Development of Uclinux Platform for Computer Vision Algorithm in FPGA Devices

Debyo Saptono
LE2I - CNRS 5158 Laboratory
University of Burgundy
21078 Dijon France
Debyo.saptono@u-bourgogne.fr

Eri Prasetyo
Faculty of Computer Science
University of Gunadarma
Jakarta - Indonesia
eri@staff.gunadarma.ac.id

Abstract

This paper describes the use of the Xilinx Microblaze 32-bit, soft-core processor in a series of senior design projects. The Microblaze was implemented on a commercial-off-the-shelf FPGA- based single board computer. The FPGA is pre-configured with the Microblaze running a version of Linux called uCLinux. Using this platform, students can develop custom hardware that can interface to the Microblaze using the OPB bus, and custom software that runs as a thread (or threads) on uClinux. Computer Vision Algorithm with sobel filter will be implemented in Microblaze system to know resource that be used by application.

Keyword : Microblaze, uClinux, Computer Vision Algorithm, Linux

1. Introduction

The primary function of vision is to extract enough information from an image, or series of images, to provide guidance to a host system.This applies not only to organic vision systems, but also to artificial vision systems. Much research has been undertaken in the study of techniques and algorithms for extracting useful data from pictures. Since the beginning of computer vision in the late 1950s/early 1960s several methods have emerged for obtaining pertinent information from images and exposing it to the host system in an understandable format. In more recent years, the development of reconfigurable hardware logic devices (CPLDs and FPGAs) have prompted research into implementing and accelerating image processing algorithms in hardware logic. Most image processing algorithms are both data-parallel and computation-intensive, making them well suited for implementation on FPGAs. Research has shown that the use of FPGAs in computer vision systems can lead to sizeable performance benefits .

2. Computer Vision Algorithm

2.1. Binary Images

The analysis of binary images is one of the simplest ways to extract meaningful data from pictures. It is particularly useful when trying to determine the location orientation of an object within an image. This method of object location has how itself to be amenable to FPGA acceleration, although the implementation on FPGA in several ways from that proposed by previous researchers. A binary image is first constructed from the original picture by marking all the pixels which correspond to the object of interest.

2.2. PBMPLUS Utilities

These three formats are intermediate formats used by the PBMPLUS utilities. The acronyms stand for Portable BitMap/GrayMap/PixMap. PBM is for monochrome images, PGM for grayscale images with up to 256 shades of gray, and PPM for color images using up to 24-bits of true color. A fourth format is the Portable AnyMap, PNM. PNM is not actually a format itself. A program that uses PNM can read and write PBM, PGM, and PPM files. PNM is used for utility programs that support multiple image types. For instance, since the image type of a TIFF file may not be known, PNM reads the TIFF file and writes the appropriate file type.

Each of the four formats can read the other ones that carry less information. That is, a PGM utility reads PGM and PBM, a PPM utility reads PPM, PGM, and PBM. PBM, PGM, and PPM utilities always write in their own format, while PNM utilities generally write whatever format they have read. The formats store data either as ASCII or binary data and are otherwise basic formats consisting of a header and image data. The header consists of a magic number to identify the format, image size, and (except for PBM) the

number of colors/gray shades. The magic strings are PBM P1(P4), PGM P2(P5), and PPM P3(P6), where the first code is the code for ASCII data, and the code in parentheses is for binary data. True color images store pixel data as a triplet of numbers for RGB data.

2.3. Edge Detection

Edges in an image represent the boundaries between objects and the background, while corners represent the intersection of two or more edges. In image processing, edge points are the points in the image where the difference in intensities of neighbouring pixels is at a maximum. Similarly, corner points are points in the image for which the curvature of edge lines is maximised. Two of the most common edge detection algorithms are the Laplace and Sobel operators, both of which can be defined in terms of 3 x 3 convolution kernels. The results of applying these operators to an image is hown in Figure 1 :

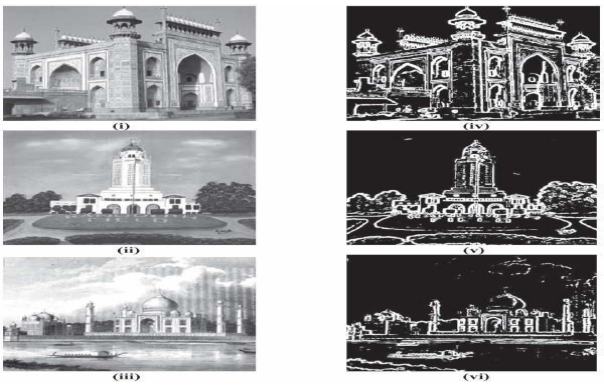


Figure 1. *Edge Detection sample*

3. System Architecture and Desain

The steady progression of Moores Law, combined with the improvements in manufacturing techniques have enabled increasing sophistication in embedded systems controllers. The increase in processing power has reached the point where current embedded CPUs are comparable in performance to the desktop CPUs of a decade ago. With better manufacturing techniques, these new embedded CPUs are also cheaper and more power efficient than the previous generation of processors. This increased capability has enabled the application of embedded systems to tasks which in the past would have been performed by much larger systems. In particular, there is increasing interest in the use of small, portable, imaging devices.

Of course, one of the most important parts of embedded system development is design. Spending time to thoroughly analyze the architecture of uClinux and to provide only the services needed by an embedded system like a Spartan 3E FPGA and its associated development board was top priority for the uClinux and Petalogix development teams. Because of this, large amounts of documentation are available on the Internet detailing every aspect of the boot loaders and kernels in the uClinux and Petalinux projects. In this section, we will explain how the various components work together and detail their key roles in the entire architecture.

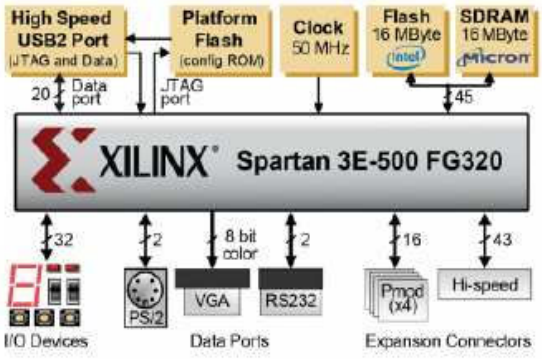


Figure 2. *Xilinx Spartan 3E Starter Board Hardware Features*

3.1. Microblaze system

The MicroBlaze processor is a 32-bit Harvard Reduced Instruction Set Computer (RISC) architecture optimized for implementation in Xilinx FPGAs with separate 32-bit instruction and data buses running at full speed to execute programs and access data from both on-chip and external memory at the same time. The backbone of the architecture is a single-issue, 3-stage pipeline with 32 general-purpose registers (does not have any address registers like the Motorola 68000 Processor), an Arithmetic Logic Unit (ALU), a shift unit, and two levels of interrupt. This basic design can then be configured with more advanced features to tailor to the exact needs of the target embedded application such as: barrel shifter, divider, multiplier, single precision floating-point unit (FPU), instruction and data caches, exception handling, debug logic, Fast Simplex Link (FSL) interfaces and others. This flexibility allows the user to balance the required performance of the target application against the logic area cost of the soft processor.

Figure 3 shows a view of a MicroBlaze system. The items in white are the backbone of the MicroBlaze architecture while the items shaded gray are optional features available depending on the exact needs of the target embedded application. Because MicroBlaze is a soft-core micro-

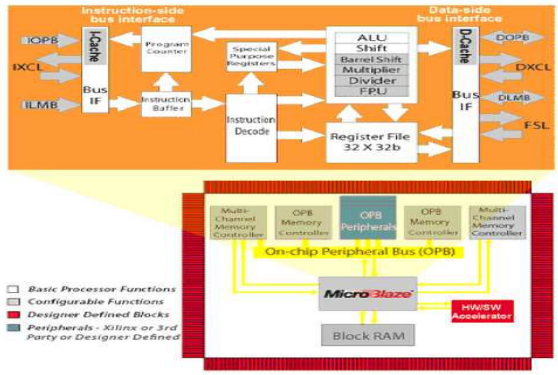


Figure 3. *View of MicroBlaze System*

processor, any optional features not used will not be implemented and will not take up any of the FPGAs resources.

The MicroBlaze pipeline is a parallel pipeline, divided into three stages: Fetch, Decode, and Execute. In general, each stage takes one clock cycle to complete. Consequently, it takes three clock cycles (ignoring delays or stalls) for the instruction to complete. Each stage is active on each clock cycle so three instructions can be executed simultaneously, one at each of the three pipeline stages.

MicroBlaze implements an Instruction Prefetch Buffer that reduces the impact of multi-cycle instruction memory latency. While the pipeline is stalled by a multi-cycle instruction in the execution stage the Instruction Prefetch Buffer continues to load sequential instructions.

The MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data accesses and instruction accesses. MicroBlaze does not separate between data accesses to I/O and memory (i.e. it uses memory mapped I/O). The processor has up to three interfaces for memory accesses: Local Memory Bus (LMB), IBMs On-chip Peripheral Bus (OPB), and Xilinx CacheLink (XCL). The LMB provides single-cycle access to on-chip dual-port block RAM (BRAM).

The EDK tools have built in C/C++ compilers to generate the necessary machine code for the MicroBlaze processor. The MicroBlaze processor is useless by itself without some type of peripheral devices to connect to and EDK comes with a large number of commonly used peripherals. Many different kinds of systems can be created with these peripherals, but it is likely that you may have to create your own custom peripheral to implement functionality not available in the EDK peripheral libraries and use it in your processor system.

To maximize the automation that EDK tools provide with you, when creating your own custom peripheral you must take into account the following considerations: The processor system by EDK is connected by On-chip Peripheral Bus (OPB) and/or Processor Local Bus (PLB), so your custom peripheral must be OPB or PLB compliant (see note). Meaning the top-level module of your custom peripheral must contain a set of bus ports that is compliant to OPB or PLB protocol, so that it can be attached to the system OPB or PLB bus.

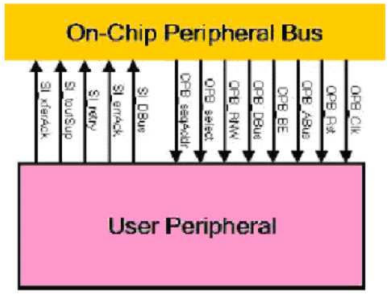


Figure 4. *OPB bus protocol in Microblaze system*

- **Determine Interface**: Identify the bus interface (OPB or PLB) your custom peripheral should implement, so that it can be attached to that bus in your processor system,
- **Implement and Verify Functionality**: Implement your custom functionality, reuse the common functionality already available from EDK peripheral libraries as much as possible, and verify your peripheral as a stand-alone core,
- **Import to EDK**: Copy your peripheral to an EDK recognizable directory structure and create the PSF interface files (.mpd/.pao) so that other EDK tools can access your peripheral,
- **Add to System**: Add your peripheral to the processor system in EDK.

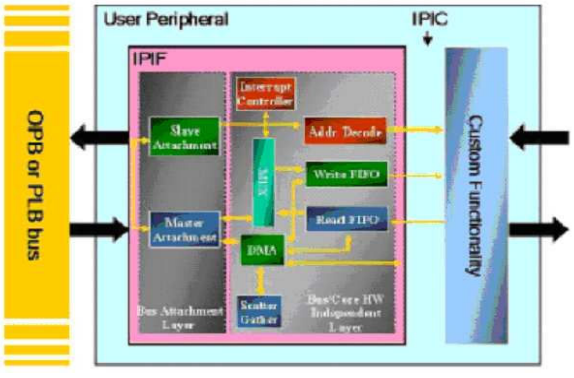


Figure 5. *Using IPIF Module in User Peripheral*

3.2. Xilinx EDK System

One of tools that the Xilinx EDK provides developers is a system block diagram, which displays a high level model of the entire project design. In this case, we will describe the main components of the FSBoot system block diagram, which are broken up into the three main classes of memory, processor, and peripherals.

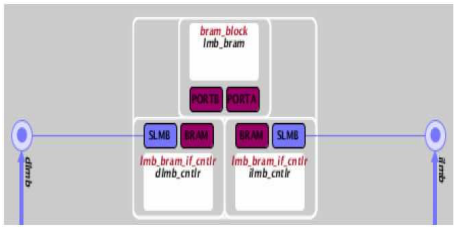


Figure 6. *FSBoot Memory Components*

The first main class is the system memory. In fact, FSBoot actually contains several different types of memory, one of which is shown above in Figure 6. The BRAM, or block RAM, is the Spartan 3E Starter Boards onboard RAM, and is used primarily for storing system data like hardware configuration information. Also, it has extremely high data rates due to its built in synchronous FIFOs. It is also very scalable as each module of the block RAM ranges from 18Kbit to 36Kbit in size and can be cascaded to grow even larger. Specific to FSBoot and many other systems that integrate with the MicroBlaze soft core processor are the data and instruction memory controllers at the bottom. They simply guide each type of data to the respective memory spaces in the MicroBlaze processor, which will be discussed below.

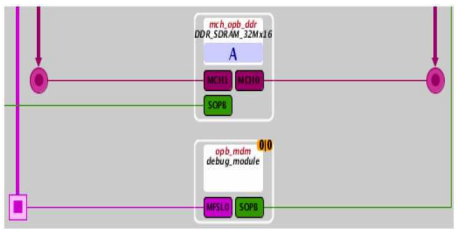


Figure 7. *Additional FSBoot Memory Space*

Extra memory addressing was also built into FSBoot, in case more sophisticated second stage boot loaders outside of UBoot were needed. FSBoot can access DDR SDRAM chips provided the board supports them. While the Spartan 3E Starter Board does not have this capability, the Virtex boards do, so this addition is welcome.

The second main system class is, of course, the processor. FSBoot, written for Xilinx development boards, took the route of using the MicroBlaze soft core processor. This was simply a design decision, since the uClinux kernel was actually compatible with MicroBlaze and PowerPC. In our senior project, however, we welcome this choice since we are intimately familiar with development systems for it. It also doesnt hurt that it was designed by Xilinx specifically for their FPGAs, making it very fast and reliable on their boards. The processor incorporates RISC based architecture and includes a 5 stage pipeline that simply completes one instruction per cycle. It is also highly configurable. Nearly everything, from its cache sizes to its bus interfaces, can be customized. While it has optional support for the EDK memory management unit IP core, we did not opt to use it in this project. It is literally the bridge between the system peripherals (green), and the system memory (blue). The processor can link its data and instruction on-chip peripheral bus to the main CoreConnect OPB bus for access to a wide range of different modules, some of which will be described below. Also, MicroBlaze links to the data and instruction memory bus controllers above it, for fast access to block RAM.

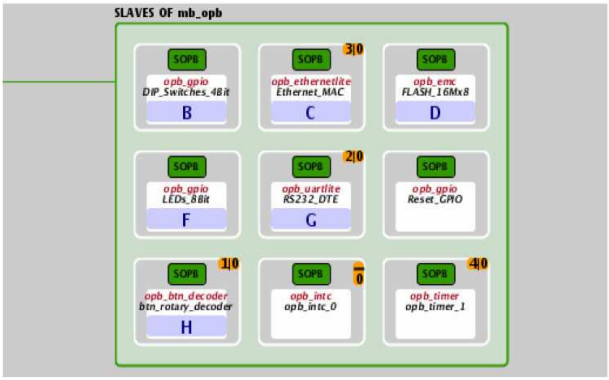


Figure 8. *FSBoot Supported Peripherals*

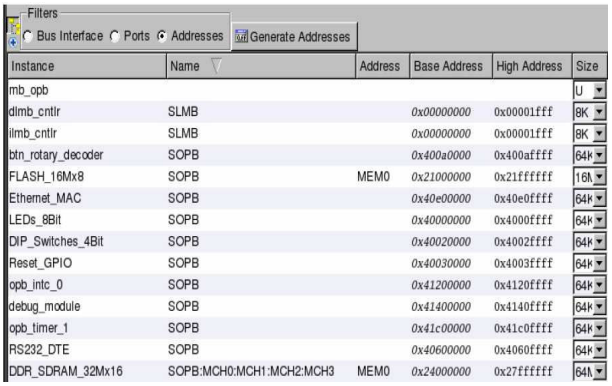
The last main system class is FSBoots supported peripherals. Since FSBoot is still only a boot loader, the only peripheral needed would be the interface in which it could download the UBoot image. The primary interface is the UART, but it also supports network transfer of binary images, so an Ethernet GPIO is also included.

3.3. Xilinx EDK Address Mappings

The final requirement for the FSBoot system design is the memory addresses in which all peripherals are mapped. Figure 9 below details the address locations for these major system components:

- Flash memory,
- Ethernet,

- Rotary decoder,
- Switches,
- DTE serial port,
- DDR SDRAM slot,



Instance	Name	Address	Base Address	High Address	Size
mb_opb					U
dlmb_cntlr	SLMB		0x00000000	0x00001fff	8K
ilmb_cntlr	SLMB		0x00000000	0x00001fff	8K
btn_rotary_decoder	SOPB		0x400a0000	0x400affff	64K
FLASH_16Mx8	SOPB	MEM0	0x21000000	0x21ffffff	16M
Ethernet_MAC	SOPB		0x40e00000	0x40e0ffff	64K
LEDs_8Bit	SOPB		0x40000000	0x4000ffff	64K
DIP_Switches_4Bit	SOPB		0x40020000	0x4002ffff	64K
Reset_GPIO	SOPB		0x40030000	0x4003ffff	64K
opb_intc_0	SOPB		0x41200000	0x4120ffff	64K
debug_module	SOPB		0x41400000	0x4140ffff	64K
opb_timer_1	SOPB		0x41c00000	0x41c0ffff	64K
RS232_DTE	SOPB		0x40600000	0x4060ffff	64K
DDR_SDRAM_32Mx16	SOPB:MCH0:MCH1:MCH2:MCH3	MEM0	0x24000000	0x27ffffff	64M

Figure 9. *FSBoot Memory Address Mappings*

The most important thing to note is that the designers of FSBoot maxed out the flash memory at sixteen megabytes. This is due to the total of FSBoot, UBoot, and kernel images equaling about six megabytes, not including all the user created drivers and applications that could be added in the future.

3.4. Boot Loader and Kernel Hierarchy

After FSBoot completes its boot process, UBoot is pushed to the board as per our instructions in the next section and then completely takes over control of the board. UBoot is the secondary boot loader chosen specifically because of its reliability and ability to adapt to nearly any embedded architecture. In addition to MicroBlaze and PowerPC, it can also handle x86, MIPS, and ARM architectures. It is this versatility that compelled the uClinux developers to integrate it into their tool chain. Although UBoot has several commands that could be run in its command prompt, one being a command to erase all flash memory that we will use frequently in kernel development, the main use is to bind the uClinux kernel to flash memory. Again, a serial connection is the primary choice to pull the kernel image from the local workstation to the board. Once transfer completes, UBoot clears the flash memory, assigns the image a checksum and Ethernet MAC address if network transfer was enabled, and binds the image to a user specified location in flash. We detail the entire three stage boot process below in Figure 10:

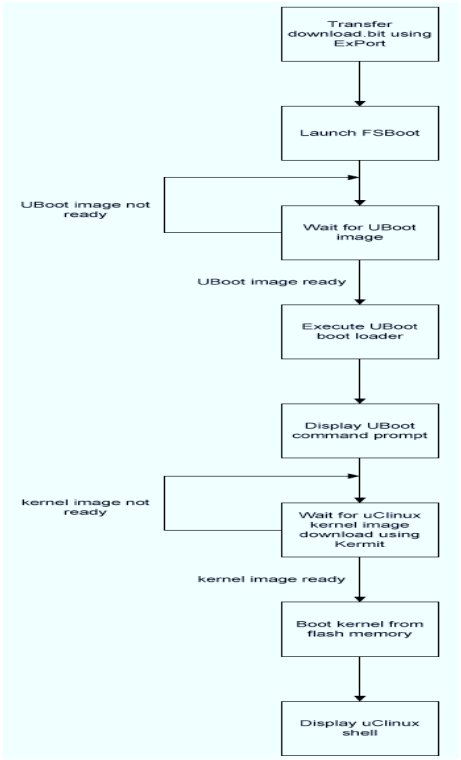


Figure 10. *Three Stage Boot FLow Process*

4. Installing uClinux on a Spartan 3E Starter Board

Embedded system development is unique in that it combines both software and hardware development in parallel. Part of the difficulty, then, is finding a balanced work setup that can speed up the development process of both the software and hardware aspects of an embedded systems project. The same can be said for installing a Linux variant on a development board like the Spartan 3E Starter Board. There are many different development setups one can pick from, but only one reliable and economical choice.

4.1. Installing the Xilinx ISE WebPACK and EDK free for 60 days on a Linux

The first step to configuring the entire blended OS development environment for this project was choosing the correct Linux distribution that worked both with uClinux and the Xilinx ISE and EDK. The choices came down to Opensuse 10.3.

Installing the Xilinx ISE and EDK is also simple, but there are some things to note during and after the initial installation. First, the recommended path prefix for uClinux is /opt/, so install the ISE and EDK in version

tagged folders under this directory as follows: /opt/ise9.2i and /opt/edk9.2i.

Second, all web updates must be installed immediately after the initial installation. Lastly, before launching an instance of the EDK, change to the directories in which the ISE and EDK were installed in a shell client and run the following commands:source settings.sh and source settings.sh.

These commands assume that the user is using bash shell. A corresponding settings.csh works for those using C shell. Also, these source commands must be run for each instance of the EDK created.

4.2. Installing XILINX JTAG tools on linux without proprietary kernel modules

When using Xilinx JTAG software like Impact, Chipscope and XMD on Linux, the proprietary kernel module windrvr from Jungo is needed to access the parallel- or usb-cable. As this module does not work with current linux kernel versions (more than 2.6.18) a library was developed, which emulates the module in userspace and allows the tools to access the JTAG cable without the need for a proprietary kernel module. We first installing the libusb-dev package, usb-driver-HEAD.tar.gz, and fxload. Doing Export LDPRELOAD=/usr/lib/libusb-driver.so (sh and bash). After that, we can added statement in the file /etc/udev/rules.d/50-xilinx-usb-pav.rules with instruction :

ACTION==”add”,BUS==”usb”

SYSFSidVendor==”03fd”,MODE=”666”.

4.3. Installing PetaLinux on a Linux

While uClinux is the Linux variant that we installed on the Spartan 3E Starter Board, there have been several versions of the distribution over the years. The PetaLinux project was created to organize the various versions and their precompiled counterparts. The entire PetaLinux project must be installed on the same workstation and same OS as the Xilinx ISE and EDK. The current PetaLinux tarball is available at http://developer.petalogix.com/ and must simply be extracted in a folder convenient to development, like the users home folder. In addition to PetaLinux, several extra tasks must also be completed to compliment PetaLinux and the Xilinx EDK. First, a folder named tftpboot must be created at root level . This folder will contain all kernel and boot loader images that PetaLinux creates during the compilation process. This folder name is tied to the PetaLinux setup process, so we recommend against changing it. Second, like in the case of the Xilinx EDK, every instance of the PetaLinux setup process must begin with a source command as follows: cd petalinux source ./settings.sh

Again, a corresponding settings.csh for C shell users is also included. Lastly, the main interface for downloading data to the Spartan 3E Starter Board, the RS232 serial port, requires some type of serial console to communicate with OpenSuse 10.3. For this project, we chose the free Kermit console program, available at http://www.columbia.edu/kermit/ck80.html. After installation, a script file with all Kermit settings named .kermrc must be created in the user home directory / exactly as pictured in Figure 11:

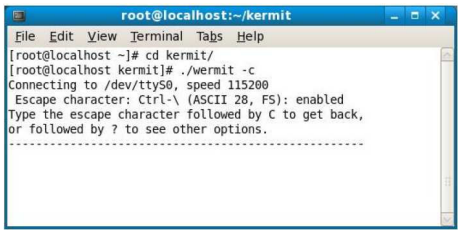


Figure 11. *Kermit free console program*

4.4. Configuring uClinux for a Spartan 3E Starter Board

Begin by connecting the serial, power, and JTAG cable to the Spartan 3E Starter Board. The serial cable should be connected to the DTE serial port on the Spartan 3E. Again, before working with PetaLinux, make sure to always source the settings.sh script in the main PetaLinux directory. Next, open a terminal and change directories to software/petalinuxdist. Then run the make menuconfig command as shown in Figure 12:

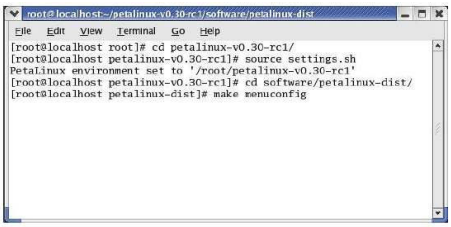


Figure 12. *Setup in PetaLinux Config Menu*

Finally, issue a make all command to compile the kernel images. As with any code, every change made to the kernel, either by hand or through the PetaLinux Setup menu, must be followed by a compile. After the compilation completes, your /tftpboot/ folder should include the files shown in Figure 13.

Before moving on to the actual Spartan 3E configuration and downloading of the kernel images, be sure to create the download.bit file for the Spartan 3E-500 Revision D board

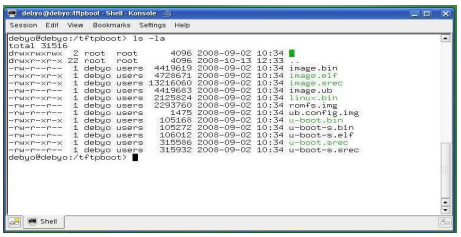


Figure 13. *Kernel Images in directory tftpboot*

by opening and updating the reference design thats located in the hardware directory of the PetaLinux folder.

The Kermit window should show the First Stage Boot Loader executing, as shown in Figure 14. Press S after the bit file finishes downloading:

```
FS-BOOT: Waiting for SREC image....
FS-BOOT: Image download successful.
FS-BOOT: Warning image location differ from default boot location.
FS-BOOT: Press 'n' to boot old image.
FS-BOOT: Use new image.
FS-BOOT: Booting image...
SDRAM :
        U-Boot Start:0x23fc0000
        Malloc Start:0x23f60000
        Board Info Start:0x23f5ffd0
        Boot Parameters Start:0x23f4ffd0
FLASH: 16 MB
ETHERNET: MAC:<NULL>

*** Warning - bad CRC, using default environment

U-Boot>
```

Figure 14. *FS-BOOT Executing when Startup*

On a separate terminal session, input the following command to transfer the second stage boot loader UBoot to the board:

```
\$ cat /tftpboot/u-boot.srec > /dev/ttyS0
```

In the Kermit window, a spinning character should appear to confirm that the image is indeed being transferred. Since this is the first time that should be setting up UBoot, it should default to the UBoot environment and shell as shown in Figure 15:

```
(/usr/local/petalogix/projects/fs-boot/) C-Kermit>connect
Connecting to /dev/ttyS0, speed 115200
 Escape character: Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
----------------------------------------------------
## Total Size      = 0x0000034E = 846 Bytes
## Start Addr      = 0x24000000
U-Boot>
```

Figure 15. *UBoot Default Environment*

The UBoot environment variables specific to the Spartan 3E Starter Board were compiled into a separate script file alongside the uClinux compilation, so the next step is to transfer that script to the board and bind it to flash memory. Input the following command into UBoot to load the script. A follow up message should appear just after the command confirming that the board is ready for transfer:

```
U-Boot> loadb 0x24000000
```

Ready for binary (kermit) download to 0x24000000 at 115200 bps

Next, press and hold the Control and buttons down on your keyboard, and then press C. This will escape out to the Kermit shell, allowing to send files to the running UBoot. Run the following command to transfer the UBoot script to the board:

```
C-Kermit> send /bin /tftpboot/ub.config.img
```

Typing connect in the Kermit window will connect back to the running UBoot session and show the amount of bytes transferred to the board, as shown in Figure 16:

```
U-Boot> autoscr $(fileaddr)
## Executing script at 24000000
PetaLogix MicroBlaze-Auto Board Configuration
---------------------------------------------
Network Configuration:
MTD Configuration:
Clobber DRAM Configuration:
Bootloader Configuration:
Kernel Configuration:
Boot Configuration:
Saving Configurations...
Saving Environment to Flash...
Un-Protected 2 sectors
Erasing Flash...
. done
Erased 2 sectors
Writing to Flash... done
Configuration Completed
U-Boot>
```

Figure 16. *Transfering Files to UBoot*

Issue the following command to run the script:

```
U-Boot> autoscr \$(fileaddr)
```

The script should output configuration lines as shown in Figure 17:

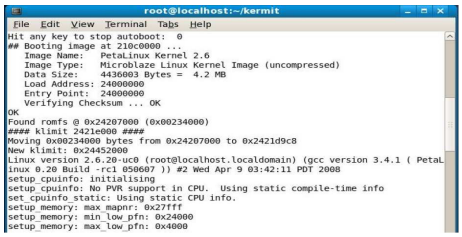


Figure 17. *UBoot Configuration Script Output*

Input the loadb command with a 0x24000000 starting address in the UBoot shell. Enter the Kermit session and transfer the UBoot binary image to the board using this command:

```
C-Kermit> send /bin /tftpboot/u-boot-s.bin
```

Connect back to UBoot after the transfer completes and issue the following commands to bind the image to flash memory

```
U-Boot> protect off \$(bootstart)+\$(bootsize)
U-Boot> erase \$(bootstart)+\$(bootsize)
U-Boot> cp.b \$(clobstart)\$(bootstart)
        \$(filesize)
```

Finally, transfer the kernel image image.ub to the board using Kermit and bind it to flash with the following commands:

```
U-Boot> protect off\$(kernstart)+\$(kernsize)
U-Boot> erase\$(kernstart)+\$(kernsize)
U-Boot> cp.b \$(clobstart)\$(kernstart)\$(filesize)
```

When prompted for a username, enter root. For the password, again input root. After the login, a generic shell should be available to browsing the uClinux file system and running applications.

4.5. Clearing Flash Memory and Resetting UBoot Environment Variables

Since every kernel change requires a recompile, retransferring a newly updated kernel to the Spartan 3E Starter Board is a common task. Unfortunately, however, this retransfer requires to clear the flash memory and start fresh. Without UBoot, this would require manually finding the start and end memory addresses that the kernel occupies and freeing every block. Thankfully, UBoot has a built in command for clearing all environment variables. Simply issue:

```
U-Boot> run eraseenv
```

Be sure to execute this command before transferring over an updated kernel image. Also, the entire process of running the UBoot configuration script and binding the UBoot binary image to flash must be completed again as well.

4.6. Implemented of Computer Vision Algorithm in uClinux

uClinux stores all user applications in the $PETALINUX/software/user-appsdirectory. This is where any custom user applications can be placed. In order to create a template for a new application, the following commands must be issued in a command prompt after uClinux is installed:
$ cd $PETALINUX/software/user-apps
$ petalinux-new-app my_new_app

New application template successfully created in software/userapps/my_new_app In this example, replace my_new_app with the desired application name. This command will create a new directory for the application in $PETALINUX/software/user-apps/my_new_app. In order to add an already existing application to uClinux, enter the exact same commands into the prompt. After adding the application, copy the applications source files to the newly created directory and edit the Makefile to include the new application in its build rules. This Makefile automatically adds the necessary flags to the compilation process in order to include the libraries needed to run the application on the Spartan 3E architecture. This entire compilation process is handled by the uClinux cross compiler provided by the uClinux developers, which is executed during the kernel build process. This cross compiler allows C code written in a Linux development environment running on Intel architecture to run as intended on a different architecture, like the Xilinx Spartan 3E. After an application is added, it must be installed into the root filesystem. In order to do this, it must first be installed into the romfs of the target system, which is then added to the kernel image at the image build process. To install the application, enter the following commands into the prompt:
$ cd $PETALINUX/software/user-apps/my_existing_app
$ make romfs

5. Conclusion and Perspectives

We have presented a Microblaze 32-bit softcore technology that incluced in Xilinx system. We have proposed implementation of uclinux port that it can access Microblaze. For this first experiment with Sobel filter, we wanted to know more about Microblaze hardware et uclinux software opensource characteristics and their performances. After we can implemented two system, we will also used the Computer Vision Algorithm in order to exploit architecture Microbaze with using Petalinux distribution for implemented in VHDL model. In future work, we hope to embed VEX C sytem technology for Microblaze 32 bit soft-core processor in FPGA Xilinx, because modern FPGA chips, with their large memory capacity and reconfigurable potential, are opening new frontiers in rapid prototyping of embedded systems. To that aim, we will analyze architecture Microblaze and resources needed for the implementation.

References

[1] Chin, Lixin. *FPGA Based Embedded Vision Systems*, Centre for Intelligent Information Processing Systems, The University of Western Australia, October 2006.

[2] Estrada, Brian and Patrick Mariano., *Development of uclinux Platform for Cal Poly Super Project*, Computer Engineering Departement, California Polytechnic State University, San Luis Obispo, 2008.

[3] HOWTO-1., *Howto create a project for a simple uclinux ready microblaze 4.0 design on xps (xilinx platform studio) for spartan-3e.*, http://www.teknologisk.dk/20356.

[4] HOWTO-1., http://ecasp.ece.iit.edu/mbtutorial.pdf.

[5] Klenke, Robert H., *Experiences using the xilinx microblaze softcore processor and uclinux in computer engineering capstone senior design projects.*, Microelectronic Systems Education, 2007. MSE apos;07. IEEE International Conference, pages 123 – 124, 3-4 June 2007.

[6] Graef, Gerald., *Graphics Format for Linux.*, http://www.linuxjournal.com/article/1119.

[7] Gemoth, Michael., *Installing XILINX JTAG tools on linux without proprietary kernel modules http://www.rmdir.de/%7Emichel/xilinx/.*

[8] *Petalogix,Linux Solutions for a Reconfigurable Word*, http://developer.petalogix.com/