RUN-TIME RECONFIGURABLE IMPLEMENTATION OF DSP ALGORITHMS USING DISTRIBUTED ARITHMETIC

Zoltan Baruch

Computer Science Department, Technical University of Cluj-Napoca,
26-28, Bariţiu St., 3400 Cluj-Napoca, Romania
Tel. 0264-194834/165, E-mail: Zoltan.Baruch@cs.utcluj.ro

Abstract: This paper describes the reconfigurable implementation of a digital filter in an FPGA device. The filter implemented is a Laplacian filter which is used in DSP and image processing applications. For an efficient FPGA implementation of the filter, distributed arithmetic techniques were used. For a run-time reconfigurable implementation, the JBits SDK was used, which allows partial reconfiguration of the Xilinx Virtex FPGA devices.

Keywords: Reconfigurable computing, DSP, FPGA, distributed arithmetic, Laplacian filter, JBits.

1. INTRODUCTION

DSP (Digital Signal Processing) applications are usually implemented using general-purpose DSP processors or special-purpose DSP chipsets and ASICs (Application Specific Integrated Circuits). Although the DSP processors and chipsets are optimized for mathematical operations, their architecture is serial. Multiply and accumulate (MAC) operations, typically found in DSP applications, are implemented using shared resources.

Many applications such as telecommunications, video/image processing, and networking require to process signals at very high rates. These rates may be beyond the capabilities of conventional DSPs available today or in the near future. Therefore, often several DSP chips are necessary to meet the performance required by these applications.

Field Programmable Gate Arrays (FPGAs) represent an alternative to implement DSP applications. These devices are suitable for arithmetic intensive DSP functions. By implementing a DSP application in an FPGA device, the design can take advantage of distributed resources (look-up tables, registers, multipliers, memory) and parallel processing to exceed the performance of single or multiple DSP processors (Goslin, 1995).

Another advantage of FPGA devices is that they can be partially or completely reconfigured during system operation (however, not all devices can be partially reconfigured). This feature means that 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 DSP functions, and the device can be reconfigured during operation to switch from one function to another.

This paper presents several techniques that can be used for the reconfigurable implementation of DSP and image processing applications, as well as an example application implemented in an FPGA device. Section 2 presents background information related to reconfigurable computing, the JBits SDK, and distributed arithmetic. Section 3 describes the recon-

figurable implementation of a Laplacian filter. Concluding remarks follow in Section 4.

2. BACKGROUND

2.1 Reconfigurable Computing

Reconfigurable computing (also called custom computing) allows to define the computing resources required by each application and to dynamically configure these resources onto a programmable logic device, usually a Field Programmable Gate Array. The reconfiguration of the target device is performed under software control. In this way, applications that are computationally demanding can be executed efficiently by allocating more hardware resources.

The capacity of today's FPGA devices has increased significantly in the last few years. However, the performance requirements have also increased, which require the use of parallel processing and distributed resources. Due to the limited configurable hardware available in a device, there is a need to swap the configurations in and out of the device upon demand and in real-time. FPGA devices require a relatively long reconfiguration time. To achieve run-time reconfiguration, a very high reconfiguration data rate is needed if the configuration has to be changed at a high frequency.

The attempts to reduce the reconfiguration data rates led to different reconfigurable architectures, such as multiple-context and partially reconfigurable. A *multiple-context* architecture stores multiple layers of configuration information, referred to as contexts. Only one context is active at a time, but a very fast context switch is possible. Each layer of the configuration memory can be independently written, so that the circuit defined by the active layer may continue its operation. A *partially reconfigurable* architecture allows a selective reconfiguration of the target device. The parts of the architecture which are not being configured may continue execution.

Usually, reconfigurable architectures are implemented using FPGA devices. These devices were originally designed as user-programmable alternatives to mask-configured gate arrays. An FPGA device is an array of logical blocks whose function and interconnection can be configured by the user. Most FPGA devices use small look-up tables as programmable computational elements. These tables are wired together with programmable interconnects.

Like processors, FPGA devices are "programmed" (configured) after fabrication to solve a particular computational task. In traditional processors, operations are temporally composed by sequencing them in time, using registers or memory to store intermediate results. In contrast, in configurable devices tasks are implemented by spatially composing primitive operators (DeHon, 2000). Because computations are performed using spatial pipelines composed of a large number of active computing elements, rather than sequentially reusing a small number of computing elements, higher performance can be achieved. However, this performance can only be obtained when the device performs the same operation from cycle to cycle.

Another advantage of FPGA devices is that they can control operations at the bit level, as opposed to processors which can control the operators only at the word level (DeHon, 2000). As a result, processors often waste part of their computational power when operating on custom data widths.

Reconfigurable computing has shown its effectiveness in several areas, such as video image processing, microprocessor emulation, encryption/decryption, or signal processing. The performance achieved by several reconfigurable architectures is often with one or two orders of magnitude greater than that of programmable processors (Vemuri and Harr, 2000).

2.2 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. 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 (Xilinx Inc., 2001). 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.

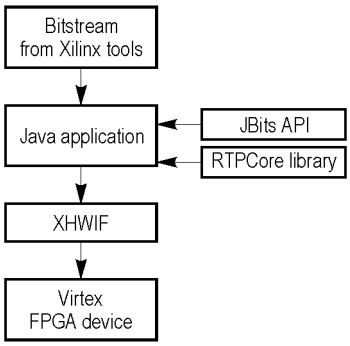

Figure 1. The JBits design flow.

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.

2.3 Distributed Arithmetic

Distributed arithmetic (DA) is one of the techniques that can be used to reduce the hardware resources required to implement DSP algorithms in FPGA devices. DA differs from conventional arithmetic in the order in which it performs operations. This technique targets the sum of products (also referred to as the vector dot product) computation that is essential in many DSP filtering and frequency transforming functions. DA rearranges the multiplication and addition operations in a sum of products to take advantage of small tables containing precomputed sums. These tables can be implemented using the look-up tables (LUTs) that are contained in most FPGA devices (Goslin, 1995).

The arithmetic sum of products that defines the response of linear networks can be expressed as:

$$y(t) = \sum_{i=1}^{N} C_i \cdot x_i(t) \quad (1)$$

where $y(t)$ is the response of the network at time t, $x_i(t)$ is the i^{th} input variable at time t, and C_i is a weighting factor of the i^{th} input variable that is time-invariant. In filtering applications the constants C_i are the filter coefficients, and the variables x_i are the prior samples of a single data source (such as an analog-to-digital converter). In frequency transforming the constants are the sine/cosine basis functions and the variables represent a block of samples from a single data source. In image processing the C_i coefficients are usually the components of a filtering operator and the variables are pixels of the image.

The computation described by Equation (1) implies a large number of multiplications, since a single output response requires the accumulation of N product terms, and usually a large number of outputs are to be generated in each second. With DA the operations of generating and summing the product terms are replaced by table look-up procedures that are easily implemented in FPGA devices.

Consider a four-product MAC function that uses a sequential shift-and-add technique to multiply four pairs of numbers and sum the results. This is shown in Figure 2, where SREG represents a shift register. Each multiplier forms partial products by multiplying a coefficient C_i by 1-bit of the data x_i at a time in an AND operation. That is, the coefficient is taken in parallel form and the data is taken serially. Each partial product is then iteratively shifted and accumulated with the previous result. The serial multiplier which contains an AND gate, an 1-bit adder, a register and a shifter is called a *scaling accumulator*. Therefore, the complex multiplier circuitry is reduced significantly. The four multiplications are performed simultaneously and the resulting products are summed in an adder tree. This process requires n clock cycles for data samples of n bits. The processing clock rate is therefore equal to the data rate divided by the number of data bits.

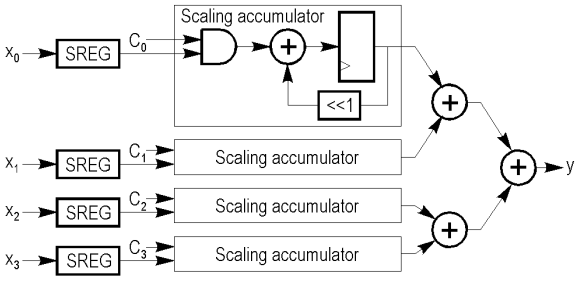

Figure 2. Four-product MAC operation using a shift-and-add technique.

Using distributed arithmetic, the order in which the partial products are summed is changed. Instead of individually accumulating each partial product and then adding the results, the partial products are added

first, and then the accumulate function is performed (Figure 3). This simple rearrangement replaces *N* multiplications followed by an *N*-input addition with a series of *N*-input additions followed by a multiplication. The number of scaling accumulators is reduced to one, which is an important saving in the case of a large number of product terms.

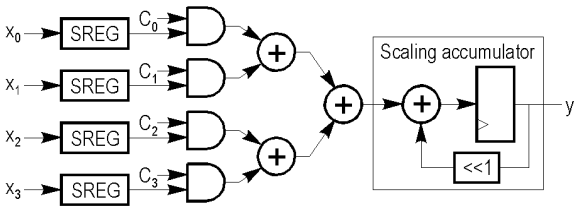

Figure 3. Four-product MAC operation using serial distributed arithmetic.

The coefficients C_i in many filtering applications are constants. Therefore, the outputs of the AND gates and the three adders in Figure 3 depend only on the four input bits from the shift registers. These AND gates and the three adders can be replaced with a 4-input (16-word) look-up table (Figure 4). The 16 entries in the table are sums of the constant coefficients for all the possible serial input combinations. The size of each word is wide enough to accommodate the largest sum without overflow. Negative values are sign-extended to the width of the table (Goslin, 1995).

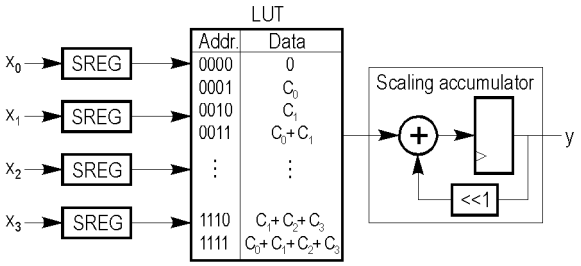

Figure 4. LUT-based four-product MAC operation using serial distributed arithmetic.

The structure in Figure 4 represents a completely serial DA technique. The serial processing limits the performance of such a circuit. Depending on the performance required and the resources available, the implementation can be optimized in several ways. A fully parallel DA technique can be used to achieve the fastest sample rates, while a serial technique can be used when the hardware resources should be reduced to the minimum and when lower sample rates are acceptable. In practice, often a combination between the two techniques is used.

3. IMPLEMENTATION OF A LAPLACIAN FILTER

3.1 Laplacian Operator

Filter operations are commonly used in DSP applications. One-dimensional DSP operations are widely used in image processing or video systems to perform resampling or filtering. In video systems, one-dimensional operations are typically performed in the horizontal dimension, because video images are sampled horizontally in real time. Two-dimensional operations are used for various graphic effects or to change the size and shape of the image. Such filter operations include high-pass filters, which sharpen edges in all directions, or low-pass filters, which soften edges or limit high-frequency noise (Kreuger, 2000).

Most filters are based on a square convolution operator (or kernel), which is simultaneously superimposed on a group of pixels. Typical operators range from 3 x 3 pixels to 15 x 15 pixels. Improved results may be obtained with larger operators (up to 63 x 63 pixels) in applications such as medical imaging. Figure 5 shows a Laplacian operator that may be used for edge enhancement in image processing. The operator is represented by 25 weights arranged in a 5 x 5 matrix. To convolve it with an image, the operator is moved over the image pixel by pixel and line by line. The filter examines the center pixel and modifies it according to the surrounding pixels. Each pixel is multiplied by the corresponding coefficient of the operator and the 25 products are summed. This operation results in one pixel of the new image. The operator then moves one pixel to the right, and the operation is repeated until the entire image is processed.

-16	-7	-13	-7	-16
-7	-1	12	-1	-7
-13	12	160	12	-13
-7	-1	12	-1	-7
-16	-7	-13	-7	-16

Figure 5. Example Laplacian operator for edge enhancement in image processing.

3.2 Parallel Implementation

As an example application, a run-time reconfigurable filter based on the Laplacian operator of Figure 5 was implemented in several variants. First, a parallel implementation of the Laplacian filter was carried out using an FPGA device. Five lines of the image are accessible simultaneously, providing five data streams. Four line buffers are used of 1 KB each, and each pixel is represented on 8 bits (grayscale images). Five consecutive pixels in each line are stored in five parallel shift registers. The array of 5 x 5 shift registers stores a window from the image. Instead of loading the entire window from the buffers for each pixel that is processed, the registers are shifted to the right to form a sliding window. Only five registers are loaded at each new pixel.

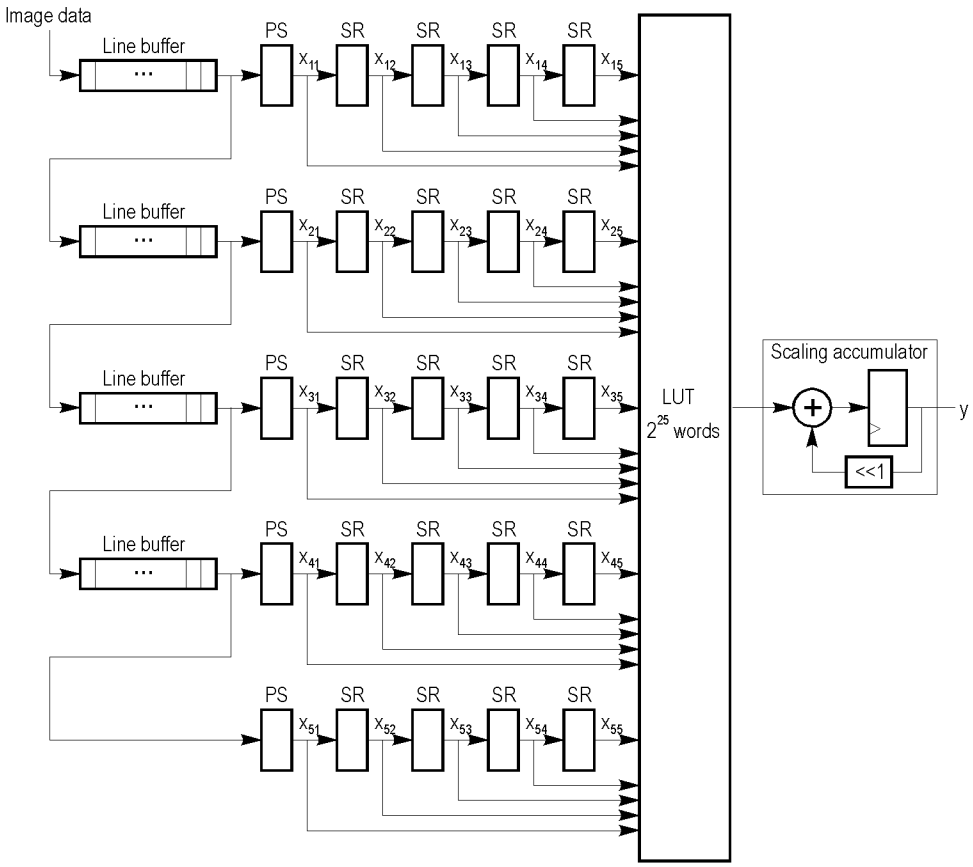

Figure 6. Implementation of the Laplacian filter using serial distributed arithmetic.

The pixel values in the registers are then multiplied with the filter constants using constant coefficient multipliers (KCMs). These multipliers are provided by the JBits SDK as RTP cores and they correspond to the Xilinx Unified Library primitives (Xilinx Inc., 2001). The results are then summed using an adder tree. In addition to the line buffers and the associated multiplexers, this implementation requires 25 shift registers of 8 bits each, 25 KCM multipliers and 24 parallel adders.

Another implementation is based on serial distributed arithmetic and look-up tables (Figure 6). For this implementation, each data stream is serialized using a parallel to serial shift register (PS) and passed through four serial shift registers (SR), each of which delay the data by one pixel. This provides simultaneous bit-serial access to five adjacent pixels from five adjacent lines of the image. For the serial multiplication and accumulation of the 25 pixel values with the filter constants, a LUT with 2^{25} words would be required, with its output connected to a scaling accumulator.

Several techniques can be used to reduce the size of the LUT. A possibility is to partition the circuit into smaller groups and to add the outputs of each LUT. If the FPGA architecture contains four-input LUTs, the optimum partition is to combine four products with a LUT. However, for the particular operator used more efficient techniques can be employed. The coefficients that are used several times can be combined with serial adders before they address the LUT, as shown in Figure 7. In this figure, SA represents a serial adder. The pixel values are therefore added and then weighted, instead of being weighted and added subsequently. Since the coefficient 160 (corresponding to pixel x_{33}) is used once, the pixel value only needs to be delayed in order to enter the LUT at the right time.

The size of the LUT has been reduced to 2^7 words, and this LUT can be implemented by two 16-word LUTs and an 8-bit adder. Therefore, the implementation using serial distributed arithmetic requires 18 serial adders, two 16-word LUTs, an 8-bit adder and a scaling accumulator.

3.3 RTR Implementation

The parallel and serial versions of the Laplacian filter were described in the Java language, using the RTP cores provided by the JBits SDK, version 2.8. The partial reconfiguration API of JBits was used to obtain run-time reconfigurable implementations of the filter. The partial reconfiguration model of JBits allows to only make the changes necessary to a device that will bring it into a desired configuration. This model determines changes made between the last configuration sent to the device and the present configuration in memory. Then it creates a sequence of packets that will partially reconfigure the device.

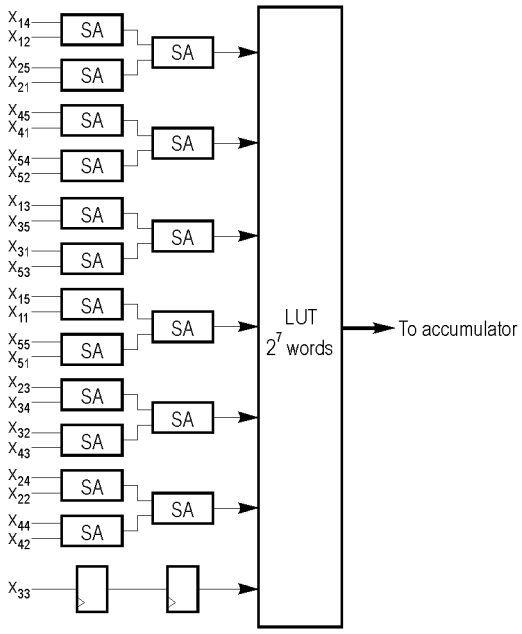

Figure 7. Combining the pixel values with serial adders before weighting.

After configuring the device, the model marks the device and memory as synchronized. The partial reconfiguration API of the JBits tool performs the synchronization functions automatically (Xilinx Inc., 2001).

The implementations of the filter were tested using the BoardScope interactive debug tool and the VirtexDS device simulator, which are provided with the JBits SDK. BoardScope allows to graphically examine the operation of FPGA devices on a reconfigurable computing 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.

The image was sent to the filter simulated by the VirtexDS tool from a file by a Java application program. The processed image was read back from the filter and was displayed on the screen. By changing the coefficients of the filter and partially reconfiguring the simulation model of the device, the effects on the processed image were visible.

4. CONCLUSIONS

This paper presented several techniques which can be used for the implementation of DSP operations in FPGA devices. The reconfigurable computing paradigm has been introduced, and the advantages of using FPGA devices as reconfigurable architectures have been illustrated. The main features of the JBits SDK were presented. Distributed arithmetic has been described as a useful technique which can significantly reduce the hardware resources required for the implementation of DSP operations.

As examples of implementations for DSP algorithms, a parallel and a serial version of a Laplacian filter were presented. For the run-time reconfigurable implementation of this filter, the JBits SDK has been used. This tool allows access to the configuration data of the Virtex FPGA devices and provides RTP cores that can be used to implement RTR systems. Currently, the JBits SDK has several limitations due to which it is difficult to use it for complex designs.

REFERENCES

DeHon, A. (2000). The Density Advantage of Configurable Computing. *Computer*, **Vol. 33, No. 4,** pp. 41-49.

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

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

Vemuri, R. R. and **Harr, R. E.** (2000). Configurable Computing: Technology and Applications. *Computer*, **Vol. 33, No. 4,** pp. 39-40.

Xilinx Inc. (2001). JBits Tutorial. JBits SDK Version 2.8, JBits@xilinx.com.