Computer Science • 14 (3) 2013 http://dx.doi.org/10.7494/csci.2013.14.3.459

Wojciech Wójcik
Jacek Długopolski

FPGA-BASED MULTI-CORE PROCESSOR

Abstract *The paper presents the results of investigations concerning the possibility of using programmable logic devices (FPGA) to build virtual multi-core processors dedicated specifically towards particular applications. The paper shows the designed architecture of a multi-core processor specialized to perform a particular task, and it discusses its computational efficiency depending on the number of cores used. An evaluation of the results is also discussed.*

Keywords microprocessor, FPGA, parallel computing, block cipher

1. Introduction

Multi-core processors are currently becoming more and more popular. Nowadays, most desktop and high-performance computing processors feature a multi-core design. Some of the most common featured architectures include Intel® Core^{TM} 2 or AMD® K10. The presented project followed this notion and tried to compare the effectiveness of multi-core parallel computing, albeit on a very modest scale. As the basis of the project, an FPGA (Field Programmable Gate Array) chip family was chosen to allow for the design of a desired architecture at the register transition level (as opposed to simulating the architecture on a ready traditional processor). The processor was designed in VHDL [1] hardware description language, using Altera Quartus II Software [2].

FPGA chips are special programmable logic devices designed to be reconfigured by users according to their needs and desires. This makes it possible to build, develop, and test any user-oriented hardware data processing architecture on a single integrated circuit. The description of the necessary configuration might be specified as an electronic circuit diagram or using a hardware description language like VHDL, Verilog [3], or others. A massively-parallel computing feature of FPGA technology gives designers a chance to accelerate the speed of a data-processing algorithm by splitting (if possible) any single computing process into many independent concurrent threads. There are a lot of computing problems which, by their nature, are easy to parallelize. In all of these circumstances, the FPGA chips show their immense benefits. Of course, an FPGA solution also has some disadvantages; for example, design time is much longer than for a regular software-based system. However, this does not diminish the advantages of FPGA.

FPGA-based processors are utilized more and more in embedded application-specific systems. For instance, Yiannacouras, Rose, and Steffan [4] propose an FPGA-based, automatically-generated, application-specific vector processor and show the possibility of scaling its performance. In their paper [5], Coyne, Cyganski, and Duckworth present an FPGA-based co-processor for accelerating the SART method for RF source localization. The system has been developed in VHDL and uses parallelism at many levels of the SART algorithm. Another example is an FPGA-based, massively-parallel, single-instruction, multiple-data-stream (SIMD) processor presented in [6]. The authors integrated 95 simple processors and memory on a single FPGA chip and showed that it can be used to efficiently implement an RC4 key search engine.

The authors' main goals of the presented work were to research and verify some possibilities of flexible, resource-efficient, and problem-specific implementation of parallel hardware data processing systems. In a standard data processing approach, systems usually use general-purpose processors (cores) which run application-specific software. In such circumstances, the processors usually have more features and greater potential than needed for a particular goal. At the same time, those features which are vital from an application point of view might be not optimal for a needed task. With that in mind, the authors believe that using FPGA technology may let us create flexible and well-fitted hardware data processing architectures directly dedicated to problems we want to solve. In such an approach, created sub-circuits and sub-processors own and use only especially-necessary resources such as registers, memory, instruction sets, etc. Thanks to this, it might be possible, for example, to improve hardware flexibility while decreasing resources, energy consumption, and data-processing time.

In our paper, we show a simple application-specific, multi-core processor architecture example, designed to perform a simple cipher algorithm. We also check and discuss its performance dependent on the number of cores used.

2. The chosen computational problem

The chosen problem was to encrypt/decrypt using a symmetric block cipher. The general idea of symmetric block ciphers is as follows: source data is first split into separate blocks of fixed and equal length. Next, a user-derived secret key is chosen, which is the same length as the data block. The key is combined with the data block using a specified algorithm which defines a particular block cipher type. The resulting combination is the output cipher text. The same key can be later applied with the same algorithm to the cipher text to decrypt the data back to normal shape — that is why the cipher is symmetrical.

The above procedure describes only how the cipher works with one data block. There are a few ways this behavior can be extended to multiple blocks. The one that was chosen for this project is called Electronic Code Book mode (ECB). In this mode, the key is simply applied to each data block separately. The data blocks are treated in a fully-independent manner, and therefore, this mode is well-suited for designing a parallel algorithm. It is not cryptographically safe, though, and is rarely used in practice. Still, the type of algorithm that can be used is not limited by the mode, and it can be anything from the simplest (like xoring data with the key) to more advanced (like DES or Blowfish). By choosing the right algorithm, the amount of time needed for processing can be adjusted. For this project, the simplest method (xor) was chosen to focus more on processor architecture rather than the implementation of complicated algorithms.

3. Processor design and architecture

In this section, the architecture of the processor and its interface is first presented. After that, the scalability of the proposed architecture is discussed.

3.1. Interface

The processor has its own simple instruction set. It allows the building of cipher programs which operate on different amounts of data stored in memory. The set contains one main instruction called `exor`, which tells the processor to actually cipher the data, and 8 auxiliary instructions with rather-standard semantics: `mov, add,` `sub`, `jmp`, `jz`, `jnz`, `nop`, `end`, which help to calculate appropriate parameters for `exor` and also control the flow of the program (e.g. loops). A sample code can be found later in section 4.1. The `exor` instruction has two parameters: `ri`, `rj`, which denote the registers holding the initial address of data and key respectively. Data block size is a parameter that is set within the design but can be easily changed.

A ready-made program still needs to be placed in the instruction memory. As for now, it has to be done by directly inputting the binary content into the VHDL design file. The reason for this was the lack of external memory on some of the available FPGA boards. The same has to be done with the data memory content. Still, the results of data processing can be monitored on a set of appropriate outputs, like LEDs.

This can be upgraded by adding support for the desired memory or interface type to the design, thus allowing data input and output to be independent from PLD programming files.

3.2. Schematics

The general architecture of the processor is presented in Figure 1. The processor core consists of four main modules, called (in short) A, B, C (multiple) and D. Additionally, there is also a monitoring module which interprets user-control signals and sends back their results via a specified output.

The monitoring module allows the switching of the output signals indicators between different core modules, which is especially useful when the number of outputs (e.g. LEDs) is limited. This module is also able to trap execution of the program, count the elapsed clock ticks, and detect idle statuses of particular modules. Therefore, it is able to determine the end of processing and measure elapsed time automatically.

The sequential processor (labeled A) is responsible for the interpretation and execution of a user-defined program (which is briefly covered in section 3.1). It executes all instructions of the program one by one. When it reaches a `exor` instruction, it delegates it (with the corresponding actual parameters) to module B and continues to process further instructions. Of course, module B has to be free and ready to accept a new task. It means that A can work in parallel with the B and C modules, but sometimes processing has to wait and synchronize itself during the exchange of needed data.

The queuing module (labeled B) is responsible for scheduling tasks received from the module A for the C processors. A new task is scheduled for the first idle C processor. If all C's are busy, the task is put into an internal FIFO queue. When the queue becomes full, further tasks are not accepted until a place in the queue is freed. It happens when one of the C's becomes idle and accepts the first task from the queue.

Each parallel processor (labeled C) is idle until it gets a task from module B. After that, it starts processing the task. The processor communicates with the memory module, requests the needed data and key fragments, combines them, and stores the results back in the memory. The procedure is repeated a desired number of times until

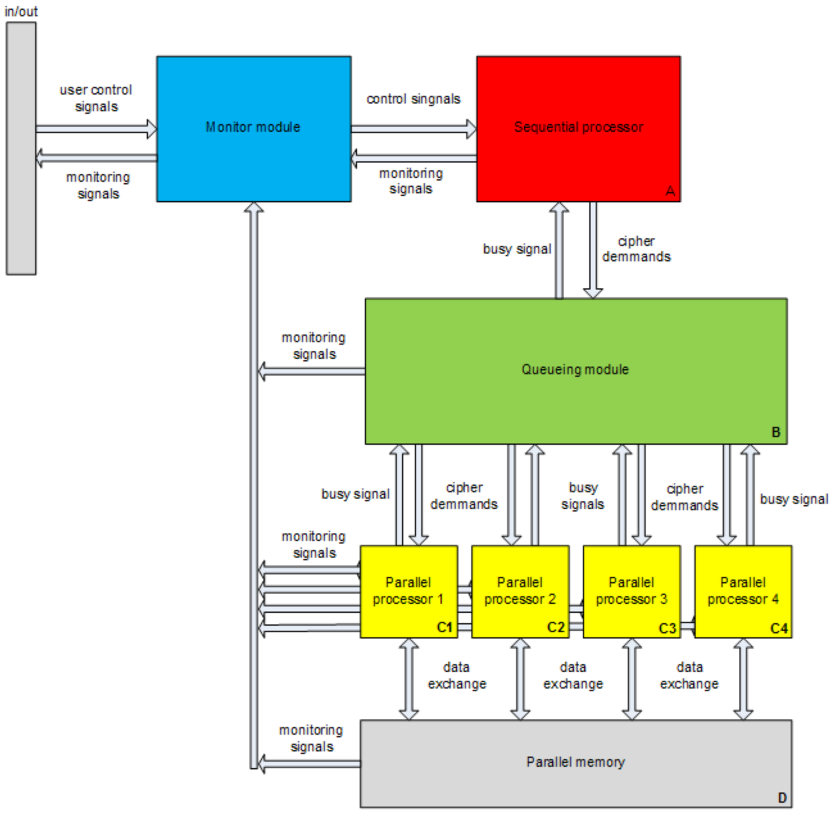

Figure 1. Overall schematics.

one data block is fully encrypted. The cipher text replaces the original data in the memory, so subsequent executions of the same program encrypt and decrypt data by turns.

The parallel memory (labeled D) is the most-passive module. It holds the data and key for processing and allows for concurrent access from all C modules. According to the block cipher idea, the parts of data on which the processors C operate should be separate (different data blocks), although some safety mechanisms still have to exist in the design to cover concurrent read/write problems. The priority solution was chosen to minimize unnecessary overheads.

3.3. Scalability

The processor architecture can be scaled. The main factor which determines the speed of the processing is the number of implemented C modules. The C modules can be treated as the actual cores of the processor. Of course, there has to be proper support for them from the adjoining B and D modules. It mostly means additional ports and buses, and rescaling of the internal algorithms, which can be done by a modification of the design files. Module A is — according to its name — sequential and not meant to be scaled.

The initial version of the processor had only two C cores. This was upgraded to four cores before running tests presented later in this paper. A further upgrade would surely require some automated means of generating the design elements (connections, modules, repeating algorithm parts) for it to be convenient. For instance, VHDL `for ... generate`, `entity ... port map`, and `for ... loop` commands could be used [7].

4. Parallel characteristics

In order to determine the quality and correctness of the design, processor performance had to be measured. The most basic metric that can be measured is the amount of time needed to complete the processing, depending on data size and number of cores used. From that, the standard parallel computing characteristics (like speedup and effectiveness) can be derived. The correctness of the processing was verified fairly easily. All that was needed was to run the program twice and check whether the memory content first shifts to an encrypted form and then back to its normal shape.

Initial tests showed that a little-coarser granularity is required to make full use of all 4 cores. That is why both setup and result sections are split in two parts and cover both the initial and final tests.

4.1. Experimental setup

For test purposes, both a program for module A as well as sample data and key (stored in module D) had to be prepared. Additionally, all the adjustable parameters of the processor had to be determined.

The general test configuration is as follows:

- there are at most 4 parallel processors available ($p = 1, \dots, 4$),
- module B internal queue has 4 places (which is more than enough),
- the memory consists of 8-bit words,
- the number of cipher operations that are actually performed is adjustable $n = 1, \dots, 8$. In each case, the user-defined program is the same — it only has a different initial loop counter value (making the results easier to compare),
- there is only one key block applied to all data blocks.

4.1.1. Finer granularity

The test configuration specifications for this part are as follows:

- each data block and the key block consist of $x = 4$ memory words (one `exor` instruction processes four words of data and key),
- there are, at most, 32 words of source data to encrypt using, at most, 8 `exor` instructions.

module A program is shown below:

```
; initial address of the first data block
mov r0, 0
; initial address of the key block
mov r1, 32
; loop counter - adjustable part
mov r2, 8
; data block length (address offset)
mov r3, 4

mov r4, 1
; exor loop
loop1: exor r0, r1
add r0, r3
sub r2, r4
jnz r2 loop1
end
```

4.1.2. Coarser granularity

The test configuration specifications for this part are as follows:

- each data block and the key block consist of $x = 8$ memory words (one `exor` instruction processes eight words of data and key),
- there are, at most, 64 words of source data to encrypt using, at most, 8 `exor` instructions.

module A program is shown below:

```
; initial address of the first data block
mov r0, 0
; initial address of the key block
mov r1, 64
; loop counter - adjustable part
mov r2, 8
; data block length (address offset)
mov r3, 8

mov r4, 1
; exor loop
loop1:  exor r0, r1
add r0, r3
sub r2, r4
jnz r2 loop1
end
```

4.2. Results

The results cover all possible combinations of problem size n and number of processors p. The time $T(n,p)$ was measured in clock ticks automatically by the monitoring module, so it is exact and independent of clock frequency. For each set of parallel characteristics, two charts are provided: one with a standard view: $f(p)$ and n-dependent data families and an alternate view: $f(n)$ and p-dependent data families.

Two main characteristics were taken for each case [9]:

- speedup $S(n,p) = \frac{T(n,1)}{T(n,p)}$
- efficiency $E(n,p) = \frac{S(n,p)}{p}$

4.2.1. Finer granularity

The processing times for all cases are shown in Table 1. The values are ordered by the indices p and n. Always $T(n_1,p_1) \leqslant T(n_2,p_2)$ where $n_1 \leqslant n_2$, $p_1 \geqslant p_2$. So, the more processors used or the smaller the problem size, the shorter the processing time. It proves the correctness of the design on a very basic level.

Table 1
Elapsed time [clock ticks] – finer granularity

p/n	1	2	3	4	5	6	7	8
1	39	64	89	114	139	164	189	214
2	39	48	64	73	89	98	114	123
3	39	48	57	66	75	84	93	102
4	39	48	57	66	75	84	93	102

For $n = 1$, the values are all the same no matter how many processors are used. A similar situation presents itself for $n = 2$ and $n = 3$. It is normal that if the number of tasks is smaller than the number of processors; in this case, some of the processors are idle. It is not correct, however, that the time is exactly the same for 3 and 4 processors for all matching problem sizes. The reason for this is the fact that module A cannot generate new tasks fast enough, at least compared to the speed they are processed by 3 C processors. By the time the fourth task is generated, one of the Cs finishes its job and accepts the new task, and so the fourth C processor is idle all the time. To make real use of 4 cores, the task size should be bigger (coarser granularity) or module A should work faster. The first solution is presented in section 4.2.2.

The Figures 2, 3 and 4, 5 show speedup and efficiency characteristics respectively.

The Figure 3 indicates that, for a fixed number of processors $p = 2$ and varying problem size, the speedup is not uniform. It is slightly better for even values than for odd. The odd value is usually similar to previous even one, or sometimes worse. That is because it is easier to fit an even number of tasks on an even number of processors.

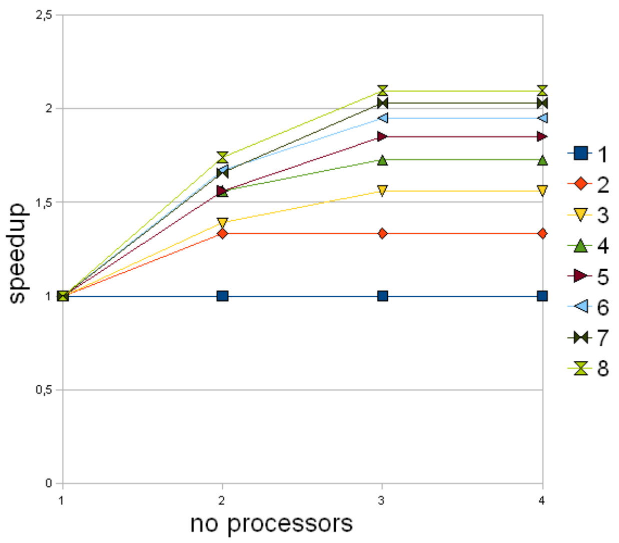

Figure 2. Speedup – finer granularity (standard view).

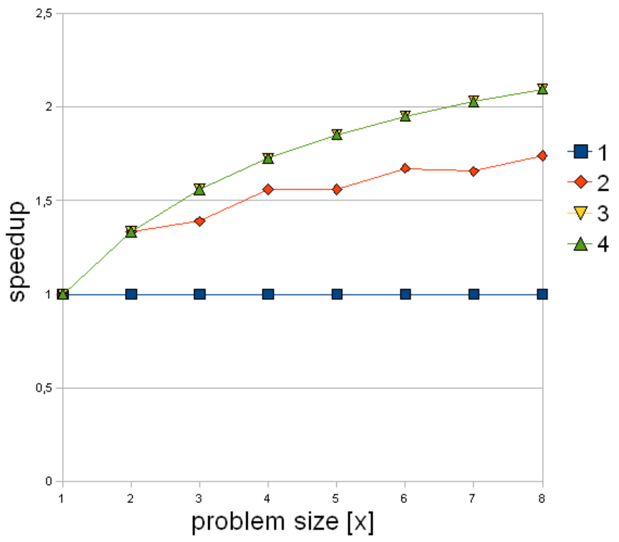

Figure 3. Speedup – finer granularity (alternate view).

If the number of tasks is odd, then one of the cores is usually idle for some time (when it could be handling another task).

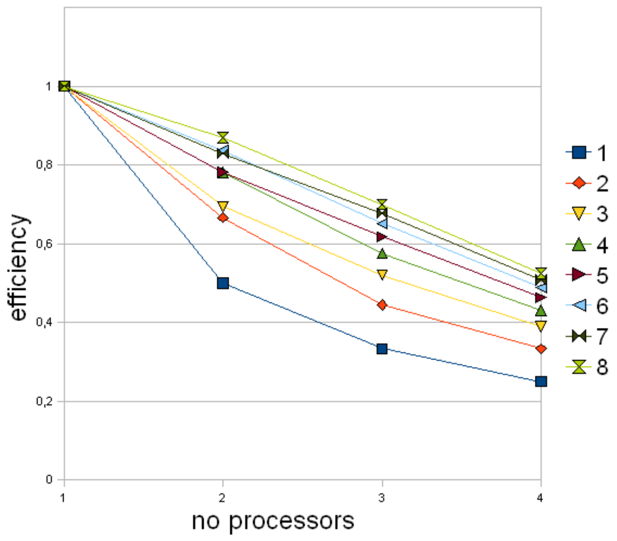

Figure 4. Efficiency – finer granularity (standard view).

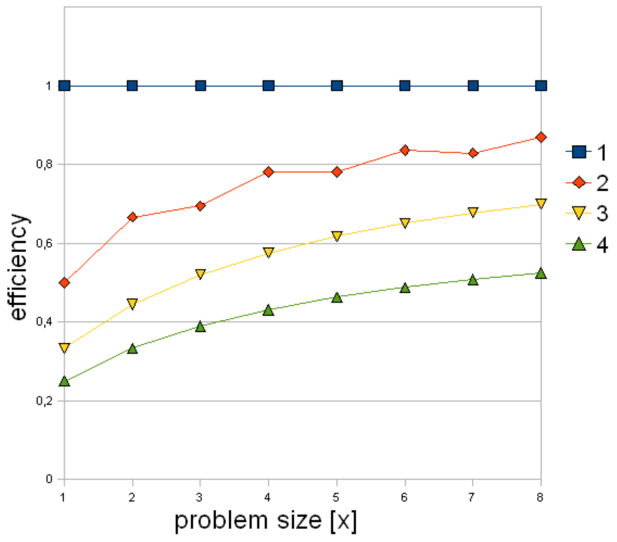

Figure 5. Efficiency – finer granularity (alternate view).

Figure 5 shows that efficiency rises along with the problem size. It means that the sequential part must decrease at the same time. In this architecture, the sequential part can be defined as the time when at least one C unit is idle, despite the fact that there are still tasks that need to be processed (which haven't been generated by module A yet). The B and D modules are working fast enough not to produce any delays, so the only source of possible delays in the design is the sequential processor A. Speedup and efficiency increase along with the problem size because, after the initial part, all modules can work in parallel until they finish their jobs. Of course, if the granularity is too fine (like here), there will be some consistent wait states throughout processing. In this case, they weren't big enough to change the characteristics.

4.2.2. Coarser granularity

Processing times for all cases are shown in Table 2. The results are quite similar to the ones presented in section 4.2.1; but this time, the fourth core really makes a difference. The increased granularity gave module A a chance to occupy all of the C cores, while working at the same speed.

Table 2

Elapsed time [clock ticks] – coarser granularity.

p / n	1	2	3	4	5	6	7	8
1	63	112	161	210	259	308	357	406
2	63	72	112	121	161	170	210	219
3	63	72	81	112	121	130	161	170
4	63	72	81	90	112	121	130	139

It can also be noted that, for $n = 4$, the values are slightly better than in the corresponding finer-granularity case with $n = 8$. This is because module A had fewer tasks to generate, which led to smaller communication overheads. There is one exception, however: for $p = 3$, the value is worse because it is harder to fit 4 tasks on 3 processors than it is to fit 8.

Figures 6 and 7 show the speedup characteristics. The corresponding finer granularity chart presented in Figure 2 had no crossed lines. Here in Figure 6, it can be seen that speedup for $p = 2$ is better for $n = 2$ rather than 3. The same situation is for $p = 3$ and $n = 3, 4$, or $p = 4$ and $n = 4, 5$. Also, the alternate view presented in Figure 7 now has a more-distinct shape. This is because the time needed to process one data block is longer now and much greater than the time needed to generate a task.

It all leads to a generalization of the facts concluded earlier in section 4.2.1: for a given number of processors p, it is best to fit the number of tasks n which is a multiple of p. Any other values of n are less optimal and cause speedup and efficiency drops.

Figures 8 and 9 present efficiency characteristics. Similar to speedup, the peaks are located at full multiples n of p values. Also, the bigger the problem size, the more the distortions are scaled down; but, they still have the same shape.

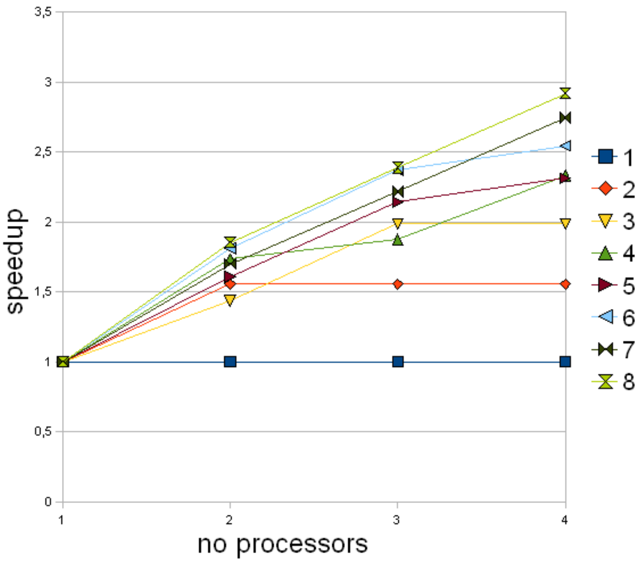

Figure 6. Speedup – coarser granularity (standard view).

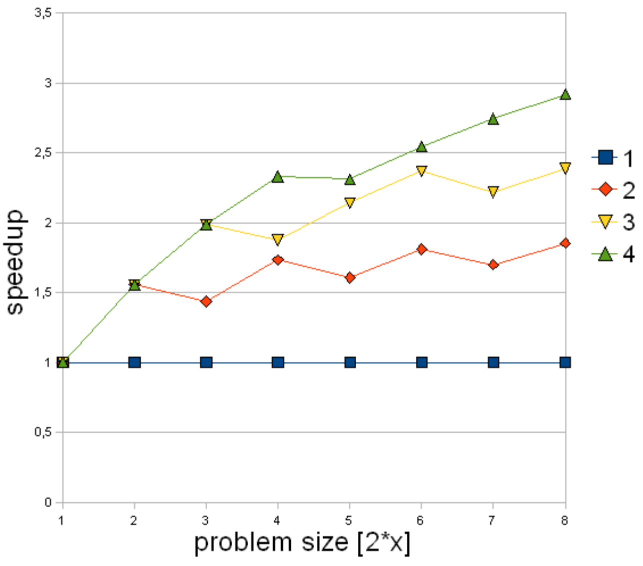

Figure 7. Speedup – coarser granularity (alternate view).

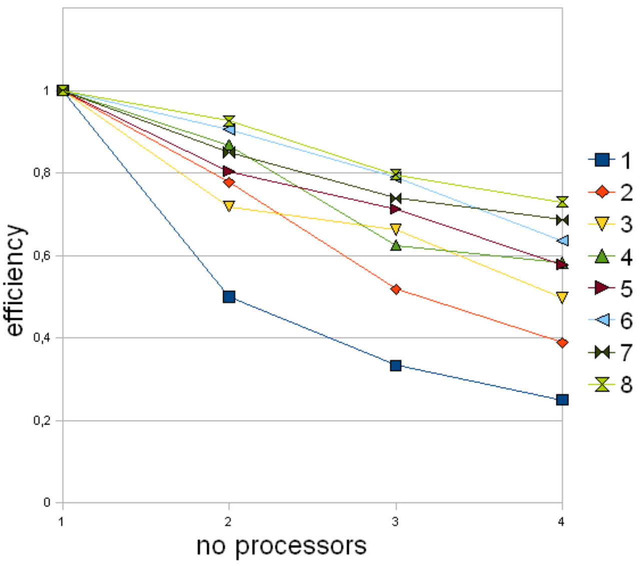

Figure 8. Efficiency – coarser granularity (standard view).

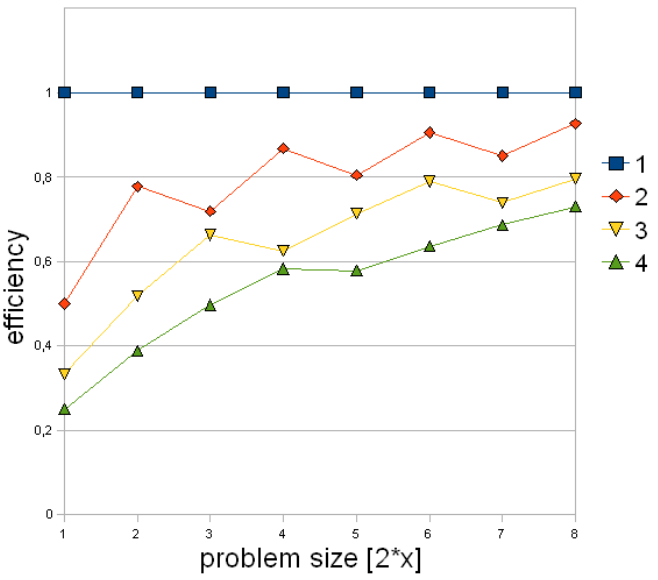

Figure 9. Efficiency – coarser granularity (alternate view).

4.3. Corollaries

The time measurement is strict — it is not measured in real time units (like seconds) but in the number of processing steps. Also, the processor itself works in a determined way; and, for a given task, it always needs the same number of steps to complete it. Therefore, the measured values should not be random. Indeed, a closer look reveals that all of them can be described using a single formula:

$$T(n,p) = \begin{cases} d_i + y + d_l(n-1) + (y - d_l \cdot p)\lfloor\frac{n-1}{p}\rfloor & \text{if } y > d_l \cdot p \\ d_i + y + d_l(n-1) & \text{otherwise} \end{cases} \tag{1}$$

where:

d_i – initial delay [clock ticks] – total processing time before the first `exor` command actually starts being executed by one of the C processors;

d_l – loop delay [clock ticks] – delay between generating subsequent `exor` commands by the A processor (processing time of all the instructions inside the loop1 of the program);

y – time needed to encrypt one data block consisting of x memory words — it depends on a chosen algorithm and its implementation in module C. For current xor implementation $y = 6x + 1$.

The additional part $(y - d_l \cdot p)\lfloor\frac{n-1}{p}\rfloor$ describes the fact that, when the task size is too big (or, conversely, there are too few processors), some tasks will have to wait until they are generated before they will actually be processed. When $y < d_l \cdot p$, it means that there is always a free C module to handle a new task, and the speed of the processing is only limited by the speed of module A. Adding more processors to a job that already satisfies this condition will not provide any speedup, which is exactly what was observed in section 4.2.1. The y element in $d_i + y + d_l(n-1)$ corresponds to the processing time of the last task, after module A has finished its job.

The factor $\lfloor\frac{n-1}{p}\rfloor$ is an integer division. When $y \geqslant d_l \cdot p$, each p-th task has to wait $y - d_l \cdot p$ cycles in queue before being processed. The first task that has to wait is $(p+1)$-th task, followed by $(2p+1)$-th, $(3p+1)$-th, and so forth.

Of course, the formula (1) is only true for a user-defined program shaped like the ones in sections 4.1.1 and 4.1.2. For both of these, the actual parameters values for the equation are as follows:

$$d_i = 14$$
$$d_l = 9$$

Each instruction requires 2 cycles except `exor`, which requires 3. Plus, there is one cycle delay before one of the C processors actually starts processing the first task, which gives 14 cycles total for d_i. Inside `loop1`, there are 3 regular instructions and 1 `exor` instruction. There is no additional delay here, because the wait state between subsequent tasks of a C module is already included as +1 in y parameter.

The formula (1) is true for $p = 1$, where the C processor is always busy. It is also satisfied for both considered granularities even though the finer granularity does not use fourth C processor at all. Based on that, the formula should also be true for any valid range of parameters n, p, x, y, but it was not tested in this experiment.

5. Conclusion

The project had two main goals to accomplish: first, design a microprocessor multi-core architecture which would show significant improvement over sequential computing; and second, make this architecture scalable.

The achieved results did show an improvement with speedup reaching 1.85 with two cores and 2.92 with four cores active. This yields the efficiency of 0.93 and 0.73 respectively. Additionally, the sequential part decreases with the problem size, so the characteristics should be even better with more tasks to process.

The architecture is also scalable. It had already been rescaled from 2 cores to 4 in its current shape, so further expansion seems limited only by the number of necessary repeatable changes in the design files. But this still could be managed with the help of appropriate VHDL commands.

References

[1] The VHDL Cookbook, `http://ecad.tu-sofia.bg/soc/data/vhdl/vhdl_cookbook.pdf`

[2] Quartus II Subscription Edition Software, `http://www.altera.com/products/software/products/quartus2/qts-index.html`

[3] Verilog Hardware Description Language Reference Manual, `http://ecad.tu-sofia.bg/soc/data/verilog/verilog.pdf`

[4] Yiannacouras P., Rose J., and Steffan J. G.: The Microarchitecture of FPGA-Based Soft Processors, *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, September 2005, San Francisco, CA.

[5] Coyne J., Cyganski D., Duckworth R. J.: FPGA-Based Co-processor for Singular Value Array Reconciliation Tomography. In Kenneth L. Pocek, Duncan A. Buell, editors, *16th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2008*, 14–15 April 2008, Stanford, Palo Alto, California, USA. pp. 163–172, IEEE Computer Society, 2008.

[6] Stanley Y. C. Li, Gap C. K. Cheuk, Kin-Hong Lee, Philip Heng Wai Leong,: FPGA-based SIMD Processor. In *11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, 8–11 April 2003, Napa, CA, Proceedings. pp. 267–268, IEEE Computer Society, 2003.

[7] Zwolinski M.: *Projektowanie układów cyfrowych z wykorzystaniem języka VHDL.* wyd. 1., Warszawa, WKŁ, 2007, ISBN 978-83-206-1635-4

[8] Zainalabedin N.: *Digital design and implementation with field programmable devices*, XVI, Norwell, Kluwer Academic Publishers 2005, ISBN 1-4020-8011-5

[9] Karbowski A., Niewiadomska-Szynkiewicz E., eds.: *Obliczenia równoległe i rozproszone*, Warszawa, Oficyna Wydaw. Politechniki Warszawskiej 2001, ISBN 83-7207-234-5.

Affiliations

Wojciech Wójcik
AGH University of Science and Technology, Institute of Computer Sciences, Krakow, Poland, wwojcik@student.agh.edu.pl

Jacek Długopolski
AGH University of Science and Technology, Institute of Computer Sciences, Krakow, Poland, dlugopol@agh.edu.pl

Received: 18.02.2013
Revised: 12.03.2013
Accepted: 02.06.2013