1 Introduction

1.1 Overview

1.1.1 Multimedia applications

The possibility to take pictures and films anywhere is taken by more and more people are used. Digital cameras are now dominating the market and even mobile phones are able to take pictures with up to megapixels. However, the recording is only the first step, the resulting media data must also be compressed and compressed, stored. Often also a visual improvement of the image through image processing algorithms.

In order to perform these steps without great delays, enough computing power is available. A processor that records the data purely sequentially, requires a very high clock frequency. High clock frequencies also result in high power consumption. A circumstance of the in mobile applications because of the dependence of current from batteries is extremely inconvenient. Instead, a different approach is appropriate. The necessary for every image point calculations are largely independent of one another in parallel. A processor that takes advantage of this circumstance. And the values of several pixels are calculated simultaneously. Not so high clock frequencies and thus saves energy.

This principle is not only limited to digital cameras, but is also useful Can be implemented in many applications where media data, be it images, Movies or sound. Often can help with such as processors, even mathematical problems (eg, calculations with Matrices) can be solved very efficiently.

1.1.2 Hardware development and open source

In the area of hardware development is becoming increasingly programmable Such as FPGAs. It is also the trend to observe, That complete systems are realized on a single chip. This is done by selecting suitable IP cores (reusable descriptions of semiconductor components) according to a modular principle With each other.

Such IP cores can either be purchased or developed by commercial IP manufacturers. On the other hand, there is also the possibility to use ready-made IP cores, which are run under an open-source-License. There are already established ones on the Internet Websites, on which a variety of different building blocks for download Is offered. The spectrum ranges from multipliers over Ethernet To microprocessors, some of which already exist as a complete system with memory, controllers and many interfaces. What is currently still in vain in this area, however, are SIMD processors suitable for multimedia applications.

1.2 The aim of the work

The goal of this work is the development of an SIMD processor for embedded applications, which has the following characteristics and is used in the situation to process a large number of data in parallel:

- The processor is ready
- Flexible as a soft core in FPGAs.
- The instruction encoding and the instruction set are extensible.
- The amount of data that can be processed in parallel is configurable
- Parts of the functions can be switched on or off to adapt it to different applications.

The processor must be validated and in real hardware, for example a FPGA, be lukewarm. For this purpose, a test environment in the form of memory and an on-chip debugging unit.

The development of an assembler and the debugging software for the development computer is also part of the task.

In addition, the performance of the processor must be determined by means of an example application.

1.3 Structure of the document

This work is divided into seven chapters. Chapter 1 discusses the topic, describes the goal of the work and gives an overview of the structure of the document. Chapter 2 discusses the basics needed for further understanding are necessary. In addition, the current status of the as far as vector extensions are concerned. Chapter 3 with the design of the processor. Basic principles, command set, microarchitecture and existing interfaces. Details about the implementation are to be found in chapter 4. This is about optimization, which components have caused problems during implementation and how these were solved. Chapter 5 describes the environment in which the processor as well as the development tools developed. The validation of the entire system is also explained here. The results the performance analysis carried out and the impact of the different parameters of the configuration to the synthesis are in Chapter 6. The work is completed by chapter 7. The results of the development are summarized and future implemented ideas for the processor.

2 Basics

2.1 Programmable hardware

If a digital system is developed in the electronics industry, two relevant manufacturing processes. It can be applied to suitable standard blocks (e.g. microprocessors, memory blocks) and programming it, but also its own application: Integrated circuits (ASICs). Since requirements, for example the speed or the power consumption of systems, which do not always allow the use of standard components. The design of their own integrated circuits in many cases is inevitable.

The use of ASICs is particularly suitable for devices that are used in high-pounds can be deposited. The high development effort can then be caused by the cost savings in the production of the individual chips be balanced.

However, there are also alternatives for devices manufactured in small series allow programmable hardware such as CPLDs or FPGAs. The fast and cost-effective development of application-specific products circuits, but with a significantly higher price for the individual chip

is purchased. FPGA stands for "Field Programmable Gate Array", not as an ASIC from its manufacture a fixed function, but can be recruited at any time. Thus, the most varied from a simple adder to a complete computer system (System-on-a-programmable chip). The Rekon FPGAs are very well suited in systems where without adjustment of the hardware changes. Thereby the functionality can be extended or changed, for example, or rectify faults in equipment already delivered. In addition, the same chip used for several different tasks, as well as the development of new applications, through the possibility at any time, can be substantially simplified.

The configuration for FPGAs is usually done using a hardware description language such as VHDL or Verilog. Rare is also the possibility is used, circuits by a graphical editor or on the basis of the programming language C.

An FPGA is mainly based on many programmable logic blocks, which usually consists of look-up tables (LUT), one or more flip-flops and multiplexers (Figure 1). These blocks are then used by means of the configuration designed by the programmer up to get the desired functionality. Current FPGAs provide the user with up to tens of thousands of logic blocks. The configuration is usually realized by SRAM memory cells and every time the FPGA boots.

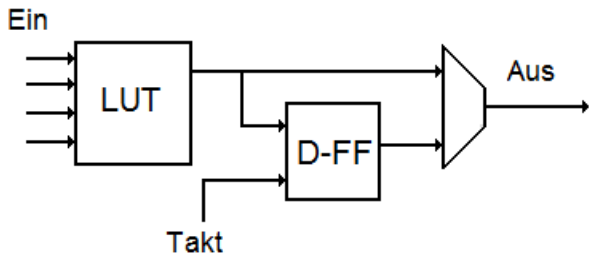

Figure 1: Simplified representation of a logic block

A LUT defines a combinatorial function (NAND, AND, OR, XOR, multiplexers, etc.) from a plurality of input signals. The number of available in this case, fixed inputs depends on the FPGA used. Common values are between 4 and 6 inputs per LUT. Submit the input a LUT for a function not out, can have multiple LUTs directly to each other.

Flip-flops, on the other hand, are used to store signals. Each flip-flop stores the value of one bit to ensure it in the following clocks can be reallocated.

In addition to logic blocks, FPGAs can also have other components, depending on the type contain.

Input and output blocks (IO blocks) serve as the interface of the FPGA to the exterior, including various TTL levels, pullup resistors and different signal standards.

RAM blocks allow the storage of larger amounts of data, in many situations flip-flops and thus also logic blocks can. This, often dual port-capable RAM, is in several small blocks organized and made possible flexible to larger units.

Multipliers allow the particularly fast execution of the multiplication-operation, without using FPGA logic cells. This is special interesting in digital signal processing.

In order to provide the clock signal synchronously on the FPGA whose to multiply or multiply the frequency or to change the phase FPGAs in the normal case also elements for clock preparation.

The question of the speed of an FPGA is not easy to answer. FPGAs with maximum system clock frequencies of several hundred MHz, the actual clock frequency of the circuit is however to large parts from the design of the on the FPGA applied configuration. In general, FPGAs are clear slower than ASICs with a comparable circuit.

2.2 VHDL

With the hardware description language VHDL (short for very high speed Integrated Circuit Hardware Description Language) can provide complex circuits at a high abstraction level. With this language can be simulated by software tools, verify and convert into a netlist (synthesis).

The network list of a circuit describes the connections between the individual components. In the case of a network list for an FPGA, the connections between the logic, RAM, IO blocks, or similar.

To create this network list is a synthesis program as well as the library of the chip maker. VHDL itself is technology independent. This means that a circuit developed with VHDL can be applied to different circuits ways. This includes the configuration of programmable logic modules such as FPGAs or CPLDs, but also the production of ASICs with standard cells as well as gate arrays.

The VHDL language makes it possible to build up a circuit hierarchically. The following example shows how many small components a complete processor can be assembled.

An adder network is formed by means of some full adders. This will then with a multiplier and a slider logic to an arithmetic logic Unit (ALU). From the ALU, several registers, memory and a control unit, the entire processor is created. It should be noted here that the individual components (e.g. registers) can again consist of smaller partial components.

The interface of a component (entity) to the outside is defined by its port. This is a list of input and output signals. The interior life (Architecture) is carried out with the help of language constructs such as minority, sequential processes, and through the integration of other components. But also functions and procedures can be used.

In VHDL are assignments of signals, as opposed to assignments in traditional programming languages such as C, adjacent (Figure 2) respectively.

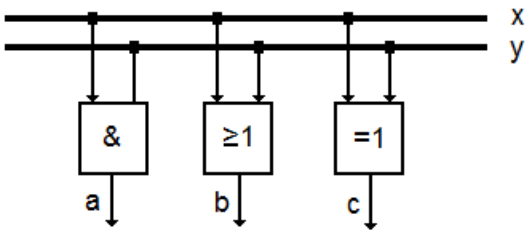

Figure 2: Auxiliary assignment in hardware

2.3 SIMD and Vector Processor

The Flynn [Fly72] class is divided into architectures their number of parallel command and data streams into four groups:

SISD - Single instruction stream, single data stream

SIMD - Single instruction stream, multiple data streams

MISD - Multiple instruction streams, single data stream

MISM - Multiple instruction streams, multiple data streams

Representatives of SISD systems are architectures in which a single control unit, exactly one computational or operational unit. The operations are executed sequentially; In each command only one operand pair can be processed. In personal computers were used until the introduction of the Intel Pentium MMX processor, mainly pure SISD architectures are used and in embedded systems these are still widespread.

SIMD computers differ from SISD architectures in that the vontrol unit instead of a plurality of operations with commands. This allows the same command on several parallel operations at the same time, and thus several pairs of operands can be processed simultaneously. Under the category SIMD field computers fall as well as processors with multimedia or vector extensions.

The two categories MISD and MISM of Flynn's schema have is of no relevance to the further understanding of this work. They differ from the two already presented architectures in that several command streams are present.

However, “Vector Processor” is used with two different meanings: On the one hand as a computer type with a pipeline-like structure arithmetic unit (s) for processing vectors to data, on the other hand but also occasionally as a general for SIMD systems, which allow operations on vectors of data. In the following, “Vector Processor” and “Vector Calculator”. For such systems, “vector extension” is used as a synonym for SIMD extension.

The processor developed in this work is a scalar (SISD) architecture with vector expansion. This extension allows it is to apply a single command to multiple data simultaneously. The parallel operation, rather than on individual operands, on a complete vector of data.

Each of these vectors includes several data words whose length is in many of today's common SIMD processors can be encoded in the command. A 128-vector registers could, for example, be in two 64-bit, four 32-bit, eight16-bit or sixteen 8-bit words.

The processing of an operation takes place for each date in the vector register taking the word length parallel (Figure 3); Thereby there are serious performance increases against scalar architectures, which each date in a single command. At an eight words, comprehensive vector registers can come with a single addition command up to seven commands. This creates an acceleration up to eight.

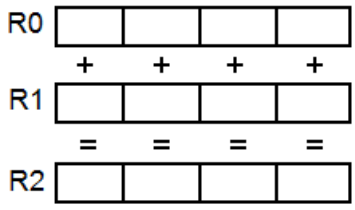

Figure 3: Addition of vector registers comprising four words

However, the profit in terms of maturity can be significantly greater. Often are used for processing such operations in scalar processors loops used for program memory and programming saving. This results in additional commands for increments, comparative operations, and conditional controversies. In addition, the operands must individually be loaded and stored in memory, while vector processors (assuming parallel memory access) also in one command can do.

An example will illustrate how many commands are executed by the use of a vector processor. It deals the addition of two 16 word arrays. Of the following C program code is developed for the thesis in this thesis, processor in two versions. On the one hand, in the variant, the only scalar unit is used, and the other is optimized for the vector unit.

Algorithmus unter Verwendung der Skalareinheit:

```
        OR      Y, 0, 0         ; i=0
LOOP:   LD      A, [Y + OP1]    ; lade op1[i] in A
        LD      X, [Y + OP2]    ; lade op2[i] in X
        ADD     A, A, X         ; addiere A und X in A
        ST      [Y + RES], A    ; schreibe res[i]
        INC     Y, Y            ; i++
        SUB     0, Y, 16        ; vergleiche i mit 16
        JNZ     [0 + LOOP]      ; springe, wenn i < 16
```

Algorithmus unter Verwendung der Vektoreinheit:

```
        OR      A, 0, OP1       ; lade Adresse op1 in A
        OR      X, 0, OP2       ; lade Adresse op2 in X
        OR      Y, 0, RES       ; lade Adresse res in Y
        VLD     R0, [0 + A]     ; lade op1 in R0
        VLD     R1, [0 + X]     ; lade op2 in R1
        VADD.DW R2, R0, R1      ; addiere R0 und R1 in R2
        VST     [0 + Y], R2     ; schreibe res
```

For the scalar variant, 113 commands are required for execution, while the counterpart using the vector unit with only seven out. Thus, a significant reduction in commands is observable, which is noticeable in a considerably less computation time. In many for all, program memory can be saved as fewer commands must be encoded. Unfortunately, this could hardly be shown in the example. Since the vector unit of this processor does not have direct values for vector-operations. This gave three extra commands to load of the addresses in registers.

This example has made clear that the vector unit is already at very high speed for simple problems. But there are also more complex one problems and especially in the processing of media data application possibilities. In image processing, high filters are used. Pictures to make them glow or edge out. For a glitter (with a box filter) for each individual pixel whose neighboring pixels are added up and the result by the factor 9 to determine an average value [Gon02]. This mean value is then the new value of the pixel under consideration. Here you can use vector processors by calculating the mean values for several pixels at the same time. The power gain is thereby over the length of the vector register limited. The more words fit in one of the registers, the faster the processing can be.

2.4 Reference architectures

Processors are nowadays commonly used by the von Neumann-reference model. In this model data and program code are used in the same memory.

An alternative to the Neumann calculator is the Harvard architecture development. It does not contain data and program code stored in the same memory, but separately. On commands and data can be concurrently allocated, while Neumann architectures can’t. Besides, it is with such architecture, other word-widths for data and commands to use. The price for a Harvard computer is that commands no longer treated like data and thus no longer by the program can be changed. In addition, two separate ones are held common memory.

2.5 State of the art

There are a variety of different processors for computer systems. In the field of personal computers are now used by every important manufacturer vector extensions in their processors. At the same time, there are different extensions with different properties.

2.5.1 MMX

The most famous extension is MMX [Int97] for Intel processors, which was introduced in 1997 with the Pentium MMX. Intel defined new register formats for data representation. The 64 bits, which are available in a register, can be found as "Packed quadword \ (1x 64 Bit)," Packed doubleword \ (2x 32Bit), "Packed word \" (4x 16 bits) or "Packed byte \ (8x 8 bits).

For the data of the MMX commands, the already existing registers of the Flying unit of the IA-32 architecture. Therefore, the MMX unit does not require any additional support from the

operating system. Unfortunately, it is also not possible, however, to do floating-point and MMX operations at the same time.

A simple variant of conditional execution is supported. Comparisons by MMX statements result in a bitmask that is of the length of the operators. This bit mask can then be used with the help of logical links are used to form the result.

Media data are often composed of many small values and calculations with their words, lead to a very high degree of overvaluation. For this reason, the MMX unit performs many calculations with a saturation arithmetic. That is, if an overflow at a calculation occurs, the result of the operation is the largest possible and for an underflow the smallest possible value is held as a result.

Commands are available for reordering data between and registers can be moved. Thus, it is possible, for example, data selected to move to another register, intermediate calculations with a higher word width and the result with the original word width.

MMX does not support flight operations, but allows calculations only on an integer basis.

With the introduction of SSE2, the MMX unit has also been reworked somewhat. MMX operations, 128-bit registers can now be applied.

2.5.2 SSE

2.5.3 3DNow!

2.5.4 AltiVec

2.5.5 SIMD extensions for embedded systems

For embedded systems, SIMD extensions are primarily used in low-cost area. For more expensive and faster processors, extensions are available that are very similar to those of the desktop systems remember. This includes NEON [neon] for ARM processors, for example, for XScale but also complete AltiVec units for CPUs from Freescale.

2.5.6 Open Source

Instead of using these expensive standard processors, however, there is also the possibility, specialized hardware in the form of ASICs or equivalent converged FPGAs. Unfortunately, it looks in the open-source area does not look good. On the inclusive web pages (such as http://opencores.org), so far, no project of a free processor with serious SIMD support. Instead there are more copies or proprietary developments of simple RISC processors.

There are, however, approaches of an SIMD extension for Leon2 processors. There, however, there are no additional, wide registers in the architecture, but only the data in the existing 32-registers are interpreted as four by 8-bit or two by 16-bits and by new commands.

In addition, a master thesis [Sha04] from the USA is to be found on the Internet, in which a configurable, specially designed for wavelet video compressions vector processor and investigated its conductivity becomes. Whether the processor is freely distributed is from the document unfortunately not visible.

3 Design

3.1 Basic Principles

The processor developed in this work is based on the ideas of parallelism, configurability, expandability, and openness.

It consists of the two components scalar unit and vector unit. While the scalar unit is individually classified as SISD, the vector unit processes several data in the style of a SIMD architecture. Parallelism at here means that both units, from each other, can do independent tasks at the same time.

The processor is not intended for a specific purpose, but instead should be a soft core flexible for various applications. Because different applications often have different requirements to a processor, it can be carried through several configurable parameters:

- The number of words (K) per vector register can be set. The higher this value is, the more data can be used parallel to a command.

- The number of vector registers (N) is also changeable. With more vector registers allow more data to be buffered and thereby reduce memory access.

- Allow the multiplication functions in the scalar and vector units can be switched on and off separately from each other.

- The ability to mix data using the Shuffle command can also turn on or off. In addition, the bit width, with which data can be mixed at maximum, configurable. This is necessary in order for values for K that do not have two-potentials to achieve meaningful bit-widths of the results (cf. Section 4.6)

- To move data over the entire vector register, move it to the left or to the right. This must be activated by configuration. Here the number of bits can be set to match the number of bits such command.

The boundaries of the congruence result from how much logic chip the processor is allowed to use. In addition, most the configuration also the maximum possible clock frequency. Shall be there certain values must be met, under certain circumstances smears of the configuration.

As new applications may require completely new commands, a simple extensibility of the processor is ensured. That closes both an extensibility of the command set, as well as the microarchitecture on.

Openness can be understood in two different ways. The processor is loaded with complete source code under an open- but it also refers to the implementation of the command set. This could, besides the implementation through the microarchitecture here, also with the help of pipelining or out-of-order execution.

3.2 Command set architecture

The command set defines the set of executables on a processor commands. Based on the command set architecture of the processor developed here, you can get some feedback about the underlying implementation to be pulled. It is the set of registers, addressing possibilities, the type of memory access, the RISC design, and the scope of the status register.

Since the processor is not developed for a specific application, could be both the microarchitecture and the command set in large parts freely defined. It became aware of complex and not essential commands to keep the processor fast and easy.

The set of commands created in the development process can be roughly divided into three groups. There are commands for the scalar unit, commands for the vector unit and instructions which are processed in cooperation between the two units become.

3.2.1 Register and Command Overview

The scalar part of the processor has three data registers. Because a command word of 32 bits, the registers and their data paths also have this length.

The data registers have the identifiers A (accumulator), X and Y. These registers are also used for scalar or cooperative operations possible to specify a scalar source or destination “0”. The value of the operand is zero or the result of the operation is not into a register. In addition, for many commands can be a direct value can be used as an operand from the command word.

The number of data registers is determined by the fact that the source and the destination registers are encoded with two bits each. This can result in a total of selection from four sources. Three of these refer to the register, the fourth to the value zero or the direct value.

The vector unit has a configurable number of vector registers (R0. .Rn), which serve as source and destination registers.

In order to make the instruction set more transparent, possible operands and parameters of commands are grouped into groups:

3.2.2 Command coding

The word length of the commands for the processor is 32 bits. This area is in 12 bits for instructions for the scalar and 20 bits for the vector unit (Figure 4). So, in a word it can be a partial command for each of the two units and also processes them at the same time become.

If the first three bits of a partial instruction are set to 0, the unit only performs a NOP instruction. This command does not cause any changes to registers, etc., so in the end does nothing. In such a case, the remaining bits of the partial instruction can be used for this, direct values or additional parameters for the command of the other unit there. In this way, it is possible, for example, to have a 16-bit direct value for an addition operation (Figure 5) directly in the command word.

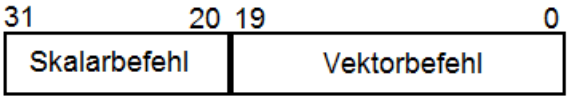

Figure 4: Distribution of a 32-bit command word

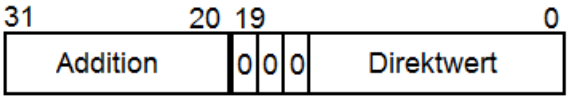

Figure 5: Addition command with direct value

3.2.3 Commands for the scalar unit

3.2.4 Commands for the vector unit

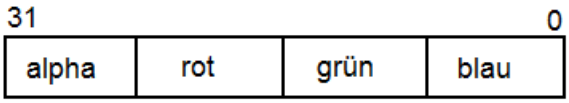

Figure 6: 32-bit Truecolor encoding

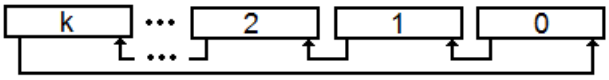

Figure 7: Effect of the "VMOL" command with a 32-bit word width

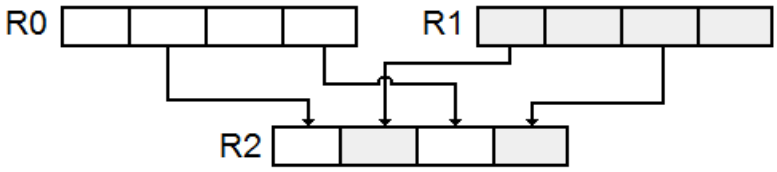

Figure 8: Rearrange with the Shuffle command

3.2.5 Cooperation commands

3.3 Microarchitecture

As was already apparent in section 3.1, the processor consists of the component scalar unit and vector unit. The units act largely autonomous and only have a few data and control lines (Figure 9) are in contact with each other. In the memory interface (also in this figure) is not a direct component of the processor, but it is necessary to put the CPU into operation to take.

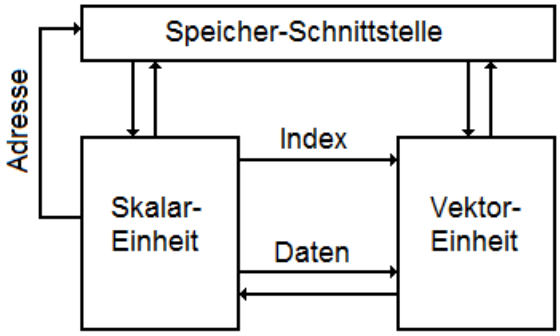

Figure 9: Data connections between units

3.3.1 Microarchitecture of the scalar unit

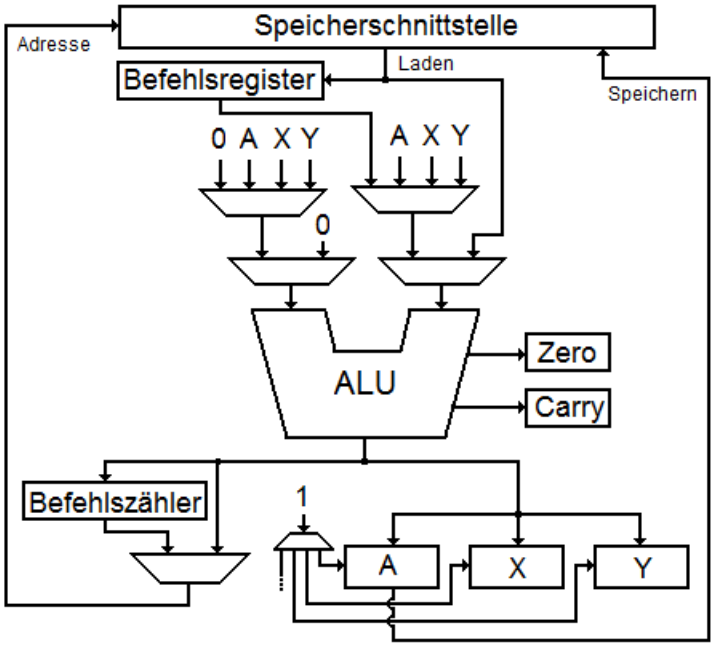

Figure 10: Scalar unit with memory interface (Simplified)

Figure 10 shows the structure of the scalar unit.

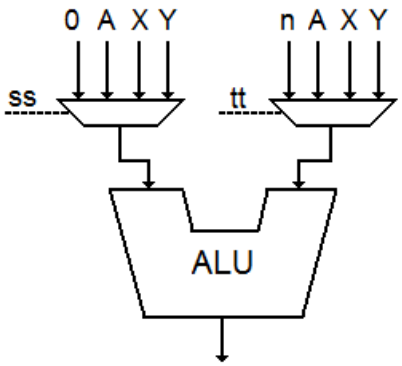

Figure 11: Multiplexer on the ALU input (Simplified)

Figure 11 shows one of the two inputs of the ALU multiplexer. The control signals ss and tt are derived from the command word and encode the first and second data sources which are used for the operation should be.

With ss = “00” the first input of the ALU can be assigned the value “0” become. This is always useful when an operation is the first input to ignore the ALU or to consider it as zero. Such cases were all has already been explained in Section 3.2.3.

The assignment of the second control signal with tt = “00” has again a special function. In this case, the 16-bit direct value n becomes from the instruction register through.

For each other combination of the bits in ss and tt, one of the data registers A, X or Y are selected.

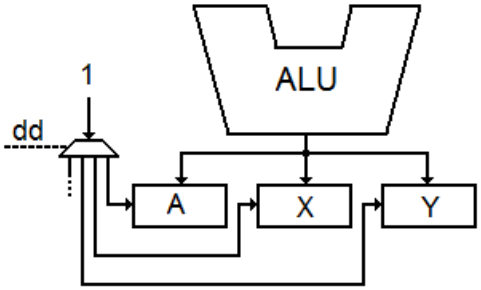

Figure 12: Data register at the ALU output

The inputs of the data registers are connected to the output of the ALU (Figure 12). The control signal dd, which is also coded in the command, where the result of an operation is stored. This is by a de-multiplexer which, depending on the choice of dd, assigns one of the registers to the command to store the value present at the input as a new value. In the case of dd = “00”, no data register receives this statement. The result of the operation is lost, but the flags Zero and carry in the status register (not on the figure) is still one new value. This allows the processor to perform comparisons between two values, without the result of the current content of registers.

The output of the ALU is in addition to the input of the command counter and the address line to the memory interface (Figure 11). This allows the ALU to calculate the addresses of jump instructions as well as load and store operations.

A load operation starts in the processor as follows: After loading the (Instruction Fetch) and decode the command (Decode) are the two multiplexers shown in Figure 11 is switched so that the address of the command is present at the ALU output. At the same time, the value of the command is given to the memory interface of the address to the memory output. As soon as the memory output is applied to an input of the ALU, the other is set to zero and an addition is executed (here further multiplexers are necessary, which are not shown in Figure 11). The result is saved as shown in Figure 12. This completes the charging process.

However, a memory operation is easier to implement by the control unit. Only the address has to be calculated by the ALU and the command for saving the memory is given to the memory interface. The value of the accumulator register A is permanently at the data input of the memory interface - only data from this register can be written become. This again has the simple background, logic, and thereby saving chip area.

The status register of the processor is mainly used for conditional jumps and is also designed to be minimalistic. There are only the flags for Carry and Zero, which can be used either according to a load, memory or ALU operation is automatically set or targeted by specific commands manipulated.

The operations of the scalar unit are connected via three data lines with the vector unit. This is a line for sending and receiving data, and one by one word from one select vector tabs.

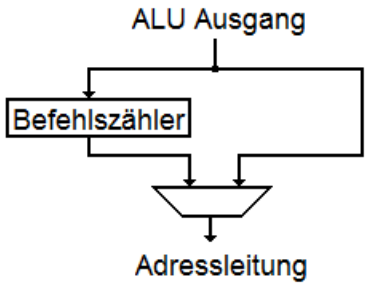

Figure 13: Command counter and address line

3.3.2 Microarchitecture of the vector unit

The vector unit (Figure 14) is simpler than the scalar unit, since this must only process data and not the program sequence and memory addressing. The central element this unit is the vector register set since each command is based on that resource access.

The vector register set includes a plurality of vector registers whose number (N) in the configuration. Each of these vector registers consists from any number of 32-bit words. Also, the number of words (K) will be in the configuration file. Higher values for K and N, however, result in a much greater need for chip area for the processor.

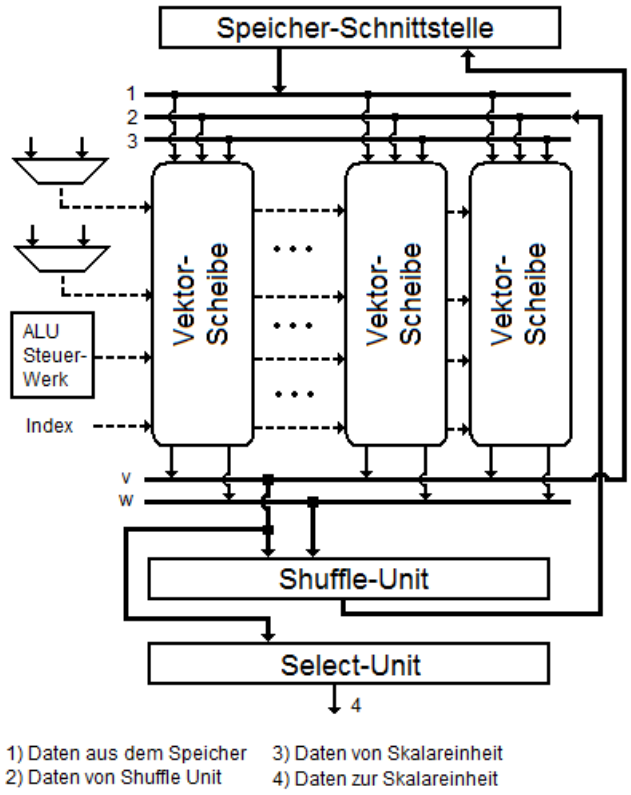

Figure 14: Vector operation

Figure 15 shows the structure of the vector register set. One line represents is a vector register (R0 ... RN), which in turn consists of K single words.

The vector unit is roughly divisible into two groups:

1. Logic that exists only once in the entire unit.

2. Logic in 32-bit slices used for each word in a vector register (also K times) is instantiated.

For example, all words from two vector registers really simultaneously, with K ALUs are instantiated. The Select Unit on the other side is available only once, as these are from all words of a vector register is only one selected and passed to the scalar unit.

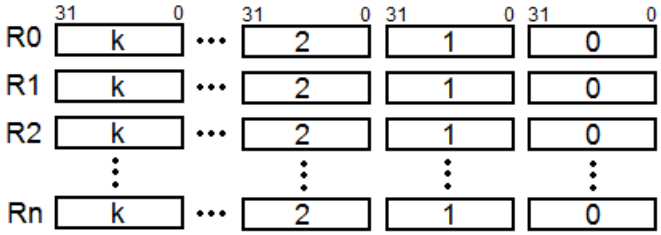

Figure 15: Structure of the vector register set

A 32-bit disk (Figure 16) of the vector unit consists of a 32-bit disk of the vector register set, a 32-bit optimized ALU and a multiplexer. The multiplexer is used to decide which source is the input data for the slices of the register set. Here, between the output of the vector ALUs, the memory interface, the Shuffle unit, or the scalar unit to get voted. It was attempted to use the 32-bit disks as much as possible slim design in order to claim even the smallest chip area for the coarse values of K.

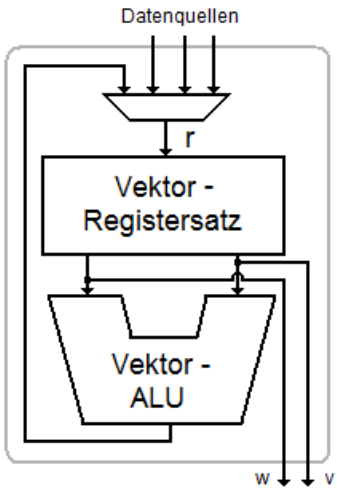

Figure 16: 32-bit disk of the vector unit

Each disk of the vector register set has a data input at which new value to be stored, two data outputs, two registers and a total of three lines to select the register number for inputs (v and w). In addition, the data line for selecting a word is applied to each disk (Index) from the scalar unit. This allows specific user data from the scalar unit into the desired disk of the vector register set to be copied. Each of the vector slices has a number between 0 and K-1, depending on its position in the vector operation. For data from the scalar to the vector unit, the input of each disk is applied and only the disk whose number identical to the index, stores it.

Components which are instantiated only once in the vector unit is the already-mentioned Select Unit for selecting a word from a Vector register, the Shuffle-Unit for re-mixing, a common (See section 4.4) for the ALUs and two multiplexers from which it is decided whether the selection of the target or source register. For a copy operation from the designated place of the command word or the direct value range. Furthermore, the vector unit has a data bus, which is in each case K-wide, which is distributed to the memory interface and returned to the 32-bit slices.

3.4 Memory Interface

The memory interface abstracts the access to the working memory. The processor does not have to be aware of the technical implementation of memory (distributed RAM, block RAM, external SRAM, external DRAM, etc.) is actually used. This allows the type of RAM easily can be changed without adapting the control unit of the processor have to.

In addition, the memory interface has the task of converting the vector unit of the processor's parallel access to the memory. In implementation, this will in some cases result in the fact that

the data internally only serially retrieved / written externally. But the data is always available in parallel.

Because the processor is never simultaneously data for the vector and the scalar unit. From the memory can read or write, has the memory interface only one address input. The desired type of draw is transmitted via a three-bit signal line. This is a bit which determines whether data should be written (WE), one which determines whether data should be read (OE) and one bit specifies whether the other two bits refer to a single data word for the scalar unit or a complete vector for the vector unit (US).

The memory interface (Figure 17) has separate lines to the scalar and vector unit, as well as separate lines for input and output outgoing data.

Because the controller of the processor must know when the data is at the memory interface is ready to be picked up, it will communicate its status with the help of ready signal. When the memory interface receives ready commands, the ready signal is set and remains on for a long time until another command is received. Once this is the case, set the memory interface ready back until the data is applied to the output.

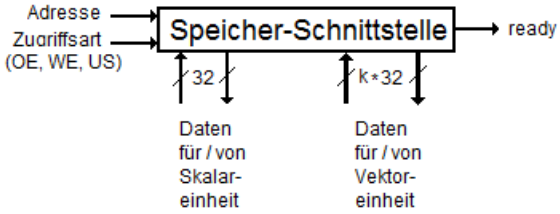

Figure 17: External view of the memory interface

3.5 Interface between scalar and vector unit

The scalar and vector units each have their own control unit, which work largely autonomously but not completely equal are. The control unit of the scalar unit is used in addition to the control unit of the scalar operation in addition to the task of implementing the sequence of the program. This means it is additionally responsible, to load the commands into the command memory, conditional and unconditional or to increment the instruction counter, to the memory for and the data transfer between the two units, to coordinate as well as with the control unit of the vector unit to synchronize. In addition, this control unit is used to reset the processors.

There are six signals for synchronization between the scalar and vector units available:

1. ir_ready, scalar unit → vector unit: The command was loaded into the command memory, the vector unit can begin its work.

2. v_done, vector unit → scalar unit: The command has been completely executed, the scalar unit can load the next command as soon as it is also finished.

3. s_ready, scalar unit → vector unit: Data for the vector unit is available. This can be both data on the memory interface and data for transfer between the two units.

4. v_feteched, vector unit → scalar unit: The data were fetched from the vector unit, the scalar unit can continue its work.

5. v_ready, vector unit → scalar unit: The data on the vector unit is available and can be read by other components. This is either the memory interface or the scalar unit.

6. s_fetched, scalar unit → vector unit: The data on the vector unit has been collected, it can continue its work.

In addition to these signals, the control unit of the scalar unit has only a few inputs. This is the command word, the ready signal of the memory interface, the reset signal, and the two flags carry and zero of the status register. The scalar operation is designed in such a way that it only manages with individual control signals and many multiplexers are switched directly from parts of the command word.

The state machine in the scalar control unit was equipped with the Instruction Fetch, Instruction Decode, and Execute. Based on this, the state synchronization is also present, in which the control unit waits for the vector unit to be finished with its work. After the hardware is switched on, the processor is initially in the hold state and starts processing the program only after it has received the reset signal.

In the control unit of the vector unit, the phase Instruction Fetch is replaced by the fact that only the message of the scalar unit is waited for whether the new instruction is in the command register. The vector unit also does not require a synchronization state since, after execution of an operation, it is simply possible to switch back to the waiting state.

4 Implementation

A prerequisite for the processor is that it should be easily expandable. This is achieved by three approaches:

1. The command encoding is designed to allow still space for new commands is free or can be easily captured.

2. The processor is called soft core in the hardware description language VHDL (see section 2.2) and can thus also be used by or modified to the one that wants to use the processor.

3. The source code is divided into several distinct components organized. This can be the right place for changes quickly found and the risk of undesirable effects to other areas of the processor is minimized.

In addition to the processor itself, programming tools in the form of assemblers and a debugger in the Python programming language. Thus, programs could be generated and executed and validated using the debugger on the test hardware, an Xilinx Spartan 3A (XC3S700A) FPGA (see section 2.1).

4.1 Hierarchical structure

Since the source code for the processor has a considerable scope, the components were constructed hierarchically over several levels. This makes the program more straightforward and reduces the likelihood of errors in signal assignments, since this can often occur within one component and not everything must be done in one level. In addition, some of the components can be tested with test benches in smaller groups and not just individually or in the entire system.

The top level is called a system. Here are the components "Debugger \ (see section 5.3) and" Memory \ connected to the processor.

The processor or CPU level, in turn, consists of the components "Scalar control unit", "Vector control unit", "Vector operation unit" and various component groups for the scalar operation (ALU, register set, etc.).

The vector operation consists of once-instantiated components, such as the vector ALU controller and the multiple (K times) generated slices, consisting of vector ALUs, multiplexers, and a register set respectively.

4.2 Configurability

What can be configured at the processor has already been described in Chapter 3.1. Each of these parameters, however, affects not only the functionality, but also the required chip area and, in some cases, the speed of the circuit. Thus, depending on the application, a suitable configuration must be created. This means that it is possible to fill a complete chip in order to achieve the maximum computing power or to occupy as little space as possible in order to achieve exactly the required performance.

The configuration is performed centrally for the entire processor in the config.vhd. No VHDL components need to be replaced but only the values of constants are adapted.

4.3 Optimization of control units

In the optimization of the processor is a very large part of the time which had to be spent on the whole development. Besides the increase of the clock frequency, care was also taken to reduce as little chip area as possible and to reduce the number of clocks required for a command.

In some places, however, it was also useful to use less-used commands to several bars to distribute the clock frequency of the overall system or to save a lot of chip space.

The controllers of the processor itself (scalar and vector unit) were using finite automata. This was the alternative between Moore and Mealy automata.

For Moore automata, the output of the output is only dependent on its current state, while the output of a Mealy automaton is determined by its state and its input.

The control unit of the scalar unit was initially realized as a Mealy automat. The ability to perform actions at the transition between the states and to react directly to events can save valuable bars during the execution of the commands.

However, it has been found that the Mealy machine, though less state, but also the paths for signals become longer and the possible clock frequency of the entire processor drops. In addition, there is the risk of a combinational loops in a Mealy machine.

For these reasons, the control units were converted to a Moore automaton and an increase in the clock frequency from 19 to 28 MHz (depending on the command) was achieved for one or two additional clocks.

In addition to the conversion to a Moore automaton was above all things the interplay with the memory interface is improved. The control unit of the memory interface was converted from a Moore to a Mealy machine to enable the scalar unit to communicate changes to its state as quickly as possible. In addition, the memory interface is no longer waited for the readiness of the memory interface to be confirmed with the ready signal. Since the scalar unit is, in principle, the only unit which induces a train, it can be assumed that the memory interface must always be ready before reading or writing commands and data. Although the debugger is allocated to the memory, this ensures that the memory interface is again in the state that the control unit expects after an operation. This makes it possible for the debugger to clock the processor independently of the memory and memory interface. If other components of the system also want to access the memory, the task of the memory interface is to wait for the readiness of the memory. By means of these changes, charging and memory instructions can be concluded in two instead of four clock cycles.

4.4 Optimization of vector ALU

The 32-bit-wide ALUs of the vector unit must be able to detect their operations to data in 8, 16, 32 and 64 bit formats. For this reason, each of these 32-bit ALUs were composed of four 8-bit ALUs (Figure 18) which are connected to each other via a carry line and a line for the transfer of the shift operations to the right. Thus, if an operation is not to be performed on the entire width, only the transfer lines must be separated.

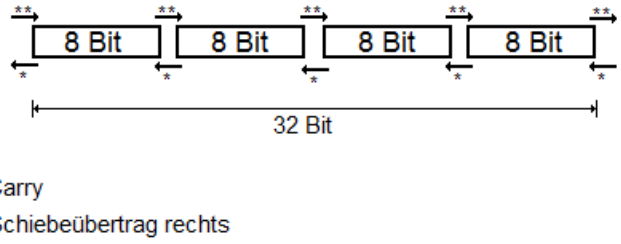

Figure 18: Composite 32-bit vector ALU

In order to also be able to process data with 64 bits, two 32-bit ALUs or vector slices must also be connected to each other (FIG. 19) through the two above-mentioned transmission lines. This was implemented in the VHDL source code by a generate statement, for which the "transfer line" is connected differently for odd values and for even values of the index variable. The carry output of the even instance goes to the carry input of the odd one, the carry output of the odd one goes to the input of the even instance, which is provided for the “transfer of shift operations to the right”.

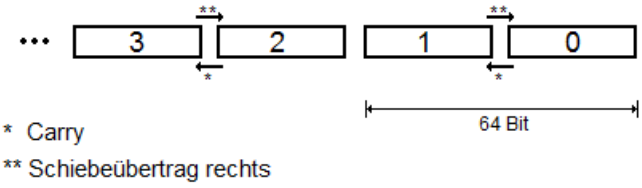

Figure 19: Connecting the vector ALUs for 64-bit mode

Due to the partitioning of a 32-bit ALU into 8-bit components, no optimized 32-bit arithmetic units (e.g. carry-look-ahead adders) are generated by the synthesis, but four 8-bit versions thereof series. Thus, in the case of addition with a 64-bit data width, eight series-connected 8-bit adders are obtained. This series circuit ensures very high signal propagation times and was the reason why the processor ran only at 28 MHz after optimization of the control units.

For this reason, the 8-bit ALUs were first transferred to the carry-principle (parallel pre-calculation, Figure 20). Everyone 8-bit ALU was generated twice. One block is connected to the transfer element fixed zero, on the other fixed one. Thus the results for both cases are calculated in parallel. The selection of which of the two results is actually to be used is effected via a multiplexer, which is switched on the basis of the transfer value of the predecessor block. As a result, the signal propagation time is limited to performing an 8-bit operation and switching the multiplexers on the remaining blocks one after the other.

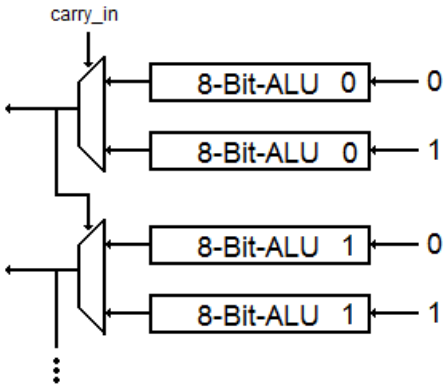

Figure 20: Carry-Select principle

By this method, the clock frequency could be increased from 28 MHz to 49 MHz, but the chip area required for the ALUs was also approximately doubled. For this reason, the configuration could be used to determine whether the ALUs should be set up according to the carry-select principle or not. In addition, the 64-bit mode could also be switched off and thus a higher clock frequency could be achieved without carrying-select.

Another, general approach to increasing the clock frequency is to use one calculation to several bars, and in each cycle only one subproblem.

This approach has been implemented by completely rebuilding the 32-bit ALUs as a switchgear (Figure 21) instead of as a switching network. Instead of performing four 8-bit operations in parallel in four 8-bit ALUs, 8 bits are selected from the input data in succession, processed by a

single data processing unit, and the result is written into a 33-bit register (Transfer bit + 32 data bits).

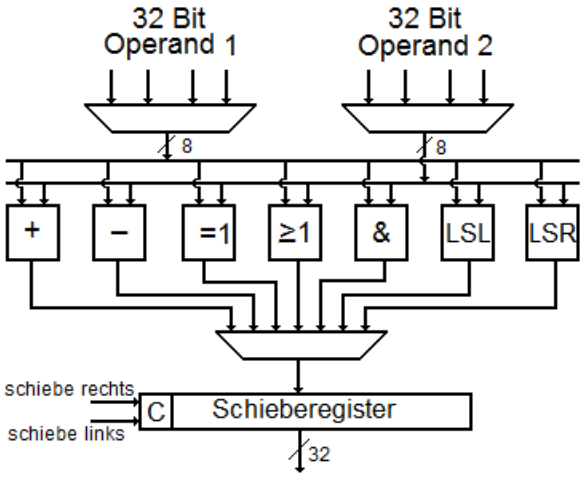

Figure 21: Simplified Representation: Vector ALU Switchgear

This eliminates three of the four 8-bit ALUs in each 32-bit ALU, but additional logic for the register, the multiplexers and the control unit of the vector ALUs is also necessary. Since the control of the vector ALU instructions is independent of the data applied thereto, there is a common vector ALU controller which is outside the 32-bit slices.

Since the handling of an ALU implementation in the forum of a derailleur is more difficult to handle than a switching network, additional control signals must be derived from the control unit of the vector unit to the control unit of the vector ALUs and vice versa. A signal determines that the vector ALUs should start their work. Another status signal confirms when the vector ALUs have performed their work and the data can be fetched at the output. This signal could in principle be omitted, but the control unit of the vector unit would then have to know how long the individual ALU operations last. 64-bit operations require more clocks than 8, 16, or 32-bit operations. For this reason, and taking into consideration the circumstance that the processor should be easily expandable, this signal has been introduced. This means that operations that require more or less bars (e.g. multiplying by additions) can later be integrated more easily since the control unit of the vector unit does not have to be adapted, but the "changes" locally of the ALU.

In total, the clock frequency could be increased from 49 to 67 MHz and the chip area for the transport unit could be reduced to a good third. All this is paid for by using a vector ALU command for additional bars. However, since the chip area can be used in this manner to increase the number of words per vector register by about three times. The performance for all other commands is increased by the increased clock frequency, and the 64-bit mode does not cause any further costs, this implementation was retained and the previous system was completely rejected with the carry-select principle.

4.5 Multiplier

To integrate the multiplication function into the vector ALUs, an additional multiplexer has been inserted before the 33-bit register within the vector ALUs (Figure 22). This can be switched so that either the result from the multiplication or that of the other operations is applied.

Since the multipliers greatly lengthen the signal propagation times and thus block-RAM resources have to be used, the multiplication can be activated or deactivated by configuration. If no multiplication is used, the multiplication instruction sets the destination register to the value zero.

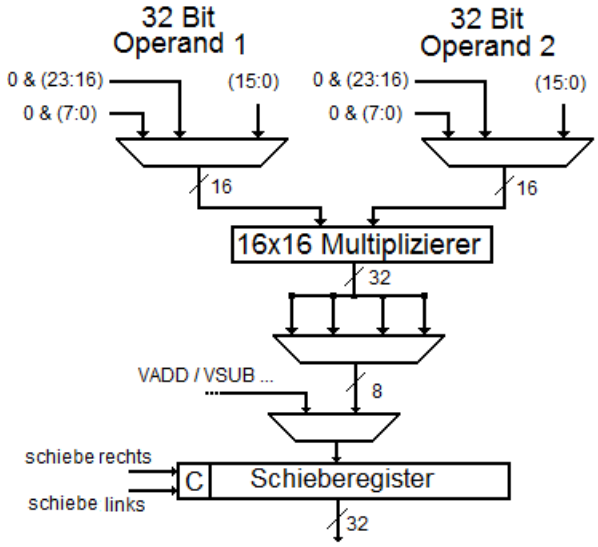

Figure 22: Integration of the multiplier

4.6 Shuffle

The shuffle unit is responsible for the horizontal shifting of data. For the mixing process, different word widths can be specified in the command. In order to make the following explanation easier to understand, the effect of the Shuffle command is first explained with full word width. Furthermore, it is assumed that the Shuffle unit has been configured and synthesized with the maximum possible bit width.

4.6.1 Definition of the operation

The shuffle operation works with the data from two source vector registers v and w. These two vector registers are each decomposed into four equally large parts, and the output vector register r is formed from the resulting parts in any order. The output vector register has the same size as a source vector register. This means there is a maximum of four of the eight parts in it. However, a part of v or w can also be copied into two, three, or four times in different positions in r.

The second bit (ww) is defined by the 14 bits of the "VSHUF" command. Ssss), it is decided which part of r is to be copied from v or from W. For a binary zero, data from v is used, for one data from w, the remaining eight bits (nnnnnnnn) indicate the position of the parts from which

the data from the source registers are to be copied, where two bits are provided for a position, where "00" denotes the least significant part and "11" the most significant part of the source register.

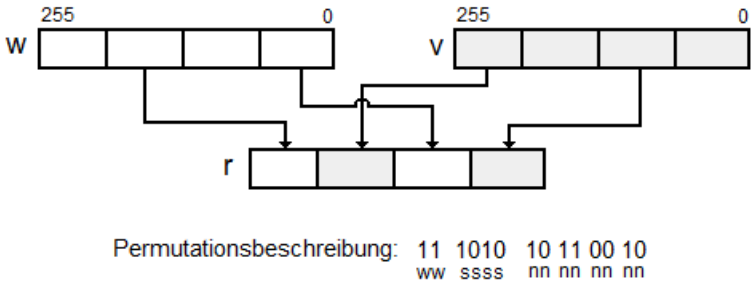

Figure 23: Shuffle operation at full word width and K = 8

The word width for the Shuffle operation can be specified in four steps. Full word width (ww = "11\) means that entire vector registers are split into four parts. At half the word width (ww =" 10 \) the Shuffle-Unit divides the vector registers into two halves and treats them as they would be an entire vector register. That is, the halves are each decomposed into four parts, mixed according to the method described above, and the result written into the half of the vector register r. For the remaining word widths ww = "01 \ and ww =" 00 \, the Shuffle unit operates analogously on each quarter or one eighth of the size of a complete register.

In the configuration, the bit width can be set, which can mix the Shuffle unit to a maximum. If this value is less than the length of a vector register, the unit treats the registers only up to this limit. The remainder of the output vector register r is filled with data from v, which are located at the appropriate position.

4.6.2 Implementation of the shuffle unit

In a simple implementation of the Shuffle-Unit as a switching network it has been shown that this requires an extremely large chip area (about 4100 slices and 8200 LUTs on the test FPGA XC3S700A at K = 16). This is based on the fact that four multiplexers per word width exist for each of the source registers. In addition, you have to select whether the data are transferred from v or w. Since the vector registers can have a considerable width, depending on the choice of the parameter K in the configuration, very large multiplexer structures are formed. For this reason, the processing of the operation was divided into several bars.

The following approach (Figure 24) has proved to be the most suitable of the tested:

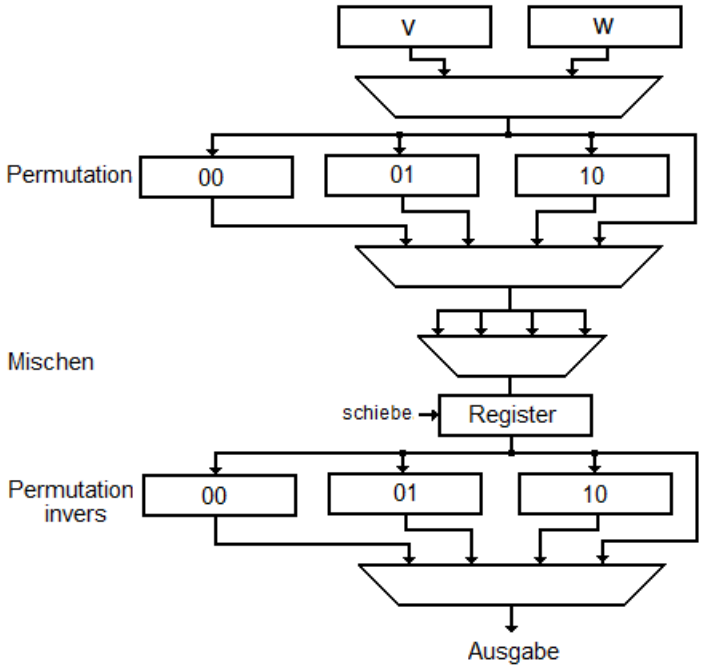

Figure 24: Structure of the shuffle unit

The processing can be described in three logical steps. The data sources are permuted, mixed, and then permuted with the inverse logic.

The permutation (Figure 25) is dependent on the word width of the shuffle-operation, in the widest range of words it can even be completely omitted. The source register is considered partitioned into four equally rough parts. Depending on the word width of the operation, these four parts are again divided into two (ww = "10 \), four (ww =" 01 \) or eight (ww = "00 \) regions.

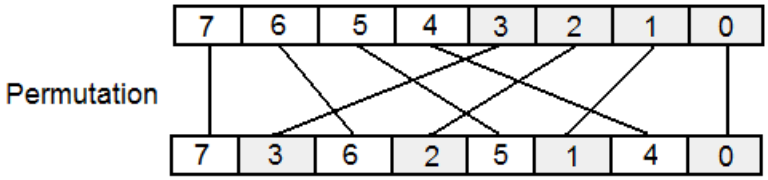

Figure 25: Permutation for word width “10”

When mixing (Figure 26), the four parts created by the permutation are inserted into a shift register in four steps. The order of the parts is changed according to the specification of nnnnnnnn in the shuffle command. This step is not dependent on the word width.

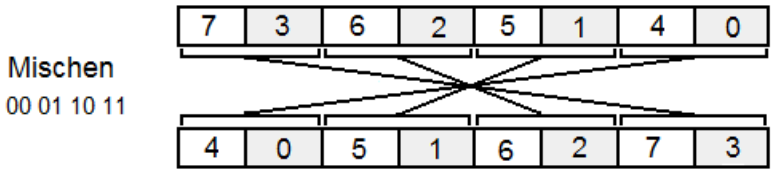

Figure 26: Mixing with word width “10”

The last step (Figure 27) for the shuffle operation is to re-sort the data from the shift register by means of a rule which is exactly inverse to the already applied permutation.

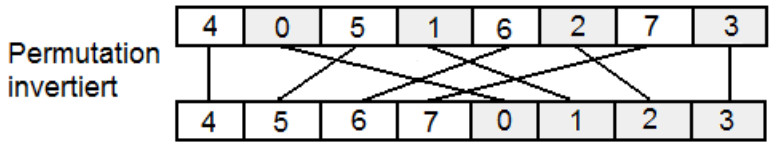

Figure 27: Inverse permutation for word width “10”

This provides the result of the Shuffle operation at the output. The chip area, which must be additionally used for the permutations, is saved by the fact that only a single word width has to be used for shift operations in the register and the actual mixing. This implementation results in less required chip area (1135 slices and 2383 LUTs on the test FPGA XC3S700A at K = 16) than all the other methods tested, and performs the mixing operation independent of the word width in four bars.

It would also be interesting to implement the Shuffle operation in which the mixing is handled in 8-bit units within the 32-bit vector slices. Logic in the vector slices can be optimized better than logic, which uses whole vector registers as input or output because of the smaller scope of the synthesizer. Unfortunately, this could not be implemented for time reasons.

4.7 Validation of the components

The validation of a system is an important development component and should be started as early as possible.

Using the method described in Section 2.2 to validate VHDL circuits, the individual components of the processor could already be tested when the system as a whole was not yet lukewarm.

This ensured that errors which arose during the testing of the entire system were not caused by the internal life of faulty components, but by the interconnection.

5 Test environment and development tools

5.1 Test environment

Since the processor alone is not executable, additional memory had to be provided in which programs, intermediate and final results can be stored. The generated memory is controlled by the memory interface and, for the sake of saving time, is only an SRAM implementation.

This implementation is based on a 32-bit wide array. The length of the array is selected so that the program as well as intermediate values and results are placed there. Since the data is only accessed synchronously, the block RAM of an FPGA can be automatically used by the synthesis tool for the memory. However, it is also possible to map the SRAM into LUTs in order to save the block RAM for the multipliers.

It is also conceivable, in addition to real memory, to additionally address so-called "memory-mapped IOs." This means that, behind certain memory addresses, no memory hides, but input or output of the chip whether a button has been pressed or an LCD display can be controlled. This possibility can be made in the "top-level module" of the system by selecting the address of the memory access, whether the request is directed to the SRAM module or to the input or output signals.

5.2 Assembler

With the help of the assembler, assembler programs, as described in some places in this document, can be converted into machine programs. The assembler was written with the Python programming language and is therefore platform-independent.

The assembler also supports the assembler directives "ORG \," EQU \ and "DC \.

With "ORG", the assembler can instruct the user to place the following machine commands or values from a specific address or to set jump marks there. This makes it possible, in particular, to keep coarse areas free in the memory for intermediate results.

The statement "EQU" corresponds to the definition of a constant, which then can be used as a direct value in the program.

```
EQU K 8       ; Konstante K mit Wert 8 definieren
SUB 0, A, K   ; Register A mit 8 vergleichen
```

With the help of "DC \" values can be stored directly in the memory. This is necessary, for example, if a required value is too rough an immediate value in the command word.

```
MASK_FF: DC 255   ; Wert 0xFF an Marke MASK_FF ablegen
```

Direct values and constants are specified either as a decimal number or by prepending a $ symbol in the hexadecimal or % symbol in binary format. When calling the assembler, you can specify as the parameter to which output format the program is to be translated.

- Symbol table: A symbol table is created by the debugger which is presented in the next section.
- Coe: A COE file is created, which is used for this purpose can initialize memory blocks instanced by the Xilinx core generator.

- Vhdl: Creates the VHDL code of a complete SRAM device that contains the program. This VHDL code can be copied unchanged into the SRAM module of the system.
- Bin: The program is translated into machine commands in bin format. The resulting .BIN file can be saved into the memory using the debugger of the system.

5.3 Debugger

A software developer requires a system to run not only programs, but also a feedback about the progress. It is important to be able to figure out where and why an error occurred. While a developer has already many tools available for the PC platform, embedded systems often have to find ways to get the desired information.

For this reason, an on-chip debugging extension was integrated into the system as an additional hardware component (debugger). The debugger represents the intermediary between the processor and a serial (RS232) interface. Commands can be sent to the debugger from another computer system via this interface. This responds to the commands by executing the desired actions and then sending a response.

5.3.1 Generation of clock signals

If the processor was synthesized with a debugger, it is in a state in which each clock must be specified explicitly via the serial interface. However, a debugger statement allows you to set the processor to a free state. This means that there is no waiting for clock commands, but clock signals are automatically generated by the debugger unit until the command to leave the free state is received via the interface. The frequency of the clock that is present at the debugger must be 50 MHz as the length of the signals for communication via the RS232 interface is derived from this. In the free state, the processor is addressed with the half of this frequency (25 MHz). From similar to the free state, the debugger still supports a state in which it generates self-sufficient clocks until the command counter has reached a predetermined value. Here, too, the clock frequency is 25 MHz.

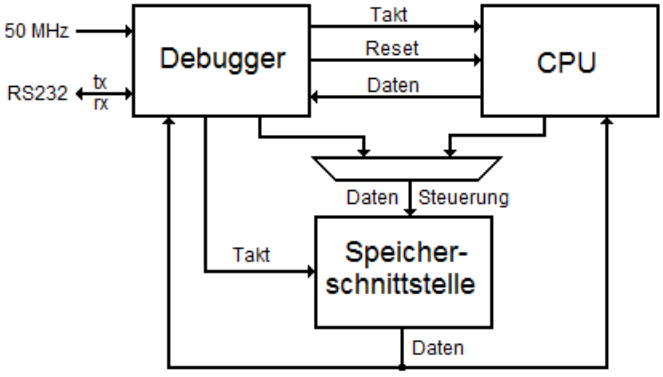

Figure 28: Integration of the debugger

5.3.2 Memory Connection

Since the debugger accesses the memory, a separate clock line exists for the memory interface, including memory, which can be switched independently of the rest of the system. This is necessary so that the access to the memory does not affect the execution control or data within the CPU. If the debugger wants to address the memory interface, the operation which the CPU has started must first be completed. During this time, only clock signals are generated for the memory interface and the memory itself. Finally, the debugger completes his request. If this is completed, the state that the memory interface had before the debugger had to be restored. The restoration is however very complex, since the internal state of the memory interface to the outside is not known. Instead, the debugger ensures that access to memory is complete as soon as the control is returned to the CPU. This means that the CPU can continue to work in any case, even if some cycles have been saved during memory access.

5.3.3 Communication and commands

5.3.4 PC-Software

5.4 Validation of the overall system

The entire system was initially validated using VHDL testbenches. The processor was simulated together with memory, which contained simple programs. The contents of the data registers and the status as (carry, zero) were able to detect whether the processor had worked properly.

Since the simulation of the processor but possibly occurring timing problems. It had to be realized and tested in real hardware. The processor was synthesized for the Xilinx Spartan 3A (XC3S700A) FPGA and played on such a chip. With the help of the on-chip debugger, various programs could be executed and the operation be followed.

In order to be able to validate the processor quickly and easily, a program was created, which executed all possible commands and results of the operations checked for validity. This was as follows realized:

In the first place, the correct operation of conditional jumps, unconditional jumps and commands for modifying the status register checked. If an error occurs at one point, the processor automatically starts into a command that puts it in the hold state.

If these steps are completed successfully, the program runs with the Check the load and store operation. Subsequently, the results of ALU operations of the scalar unit checked for validity,

By using reference values from the memory or with a direct value are compared. With the help of the already - verified functionality, the program validates the correct operation of the remaining commands. The correct results for Vector ALU operations are not pre-computed in memory. Instead the scalar ALU is used to set date for a second time and compare the results with those of the vector unit.

The processor is validated by adapting the program to the configuration of the processor, and debugger software into the memory is loaded. Because, depending on the configuration, many thousand commands are executed. It is most useful to the processor's instructions in the free state (See section 5.3). Whether the program is successful has been traced by the fact that the command counter is stable. The value of the address of the program end ("FINISH" symbol).

5.5. Resource used

6 Results

6.1 Synthesis results depend on the configuration

The data from Table 5 show that by increasing the number (K) per vector register by the value of one, the number of required slices for the vector unit on the above-mentioned FPGA increases by an average of 321 and the number of LUTs by 599 increases. (Note: The "slices \" and "LUTs" columns refer to the entire system.) These high values result from the fact that a separate 32-bit disk is instantiated for each word and multiplexer structures in the select unit or the Shuffle-Unit become coarser.

If the part of the vector unit, which is not dependent on K, is neglected (e.g. its control unit), the chip area thus increases approximately linearly with K in the tested range (K = 4. It should be noted that for K, because of the 64-bit mode of the processor, only straight values can be used. An increase by the value of one is thus only possible by calculation.

Tested with the following configuration: Shuffle with maximum width activated – Vector shift activated with 32-bit width - 8 vector registers - no multiplier - optimization on speed.

The value of the change in the number of vector registers (N) is strongly dependent on the realization of the registers during the synthesis. In FPGAs, the two possibilities are to generate registers in block RAM or with the aid of many LUTs (parameters for synthesis). In the first case, the number of required LUTs and slices increases only minimally with N, but valuable block RAM resources are used. In the second case this is exactly the opposite. Table 6 shows that the increase in slices and LUTs increases non-linearly with N, but instead, higher resources are used for higher values. The number of required slices per vector register is in the range of 391 to 213, the number of LUTs between 318 and 130. In addition, the results at N = 32 indicate that a very good optimization is achieved by favorable selection of the parameters Can

The following configuration was used: 8 words per vector register

- Shuffle disabled - Vector shifter disabled - no multipliers

- Optimization on speed - disabled "RAM-Extraction \.

6.2 Performance Analysis

To determine the performance of the processor against other architectures, a simple form of performance evaluation was applied. This is the implementation and comparison of the runtime of a sample application for the processor developed here as well as modern desktop computers and a Motorola 68000 microcontroller.

6.2.1 Example application

In the example application, a 1600x1600 pixel coarse image is displayed by a filter in the local area.

The filter core used is shown in Figure 29.

For filtering, a pixel is assigned to each pixel in the image. This value results from the weighted sum of the values of the pixel itself and its eight adjacent pixels in the original image. The weighting depends on the position and the value provided in the filter core.

$$\frac{1}{16}\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

Figure 29: 3x3 filter core

If, for example, the new value of the central pixel from Figure 30 is to be determined, this happens by the following calculation:

$$\text{Neuer Wert} = \frac{1*30+2*16+1*12+2*34+4*18+2*18+1*22+2*16+1*16}{16} = 20$$

30	16	12
34	**18**	18
22	16	16

Figure 30: Sample data for image processing

The filter mask has been selected so that multiplications or divisions in the program can be replaced by shift commands. The working memory for the processor is implemented on the FPGA as block RAM and thus shares resources with the hardware multiplying. For this reason, no support for multiplications in hardware for the test FPGA could unfortunately be applied. Since shift operations and multiplications in the processor developed here require the same number of clocks, this does not result in any falsification of the results.

The application is well-suited for SIMD processors. Instead of one pixels per loop pass, each K-2 pixel can be simultaneously processed by the vector unit. K-2 results from the fact that the first

and last pixel in the vector register contains invalid values since the data for the pixels on the left and right of the central pixel must be shifted by one word in the vector register.

Pixels are interpreted as a 32-bit integer in the example application because the developed processor does not have any sibling arithmetic. A more realistic approach would be if each pixel consists of four, each with 8-bit coded color values, and the processing would be based on the word length "byte\".

Unfortunately, 40% of all commands are not vectorized in the parallelized algorithm, but are processed by the scalar unit alone. These are control structures for loops and the need to mirror pixels at the edge of the screen.

The RAM stored in the block RAM for the processor developed here is not large. Enough to capture a complete 1600x1600 pixel image. In order to be able to perform the performance evaluation nevertheless, only a single line of the image is stored in the memory. The algorithm then reads the image as if each line of the image contained the same data as the row in the memory. This results in the same runtime, which would also have resulted from filtering a real image. This methodology has also been applied to the Motorola 68000.

The example application was written for the configurable vector processor and the Motorola 68000 as an assembler program and for the desktop processors with the programming language C. The C program has been optimized with the GCC compiler for runtime (GCC flag -O3) and for the respective process location type (GCC flag -mtune = CPU type). MMX, 3DNow!, SSE or the second processor of the Intel Core 2 Duo were not used by the compiler to accelerate the application.

6.2.2 Results of performance analysis

In order to compare the results, the sample application was run on different systems and recorded how long it takes to filter a 1600x1600 pixel image. The measurements for the desktop processors were carried out under Linux with Kernel 2.6, with the Motorola 68000 and the processor developed here no operating system was active. From the measured time, the number of clocks that the processors need to process an image point can be calculated. The table below also shows the clock frequency at which the tested processors are operated and the number of transistors or FPGA resources they have built up.

The required number of clocks for the calculation of a pixel with the configurable vector processor can be estimated with a formula as a function of K. It should be noted that only the maximum possible power can be calculated. In the case of coarse values for K, there is increasingly the risk of wasting valuable computing time because of registers which are partially filled with user data. For this reason, K should be selected so that the number of pixels per line can be divided by K-2 without remainder.

……

7 Summary and outlook

7.1 Result of the work

In this diploma thesis, a new processor was developed, which is not comparable in the open-source field. The resulting command set is Turing-complete and independent of the width or number of vector registers.

The processor is validated and running in real hardware. This was checked with the help of VHDL testbenches and a Xilinx FPGA.

In addition to the processor itself, an assembler and an on-chip debugger were created. This provides the basic software tools necessary for the development of applications.

Using a sample program from the image processing area, performance was investigated and demonstrated that actually real applications can be executed on the configurable processor.

7.2 Possible extensions

Unfortunately, it was not possible in the limited time to find all the oen ideas for the processor presented here also. Its development must but do not end with giving this work. It is expected as an open-source project on the Internet and can thus be arbitrary and further developed.

A point that provides new application possibilities for the processor would be the extension that interrupts are supported. The processor could also be used in an environment in which quick reactions to events is important.

The comparison with other SIMD architectures shows further points to which the processor can be extended. For example, it uses vector operations on no saturation arithmetic. This means that the value ranges are likely in many situations and must be performed by the application. But also in the range of "horizontal \ operations or the processing of flow commands can still do a lot be learned from these architectures.

Also, interesting would be the development and performance analysis of a processor, which operates on the same command set, but its structure pipeline-like structure. A pipeline in the scalar part of the processor would increase the throughput of scalar commands per clock, whereas a pipeline-like vector unit might have a higher value parallelism at the same chip.

Also outside the CPU there are still areas where improvements can be introduced. For example, the memory interface so far only use the internal block RAMs. Because the processor is designed to process media data, it would be very useful, also access external SDRAM devices with high memory capacity

Nevertheless, at the end of the period of this diploma thesis is a more effective one processor, which is so far unrivaled in the open-source field.