An FPGA Implemented Processor Architecture with Adaptive Resolution

Jim Torresen and Jonas Jakobsen
Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
{jimtoer,jonjakob}@ifi.uio.no
http://www.ifi.uio.no/∼jimtoer

Abstract

Reconfigurable software has been applied for a long time. Reconfigurable technology also provides possibility for reconfiguring hardware but this has not been much exploited so far. In this paper, a flexible processor architecture is proposed that allows for variable resolution in data variables at run-time. Experiments are undertaken for an image processing task where the results show that the approach is beneficial.

1. Introduction

Today it would be unthinkable not to swap software at run-time in a processor. This allows for many programs to apperently run concurrently on a single processing unit. With the introduction of reconfigurable technology – Field Programmable Gate Arrays (FPGA), this is also now possible for hardware. Instead of having every function – whether used or not, static in hardware, it would reduce hardware cost and power consumption if only those functions active at a given time are placed in hardware. Speed could also be increased since more specialized hardware units can be implemented rather than having a more general hardware implementation. However, long reconfiguration time has limited the applicability of run-time reconfigurable hardware. With the devices becoming larger and faster, it is now possible to design run-time reconfigurable systems without having to reconfigure the complete device.

Reconfigurable computing has grown to become an important and large field of research. The FPGA devices consist of a number of logic blocks connected with configurable interconnection lines. Each logic block consists of configurable combinational logic together with one or a few flip-flops. The logic is implemented in Look-Up Tables (LUTs) in Random Access Memory (RAM). The configuration of the function of each logic block and its connections to other blocks are given by the configuration bitstream loaded from outside the device and stored in distributed RAM on the FPGA.

Work is conducted either by using commercial FPGAs or by having new FPGA-like devices developed. A survey of can be found in [1]. The benefit of the reconfigurable technology compared to processors is the inherent parallel architecture. That is, the devices consist of a number of simple processing units (logic blocks) with distributed communication links. The computation could be undertaken in a parallel scheme rather than by a single unit (processor) – as in a personal computer (PC).

The approach of applying reconfigurable logic for data processing has been demonstrated in some areas such as video transmission, image-recognition and various pattern-matching operations (handwriting recognition, face identification) [8].

Commercially available FPGAs do not yet provide rapid configuration switching and each new configuration would have to be downloaded externally. The main experience from work on switching configurations at runtime seems to be that today's FPGAs are still requiring a (too) long reconfiguration time [5]. A task should be partitioned into *coarse* grained parts to reduce the overhead of switching between different configurations. Most devices require the complete configuration bitstream to be downloaded in one operation. The downloading time then increases with the size of the device. To reduce this overhead *partially* reconfigurable devices have been introduced – e.g. the Xilinx Virtex family offer this feature. Then, the user can program a part of the device at a time and then reduce the time for switching to a new task to be executed. Another approach to reduce the downloading time, is to apply prefetching of the configuration [2]

To overcome the problem of long reconfiguration time, a structure can be implemented in an FPGA which can provide switching between a *set* of circuits in short time. In this way, fast reconfiguration is obtained without having to reprogram the FPGA configuration register. This was first introduced by implementing a *user defined FPGA* inside an ordinary FPGA [3, 4] – also named Virtual Reconfigurable Hardware (VRH). It has later been extended to include multiple on-chip configuration bitstreams [5] and switching between them would be possible in one clock cycle [6]. In this paper, such methods are applied for designing a high performance flexible processor in FPGA. At run-time, the data buses are configured to the necessary data bus width in a single clock cycle. When the buses consist of few lines, a larger number of parallel processors can be configured.

The proposed reconfigurable processor is presented in the next section. Results from real experiments on a template matching problem are included in Section 3. Finally, Section 4 contains conclusions.

2. A Flexible Processor Architecture

In this section, we propose a processor with flexible resolution. It can be configured to compute with bus width from 1 to 16 bit. This width determines the number of parallel processing elements being available. E.g. with 1 bit processing, 16 parallel processing elements would be available. A block level description of the flexible processor is included in Figure 1.

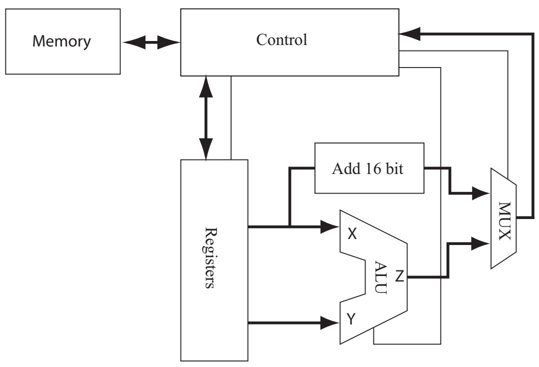

Figure 1. Processor architecture.

To limit the size of the control logic [7], the number of configuration bits is limited as much as possible. Thus, it is emphasized on using the same hardware unit for the various configurations as much as possible.

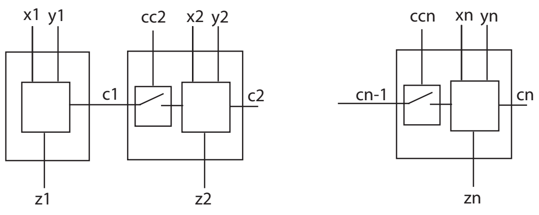

Figure 2. Illustration of flexible ALU sizes.

The Arithmetic Logic Unit (ALU) is the main part of the system and contains the reconfigurable part of the processor. Each ALU can perform either AND or adder function. More functionality would be added in the future. It is configurable for up to 16 bit data bus. These 16 bit can be partitioned by the user as illustrated in Figure 2. The splitting of the carry chain determines the size of each ALU and this is determined by the *carry configuration* (cc) bits. It can be configured from 16 one bit ALUs up to one 16 bit ALU (ALUs of different sizes are also possible). The first configuration (16 one bit ALUs) is used in the following experiments. The carry bits will only propagate *within* an ALU and not *between* different ALUs. Thus, the configuration bit cc_i determines if an ALU$_i$ is to receive the previous carry bit (c$_{i-1}$). The complete configuration can be changed in a *single* clock cycle. We are not aware of any other work applying the same architecture.

2.1. Registers

A register bank with 16 16-bit registers has been implemented. Two registers can be addressed concurrently for the input to the ALU while a third register stores the result of an operation. When less than 16 bit is used for data, several words are stored within the *same* register. E.g. with 1 bit words, there will be 16 words in a single register (a separate carry register is necessary for storing the carry bit for each word after ADD operation). To add 1 bit results together into a 16 bit number – as necessary in the following experiments, the "Add 16 bit" unit is applied.

2.2. Instruction Set

The Control unit (see Figure 1) receives instructions from the memory, and by a state machine it generates appropriate control signals to the different units in the processor. The instructions are stored in a memory bank on the Celoxica board (see the next section). The coding of the instructions is given in Table 1.

	Comand	Data	Reg1	Reg2	Out	RegOut	MUX
# of bit	2	16	4	4	1	4	1

Table 1. Instruction coding.

Each field represents the following:

- Command: Determines the instruction: Load a register (00), ALU performs AND operation (10) or ALU performs ADD operation (11). Command 11 is also used for "Add 16 bit" – see "MUX".
- Data: This is either data to be stored in a register (command 00) or the carry configuration of the ALU (command 01/11).
- Reg1, Reg2 and RegOut: Reg1 and Reg2 are input operands and RegOut is the register where the result of an operation is stored.
- Out: The parameter determines if the result is to be stored in a register or in external memory.
- MUX: This parameter determines if the result should be from the ALU or "Add 16 bit" (see Figure 1). The latter operation is used after 1-bit operations (AND in the following experiments) to add the results from 16 1-bit operations.

2.3. Prototyping Platform

The system has been implemented in VHDL (Very high speed integrated circuit Hardware Description Language) for a Xilinx Virtex 1000-6 BG560 FPGA. The experiments have been performed on a Celoxica RC1000-PP board with PCI (Peripheral Component Interconnect) interface. The FPGA is interfaced to the PCI interface through four memory banks each of 2 Mbyte.

3. Running Template Matching on the Processor

To demonstrate the performance of the processor, we selected an application from image processing. We are involved in a project on speed limit sign recognition and this includes a number of time consuming steps. One of the most computational demanding is template matching for detecting red circles on signs in an input image. This includes passing *six* different sized templates over the image as illustrated for one template in Figure 3. That is, each template of different size is moved over the image one at time. If the *best* match at a given position in the image is satisfactory, it is concluded that a sign is present in the image and recognition of the sign content follows.

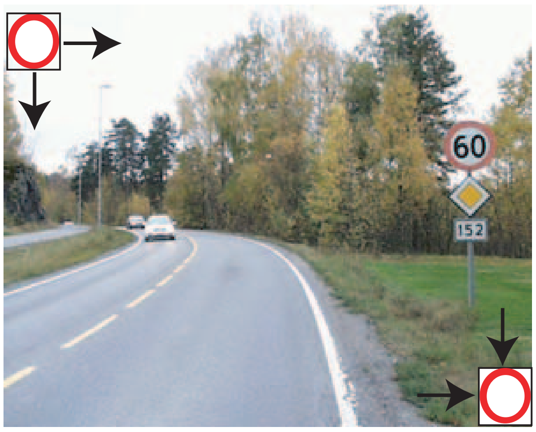

Figure 3. Template matching in image.

The template is not applied on the raw input but rather on a bit level image with red color extracted – i.e. one bit for each pixel with value "1" for red color and "0" for not red color. Thus, both the templates as well as the image are represented with *binary* data. For this reason it is an interesting application to run on our processor to demonstrate its performance.

The processing steps will be as follows for each position of the template in the red color extracted image: Perform XNOR (both pixels are either 0 or 1) between template and image. Then, add the results together. The latter operation is performed with the "Add 16 bit" unit in Figure 1. The code for the processor looks as follows for one template for one pixel position in the image:

1. Load a part of a template in register r1.
2. Load a part of the image in register r2.
3. Perform bitwise XNOR between r1 and r2 and store the results in register r3.
4. "Add 16 bit" in r3 and store the result in register r4.
5. ADD register r4 and r5 and store result in r5.
6. While (not finished with template) return to 1.

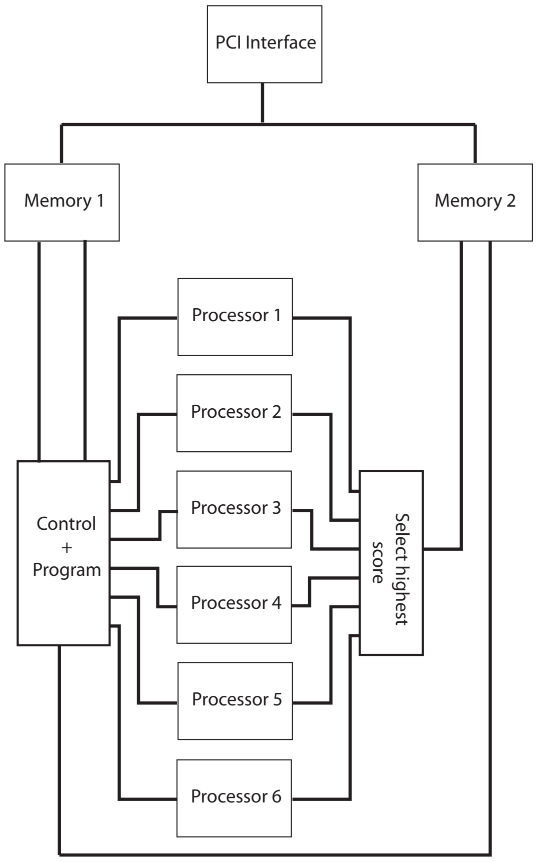

Figure 4. Architecture with six processors.

A system was implemented on FPGA with *one* processor for *each* template. The six templates were stored in on-chip RAM on the FPGA while the image was stored on external SRAM memory (Memory 1) as illustrated in Figure 4. Each processor is as outlined in Section 2. They are controlled in a Single Instruction Multiple Data (SIMD) way by the common control unit. For each position template matching is undertaken in the input image, the best match (of the six templates) is stored in Memory 2.

The size of each template is 64x64 pixels (most templates are smaller but it was practical in these first experiments having them of the same size). This corresponds to

256 words of 16 bit length each. The binary image is 640 x 480 pixels. To simplify the hand-coded processor code, there were made a few simplifications that would need to be taken care of in a real system. This includes how the templates are shifted across the image. However, this should not influence the comparison of the performance speed. The design filled the XCV1000 device to a large degree as seen in Figure 5.

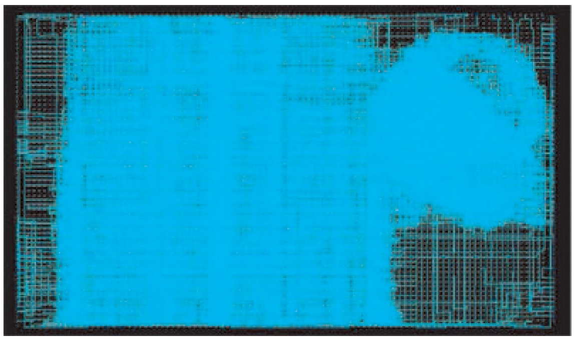

Figure 5. Layout of the FPGA design with six processors.

For comparison, equivalent code was implemented on a Pentium III PC with a clock speed of 1GHz. This is higher than what most *embedded* processors are running at today. Preliminary experiments showed that for the PC, it was more beneficial storing only one pixel for each memory location rather than 16 as for the FPGA processor. Thus, this was applied for the following experiments. Further, the PC program applied the same simplifications as the FPGA regarding template matching to make the two implementations comparable in speed of performance.

The following experiments are concerned only with one step of the processing (template matching). This is mainly due to lack of programming tools for our system requiring hand coding at a very low (and time consuming) level. However, with most of the other processing steps being on binary data, we believe the architecture will be effective for our application. Moreover, with FPGA it is also possible to implements parts of the processing as dedicated hardware.

3.1. Results

The FPGA design was able to run at 35MHz. This resulted in a execution time of 2.152s for the template matching on one image. The PC running at 1GHz needed 6.379s for the same task. Thus, the FPGA architecture was close to three times faster than the PC. There is a potential in increasing the clock speed of the FPGA by improving the design. For a real system both these implementations will be too slow. However, it is not necessary to do template matching for every pixel in the image (as here) and if this is not undertaken, execution time will be substantially reduced. Correct operation was confirmed with verifying that the two implementations gave the same output result.

4. Conclusions

In this paper, a processor with parallel and flexible ALUs has been introduced. It allows for adapting the data bus width to the problem at hand. Experiments for an image template matching problem show that such a processor realized on an FPGA can be faster than running the same application on a PC. Future work consists of improving the processor and testing it for other applications.

Acknowledgments

We are thankful for the PCI interface designed by Fearghal Morgan at National University of Ireland, Galway that was made available for this project.

The research presented in this paper is a part of the project *Biological-Inspired Design of Systems for Complex Real-World Applications* (project number 160308/V30) funded by the Research Council of Norway.

References

[1] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proc. of Int. Conference on Design Automation and Testing in Europe - and Exhibit (DATE)*. Munich, Germany, 2000.

[2] J. Resano et al. A reconfiguration manager for dynamically reconfigurable hardware. *IEEE Design and Test of Computers*, 22(5):452–460, September – October 2005.

[3] L. Sekanina and R. Ruzicka. Design of the special fast reconfigurable chip using common FPGA. In *Proc. of Design and Diagnostics of Electronic Circuits and Systems - IEEE DDECS'2000*, pages 161–168, 2000.

[4] J. Torresen. A divide-and-conquer approach to evolvable hardware. In M. Sipper et al., editors, *Evolvable Systems: From Biology to Hardware. Second Int. Conf., ICES 98*, pages 57–65. Springer-Verlag, 1998. Lecture Notes in Computer Science, vol. 1478.

[5] J. Torresen. Reconfigurable logic applied for designing adaptive hardware systems. In *Proc. of the International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet (SSGRR'2002W)*. Scuola Superiore G. Reiss Romoli, 2002.

[6] J. Torresen and K. Vinger. High performance computing by context switching reconfigurable logic. In *Proc. of the 16th European Simulation Multiconference (ESM2002)*, pages 207–210. SCS Europe, June 2002.

[7] R. Turner and R. F. Woods. Design flow for efficient FPGA reconfiguration. In *Proc. of International Conference on Field Programmable Logic and Applications (FPL 2003)*. Springer Verlag, LNCS 2778, 2003.

[8] J. Villasenor and W. Mangione-Smith. Configurable computing. *Scientific American*, (6), 1997.