A&QT-R 2004 (THETA 14)
2004 IEEE-TTTC International Conference on Automation, Quality and Testing, Robotics
May 13 – 15, 2004, Cluj-Napoca, Romania

DESIGN OF AN IMAGE PROCESSING FILTER USING THE JBITS PACKAGE

Zoltan Baruch, Zoltan Balazsi

Technical University of Cluj-Napoca, Romania, Computer Science Department
E-mail: Zoltan.Baruch@cs.utcluj.ro

Abstract. This paper describes the design of a digital filter and its implementation in an FPGA device. The filter designed and implemented is a grayscale filter used in image processing applications. For this design the JBits package was used, which contains a set of Java classes allowing to generate and modify configuration bitstreams for Xilinx Virtex FPGA devices. Using this package, all configurable resources of a Xilinx Virtex FPGA device can be individually set under software control. Therefore, a dynamic and partial reconfiguration of the FPGA device is possible from a Java application. The dynamic reconfiguration allows to modify the filter parameters during run-time, without the need to completely reconfigure the device.

Keywords: FPGA devices, image processing, reconfigurable architectures, JBits.

1. INTRODUCTION

Traditionally, image processing applications are implemented using general-purpose DSP (*Digital Signal Processing*) chips. Although the DSP chips are optimized for mathematical operations, their architecture is serial. Multiply and accumulate (MAC) operations, typically found in DSP applications, are implemented using shared resources.

FPGA (*Field-Programmable Gate Array*) devices represent an alternative to implement image processing algorithms. These devices are suitable for arithmetic-intensive image processing functions. By implementing an image processing algorithm in an FPGA device, the design can take advantage of distributed resources and parallel processing in order to exceed the performance of single or multiple DSP processors [1].

An FPGA device contains logical blocks and interconnection lines between them. The operations that are to be performed by the device are specified by configuration bits, which define the various blocks' function and the interconnections between blocks. Like processors, FPGA devices can be programmed after fabrication in order to solve any computational problem allowed by their hardware resources. This programmability differentiates processors and FPGA devices from application-specific functional units, which can perform only a function or a limited number of functions.

Another advantage of FPGA devices is that they can be partially or completely reconfigured during operation. Therefore, multiple functions can be performed using a minimal configuration. For example, an FPGA device could be used in a system that performs one of several image processing functions, and the device can be reconfigured during operation to switch from one function to another.

This paper describes the design of a grayscale filter and its implementation in an FPGA device. The filter designed and implemented can be used in image processing applications. The JBits SDK was used for this design, and therefore the design is run-time reconfigurable. The run-time and partial reconfiguration allows to modify the filter parameters during operation, without the need to completely reconfigure the device.

The organization of the paper is as follows. Section 2 presents an overview of the JBits SDK and of the design flow when using this tool. Section 3 describes the design of a grayscale filter using the JBits SDK. The testing procedure of the filter is presented in Section 4. Concluding remarks follow in Section 5.

2. THE JBITS SDK

The JBits SDK is a set of Java classes which provide an Application Program Interface (API) for generating and modifying configuration bitstreams for the Xilinx Virtex FPGA devices. This interface operates either on configuration bitstreams generated by Xilinx synthesis tools, or on bitstreams read back from the actual device [5]. Using the JBits SDK, all configurable resources of the device can be individually set under software control. Therefore, a dynamic and partial reconfiguration of the Xilinx Virtex FPGA devices is possible from a Java application.

The JBits SDK provides access to all the resources of a Virtex FPGA device, including the Look-Up Tables (LUTs) inside each Configurable Logic Blocks (CLBs) and the routing resources adjacent to the CLBs. The device architecture is represented as a two-dimensional array of CLBs, and each CLB is referenced by a row and column. The JBits SDK allows to develop run-time reconfigurable (RTR) systems in a high-level language. This SDK can also be used to produce or modify traditional static design bitstream files for Virtex FPGA devices.

Figure 1 illustrates the JBits design flow [4]. The user-written Java application configures the FPGA device by communicating with the board containing the device. The bitstream input to the Java application can be a null bitstream or a bitstream for an existing design. The application may use the bit-level interface provided by the JBits API, which allow to set or clear a single bit or a group of bits in the bitstream. This is a low-level interface responsible for knowing the bit location in the bitstream of a given configuration data for the devices supported in the Virtex FPGA family. The bit-level interface interacts with the Bitstream class, which manages the device bitstream and provides support for reading and writing bitstreams from and to files. This class can also read back the existing configuration data from the operating device, which is necessary for dynamic reconfiguration.

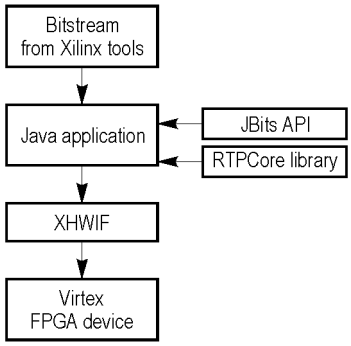

Figure 1. The design flow using the JBits SDK.

The user application may also use the *Run-Time Parameterizable Core* (*RTPCore*) library provided by the JBits SDK. This library is a collection of Java classes defining macrocells or cores that can be dynamically parameterized and relocated within a device. Examples of cores are registers, counters, adders, multipliers and other standard Xilinx Unified Library logic and computation functions. In addition to these primitive cores, other non-primitive RTP cores can be used, which are created by instantiating primitive or non-primitive subcores connected with nets and buses.

The *Xilinx Hardware Interface* (XHWIF) provides a portable layer to connect JBits applications to reconfigurable hardware. By using this layer, JBits applications can run without recompilation on various hardware platforms. For example, the host computer executes the JBits application and configures a Virtex FPGA device located in the PCI slot using the XHWIF API. This enables run-time configuration and reconfiguration of the Virtex FPGA device.

3. DESIGN OF A GRAYSCALE FILTER

3.1. Design Overview

The grayscale filter computes the arithmetic mean of the red, green, and blue components of a pixel value from an image. Then a new pixel is generated, in which all the three components are set to the arithmetic mean of the red, green, and blue components of the corresponding pixel from the original image. In this way, a gray-scaled image is generated from the original image.

For the design of the grayscale filter, a class called *GrayFilter* was created using the classes provided by the JBits SDK, version 2.8. This class is inherited from the *RTPCore* class. *RTPCore* is an abstract class which defines methods for primitive, non-primitive, and top-level cores. A primitive core does not contain subcores and uses JBits calls to implement its functions. A subset of the standard Xilinx Unified Library primitives is provided by JBits. A non-primitive core contains subcores and implements its functions by instantiating primitive and non-primitive subcores connected with nets and buses. A top-level core does not have a parent.

The JBits *CoreTemplate* package was used for the design, which allows to efficiently create dynamically parameterizable and relocatable cores. This package defines classes for *RTPCore* components such as modules, nets, buses, ports and pins [4]. Cores created with the *CoreTemplate* package may have CLB (*Configurable Logic Block*), slice, or logic element placement granularity.

The *GrayFilter* class implements a module with four input ports: a clock port, and three ports for the red, green, and blue components of a pixel value. The output port will contain the grayscale value of a pixel. These ports are connected to external signals; for the pixel ports, these signals are 8-bit buses. For each pixel component, an 8-bit constant is defined, which is multiplied with the pixel component value. The results are three 16-bit values. These values are summed using two adders, and the result is stored into an 18-bit register. To compute the arithmetic mean, a scaling factor is required, which is multiplied with the 18-bit sum. The result of the multiplication is a 26-bit value. In order to get an 8-bit result, a bounder module is used, which limits the result to a maximum value.

The steps followed for designing the grayscale filter are described next.

3.2. Computing the Core's Granularity and Dimensions

The *GrayFilter* core has fixed CLB granularities. The horizontal granularity of the core is computed with the *calcWidthGran()* method, and the vertical granularity is computed with the *calcHeightGran()* method:

```
public static int calcWidthGran() {return Gran.CLB;}
public static int calcHeightGran() {return Gran.CLB;}
```

The core has a fixed width of 24, the units being given by the width granularity (CLB). The height of the core is 16 CLB units. These dimensions are returned by the *calcWidth()* and *calcHeight()* methods, respectively:

```
public static int calcWidth() {return 24;}
public static int calcHeight() {return 16;}
```

3.3. Defining the Core's External Characteristics

An *RTPCore* constructor defines the core's ports, width, height, width granularity and height granularity. Any core parameters that affects the dimensions or granularity should be passed to the constructor. External signals connected to the ports can also be passed as parameters.

An instance of the grayscale filter is created with the following constructor:

```
public GrayFilter (String instanceName, Net clk, Bus redIn, Bus greenIn,
                   Bus blueIn, Bus grayscaleOut) throws CoreException {
```

The parameters *redIn*, *greenIn*, and *blueIn* are external signals (buses) connected to the input ports of the core. The parameter *grayscaleOut* is a signal connected to the output port of the core. The size of the input and output buses is checked in the constructor with the *CheckParameters()* method. If the size of the input and output buses is different than 8, the *CoreParameterException* exception is thrown.

Setting the external characteristics of the core consists in the following steps:

- Assigning an instance name to the core. This is done by calling the superclass constructor *super()*:

```
super (instanceName);
```

- Creating the ports and connecting external signals to these ports. For this step, the *newInputPort()* and *newOutputPort()* methods are used:

```
clkPort = newInputPort ("CLK", clk);
redPort = newInputPort ("Red", redIn);
grayscaleOutPort = newOutputPort ("GrayScale", grayscaleOut);
```

- Setting the dimensions and granularity of the core. This is done by calling the *setWidth()*, *setHeight()*, *setWidthGran()*, and *setHeightGran()* methods:

```
SetWidth (calcWidth());
SetHeightGran (calcHeightGran());
```

3.4. Implementing the Core

A core may be implemented only after it has been placed by the application that called the core's constructor. The core is placed by assigning the core's relative offset. Finally, the *implement()* method is called. For the designed grayscale filter, the *implement()* method has three parameters, which are the weighting constants for the three

color components. These constants are *redWt*, *greenWt*, and *blueWt*. The operations performed by the *implement (int redWt, int greenWt, int blueWt)* method of the core are the following:

- Checking the parameters and computing the scaling factor. The parameters are checked with the *checkParameters (int redWt, int greenWt, int blueWt)* method. The scaling factor is computed by calling the *calcScaleFactor()* method:

```
long scaleFactor = calcScaleFactor (redWt, greenWt, blueWt);
```

- Creating the buses and the internal connections. A new bus can be created with the *newBus ("BusName", width)* method, and an internal connection inside a core can be created with the *newNet("NetName")* method. The following fragment shows the creation of some buses and nets used in the *GrayFilter* core.

```
Bus redIn = newBus ("RedIn", redPort.getWidth());
Bus redMultOut = newBus ("Red Mult Out", 16);
Net clkNet = newNet ("clk");
```

- Creating the subcores. The *GrayFilter* core uses the following types of subcores defined in the JBits packages: constant, multiplier, adder, register, and bounder. The following fragment illustrates the creation of subcores for a multiplier and an adder.

```
Multiplier redMult = new Multiplier("Red Mult", clkNet,
                                    redMultConstant, redIn, redMultOut);
Adder rgAdder = new Adder("Red+Green", redMultOut, greenMultOut,
                                    rgAdderOut );
```

- Assigning buses and internal signals to ports. An internal signal may be assigned to a port with the *setIntSig(signal)* method of the *Port* class. The *signal* parameter may be an object of type *Bus* or *Net*. The following line assigns the *redIn* bus to the *redPort* port:

```
redPort.setIntSig (redIn);
```

- Obtaining the core's absolute offset. The designed core has CLB granularity, and therefore only the CLB offset is required. The offset is computed by calling the *calcAbsoluteOffset()* method, which returns an instance of the *Offset* class. Using this object, the *getVerOffset()* and *getHorOffset()* methods may be accessed to get the vertical and horizontal offset, respectively:

```
Offset offset = this.calcAbsoluteOffset();
int row = offset.getVerOffset(Gran.CLB);
int col = offset.getHorOffset(Gran.CLB);
```

- Adding the subcores. This is done by calling the *addChild (RTPCore child)* method for each of the subcores created.

- Setting the location of the subcores. This location is relative to the location of the parent core. First, the relative position of each subcore is obtained by calling the *getRelativeOffset()* method, and then the vertical and horizontal offsets are set by calling the *setVerOffset()* and *setHorOffset()* methods. Finally, the location of each subcore is set with the *place()* method:

```
private void place (RTPCore core, int verCLB, int horCLB)
```

- Implementing the subcores. After adding and placing a subcore, the *implement()* method is called for that subcore.

- Routing buses and internal connections. The *Bitstream.connect(signal)* method is called, where *signal* is a bus or internal connection. The subcores connected to the buses or internal connections must be placed and implemented before calling the routing method.

4. TESTING THE GRAYSCALE FILTER

To test the grayscale filter, the *GrayFilter* core was included into a test program, and a bitstream file was generated for the filter. This program reads the pixel values of an image from three BRAM memories and processes the pixels using the *GrayFilter* core. The resulting pixel values are stored into a BRAM memory. The pixel values are then read from this memory, converted into an image and displayed on the screen.

The operation of the *GrayFilter* core was tested using the BoardScope debugging tool and the VirtexDS simulator provided by the JBits SDK. BoardScope allows to graphically examine the operation of FPGA devices on a development board. VirtexDS allows to test Xilinx Virtex bitstream files without the need for an actual device. The bitstream generated by the application program was not downloaded to a Virtex FPGA device, due to several limitations of the current JBits version. For example, a complete design rule check (DRC) can not be performed with the JBits SDK, and an inappropriate configuration can damage the device.

5. CONCLUSIONS

This paper presented the design of a grayscale filter using the JBits package. First, the advantages of using FPGA devices for image processing were described. Then the main features of the JBits SDK were summarized. The main steps for designing and testing the grayscale filter were described.

The main advantage of the JBits package is that it allows access to all the internal resources of a Xilinx Virtex FPGA device. Therefore, it is possible to dynamically and partially reconfigure the FPGA device. The dynamic reconfiguration allows to modify the filter parameters during run-time, without the need to completely reconfigure the device. A disadvantage of using the JBits package is that the user should be familiar with the internal architecture of the target device. All the design steps performed automatically by the traditional design tools must be specified explicitly in the user program. Currently, the JBits SDK has several limitations which restrict its use for complex designs.

REFERENCES

[1] Goslin, G. R. [1995]. A Guide to Using Field Programmable Gate Arrays (FPGAs) for Application-Specific Digital Signal Processing Performance. Xilinx Application Note, http://www.xilinx.com/appnotes/dspguide.pdf.

[2] Kreuger, R. [2000]. Virtex-EM FIR Filter for Video Applications. Xilinx Application Note XAPP241 (v1.1), http://www.xilinx.com/xapp/xapp241.pdf.

[3] Sun Microsystems, Inc. [2002]. Java 2 SDK Documentation, Standard Edition, Version 1.2.2-001.

[4] Xilinx, Inc. [2001]. JBits API Documentation, JBits SDKVersion 2.8.

[5] Xilinx, Inc. [2001]. JBits Tutorial, JBits SDK Version 2.8.