MASTER'S THESIS | LUND UNIVERSITY 2016

Application Specific Instruction-set Processor Using a Parametrizable multi-SIMD Synthesizeable Model Supporting Design Space Exploration

Magnus Hultin

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-14

Application Specific Instruction-set Processor Using a Parametrizable multi-SIMD Synthesizeable Model Supporting Design Space Exploration

Magnus Hultin
magnushultin@gmail.com

May 18, 2016

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Flavius Gruian, flavius.gruian@cs.lth.se
Examiner: Krzysztof Kuchcinski, krzysztof.kuchcinski@cs.lth.se

Abstract

In this thesis, we provide a synthesizable model for supporting design space exploration of application-specific instruction-set processors. The model is written in a high-level of abstraction hardware description language Bluespec System Verilog and is parametrized to support different configurations for use in the design space exploration. To test the model, different applications from the media domain was selected to run on some of the configurations from the design space exploration. The applications were also run on a standard general processor for comparison.

The results show that there is a performance gain compared to the standard processor, but with a higher cost of resources. With the utilization of the resources the scheduling of the applications turned out to be critical for this performance gain. The synthesizable model also shows that there is a consideration of the maximum clock frequency and memory constraints that the theoretical design space exploration model does not take into account.

Keywords: MSc, VLIW, SIMD, processor, Bluespec, DSE

Acknowledgements

First, I would like to thank my supervisor Flavius Gruian for his valuable feedback and support throughout the project. This project would not be possible without his help and patience.

I would like to thank Mehmet Ali Arslan for his help and discussions.

I would like to thank my examiner Krzysztof Kuchcinski for his feedback on the thesis.

I would like to thank Lars Nilsson and Anders Bruce for their computer support.

Finally I would like to thank my family for their support and my sambo, Yuqiao Xie for her love and support.

Contents

Chapter 1
Introduction

Today's streaming applications (e.g. multimedia, networking) benefit from increased data and instruction parallelism in hardware architectures. Vector processing units (SIMD) are often employed to boost performance in standard processors. VLIW architectures and multi-core processors are commonplace today, offering support for instruction parallelism. Nevertheless, these solutions are often designed to support a wide variety of applications, resulting in compromises that may reduce performance for particular domains. Custom architectures (application-set specific) are usually the alternative adopted to obtain the right performance with the least amount of resources. Often design space exploration (DSE) is used to examine different designs before implementation. This makes it easier to do rapid prototyping, optimization and system integration. Implementing all of those designs in hardware can be a long and costly process to do by hand. Therefore it would be beneficial to have a model that could be synthesized for the different parameters explored in the DSE process.

1.1 Problem Definition

This project's focus is on supporting DSE for custom processor architectures introduced in [2]. In paper [2] there is a DSE model that uses Pareto points to identify how many scalar and SIMD units the processor should have and the width of the SIMD unit. This provides different architecture candidates that would meet the different requirements that are set and minimize the resources used. Using the model in [2], those candidates can be synthesized to hardware and run the different applications for evaluation. To support this design space exploration, we implement a synthesizable parametric model. This model has support for different vector and/or scalar units of different widths and parameterizable word size. It is written in Bluespec System Verilog (BSV) and is able to run applications written in machine code. A few architecture designs are compared to a Xilinx Microblaze [17] processor with applications chosen from the multimedia and signal processing domain.

1.2 Method

To be able to implement this synthesizable model, first the high level hardware language Bluespec System Verilog was chosen because of its high level of abstraction which generates a synthesizable Verilog model. Some time in the beginning of the project was dedicated to study the language. After that the first step was to implement a simple single instruction single data (SISD) processor. The idea was to first implement a simple scalar processor and expand upon the design later. The processor has a four stage pipeline, where the stages are instruction fetch, register fetch, execute and write-back. This model was then extended to support vector unit (SIMD). This vector unit is parameterizable with different width, word size, and an added stage for vector permutations. The simple processor was also expanded to support several scalar units in parallel, making it a very long instruction word (VLIW) processor. After this, both the SIMD and VLIW models are combined into a single model to support several scalar and SIMD units.

To test the architectures, programs were chosen from the multimedia domain and hard-coded for the processor. To maximize the resources and utilization, different types of schedule techniques were studied. The parameters that were examined was the number of SIMD units, the SIMD length and the number of scalar units. The measurements were, the execution time in clock cycles, the resources used in a FPGA, and the maximum clock frequency were examined. These measurements were compared to the Xilinx Microblaze processor.

1.3 Related Work

The reason Bluespec System Verilog (BSV) was chosen for this project was because it has been successfully used before in large complex designs. One is a 2-way out-of-order process or with the MIPS I integer instruction set architecture [7]. With that design the author concluded that designing in Bluespec was relatively simple if the design does not have to worry about performance, and that debugging the design was simplified in Bluespec compared to standard hardware description languages.

Another one is a native Java embedded processor, BlueJEP [8], where the authors had a design written in VHDL. The VHDL design was redesigned in Bluespec and features where added including a longer pipeline and speculative execution. The design was concluded to be more flexible, partly due to BSV, but also had an increase of device area with the about the same performance of the VHDL design. It was also concluded in paper [9], that compared the design process of the VHDL and BSV versions of BlueJEP, that BSV was better for fast prototyping and architectural exploration.

With the increasing capacity of FPGAs, soft-processors becoming more common. Processors such as the Nios II from Altera and the Microblaze from Xilinx are examples of soft-processors for FPGAs. Soft-processors can be customized to meet the needed requirements, but they lack of features such as VLIW or SIMD. A co-processor that adds the feature of parameterized vector processor design is the VESPA processor [18]. That processor is designed as a co-processor to a MIPS scalar processor, where the parameters are the number of vector units and the width of those units. That design has since been iterated on and support for 2D/3D vectors have been added. The memory was changed

from a vector address register file to a scratchpad memory and programming support was added in form of a compiler that could compile C code with vector extension for the architecture. It was also changed to be a co-processor to the Altera Nios II/f, the changes and optimization made it so that it could run 1.5-2x the previous clock rate [15]. Even later versions of the soft-processor add fixed-point support, 2D DMA and scatter/gather operations [16]. These processors, unlike our project, only look at SIMD style data parallelization as an extension to a scalar processor and not instruction level parallelization like a VLIW architecture.

A processor that adopts a VLIW parametrized architecture is the ρ-VEX processor [14]. It uses parametric 5 stage pipelined VLIW architecture with operand forward logic. The parameters for the processor is the issue-width, the type and number of functional units, the size of the multi-ported register files and if there should be operand forward logic or not. That design, unlike this project, does not explore data parallelism in form of vectorization of data.

The FabScalar project at NCSU is also related in terms of it developed a tool-set FabScalar, that is able to generate superscalar cores with various superscalar width, pipeline depth, and stage specific structure sizes [6]. Example of stage specific structures include L1 data cache in execute stage and fetch queue in decode stage. That project's main focus is too explore the instruction level parallelism with different types of superscalar cores, but does not look at vectorization of data.

The paper [13] explores both vectorization of data and instruction level parallelism in a VLIW-SIMD processor. The processor consists of two VLIW-SIMD cores that each take 4-slot VLIW instructions. These VLIW instructions have two four way SIMD ALU instructions and two load/store instructions. The processor was developed to accelerate object detection algorithm in embedded applications. It does not have scalar instructions like in this project and is not parameterizable.

In our project, the design flow we used was introduced in paper [2]. That design flow is proposed for speeding up the design space exploration (DSE) for custom processors. The DSE model uses a set of applications and identifies the processor configuration in terms of SIMD width, number of SIMD units and number of scalar units that is optimal for the application set. This provides different architecture candidates that meets the different requirements that are set and minimize the resources used. The work in [2] looks mainly at throughput to determine what kind of performance the design should have. In our project, we make a parameterizable processor model that can be used with the configurations as parameters.

1.4 Contribution

The major contribution from this thesis is a parametric SIMD processor model in Bluespec System Verilog, that was made to fit into the design space exploration model in paper [2]. This model includes parameters for different SIMD width, number of SIMD and scalar units like the DSE model, and is explained in more detail in the background chapter. Hopefully this will help supporting design space exploration of custom processor architectures and speed up the process of implementing custom processor architectures.

Chapter 2
Background

In order to get a better understanding of the work done in this project, this chapter will provide a brief description on the subject of processor design. We will go through the subject of pipelining and a couple of ways to achieve parallelization with data and instructions. These subjects are covered more in detail in books [12][10]. We also give a brief introduction to the hardware description language (BSV) used in this project and the different schedule techniques used when programming the architectures.

2.1 Design Space Exploration

Before a design gets implemented into hardware, it is useful to explore different designs alternatives in order to get the optimal design . In this project we are looking at applications-set specific processors, meaning the design space that is explored is for a group of applications, that in the end will determine how the processor will be designed. The reason to use this exploration process is so that the design does not have to be redesigned if the detailed implementation turns out not to have the necessary performance or if the implementation is using too much resources. This could be a waste of time and money or produce a design that waste resources like power and chip area.

The paper [2] introduces the design space exploration for application specific processor that is being supported and in some ways extended by this project. It aims at reducing the design time in the start of the design process by taking a set of applications and identify the processor configuration in terms of SIMD width, number of SIMD units and number of scalar units that is optimal for the application set. The work in [2] focus mainly on throughput to determine what kind of performance the design should have.

The design space exploration flow introduced in [2] is shown in figure 2.1. The applications chosen usually has a part that is repeated a lot and use a lot of computation time, this part is called a kernel. As input for the first automated part, the throughput requirements for these kernels and estimate clock cycles are assumed. After that the problem is

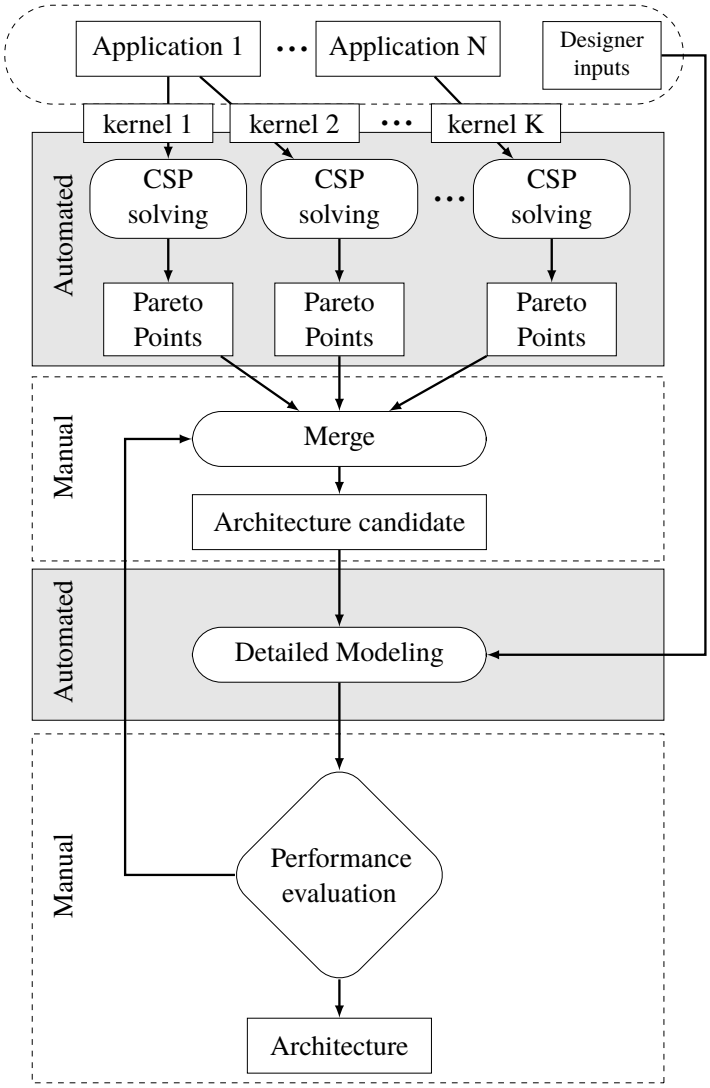


Figure 2.1: Design space exploration flow proposed in [2]

solved with constraint model resulting in Pareto points that will give the optimal solution depending on the throughput requirements. These Pareto points are the non-dominated solutions for a multi-objective function, where non-dominated mean the solution cannot be better in one objective function without making another objective function worse. Exactly how this is done can be read in the paper [2]. After that the designer can manually merge these points into one candidate. When the candidate is chosen, detailed modeling is done of the chosen architecture. This is where this project automates the process by making the detailed model out of the candidate specifications. For this some inputs have to made by the designer, because the memory sizes and programming of the applications are not a part of this design flow yet.

This model can then be evaluated by the designer to make sure that it meets the expected performance requirements. If the design does not meet the requirements, the designer can go back and reiterate the process by choosing another merge or the Pareto points and rerun the detailed modeling. When an architecture has met the desired performance it can

then be implemented in more detail, which is a time consuming effort.

This project therefore assumes the work is done in the first part of figure 2.1 and is doing the work in the second automated gray box.

2.2 RISC Pipeline

When describing processor architectures, the *reduced instruction set computer* RISC architecture is often used as the example because of its simplicity and how commonly used it is today in ARM, MIPS, SPARC processors. The DLX processor introduced in the first edition of [10] used for teaching is also using this architecture. The architecture could be simplified to the following 5 steps.

1. Instruction fetch
   Fetches the instruction from the instruction memory at the program counter (PC) location. Increments the PC.

2. Instruction decode/register fetch
   Decodes the instruction and reads register

3. Execution/effective address
   Performs the specified ALU operation or compute load/store address or compute branch/jump address.

4. Memory access/ branch completion
   Only active with branch/jump or load/store instructions. Accesses the memory if needed. Loads data or stores data to memory.

5. Write-back
   Writes the result to register files.

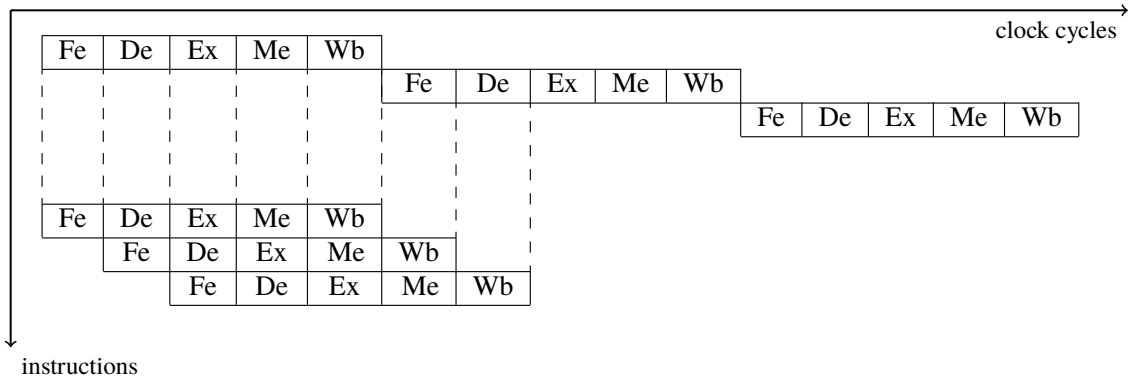


Figure 2.2: Comparison between a sequential processor and a pipelined processor. Shows a 5-stage pipeline with *Fetch*, *Decode*, *Execute*, *Memory Access* and *Write-back*.

When executing these steps it would be inefficient in most cases to wait for the instruction to go through all the steps before the next instruction would start executing. Therefore

pipelining exists which partially executes several instructions at the same time. Figure 2.2 shows the difference between a 5-stage pipeline and a sequential processor when executing three instructions. The sequential processor in this case goes through the fetch, decode, execution, memory access and then write-back stage before starting to use the fetch stage again for the next instruction. With having the pipelined stages, the processor starts the fetch stage for the second instruction right when the first instruction is processed in the decode stage and so on.

2.3 VLIW

In order to achieve more parallelism, instructions can be group together and executed at the same time. When dependencies are determined at a software level and instructions are grouped together to form a very long instruction, the type of processor is called a *very long instruction word* (VLIW) processor. Because all the dependencies are checked at compile time, the hardware does not need logic to check for dependencies, making the hardware design of VLIW processor more simple.

Figure 2.3 shows a 5 stage pipeline of a VLIW processor with 3 execution units. The instruction has three instructions turned into one very long word instruction. The execute units can then use those three instructions at the same time. The assumption is that the compiler made sure there is no dependencies of the instructions and that there will not be any memory issues, like cache misses. Compared to the RISC pipeline the instructions per clock cycles is higher.

Difficulties with the VLIW design lies in the compiler that has to check for the dependencies between instructions and schedule many instructions simultaneously, because of potentially complex branches in the application. If there is such dependencies the compiler could insert NOP (no operation) instructions until the dependency is resolved or reschedule the instructions. Also, the code generated depends on the design of the processor where if for example a processor with 3 execution units need different code than a processor with 4 execution units.

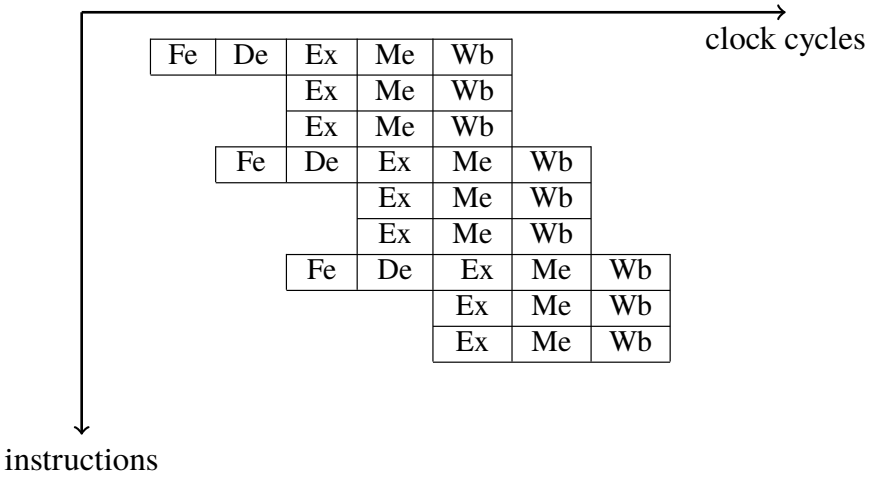
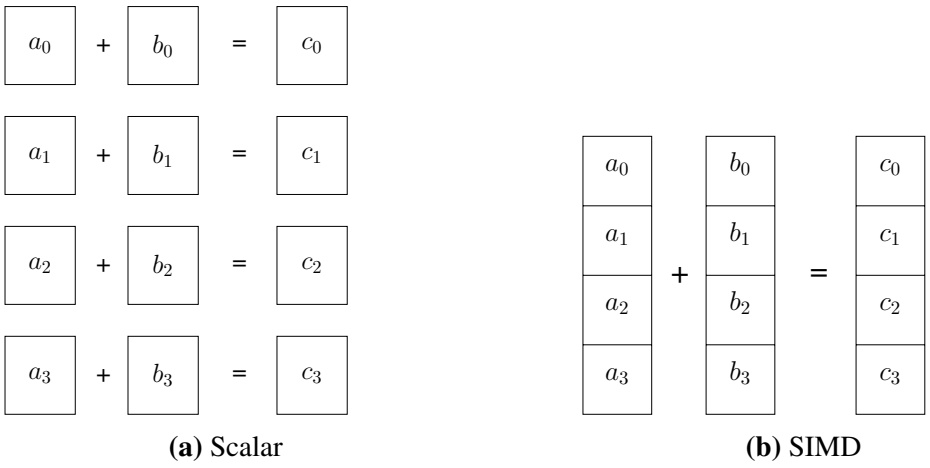


Figure 2.3: VLIW 5-stage pipeline with *Fetch*, *Decode*, *Execute*, *Memory Access* and *Write-back*.

2.4 SIMD

In many multimedia applications there exists an array or group of data that the same instruction is gonna be used on. Single instruction multiple data (SIMD) will take advantage of this and execute the same instruction on all of the parallel data. Figure 2.4 compares the four scalar operations it would take to execute a single 4 width SIMD operation. In this example it only takes one SIMD instruction to execute what would have been four scalar instructions. If the scalar instructions would had different types of operations, for example addition and subtraction then the SIMD instruction would have had to been divided to one instruction with the addition operation and one instruction with the subtraction operation. In a VLIW processor with 4 execution units the operations could be of different types and executed at the same time. For example two addition and two subtraction operations executed at the same time.



(a) Scalar **(b)** SIMD

Figure 2.4: Comparison of four scalar operations and a single 4 width SIMD operation

2.5 Scheduling Techniques

When programming custom architectures, different techniques are used to take advantage of the parallelism of the architecture and therefore increase the utilization of the hardware resources. This section will go through some techniques used for this project, but for a more detailed description the paper [1] is a comparative study that describes the techniques in more detail.

2.5.1 Single Iteration

When scheduling a single iteration of a loop/kernel, the iterations are executed in sequential order. Every iteration is scheduled as efficiently as possible in terms of using as few NOP (no operation) instructions as possible.

As an example, a part of the FDCT application that will be used in evaluating the design in this thesis, will be scheduled for single iteration. The data flow graph for the part of

FDCT that is scheduled is shown in appendix B.1. The data flow graph, scheduled for a SIMD pipeline is shown in figure 2.5. The graph is scheduled for a single iteration where the latency of the pipeline determines when the next operation, that has a data dependency can be executed. In this example, the latency is four clock cycles and the vector length is four. This makes it so that four of the same operation can be scheduled at the same time. The critical path will give the length of the schedule. A VLIW pipeline would be similar to the SIMD pipeline, but with the different types of operations scheduled at the same time as shown in figure 2.6, with the multiplication and subtraction operations scheduled at the same time as the addition operation. The latency is lower and the throughput is higher for the VLIW single iteration as shown in table 2.1.

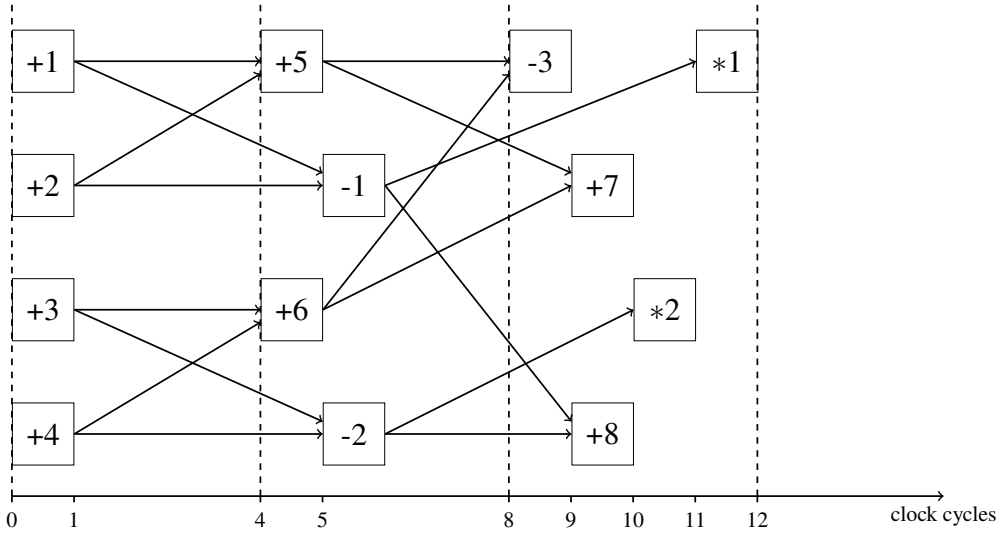


Figure 2.5: Example of a single iteration schedule for a SIMD pipeline

2.5.2 Overlapped Execution

As shown in the single iteration example, there is a delay because of the data dependencies. In order to fill the pipeline, multiple iterations could be executed instead of waiting, assuming the iterations always have the same operations. Figure 2.7 shows the same example as in the single iteration section, where the iterations are alphabetically ordered, A,B,C ... and so on. The single iteration has had its NOP instructions minimized. To fill the pipeline at all times three iterations (A,B,C) was overlapped because of the latency of the pipeline. This will fill the pipeline and increase the throughput but still does not utilize all of the vector elements in a SIMD pipeline. The overlapping comes at the price of having to store all the active data in registers from the overlapping iterations and the latency of the input to output data is increased compared to a single iteration schedule as seen in figure 2.1.

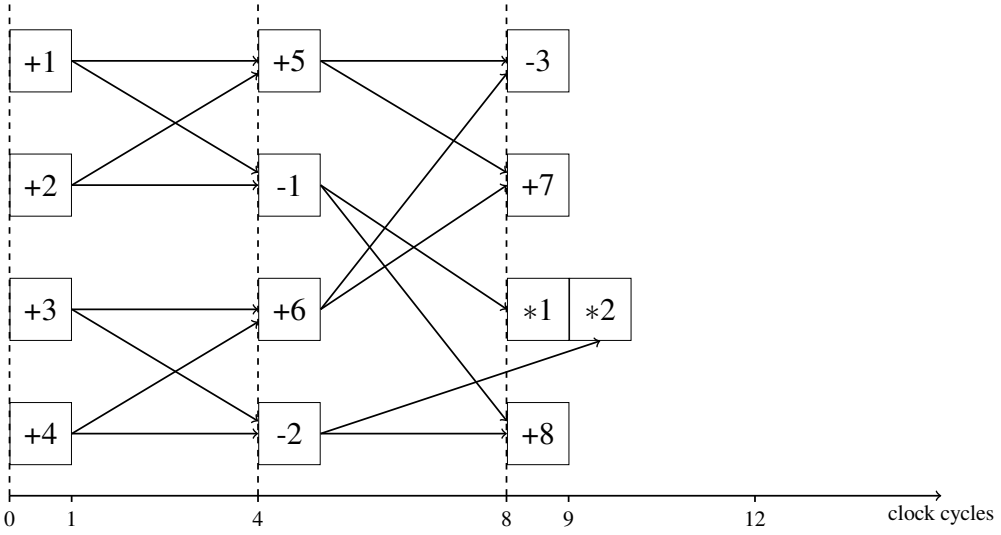


Figure 2.6: Example of a single iteration schedule for a VLIW pipeline

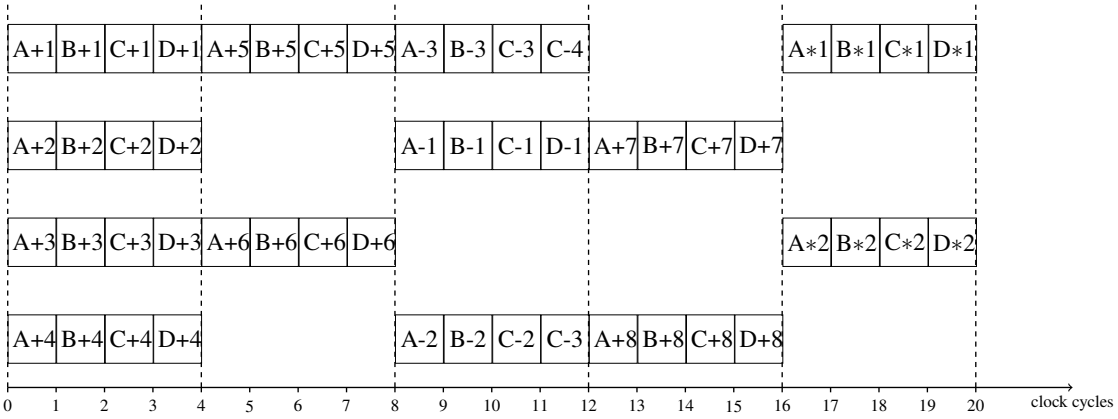


Figure 2.7: Example of a overlapped schedule for a SIMD pipeline, where A,B,C is the iterations

2.5.3 Loop Vectorization

Another technique to utilize the vector more would be to unroll a number of iterations (unroll factor) and fill the vector with unrolled operations. In figure 2.8 the single iteration example is used with the unroll factor four and the vector length four. Operations from Iterations A to D are scheduled in one vector. Although there is still a delay between data dependencies, this makes it so that every operation utilize all of the width of the vector. Dependencies could exist between iterations that would make loop vectorization impossible. In this project the applications chosen did not have such dependencies.

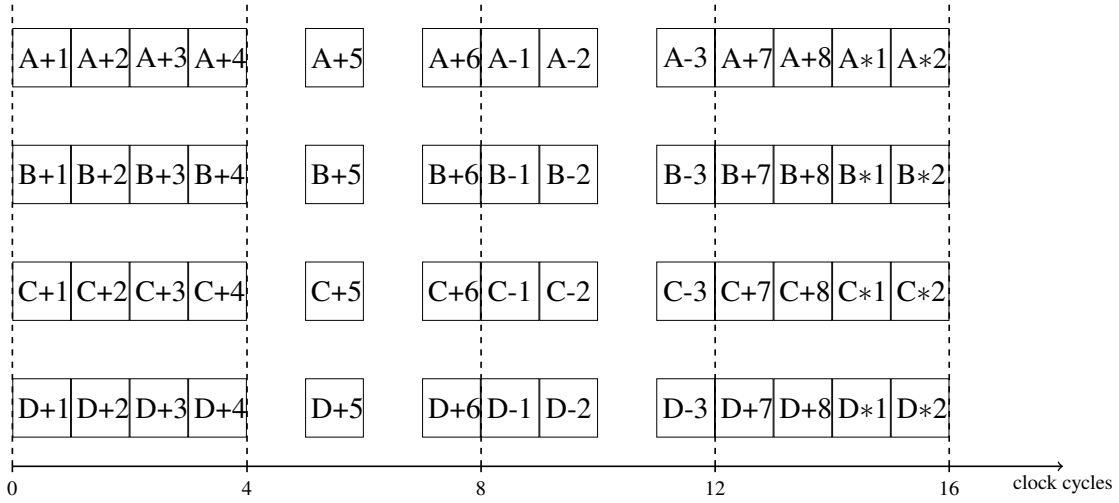


Figure 2.8: Example of loop vectorization for a SIMD pipeline

2.5.4 Loop Vectorized then Overlapped Execution

In order to increase the pipeline utilization of the previous case, the overlapped technique can be used after the loop vectorization, but only if there is enough iterations left to schedule. Figure 2.9 shows loop vectorization example overlapped with the the next iteration. The light gray operations in the figure are the overlapped operations. This way, the throughput of this example is the best of these techniques, but the latency is also the highest as seen in table 2.1.

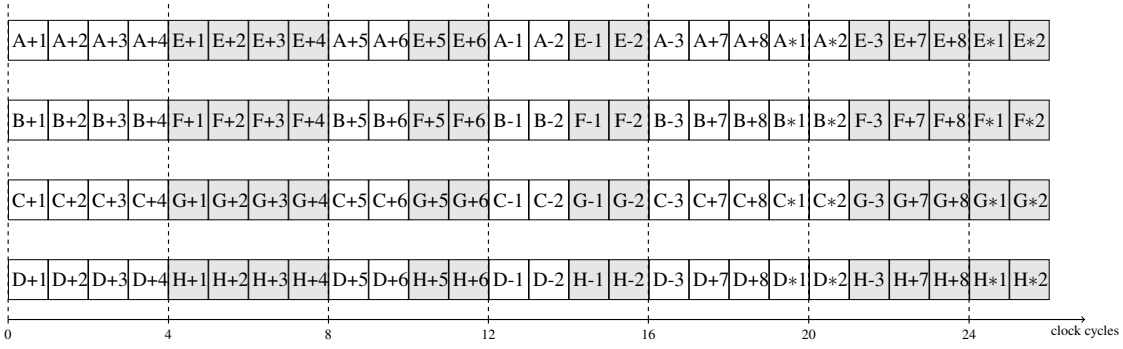


Figure 2.9: Example of loop vectorized then overlapped schedule for a SIMD pipeline

Table 2.1: Throughput and latency for a part of FDCT.

Schedule technique	Throughput (iter./cc)	Latency (cc)
Single SIMD	0.083	12
Single VLIW	0.100	10
Overlapped	0.200	17
Loop vectorized	0.250	16
Loop vectorized & overlapped	0.308	21

2.6 Bluespec System Verilog

In this project, the hardware specification language used is a high level synthesis language called Bluespec System Verilog. BSV is based on a synthesizable subset of System Verilog[5]. To be able to fit into the design exploration loop in figure 2.1, the language had to be chosen with the aspect of parametrization in mind. BSV with its general type parameterization (polymorphism) makes it so that parts of the design can be reused.

In BSV, hardware is made out of modules. A module consists of a state, rules, interfaces and there can be submodules inside a module. Figure 2.10 shows an overview over an module.

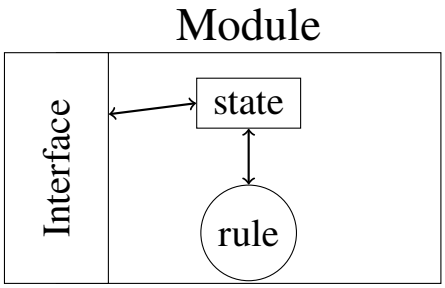


Figure 2.10: Overview of a module in Bluespec

Rules are what determines the behavior in BSV. These rules are made of a condition that determines when the rule will fire and a body with a set of actions which describe the state change. Interfaces are made of methods that describe the transactions between a module and outside circuitry. Methods contain implicit ready and enable signals that determine if the method will cause a state change or if values will be return to the caller.

Listing 2.1: FIFO interface from BSV library[4]

```
interface FIFO#(type any_t);
        method   any_t                  first() ;
        method   Action                 clear() ;
        method   Action                 enq( any_t data_in ) ;
        method   ActionValue#(any_t)    deq() ;
endinterface: FIFO
```

Listing 2.1 shows an interface of a FIFO module in. The FIFO can be initiated with any type (any_t). The interface has four methods: $first$ to get the first element in the FIFO, $clear$ to clear the fifo, enq to enqueue data of type any_t and deq to dequeue the first element. These methods create both the in/out ports and the control signals. There is an overview of these ports and signals in appendix C, figure C.1. The variable passed into the Action method enq creates the input port. The width of the port depends on the width of the type any_t. For example the type Bit#(32) has a width of 32-bit. The $enable$ signals are automatically implemented and trigger each method. The rdy signals indicates when the methods are ready can be called.

Listing 2.2 shows an example of the FIFO module initiated and used in the $mkTest$ module. There is two FIFOs created, $fifo1$ and $fifo2$. Both with integers 32bit types. In rule $r1$, the first element in $fifo1$ is compared and if it is less than 2, the rule will fire. When the rule fires the first element in $fifo1$ is enqeued into $fifo2$. The first element in $fifo1$ is also dequeued.

Listing 2.2: FIFO instantiation with Int32 type

```
module mkFIFOExample
    FIFO#(Int#(32)) fifo1 <- mkFIFO;
    FIFO#(Int#(32)) fifo2 <- mkFIFO;
    ...
    rule r1(fifo1.first() < 2);
        fifo2.enq(fifo1.first());
        fifo1.deq();
    endrule
    ...
endmodule
```

With this high level description the model can be compiled to a Verilog and then synthesized.

Chapter 3
Implementation

This chapter describes the architecture and implementation of our processor. The first section explains the early versions of the architecture and the process of getting to the finalized version. The last section describes the final version.

3.1 Early Versions

When starting out, the first version of the implementation was an single instruction single data processor. The processor had a 3-stage pipeline and the register file only consisted of one BSV register file module. Figure 3.1 shows the overview of the first version. Instructions where generated in a finite state machine that would take some test inputs for example address R0 and address R2 of the register file and operation code for add instruction and enqueue in the FIFO before the register fetch stage. The FIFO has one element making the stages stall until an item is enqueued into the FIFO. One instruction was queued in each state of the FSM. The register fetch would then get the scalar operands from the register file addresses specified and enqueue in the next FIFO before the execute stage. The execute stage had some simple operations implemented such as the add and subtract operations. The add operation would add the scalar operands together and enqueue the result to the FIFO before the write-back stage. The write-back stage would then write to the register file. There is also a function supplied in the BSV simulation environment called *$display*, that could print the result and the result address to the terminal. The output where examined to verify that the model worked as intended.

Starting out with the first version that had a scalar pipeline, the second version was to implement a VLIW pipeline. In order to do this, the register file had to be changed to something that could scale up with the number of added pipelines. The register file module, in BSV, only has support for 5 read ports and 1 write port. A register file module was designed to have register files with 2 read ports for each pipeline in the architecture. The register file module is described in more detail in the final version. Similar to the first

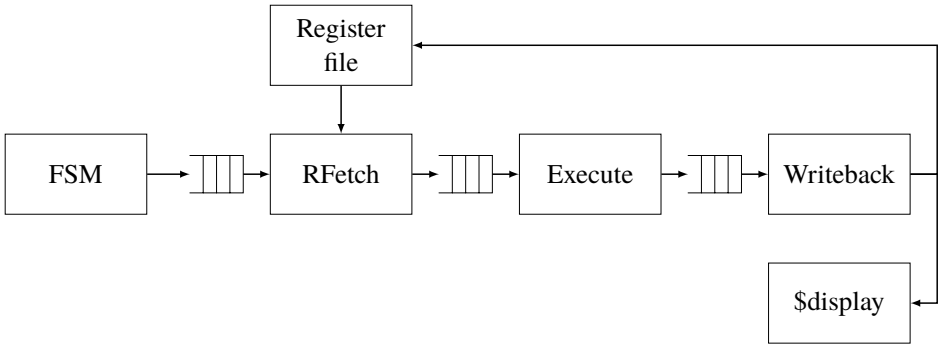


Figure 3.1: Overview of the first architecture

version, this VLIW processor was tested with a FSM, that models the instruction memory. The FSM would send four instructions to the FIFOs before the register fetch modules. The results where printed with *$display* statements to the terminal during simulation. Figure 3.2 shows an overview of this VLIW architecture with four scalar pipelines.

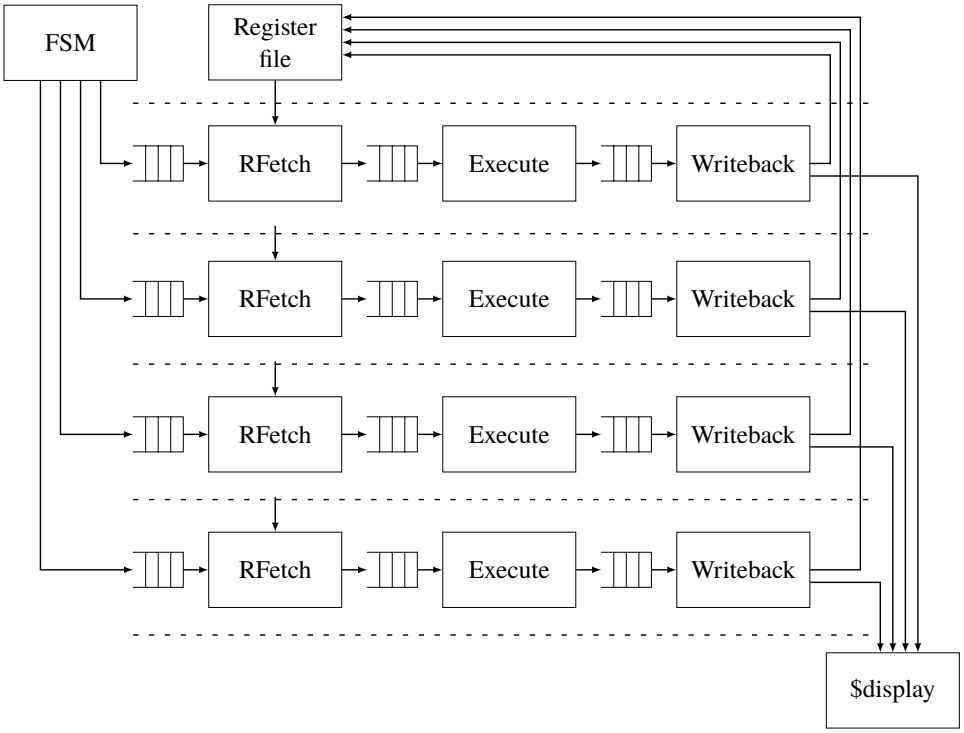


Figure 3.2: Overview of the VLIW architecture

The third version was to implement a SIMD pipeline. In this version, data is packed in an abstract data type called Vector in Bluespec (the vector type is shown in Listing 3.1). The *vsize* defines the length of the vector, and the *element_type* defines the type of the elements in the vector. The element type does not have to be bits or integers but can also be FIFOs or registers. It can also be abstract types like Rules, that will be evaluated during the static elaboration and contain no hardware implementation, which is useful

when implementing the parameters for the models. Vectors start with the first element in the least significant bit to the most significant bit.

Listing 3.1: Vector type definition

```
typedef struct Vector#(type numeric vsize, type element_type);
```

Using this vector type, the SIMD version of the processor has a FSM similar to the scalar version but with the vectors instead of scalars. Since the vector has integers stored in them, it is no problem to store them in FIFOs compared to trying to store some abstract type, that cannot be evaluated into bits. The way the register file is read to the register fetch stage, with vectors in mind, is explained in more detail in the register file section. For the SIMD version of the processor another stage was added in order to be able to permute the operand vectors. This was done in order to simplify any operations where the order of the elements in the operand vectors was different than needed. The order of the operand vectors elements is specified by vectors stored in a memory in the permutation module. Figure 3.3 shows an overview of this SIMD architecture. As with the previous architectures, *$display* statements are used to print the output to the console for testing and debugging purposes.

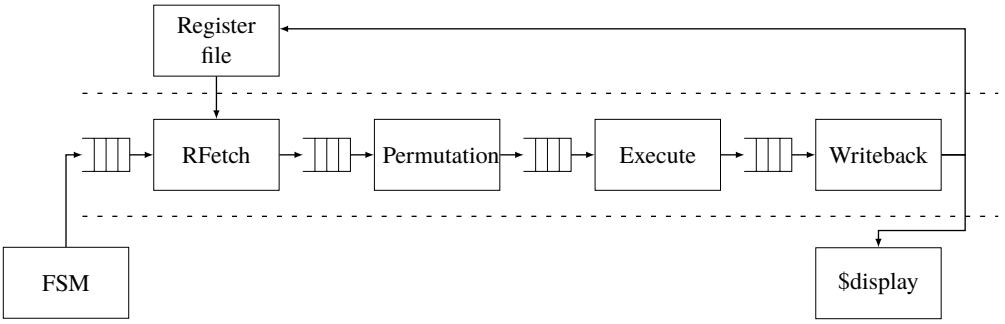


Figure 3.3: Overview of the SIMD architecture

3.2 Final Version

Figure 3.4 shows the pipeline overview for the final architecture. The number of scalar and SIMD pipelines is parameterized. The configurations are reffered to as *(SIMD units, SIMD width, Scalar Units)*. For example a single SIMD pipeline, with a vector width of four, is referred to as a $(1, 4, 0)$ configuration.

The scalar pipeline path and the vector pipeline path both can be added in multiples or removed. The stages are separated by one width FIFOs to achieve pipelining. Reading from empty FIFOs and writing to full FIFOs is not possible due to the implicit signals that Bluespec automatically implements. Both the scalar and SIMD pipelines are connected to a shared register file. Each stage of the scalar pipeline and SIMD pipeline is described first and then the register file module of the architecture.

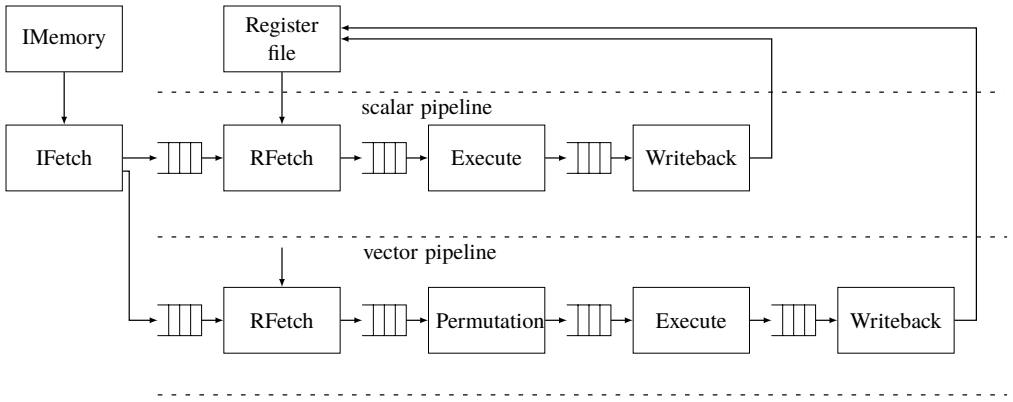


Figure 3.4: Overview of the architecture

3.3 The Pipeline

This section describes each stage of the scalar and SIMD pipelines. The stages are the instruction fetch stage, the register fetch stage, the permutation stage for the SIMD pipeline, the execution stage, and the write-back stage.

3.3.1 Instruction Fetch

The first stage fetches instructions from the instruction memory and puts each instruction into the FIFO for each stage. If there are multiple pipelines, the first set of instructions is going to the SIMD stages and the second set of instructions is going to the scalar pipelines. For example, with a single scalar pipeline and a single SIMD pipeline, the first SIMD instruction will be in the first slot and the first scalar instruction will be in the second slot and so on. If there are more stages than read ports for the register file holding the instructions, additional register files will be created to hold the instructions.

Figure 3.5 and figure 3.6 show the scalar and SIMD instruction format respectively. The length of the operation field depends on the number of operations supported. The number of operations implemented is 8 (*multiplication*, *addition*, *subtraction*, *and*, *or*, *bitwise invert*, *right shift*, *left shift*), so that makes the field 4 bits wide. The width of the

address1, address2 and result address depends on the memory size, as shown later in more detail in the register file module section. The permutation addresses width depends on the memory size of the permutation memory in the permutation stage.

op	address1	address2	result address

Figure 3.5: Scalar instruction format

op	address1	permutation address1	address2	permutation address2	result address

Figure 3.6: SIMD instruction format

3.3.2 Register Fetch Stage

In the scalar pipeline, the register fetch stage takes the specified operand addresses, reads them from the register file module, puts them together with the operation and resulting address and then enqueue them to the next FIFO. The input FIFO is dequeued after the addresses have been read from the input FIFO. The data format enqueued to the output FIFO is shown in Figure 3.7. The operand1 and operand2 width depends on the width of the data stored in the register file. For testing the design the data width was 32bit. The resulting address depend on the depth and width of the register file.

op	operand1	operand2	result address

Figure 3.7: Scalar data format to execute stage.

In the SIMD pipeline, the input FIFO before the register fetch stage is read and then dequeued. Then, using the starting address for both the operand vectors, the operand vectors are read from the register file module. It then enqueues both the operand vectors from the register file to the output FIFO. The data format enqueued to the output FIFO is shown in Figure 3.8. The vector sizes depends on the data width of each element and the number of elements in the vector.

op	vector1	permutation address1	vector2	permutation address2	result address

Figure 3.8: SIMD data format to permutation stage.

3.3.3 Permutation Stage

Different permutation options for the vector are stored in a small memory in this module. This memory content is preloaded from a hex file with the different options for permutation. The size of the memory is determined by the width of the vector and by a parameter

for the depth. The vector data is read from the FIFO and the permutation addresses. The addresses then fetch the permutation specification from the memory and rearrange the vector elements accordingly. The rearranged vector data is then put into the output FIFO. For example if the permutation memory has the order 2,3,1,0 stored in memory for a length 4 vector, the resulting vector would be $v[0] = v[2]$, $v[1] = v[3]$, $v[2] = v[1]$, $v[3] = v[0]$.

The format of the data from the permutation stage is shown in Figure 3.9. The format is the same as from the register fetch stage, but without the permutation addresses which are no longer needed.

op	vector operand1	vector operand2	result address

Figure 3.9: SIMD data format to execution stage.

3.3.4 Execute Stage

In the scalar pipeline the execution stage takes the values acquired from the register file in the previous stage and performs the operation specified. The result and its register address are then passed out to the next stage.The format of this data is shown in 3.10. The result has the same width as the operands have. The instructions we implemented are *multiplication*, *addition*, *subtraction*, *and*, *or*, *bitwise invert*, *right shift*, *left shift*. These are all integer operations and there is no support for floating point or fixed point values.

result	result address

Figure 3.10: Scalar data format to write-back stage.

For the SIMD execute stage, the operations implemented are the same as for the scalar pipeline, but here the operations on the vectors are carried out element-wise. For example, two vectors $\mathbf{a}$ and $\mathbf{b}$ of length 4, the result after multiplication is $(a_0b_0, a_1b_1, a_2b_2, a_3b_3)$ and the result after addition is $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$. The format written to the output FIFO is the resulting vector and the result address as shown in figure 3.11, where the resulting vector is of the same length and format as the operands.

vector result	result address

Figure 3.11: SIMD data format to write-back stage.

3.3.5 Write-back Stage

In the scalar pipelines, the scalar result and its register address are sent to the register file modules write interface.

Similarly, in the vector write-back stage, the result vector and its address are sent to the register file modules write interface.

3.4 Register File Module

The register file is made from several standard Bluespec library register files. They all have one write and up to five read ports. They are put into modules with two read ports and one write port. These modules are reused depending on how many scalar and SIMD pipelines are specified in the design and the length of the SIMD. For example if there is one SIMD pipeline with a vector length 4 and one scalar pipeline, the number of register file modules created would be 5. Figure 3.12 shows how the register files are connected to the interface of the register file module. SW1 and SW2 switches are made through rules in the BSV module, so that there is only one write method and two read methods used at any given moment for each register file module. If there is a read and write at the same clock cycle in the same address, the read value will be the value in the beginning of the clock cycle instead of the value being written. This way there will not be a conflict when reading and writing values to the same address.

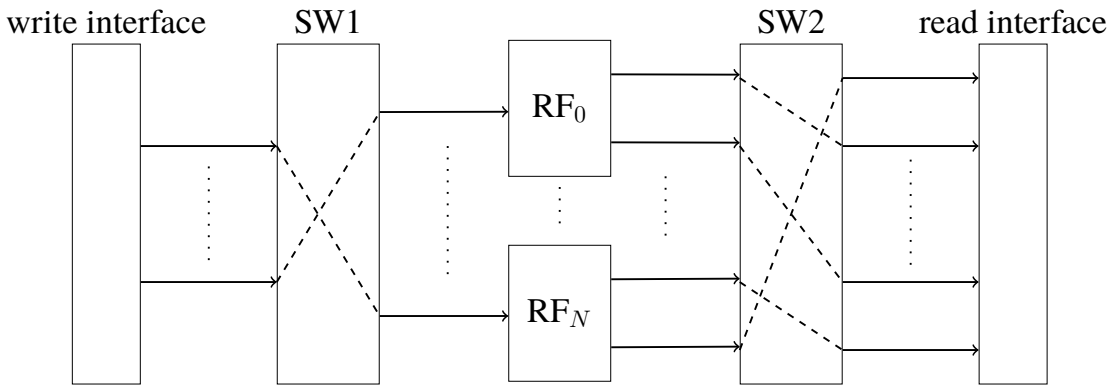


Figure 3.12: Overview of the register file module

The address for the register file module is organized in register file index and slot index. Figure 3.14 shows the register file layout and figure 3.13 shows the register file address format. The size of RF index is given by equation 3.1. It is a unsigned integer with size N depending on the ceiling function of the $\log_2$, where VectorSize is the size of the vectors in the SIMD pipeline, nbrOfSIMD is the number of SIMD pipelines and nbrOfScalar is the number of scalar pipelines. The slot index is chosen by a parameter.

$$\begin{aligned} &\text{RF index} = \text{UInt\#}(N), \text{where } N \text{ is the number of bits} \\ &N = \lceil \log_2 \left((\text{VectorSize} \cdot \text{nbrOfSIMD}) + \text{nbrOfScalar} \right) \rceil \end{aligned} \tag{3.1}$$

Some examples would be, $(1, 4, 0)$ in equation 3.1, gives $N = \lceil \log_2 \left((4 \cdot 1) + 0 \right) \rceil = 2$, so that RF index becomes an unsigned integer of size 2 with a range of [0, 3]. The $(0, 0, 4)$ configuration gives the same result. The $(1, 4, 1)$ configuration has $N = 3$, where RF index is an unsigned integer of size 3 with a range of [0, 7]. This range is larger than the needed one, because the configuration only has 5 register file modules.

The register file module layout enables parallel access for vector or multiple scalars. If the address to be written is to the first register file and it is a vector that is to be written

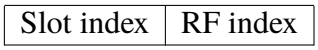


Figure 3.13: Register file address format

or read, then the vector elements will be in the same slot across all the register files. This is shown with the **A** vector in the figure 3.14. If the vector address points to the second register file, the next vector elements will be in the same slot except for the last element, which will move down one slot in the first register file. This is shown with the **B** vector in the figure 3.14. In this way there is only a need for two reads in each register file and when writing the result, only one write port is needed in each register file.

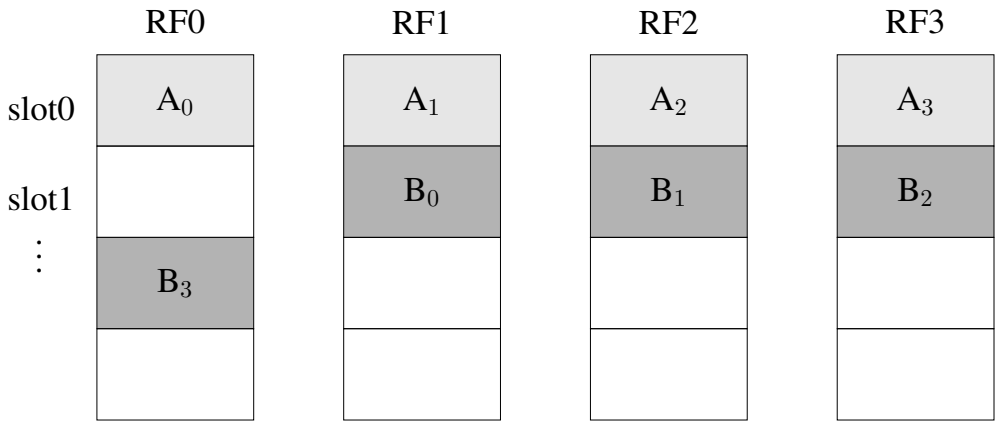


Figure 3.14: Overview of the register file module layout

3.5 Parameterization

The processor has several parameters that can be changed in order to meet the application specific requirements. The following parameters can be changed:

1. Width of the data (Int#(N))
2. Issue-width of the SIMD pipeline (number of SIMD units)
3. Issue-width of the VLIW pipeline (number of scalar units)
4. SIMD width (for all the SIMD units)
5. Register file depth (number of memory locations)
6. Permutation memory depth (number of memory locations)
7. Instruction memory depth (number of memory locations)

The parameters are set in the design before BSV compiles the design to Verilog. The width of the data will affect the range of the numbers that the processors can use. For example, an Int width of 8 has the range of –128 to 127. With the larger range, the memory needed is increased. The FIFOs after the register fetch stage also has to be larger to fit the increase of data. In the execute stage, the operation complexity when turned into hardware is increased with a larger width of the data. For the applications used in this project a width of 32bit is used, since the C applications use 32bit integers.

The issue-width is the number of SIMD or VLIW pipelines the processor is configured for. An increase of the issue-width for both leads to an increase in instructions and instruction memory needed. The instruction fetch stage also has to fanout to more pipelines. Since all of the pipelines are repeated it produces an increasing complexity. The benefit is that the theoretical throughput is higher because of all the instructions that can be executed at the same time. The register file module is increased with a register file for each pipeline and the input and output fanout from the register file module is also increased.

The SIMD width potentially increases the throughput if the full width is utilized. The register file module is increased with a register file for the width of the SIMD as explained in the register file module section. The FIFOs after the register fetch stage also have to be larger, due to the increased width vector. In the permutation stage, the memory for storing permutations will be larger. The execute stage has to increase in complexity with more functional units like add and multiply.

The memory depths of each memory, increases the memory locations that stores the instructions, permutations or data values. The memories will be larger and with FPGAs there often is not a lot of block rams or logic to fit large memories, when you synthesize the design.

Chapter 4
Evaluation

In this chapter, we evaluate three different architecture configurations . The architectures are compared in terms of resource usage on an FPGA, maximum clock speed and speed up compared to a Microblaze processor. The applications and configurations are chosen so they can be compared to the design space exploration in [2].

4.1 Setup

For compiling the Bluespec code to Verilog and for simulation, the 2014.07.A version of the Bluespec complier was used. The Bluespec compiler was run in CentOS 6.7 in VMware 12.0.0 on a Windows 7 Professional 64-bit operative system. To synthesize the design and the Microblaze processor, the Xilinx ISE Design Suite 14.7 was used. The Microblaze was running on a Xilinx Spartan-6 XC6SLX16 FGPA with speed grade -2. The machine running Windows had a Intel(R) Core i7-3820 @ 3.6 GHz processor and 64GB of RAM.

The hardware designs were tested by using three programs hard coded in machine code except with the Microblaze which is using the C code. Two of the programs are from Mediabench suite [11] and are provided by ExPRESS research group in the Electrical & Computer Engineering Department at the UCSB [3]. These programs are the forward discrete cosine transformation (FDCT) used in with JPEG compression and inverse discrete cosine transformation (IDCT) used in decoding MPEG. The FDCT and IDCT programs both operate on a 8x8 matrix. Both the programs traverse the eight rows of the matrix and then the eigth columns. Both of these programs are also used in the design space exploration paper [2]. The third program is a naive matrix multiplication referred to as MATMULT. The source of the C-program matmult may be seen in listing A.1 (appendix A). The matrix multiplication is for two 4x4 matrices. In this program both are the same matrix. Every matrix row element A_{ik}, $i = 1, 2, 3, 4$, is multiplied with the other matrix column element A_{kj}, $j = 1, 2, 3, 4$ and the result is summed over $k = 1, 2, 3, 4$.

The output of the hard coded programs was compared to the C code to verify that they worked correctly. The output was produced running in simulation using *$display* statements to print out the results.

4.2 Experimental Results

The three configurations evaluated are a single SIMD pipeline of width 4, a 4 VLIW pipeline and a 1 SIMD of width 4 with 4 VLIW pipeline design. All of them using 32-bit data, register file depth of 128, permutation memory depth of 16 and instruction memory depth of 4096. The configurations will be referred to as *(SIMD units, SIMD width, Scalar Units)*. The $(1, 4, 0)$ configuration was the configuration chosen from the evaluation in paper [2]. The $(0, 0, 4)$ configuration was not a pareto point in the paper for the IDCT or FDCT but for another application. The $(1, 4, 1)$ configuration was not a pareto point in any of the applications in the paper but was chosen to see if the extra scalar unit would increase the performance.

The configurations were synthesized for a Xilinx Spartan-6. Table 4.1 shows the resource requirements of the different configurations and the Microblaze processor including a 32kB memory, for storing instructions and data. The total usage of the resources of the FPGA is also reported in percentage.

Table 4.1: Resources used with various configurations *(SIMD units, SIMD width, Scalar Units)* and a MicroBlaze processor

Config	Slice Reg	Slice LUT	LUT RAM	DSP48A1	BUFG	BRAM
(1,4,0)	738 (4%)	5904 (64%)	1680 (77%)	12 (38%)	1 (6%)	0
(0,0,4)	522 (2%)	4509 (49%)	704 (32%)	12 (38%)	1 (6%)	27 (62%)
(1,4,1)	848 (4%)	7806 (85%)	880 (40%)	15 (46 %)	1 (6%)	12 (37%)
MB	1512 (8%)	2017 (22%)	140 (6%)	3 (9%)	2 (12%)	16 (50%)

To measure the performance of the different configurations the number of clock cycles were recorded for each applications in the simulation. With the Microblaze processor, it was synthesized with a standard IP called xpstimer to count the clock cycles for each application. The timer was initialized and stopped for the relevant C code in the applications. The C code was optimized in the GCC compiler at level O3 or O2 depending on which was the fastest.

The hard-coded applications were scheduled in three different ways. The single iteration case compared in table 4.2 with optimized C code in the Microblaze. The four iterations overlapped execution shown in 4.3 and the loop vectorized and then two times overlapped case are shown in table 4.4.

The maximum frequency for the different synthesized configurations are shown in table 4.5. The Microblaze is clocked at the frequency of 100Mhz and the default options for speed optimization were chosen when synthesized.

The speedup for the various configurations compared to the Microblaze running at 100Mhz are depicted in figure 4.1 for jpeg FDCT, 4.2 for mpeg IDCT and 4.3 for the matrix multiplier. These figures also show the speedup with the different scheduling techniques applied to the applications. The configurations are assumed to be clocked at their

Table 4.2: Single iteration execution time (clock cycles) for various configurations *(SIMD units, SIMD width, Scalar Units)* and applications.

Configuration	**matmult** (cc)	**jpeg_fdct** (cc)	**mpeg_idct** (cc)
(1,4,0)	52	1073	1437
(0,0,4)	34	769	841
(1,4,1)	50	997	1289
MicroBlaze	336	1341	1521

Table 4.3: Overlapped x4 execution time (clock cycles) for various configurations *(SIMD units, SIMD width, Scalar Units)* and applications.

Configuration	**matmult** (cc)	**jpeg_fdct** (cc)	**mpeg_idct** (cc)
(1,4,0)	48	644	1009
(0,0,4)	32	393	553
(1,4,1)	48	545	873
MicroBlaze	336	1341	1521

Table 4.4: Loop vectorized and overlapped x2 execution time (clock cycles) for various configurations *(SIMD units, SIMD width, Scalar Units)* and applications.

Configuration	**matmult** (cc)	**jpeg_fdct** (cc)	**mpeg_idct** (cc)
(1,4,0)	28	349	391
(0,0,4)	28	238	265
(1,4,1)	28	333	361
MicroBlaze	336	1341	1521

Table 4.5: Maximum clock frequency for various configurations *(SIMD units, SIMD width, Scalar Units)*

1,4,0 (Mhz)	**0,0,4** (Mhz)	**1,4,1** (Mhz)
55.9	52.2	48.9

maximum frequency. The calculated speedup is relative to the speed of the Microblaze according to equation 4.1. Where MB stands for Microblaze, cf stands for clock frequency, cc stands for clock cycles, and config the configuration.

Figure 4.4 shows the FDCT with added vector load and vector store instructions. The dimension for the matrix in FDCT is 8x8, so the added instructions are 16 vector load and 16 vector store, when the vector is of width 4. The dashed line in figure 4.4 is the theoretical maximum speedup, assuming the Microblaze has no stalling in the pipeline. The

maximum speedup is 4 for a vector of length 4 and running at the same clock frequency.

$$\frac{\text{MB}_{cc}}{\left(\frac{\text{MB}_{cf}}{\text{Config}_{cf}}\right) \cdot \text{Config}_{cc}} = \text{speedup} \tag{4.1}$$

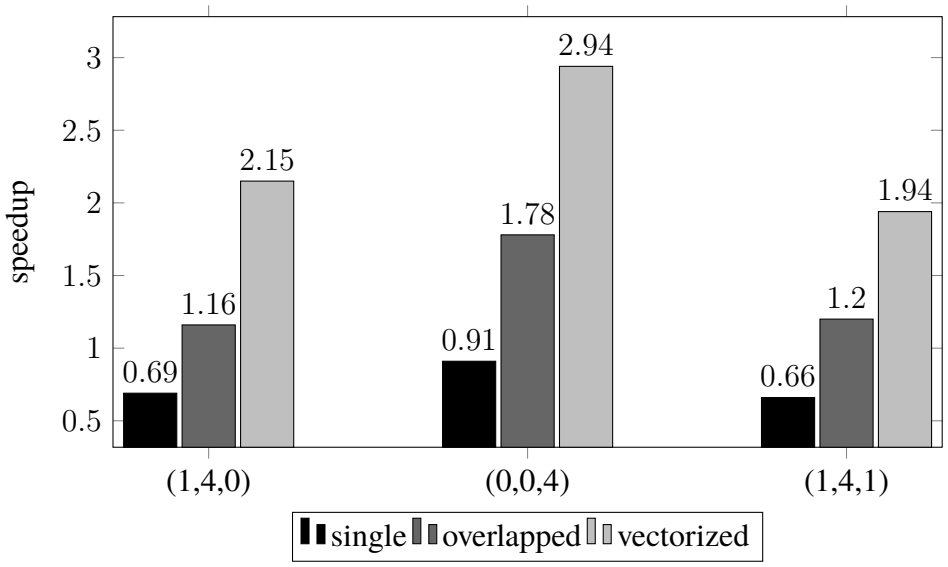


Figure 4.1: Speedup FDCT application compared to MicroBlaze

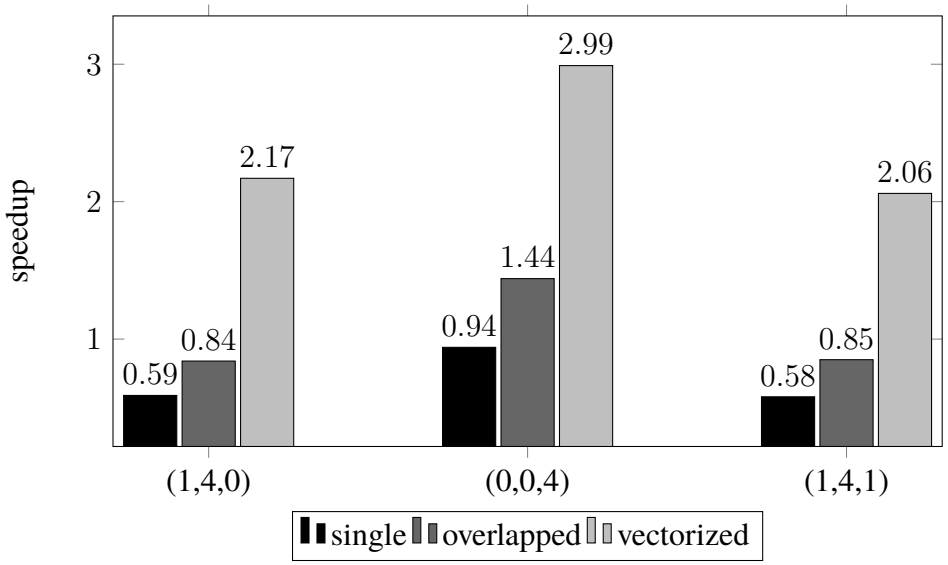


Figure 4.2: Speedup IDCT application compared to MicroBlaze

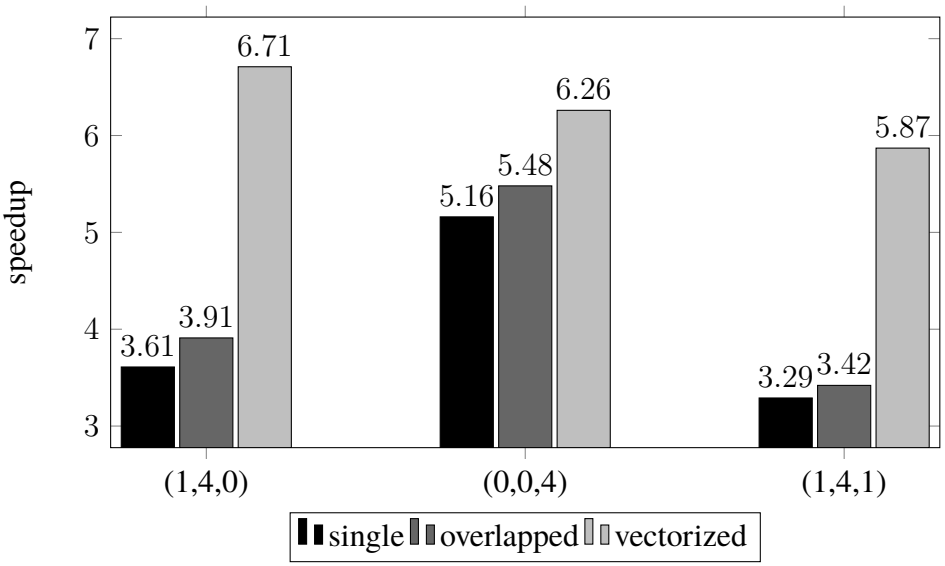


Figure 4.3: Speedup of the matrix multiplier compared to MicroBlaze

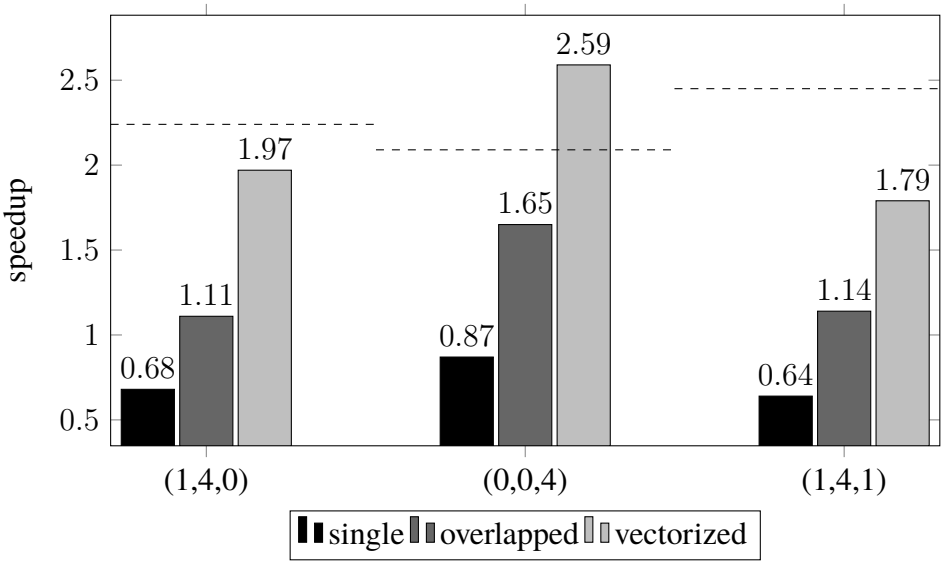


Figure 4.4: Speedup FDCT application compared to MicroBlaze, with added load/store instructions and dashed line for the theoretical limit

Chapter 5
Discussion

In this chapter we discuss the result, in terms of clock cycles, speedup and resources used compared to the Microblaze processor. We also discuss the design decisions in terms of the configurations that were chosen and the different schedule techniques that were used. We also discuss the hardware description language that was used and improvements that can be made in the future.

5.1 Performance

When looking at the performance of the configurations compared to the Microblaze, there are cases when it reaches over the theoretical maximum speedup. For a SIMD unit with width 4, the maximum theoretical speedup is 4, when the systems are at the same clock speed. One of the reasons behind this is that the processor in this project lacks the load and write to memory instructions that the Microblaze has. Instead, here the registers are preloaded with the data that is going to processed. The speedup for the FDCT with added vector load/store still is higher with the $(0, 0, 4)$ configuration, than the theoretical limit. The reason for this could be that the Microblaze has stalling when there is risk for a data dependency hazard in the pipeline, which would decrease the performance of the Microblaze.

Another reason is the branch instructions, that are lacking in our processor. This seems to be one of the reasons for why the matrix multiplier has such a big speedup over the C program. In the hard-coded matrix multiplier application, the three "for loops" are fully unrolled and therefore there is no branch instructions. In theory, the matrix multiplier has 64 multiplications and 48 additions, making it a total of 112 operations. The theoretical best operation for a vector of width 4 would then be 28 operations which is what has been achieved.

When counting the operations in the C files for the FDCT and IDCT the total amount of operations would be 944 and 1048 respectively. Again divided by four, the theoretical

best speed for a vector of width 4 would be 236 and 262 which is close to what is achieved with the loop vectorization and overlapping.

Some performance in the SIMD only configuration $(1, 4, 0)$ was lost due to the vector operations assuming the elements to be packed in consecutive addresses. The elements of the vector have to be aligned, whereas in the VLIW, each pipeline can read two data values from each register file. This was mostly a problem in the single iteration scheduling and overlapped execution, when traversing the columns of the matrices for FDCT and IDCT, since the matrices were stored in row major order. With VLIW $(0, 0, 4)$ it was easier to schedule and utilize all the resources for those cases, since the addresses does not have to be consecutive. This is why the $(0, 0, 4)$ configuration take fewer clock cycles to complete the applications.

There was a small improvement when a scalar pipeline was added to the SIMD only pipeline $(1, 4, 1)$ in terms of clock cycles. This was because of writing data in instances where there were other active data that could be overwritten by the SIMD operation. In some cases this could be avoided in the SIMD pipeline by leaving extra space between rows of data. But when overwriting the columns for the matrices for example, there had to be an extra copy in the $(1, 4, 0)$ configuration, where in the $(1, 4, 1)$ case it could be overwritten directly. Otherwise this configuration did not improve much in speed over the $(1, 4, 0)$ configuration. The 4+1 size makes it for these applications hard to schedule because of the matrices being even and the number of loops also even. Perhaps a 3+1 configuration $(1, 3, 1)$ would be more beneficial. The scalar unit could then be used while data is unaligned for the vector unit and then act as a 4 width vector unit when the data has been aligned. Another case would be for the scalar unit to be used at the same time for an application where the SIMD structure is harder to utilized for example when there is loop dependencies.

The VLIW scalar only case $(0, 0, 4)$ is the overall fastest in terms of speedup except for in the matrix multiplier application. This is because the 4 width SIMD gets a slightly higher maximum clock frequency and they both take the same amount of clock cycles to complete that application.

Because of the lower frequency of the configurations compared to the Microblaze, the single iteration case only managed to get a speedup above 1 in the matrix multiplier application. The loop vectorized technique was the only case when it always outperformed the Microblaze even with lower clock frequency, doubling the performance or more in most applications.

5.2 Resource Usage

The resources used with the different configurations seems to be half or less of the slice registers compared to the Microblaze. This might be because of the logic in the memory access stage, that only the Microblaze has. The Microblaze uses at least half of slice lookup tables that the configurations use. One of the reasons seems to be because some memory is synthesized to LUT RAM instead of BRAM. The DSP48A1 usage is higher because of the extra multipliers in the different configurations. The usage is 4 and 5 times higher so that seems to correspond to the number of multipliers. The BRAM usage is zero on the SIMD only version because the instruction memory in that configurations was synthesized into LUT RAM. Otherwise it would have been similar to the VLIW usage.

5.3 Schedule Techniques

Scheduling more than a single iteration increases the utilization of pipelines and the resources. This is shown in the speedup of the different applications with the overlapping technique. The downside of scheduling more iterations is the increased number of registers needed for the variables in every iteration that is scheduled at the same time. Also when more iterations are scheduled, the input to output time (latency) can potentially increase. If the target application is a streaming application, then the scheduling should take the latency into account so that the buffer holding the streaming data will not be empty or full due to short or long latency of consuming the data.

The loop vectorization proved to be useful in the chosen applications by increasing the utilization of the vector in the SIMD pipeline and the scalars in the VLIW pipeline. This technique would have been harder to use if there was any dependencies between data in the loops.

5.4 Bluespec System Verilog

As a higher level language Bluespec SystemVerilog simplifies a lot by using a higher abstraction layer. For example, the modules have methods that contain all the input and output ports. The methods also automatically implements the handshaking signals and logic for those ports, for example the enqueue for a FIFO module is implicit controlled so that it cannot enqueue when full or dequeue when empty. These implicit signals, that control methods and rules, can make it hard to debug the verilog output after compilation. The critical path for example, is harder to follow when the signals are not named by the designer.

Each stage of the pipeline was also tested separately with test modules that finite state machines to verify that the stage worked properly before attaching to the other stages. A FSM is easy to make because of the $StmtFSM$ module that comes with the BSV library. In appendix D listing D.1 is an example of a FSM with the SmtFSM package. The package is imported from the standard BSV library. Then the FSM is specified within the $seq..endseq$ keyword. Within the seq keyword every line is executed in sequence. The actions between $action..endaction$ only takes one clock cycle. The if statement contains a sequence that will be executed in sequence if the condition is true. There can be if statements inside one $action$ that will take one clock cycle to execute. The FSM is done at the $\$finish$ statement.

The FSM is instantiated in the $mkFSM$ constructor, that has the FSM interface with the start method used in rule $fsmstart$. The start method has an implicit condition that the FSM cannot already be running.

Writing the FSM in Bluespec compared to VHDL (appendix D.2), there is no clock or reset signal specified by the designer. It is handled by the Bluespec compiler. If the combinatorial block in VHDL is incomplete, there could be unwanted inferred latches. In Bluespec, registers are always instantiated by the designer. This is one of the reasons debugging in Bluespec was also a lot easier than in VHDL. In Bluespec you can overload the print function with your own defined types, instead of having to look at wave form signals. For example, if there is user defined instructions like add, subtract. They can be

printed out in the console as “+”, “-” instead of the bit value.

The amount of readily available modules in the Bluespec library also makes it faster to design hardware than in other languages. The fact that they are generic and can be used with different types, as with the FIFO module, makes it easy to use.

5.5 Design Space Exploration

The configuration $(1, 4, 1)$ was not one of the pareto points in the DSE paper [2]. Considering it did not perform better than the $(1, 4, 0)$ when it came to the speedup, it is understandable that it was not one of the optimal points.

The configuration $(0, 0, 4)$ was also not one of the pareto points in the paper, even though it overall performed the best in this project. This could be explained by the model not having any storage constraints. For example the $(1, 4, 0)$ configuration had to align the vector data where the $(0, 0, 4)$ configuration was more flexible.

The applications in the paper was scheduled with modulo scheduling and with a different latency on the pipelines. Therefore, it is hard to compare the evaluated results with that paper, but they have one thing in common, that with the $(1, 4, 0)$ configuration has a higher throughput in the FDCT compared to IDCT application, when scheduled with single iteration or overlapping execution.

5.6 Future Work and Improvements

A lot of time in the project was spend manually programming the different configurations for the architecture. One big improvement would be to make a compiler for this model, that can try out different kinds of schedule techniques and configurations. With the SIMD pipeline there were also a lot of time spend on packing data elements of interleaved data in the vector. This was because the vector operations assume the elements of the vector to appear in consecutive order in the register file module. Loop vectorization made this easier and it would good if the compiler would have support for auto-vectorization, since the applications in this project benefited greatly from it.

Since the configurations where only tested through simulation of Verilog files, there should be some verification done on actual hardware. Since this could be time consuming there was not enough time left on the project to do this.

This design lacks a memory access stage in the pipelines that would take the load and store instructions that were discussed earlier. This stage should be added so that different data can be loaded to the register file and so that the speed measurements would be more accurate compared to a general-purpose CPU like the Microblaze.

Since the design has around half the clock frequency of the Microblaze, some work could be done to minimize the critical path of the design. The critical path seems to be the highest around either the instruction memory or the register file module. Some more work into critical path analyzing could be done in future work.

Chapter 6
Conclusion

A synthesizable model has been made that could support the design space exploration for custom processor architectures. The model was evaluated with three potential architectures and applications. The scheduling techniques of the applications turned out to be crucial in terms of speedup compared to the Microblaze processor. Even with a lower clock frequency, the loop vectorized overlapped schedule had speedup of at least 1.79 times the Microblaze. The single iteration schedule did provide a speedup with the lower clock frequency compared to Microblaze. The execution time in clock cycles for the single iteration schedule was still lower than on the Microblaze. The performance of the custom architectures comes at the cost of an increase in resource usage overall. With the overlapped schedule techniques there is a need for more registers to hold the active data.

The configuration $(0, 0, 4)$ was overall the configuration that performed the best in this project. The configuration $(1, 4, 0)$ had a longer execution time, because the vector elements have to be aligned in memory. The $(1, 4, 1)$ configuration was the worst configuration, because of the lower maximum clock frequency and because of how hard to schedule because of its odd size.

The use of Bluespec SystemVerilog made the process of designing the model much easier due to the higher abstraction layer. The user defined types, overloading of functions and extensive standard library helped when debugging the model and sped up the design process. The explicit register instantiation also helped by not having to find inferred latches when debugging.

This model lacks a few features such as a memory access stage that would provide load and store instructions and the software support in terms of a compiler. Adding the load store instructions would make the comparison to the Microblaze processor more accurate and the addition of a compiler would make the programming of the custom architectures less time consuming. A compiler would also increase the speed of the design space exploration that this model fits into. The need to explore different scheduling techniques and different type of architecture candidates is slowed down by manually writing code.

Bibliography

[1] Mehmet Ali Arslan, Flavius Gruian, and Krzysztof Kuchcinski. A comparative study of scheduling techniques for multimedia applications on SIMD pipelines. In *DATE Friday Workshop on Heterogeneous Architectures and Design Methods for Embedded Image Systems (HIS 2015)*, pages 3–9, March 2015.

[2] Mehmet Ali Arslan, Flavius Gruian, and Krzysztof Kuchcinski. Application-set driven exploration for custom processor architectures. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 70–71, July 2015.

[3] Express DFG Benchmarks. ExpressDFG website. `http://express.ece.ucsb.edu/benchmark/`, 2005. [Online; accessed 07-April-2016].

[4] Bluespec, Inc. FIFO Interface Example. `http://wiki.bluespec.com/Home/Interfaces/FIFO-Interface-Example/`, 2016. [Online; accessed 18-April-2016].

[5] Bluespec, Inc. `http://www.bluespec.com`, 2016.

[6] Niket K. Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiel, Sandeep Navada, Hashem H. Najaf-abadi, and Eric Rotenberg. Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template. *SIGARCH Comput. Archit. News*, 39(3):11–22, June 2011.

[7] Nirav Hemant Dave. Designing a Processor in Bluespec. Master's thesis, MIT, Cambridge, MA, January 2005.

[8] Flavius Gruian and Mark Westmijze. Bluejep: A flexible and high-performance java embedded processor. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '07, pages 222–229, New York, NY, USA, 2007. ACM.

[9] Flavius Gruian and Mark Westmijze. Vhdl vs. bluespec system verilog: a case study on a java embedded architecture. In *SAC '08: Proceedings of the 2008 ACM Symposium on Applied computing*, pages 1492–1497, New York, NY, USA, 2008. ACM.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[11] Chunho Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 330–335, Dec 1997.

[12] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[13] Venkata G. Puppala. VLIW - SIMD processor based scalable architecure for parallel classifier node computing. In *Advance Computing Conference (IACC), 2013 IEEE 3rd International*, pages 1496–1502, Feb 2013.

[14] Roël Seedorf, Fakhar Anjam, Anthony Brandon, and Stephan Wong. Design of a pipelined and parameterized vliw processor: ρ-vex v. 2.0. *HiPEAC Workshop on Reconfigurable Computing (WRC)*, 2012.

[15] Aaron Severance and Guy G. F. Lemieux. Venice: A compact vector processor for fpga applications. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 245–245, April 2012.

[16] Aaron Severance and Guy G. F. Lemieux. Embedded supercomputing in fpgas with the vectorblox mxp matrix processor. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 6:1–6:10, Piscataway, NJ, USA, 2013. IEEE Press.

[17] Xilinx, Inc. `http://www.xilinx.com`, 2016.

[18] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Vespa: Portable, scalable, and flexible fpga-based vector processors. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 61–70, New York, NY, USA, 2008. ACM.

Appendices

Appendix A
Matrix Multiplier Application in C

Listing A.1: Matrix multiplier application in C

```
1  #include <stdio.h>
2
3  #define LEN 4
4
5  int main(int argc, char *argv[])
6  {
7
8      int a[LEN][LEN] = {{1,2,3,4},{2,3,4,5},{3,4,5,6},{4,5,6,7}};
9      int res[LEN][LEN] = {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
10
11     for(int i = 0; i<LEN; ++i) {
12         for(int j = 0; j<LEN; ++j) {
13             for(int k = 0; k<LEN; ++k) {
14                 res[i][j] += a[i][k] * a[k][j];
15             }
16         }
17     }
18
19     // print out the resulting matrix
20     for(int i = 0; i<LEN; ++i) {
21         for(int j = 0; j<LEN; ++j) {
22             printf("%d ",res[i][j]);
23             if(j==LEN-1)
24                 printf("\n");
25         }
26     }
27     return 0;
28 }
```

Appendix B
DFG for a part of FDCT

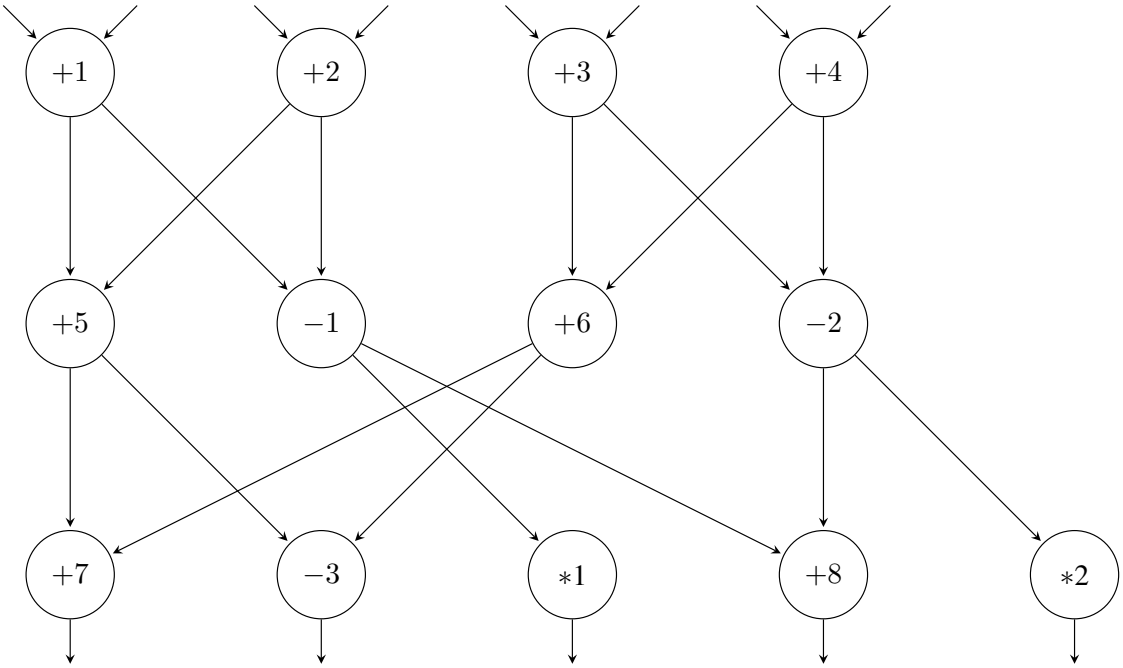


Figure B.1: Dataflow Graph for a Part of FDCT

Appendix C
Bluespec FIFO interface

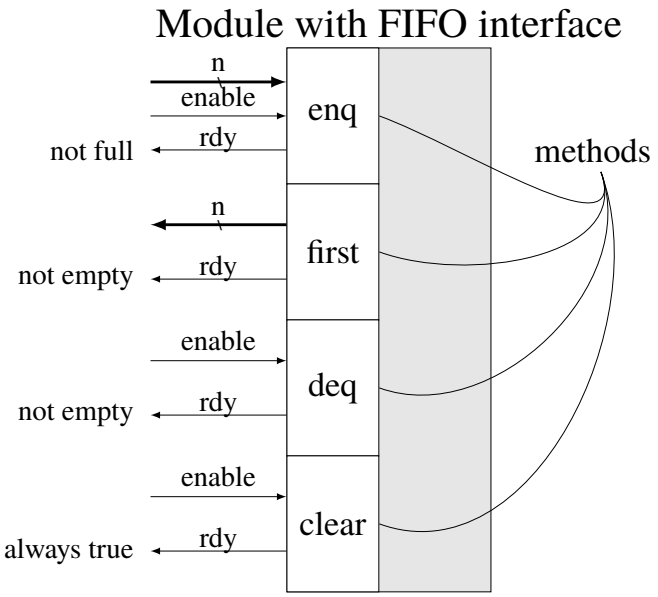


Figure C.1: Overview of a module with a FIFO interface, in Bluespec

Appendix D
Finite State Machine Example

Listing D.1: Stmt FSM in BSV

```
1  import StmtFSM :: *;
2  ...
3  module mkStmtExample
4      ...
5      Stmt test =
6      seq
7          $display("State 1");
8          $display("State 2");
9          $display("State 3");
10         action
11             $display("State 4, parallel actions");
12             $display("State 4, parallel actions");
13         endaction
14         if (condition) seq
15             $display("If-state 1");
16             $display("If-state 2");
17             $display("If-state 3");
18         endseq
19         $finish;
20     endseq;
21
22     FSM fsm1 <- mkFSM(test);
23
24     rule fsmstart;
25         fsm1.start;
26     endrule
27
28 endmodule
```

Listing D.2: FSM in VHDL

```
1  architecture structural of fsm_example is
2    type   state_type is (st1, st2, st3, st4, st5,
3                         ifst1, ifst2, ifst3);
4    signal state, next_state : state_type;
5
6  begin
7
8    combinatorial : process (state, cond)
9    begin
10     case state is
11       when st1 => report "st1";
12                   next_state <= st2;
13       when st2 => report "st2";
14                   next_state <= st3;
15       when st3 => report "st3";
16                   next_state <= st4;
17       when st4 => report "st4";
18                   if condition = '1' then
19                     next_state <= ifst1;
20                   else
21                     next_state <= st5;
22                   end if;
23       when ifst1 => report "ifst1";
24                    next_state <= ifst2;
25       when ifst2 => report "ifst2";
26                    next_state <= ifst3;
27       when ifst3 => report "ifst3";
28                    next_state <= st5;
29       when st5 => finish(0);
30       when others => null;
31     end case;
32   end process;
33
34   sequential : process (clk, rst)
35   begin
36     if rst = '1' then
37       state      <= st1;
38     elsif (clk'event and clk = '1') then
39       state      <= next_state;
40     end if;
41   end process;
42
43 end structural;
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-04-28

EXAMENSARBETE Application Specific Instruction-set Processor Using a Parametrizable multi-SIMD Synthesizeable Model Supporting Design Space Exploration
STUDENT Magnus Hultin
HANDLEDARE Flavius Gruian (LTH)
EXAMINATOR Krzysztof Kuchcinski (LTH)

Parametrisk processor modell för design utforskning

POPULÄRVETENSKAPLIG SAMMANFATTNING **Magnus Hultin**

Applikations-specifika processorer är allt mer vanligt för få ut rätt prestanda med så lite resurser som möjligt. Detta arbete har en parametrisk modell för att kunna testa hur mycket resurser som behövs för en specifik applikation.

För att öka prestandan i dagens processorer finns det vektorenheter och flera kärnor i processorer. Vektorenheten finns till för att kunna utföra en operation på en mängd data samtidigt och flera kärnor gör att man kan utföra fler instruktioner samtidigt. Ofta är processorerna designade för att kunna stödja en mängd olika datorprogram. Detta resulterar i att det blir kompromisser som kan påverka prestandan för vissa program och vara överflödigt för andra. I t.ex. videokameror, mobiltelefoner, medicinsk utrustning, digital kameror och annan inbyggd elektronik, kan man istället använda en processor som saknar vissa funktioner men som istället är mer energieffektiv. Man kan jämföra det med att frakta ett paket med en stor lastbil istället för att använda en mindre bil där samma paketet också skulle få plats.

I mitt examensarbete har jag skrivit en modell som kan användas för att snabbt designa en processor enligt vissa parametrar. Dessa parametrar väljs utifrån vilket eller vilka program man tänkta köra på den. Vissa program kan t.ex. lättare använda flera kärnor och vissa program kan använda korta eller längre vektorenheter för dess data.

För att kunna välja vilken typ av processor som är rätt för den specifika applikationen krävs det ofta att man snabbt kan testa olika prototyper. Att implementera dessa till hårdvara kan ofta vara tidskrävande och ifall det visar sig att implementationen inte klarar dem kraven man ställt för prestanda och energieffektivitet, måste man designa för nya parametrar och mer tid har blivit slösat. Om den här processen istället kan göras automatiskt utifrån dessa designparametrar kan man teoretiskt spara en massa tid. Modellen testades med olika multimedia program. Den mest beräkningsintensiva och mest upprepande delen av programmen användes. Dessa kallas för kärnor av programmen. Kärnorna som användes var ifrån MPEG och JPEG, som används för bildkomprimering och videokomprimering.

Resultatet visar att det finns en prestanda vinst jämfört med generella processorer men att detta också ökar resurserna som behövs. Detta trots att den generella processorn har nästan dubbelt så hög klockfrekvens än dem applikations-specifika processorerna. Resultatet visar också att schemaläggning av instruktionerna i programmen spelar en stor roll för att kunna utnyttja resurserna som finns tillgängliga och därmed öka prestandan. Med den schemaläggningen som utnyttjade resurserna bäst var prestandan minst 79% bättre än den generella processorn.