-0x04052de3
        section, .bss: 0x0404ac9c-0x0409f8b7
Download Progress.10.20.30.40.50.60.70.80.90.Done
Setting PC with Program Start Address 0x04000000
XMD% dow -data C:/Users/cag01/Documents/mdh/vhdl/gimme2a/gimme2_led/gimme2_led.sdk/bootimage/
    BOOT.bin 0x08000000
```

```
Downloading Data File -- C:/Users/cag01/Documents/mdh/vhdl/gimme2a/gimme2_led/gimme2_led.sdk/
    bootimage/BOOT.bin at 0x08000000
Progress
    ..........................................................................................
    Done
XMD% con

RUNNING> XMD%
```

B. U-boot Terminal Input/Output

```
U-Boot 2014.01 (Sep 29 2014 - 14:54:16)

I2C:   ready
Memory: ECC disabled
DRAM:  512 MiB
MMC:   zynq_sdhci: 0
SF: Detected S25FL256S_64K with page size 256 Bytes, erase size 64 KiB, total 32 MiB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   Gem.e000b000
Hit any key to stop autoboot:  0
zynq-uboot> sf probe 0 0 0
SF: Detected S25FL256S_64K with page size 256 Bytes, erase size 64 KiB, total 32 MiB
zynq-uboot> sf erase 0 0x01000000
SF: 16777216 bytes @ 0x0 Erased: OK
zynq-uboot> sf write 0x08000000 0 0xFFFFFF
SF: 16777215 bytes @ 0x0 Written: OK
zynq-uboot>
```

APPENDIX D
U-BOOT CONFIGURATION FOR GIMME2

The following files were added to the U-Boot file structure.

A. *zynq_gimme2.h*

`<path to u-boot>/include/configs/zynq_gimme2.h`

```
/*
 * (C) Copyright 2012 Xilinx
 *
 * Configuration settings for the Xilinx Zynq ZC702 and ZC706 boards
 * See zynq_common.h for Zynq common configs
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 */

#ifndef __CONFIG_GIMME2
#define __CONFIG_GIMME2

#define CONFIG_SYS_SDRAM_SIZE    (512 * 1024 * 1024)


#define CONFIG_ZYNQ_SERIAL_UART0
#define CONFIG_ZYNQ_GEM0
#define CONFIG_ZYNQ_GEM_PHY_ADDR0 7

#define CONFIG_SYS_NO_FLASH

#define CONFIG_ZYNQ_SDHCI1
#define CONFIG_ZYNQ_USB
#define CONFIG_ZYNQ_QSPI
/*#define CONFIG_ZYNQ_I2C0*/
#define CONFIG_ZYNQ_EEPROM
#define CONFIG_ZYNQ_BOOT_FREEBSD
#define CONFIG_DEFAULT_DEVICE_TREE   zynq-gimme2

#include <configs/zynq-common.h>

/* Default environment */
#ifdef CONFIG_EXTRA_ENV_SETTINGS
#undef CONFIG_EXTRA_ENV_SETTINGS
#define CONFIG_EXTRA_ENV_SETTINGS \
  "ethaddr=00:0a:35:00:01:22\0" \
  "kernel_image=uImage\0" \
  "ramdisk_image=uramdisk.image.gz\0" \
  "devicetree_image=devicetree.dtb\0" \
  "bitstream_image=system.bit.bin\0"  \
  "boot_image=BOOT.bin\0" \
  "loadbit_addr=0x100000\0" \
  "loadbootenv_addr=0x2000000\0" \
  "kernel_size=0x500000\0"  \
```

```
  "devicetree_size=0x20000\0" \
  "ramdisk_size=0x5E0000\0" \
  "boot_size=0xF00000\0"  \
  "fdt_high=0x20000000\0" \
  "initrd_high=0x20000000\0"  \
  "bootenv=uEnv.txt\0" \
  "loadbootenv=fatload mmc 0 ${loadbootenv_addr} ${bootenv}\0" \
  "importbootenv=echo Importing environment from SD ...; " \
    "env import -t ${loadbootenv_addr} $filesize\0" \
  "mmc_loadbit_fat=echo Loading bitstream from SD/MMC/eMMC to RAM.. && " \
    "mmcinfo && " \
    "fatload mmc 0 ${loadbit_addr} ${bitstream_image} && " \
    "fpga load 0 ${loadbit_addr} ${filesize}\0" \
  "norboot=echo Copying Linux from NOR flash to RAM... && " \
    "cp.b 0xE2100000 0x3000000 ${kernel_size} && " \
    "cp.b 0xE2600000 0x2A00000 ${devicetree_size} && " \
    "echo Copying ramdisk... && " \
    "cp.b 0xE2620000 0x2000000 ${ramdisk_size} && " \
    "bootm 0x3000000 0x2000000 0x2A00000\0" \
  "qspiboot=echo Copying Linux from QSPI flash to RAM... && " \
    "sf probe 0 0 0 && " \
    "sf read 0x3000000 0x450000 ${kernel_size} && " \
    "sf read 0x2A00000 0x780000 ${devicetree_size} && " \
    "echo Copying ramdisk... && " \
    "sf read 0x2000000 0x785000 ${ramdisk_size} && " \
    "bootm 0x3000000 0x2000000 0x2A00000\0" \
  "uenvboot=" \
    "if run loadbootenv; then " \
      "echo Loaded environment from ${bootenv}; " \
      "run importbootenv; " \
    "fi; " \
    "if test -n $uenvcmd; then " \
      "echo Running uenvcmd ...; " \
      "run uenvcmd; " \
    "fi\0" \
  "sdboot=if mmcinfo; then " \
      "run uenvboot; " \
      "echo Copying Linux from SD to RAM... && " \
      "fatload mmc 0 0x3000000 ${kernel_image} && " \
      "fatload mmc 0 0x2A00000 ${devicetree_image} && " \
      "fatload mmc 0 0x2000000 ${ramdisk_image} && " \
      "bootm 0x3000000 0x2000000 0x2A00000; " \
    "fi\0" \
  "usbboot=if usb start; then " \
      "run uenvboot; " \
      "echo Copying Linux from USB to RAM... && " \
      "fatload usb 0 0x3000000 ${kernel_image} && " \
      "fatload usb 0 0x2A00000 ${devicetree_image} && " \
      "fatload usb 0 0x2000000 ${ramdisk_image} && " \
      "bootm 0x3000000 0x2000000 0x2A00000; " \
    "fi\0" \
  "nandboot=echo Copying Linux from NAND flash to RAM... && " \
    "nand read 0x3000000 0x100000 ${kernel_size} && " \
    "nand read 0x2A00000 0x600000 ${devicetree_size} && " \
    "echo Copying ramdisk... && " \
    "nand read 0x2000000 0x620000 ${ramdisk_size} && " \
    "bootm 0x3000000 0x2000000 0x2A00000\0" \
  "jtagboot=echo TFTPing Linux to RAM... && " \
    "tftpboot 0x3000000 ${kernel_image} && " \
    "tftpboot 0x2A00000 ${devicetree_image} && " \
    "tftpboot 0x2000000 ${ramdisk_image} && " \
    "bootm 0x3000000 0x2000000 0x2A00000\0" \
  "rsa_norboot=echo Copying Image from NOR flash to RAM... && " \
    "cp.b 0xE2100000 0x100000 ${boot_size} && " \
    "zynqrsa 0x100000 && " \
```

```
    "bootm 0x3000000 0x2000000 0x2A00000\0" \
  "rsa_nandboot=echo Copying Image from NAND flash to RAM... && " \
    "nand read 0x100000 0x0 ${boot_size} && " \
    "zynqrsa 0x100000 && " \
    "bootm 0x3000000 0x2000000 0x2A00000\0" \
  "rsa_qspiboot=echo Copying Image from QSPI flash to RAM... && " \
    "sf probe 0 0 0 && " \
    "sf read 0x100000 0x0 ${boot_size} && " \
    "zynqrsa 0x100000 && " \
    "bootm 0x3000000 0x2000000 0x2A00000\0" \
  "rsa_sdboot=echo Copying Image from SD to RAM... && " \
    "fatload mmc 0 0x100000 ${boot_image} && " \
    "zynqrsa 0x100000 && " \
    "bootm 0x3000000 0x2000000 0x2A00000\0" \
  "rsa_jtagboot=echo TFTPing Image to RAM... && " \
    "tftpboot 0x100000 ${boot_image} && " \
    "zynqrsa 0x100000 && " \
    "bootm 0x3000000 0x2000000 0x2A00000\0"
#endif

#endif /* __CONFIG_GIMME2 */
```

B. *zynq-gimme2.dts*

`<path to u-boot>/board/xilinx/dts/zynq-gimme2.dts`

```
/*
 * GIMME2 board DTS
 *
 * Copyright (C) 2013 Xilinx, Inc.
 *
 * SPDX-License-Identifier: GPL-2.0+
 */
/dts-v1/;
#include "zynq-7000.dtsi"

/ {
  model = "Zynq Gimme2 Board";
  compatible = "xlnx,zynq-gimme2", "xlnx,zynq-7000";
};
```

ACKNOWLEDGMENT

Thanks to AF Inventions for the GIMME2 design.

Thanks to Xilinx for supporting this project.

Thanks to Volvo CE.

Thanks to the Knowledge Foundation.

REFERENCES

[1] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing whti the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*, 1st ed. Glasgow, Scotland: Strathclyde Academic Media, 7 2014. [Online]. Available: http://www.zynqbook.com/
[2] ZedBoard. ZedBoard home page. [Online]. Available: http://zedboard.org/
[3] Xilinx. Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT) - A Hands-On Guide to Effective Embedded System Design - UG873 (v14.6) June 19, 2013. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_6/ug873-zynq-ctt.pdf
[4] ——. Xilinx Wiki - Getting Started - Overview of the Xilinx Zynq AP SoC Design Flow. [Online]. Available: http://www.wiki.xilinx.com/Getting+Started
[5] C. Ahlberg. GIMME2 - a getting started guide file package. [Online]. Available: carl.ahlberg@mdh.se
[6] Xilinx. Xilinx Wiki - Zynq Releases. [Online]. Available: http://www.wiki.xilinx.com/Zynq+Releases
[7] ——. LogiCORE IP AXI GPIO v2.0, Product Guide, Vivado Design Suite - PG144 April 2, 2014. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf
[8] ——. Xilinx Wiki - Linux Drivers. [Online]. Available: http://www.wiki.xilinx.com/Linux+Drivers
[9] ——. Xilinx Wiki - Linux GPIO Driver. [Online]. Available: http://www.wiki.xilinx.com/Linux+GPIO+Driver
[10] OpenCV. OpenCV home page. [Online]. Available: http://opencv.org/
[11] R. Lienhart. Haar Frontal Face 2 data file. [Online]. Available: https://github.com/Itseez/opencv/blob/master/data/haarcascades/haarcascade_frontalface_alt2.xml
[12] Learn OpenCV by Examples - Haar Face Detection. [Online]. Available: http://opencvexamples.blogspot.com/2013/10/face-detection-using-haar-cascade.html#.VTEXT_msXYE
[13] DENX. U-Boot home page. [Online]. Available: http://www.denx.de/wiki/U-Boot
[14] Xilinx. Xilinx Wiki - U-Boot. [Online]. Available: http://www.wiki.xilinx.com/U-boot
[15] devicetree.org. Device Tree Project home page. [Online]. Available: http://www.devicetree.org
[16] Xilinx. Xilinx Wiki - Device Tree. [Online]. Available: http://www.wiki.xilinx.com/Build+Device+Tree+Blob
[17] GitHub. GitHub - Xilinx. [Online]. Available: https://github.com/Xilinx
[18] X. U. Community. Xilinx User Community Forum on device tree PHY setup. [Online]. Available: http://forums.xilinx.com/t5/Embedded-Linux/xemacps-e000b000-ps7-ethernet-eth0-no-PHY-setup/td-p/499018
[19] Xilinx. Xilinx Wiki - Build and Modify a Rootfs. [Online]. Available: http://www.wiki.xilinx.com/Build+and+Modify+a+Rootfs
[20] J. Lucero and Y. Arbel. Designing High-Performance Video Systems with the Zynq-7000 All Programmable SoC - XAPP792 (v1.0.1) October 16, 2012. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp792-high-performance-video-zynq.pdf