Vector Processing on FPGAs
Scalability and Performance

By
Micha Nelissen

Supervisors:
Kees van Berkel (TU/e)
Sergei Sawitzki (Philips)

Eindhoven, June 2006
# Contents

1 Introduction
   1.1 FPGAs .................................................. 5
   1.2 Xilinx Virtex-4 architecture .............................. 5
   1.3 FPGA design flow ...................................... 7
   1.4 Vector, parallel processing .............................. 9
   1.5 Global names and constants ............................ 9
   1.6 System overview ..................................... 10
   1.7 Assignment .......................................... 10
   1.8 Report structure ................................... 10

2 Key issues regarding vector processing on an FPGA
   2.1 Shuffle .............................................. 11
       2.1.1 Specification .................................. 11
       2.1.2 Implementation ................................ 12
       2.1.3 Optimization .................................. 12
   2.2 Register file ....................................... 13
   2.3 Clock frequency goal: 200 MHz ........................ 14
   2.4 Conclusion ......................................... 15

3 Instruction Set Architecture ................................ 16
   3.1 Introduction ......................................... 16
   3.2 Data types .......................................... 17
   3.3 State ............................................... 17
   3.4 Memory ............................................ 18
   3.5 I/O ............................................... 18
   3.6 Instruction encoding ................................ 18
   3.7 Load/store unit ..................................... 20
   3.8 External I/O unit .................................... 24
   3.9 Flow control unit ................................... 26
   3.10 MAC unit .......................................... 27
   3.11 Shuffle unit ....................................... 33
       3.11.1 Pattern specification ........................... 33
   3.12 Rotate multiplier broadcast unit ...................... 33
   3.13 Rotate shuffle broadcast unit ........................ 33
   3.14 Assembler .......................................... 34
   3.15 Allowed Parallelism ................................ 34
3.16 Instruction latency ......................................................... 34
3.17 Common concepts left out .................................................. 35
3.18 Conclusion ................................................................. 35

4 Implementation ............................................................... 36
  4.1 General implementation approach ..................................... 36
  4.2 Fetching ................................................................. 39
  4.3 Instruction decoding .................................................... 39
  4.4 Pointer updates ........................................................ 40
  4.5 Execution .............................................................. 41
    4.5.1 Bypasses .......................................................... 41
    4.5.2 Masked bypass ................................................... 42
  4.6 Write back ............................................................. 42
  4.7 Fixed point format ..................................................... 42

5 Optimization ................................................................. 44
  5.1 Instruction decoding ................................................... 44
    5.1.1 Lowering spatial dependencies .................................. 44
    5.1.2 Register fetching ................................................ 45
  5.2 MAC unit ............................................................... 45
  5.3 Shuffle ................................................................. 46
    5.3.1 Multiplexed pipeline ............................................. 46
  5.4 Location constraints .................................................. 46
  5.5 Bypass ................................................................. 50
  5.6 Conclusion ............................................................. 50

6 DSP Algorithms ............................................................ 51
  6.1 FIR filter .............................................................. 51
    6.1.1 Specification ..................................................... 51
    6.1.2 Implementation .................................................. 51
    6.1.3 Conclusion ...................................................... 52
  6.2 FFT ................................................................. 52
    6.2.1 Implementation, P=8 ............................................ 52
    6.2.2 Implementation, P=16 .......................................... 53
    6.2.3 Results .......................................................... 54
    6.2.4 Conclusion ...................................................... 54
  6.3 Conclusion ............................................................. 54

7 Ripple ................................................................. 56
  7.1 Introduction .......................................................... 56
  7.2 Approach 1: P copies ................................................ 58
  7.3 Approach 2: Smart gather instruction ................................ 58
  7.4 Approach 3: Splitting the screen .................................... 59
  7.5 Gather implementation ................................................. 59
    7.5.1 Gather state ..................................................... 59
    7.5.2 Gather related latency ......................................... 60
    7.5.3 Allowed parallelism .............................................. 60
7.5.4 Pipeline with gather ........................................... 60
7.6 Gather collision measurements ................................. 61
7.7 Conclusion .......................................................... 67

8 Conclusion and evaluation ...................................... 68
8.1 Evaluation ............................................................ 69

A Code ......................................................................... 70
A.1 FIR filter code ....................................................... 70
A.2 FFT code ............................................................... 71
A.2.1 Vector width 8 ................................................... 71
A.2.2 Vector width 16 ................................................... 76
A.3 Ripple code ............................................................ 79

List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPF</td>
<td>Vector Processor for an FPGA</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Large Instruction Word</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block</td>
</tr>
<tr>
<td>MUX</td>
<td>Multiplexer</td>
</tr>
<tr>
<td>LUT</td>
<td>Lookup table</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out buffer</td>
</tr>
<tr>
<td>BDC</td>
<td>Broadcast</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>EDIF</td>
<td>Electronic Design Interchange Format</td>
</tr>
<tr>
<td>NGD</td>
<td>Intermediary file format for Xilinx tools</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In this thesis we describe the design and implementation of a VLIW vector processor with a custom chosen instruction set targeted towards DSP applications on an FPGA. Its name will be the VPF, Vector Processor for an FPGA. We will implement a range of DSP algorithms, with FIR and FFT, and also a graphical “Ripple” demo, to explore scalability and performance issues.

1.1 FPGAs

FPGAs are programmable logic devices, containing a two-dimensional grid of logic blocks (lookup tables, multiplexers) and routing channels to connect these in arbitrary fashion. The location of the logic (lookup tables, multiplexers etc.) and the channels is fixed. Logic is grouped into blocks, which have a number of input and output pins, which can be connected to the lanes through the use of “switchboxes” that can connect multiple lanes, and lanes to pins on the logic block. The Xilinx Virtex-4 [7] adds fast arithmetic blocks to the FPGA concept, specifically, allowing fast multiplication, addition, accumulation, counting, and more; up to 500MHz in the fastest speed grade, when fully pipelined.

The main challenge in programming fast FPGAs is to tune the design for routing correctly. All the wires and connections are already in the fabric, so there is only a limited number of combinations possible, so designs with high fanouts have to be avoided. Unfortunately, in naive designs, fanout of some components tends to depend linearly on the vector width of our processor, in particular between the decoding and execution stages.

1.2 Xilinx Virtex-4 architecture

We will now give an overview of the architecture of Virtex-4 FPGAs as an aid in understanding optimization issues.
From a global point of view, an FPGA (see figure 1.1) is built from CLBs (see page 163 of [8]), which are “Configurable Logic Blocks”, and DSPs, that can, in essence, do fast multiplication and addition. A CLB contains four so called “slices”, each having two flip-flops, two 4-input LUTs, two MUXes to build multiplex trees, and two MUXes to build addition chains. A LUT, illustrated in figure 1.2, is the basic building blocks of functionality on an FPGA. It has, in this case, four inputs that via a lookup table determines the output. The lookup table is programmed, or preset, while initializing the FPGA. Two of the four slices in a CLB can only contain logic (called SLICEL), but the other two can also be configured to contain RAM, which can be configured to also function as a FIFO or Shift Register. For more details see the User Guide [8].

To build large multiplexers, the lookup tables (LUTs) are combined with MUX components of which the inputs are directly connected to the output of the LUTs. The first stage of multiplexing will be a 3-input LUT; after this stage MUXes are connected in a special pre-determined pattern to build up to a 32:1 multiplexer using 4 slices with practically zero routing.
delay. In the first stage the selector signal can be a bitwise combination of two signals, by using the full 4 inputs of a LUT, since all LUTs are in fact 4-input LUTs.

Figure 1.3: Switch matrices connect the blocks to lanes to other blocks

Each CLB or DSP is accompanied by a “switch matrix”, see figure 1.3, where signals to and from the logic blocks can be connected to interslice connections. They are layed out in rows and columns. Some interslice connections run in between two adjacent switchboxes, some connect more distant switchboxes. For more information on FPGAs, see [4].

1.3 FPGA design flow

This section describes the work flow of machine design on an FPGA. The previous section explained the architecture of FPGAs, this section will explain concisely, how the tools work.

A typical iteration in the (Xilinx) design flow (also see figure 1.5) uses these tools in order:

1. An editor, to write a VHDL or Verilog file, in which one describes the behaviour of the machine to be implemented

2. A synthesis tool, that produces from the VHDL file an EDIF file, which is a netlist of the individual components that exist on the target FPGA, like LUTs and DSPs; and their connections

3. Ngdbuild, that translates the EDIF format into NGD format, which is the input format for the subsequent Xilinx tools
Figure 1.4: Design flow, VHDL to netlist

Figure 1.5: Xilinx Design Flow; taken from [11], sheet 3
4. Map, maps the components in the netlist into slices; can optionally suggest a placement

5. Place, places the slices on the FPGA in a certain row and column

6. Route, routes the connections mentioned between components, through switchboxes and interslice connections

Note that the last two items are combined in one tool but listed separately for clarity. The last two items are also the hardest part of the flow (for the tools), because the placement of components affects the density of the needed interslice connections, but also vice versa: the available routing opportunities should suggest best placement locations.

1.4 Vector, parallel processing

Vector processing ([1], appendix G) is a form of SIMD[5], where processors have multiple processing elements (PEs), working on vectors of data, instead of on scalars. This introduces parallelism by doing calculations like adding, multiplying, etc. on P elements per time unit, instead of on one. It turns out that DSP algorithms are particularly well “vectorisable”, or can be rewritten in a form where they operate on vectors instead of on scalars, because usually the operations on a stream of data have intrinsic parallelism in one way or the other.

Another dimension of parallelism we will look at are VLIW[3] architectures, where we can feed the separate existing units in a processor an instruction each, also called issue slots. More information about DSP VLIW design can be found in [2].

Our goal will be to have initiation interval 1, meaning that every clock cycle a new operation can be started. On the other hand, we have some freedom with regard to pipeline depth and latency of instructions. The pipeline depth affects the cost of wrongly predicted branches, higher depth increases branch cost, because the calculation pipeline stages will be empty for more time slots. We do not expect many branches in the algorithms we are going to implement, or they will be assisted in assembly. The cost of higher latency is that the compiler and/or programmer will have to interleave instructions with other non-dependent useful instructions, i.e. it will be harder for them to write optimal assembly code. The processor will never stall, so instructions started before the specified latency for a particular instruction will not read the value produced by that particular instruction, but the value produced before.

1.5 Global names and constants

In this document we will use the variable $P$ for the width of the vector processor, that is, the number of elements in a vector processed per clock cycle. The term “broadcast” means that we have a scalar value that is broadcasted vector wide, usually within a unit.
1.6 System overview

Our system, shown in figure 1.6 will consist of algorithms that are implemented in our chosen ISA, running on our vector processor, which in turn is implemented in VHDL, a hardware description language.

![System overview diagram]

Figure 1.6: System overview

1.7 Assignment

Context: recently FPGAs have increased in clock frequency a lot relative to processors, by adding hardware multipliers and adders, such that even calculation intensive applications are implementable at acceptable speed. A vector processor, adding an abstraction layer, has become a feasible challenge. Furthermore, the EVP, a vector processor, has been developed within Philips for various DSP applications.

Assignment: to design, implement, test, and describe a scalable vector (SIMD) machine. This machine will be inspired by the EVP, and if possible, research in the area of gather load will be conducted. We want to know at what clock frequency this machine can run, and how this scales with increase of $P$, and why. Functioning of the processor will be proven by implementing a couple of non-trivial algorithms including the FFT.

1.8 Report structure

We can outline the structure of the rest of this report as follows. In the next chapter we will continue with key differences between implementing a vector processor versus a scalar processor on an FPGA. These issues and their results will give rise to design choices for the ISA in the third chapter. In the fourth chapter we will describe the implementation of this ISA. Optimization of the implementation is discussed in chapter five. In chapter six, we discuss implementation of two DSP algorithms, thereby proving the functioning of the processor. A third, complex algorithm requiring extensions to the processor will be discussed in chapter seven. Finally, in chapter eight we draw general, main conclusions.
Chapter 2

Key issues regarding vector processing on an FPGA

In this chapter we will explore and explain the key issues involved when implementing a vector processor, as opposed to a scalar processor. A key difference is the shuffle unit, to shuffle data within a vector, so that unit will be looked at. We will also look at the register file, since it is a large part of the design, and as such, we will do some research into this area to get a feeling for FPGA possibilities.

2.1 Shuffle

The shuffle operation creates dependencies across the whole vector because from every target position, one can index into an arbitrary source position via the pattern. In contrast, every target position in an addition only has a dependency on that same position in the source vector. As $P$ increases, the shuffle gets wider, requiring more space, increases routing delay, and thus limiting the clock speed. For this reason, the shuffle operation is an area of research.

Now we will describe the operation of the shuffle instruction, we look at implementing it as a separate unit, and any optimizations for (clock) speed.

2.1.1 Specification

The shuffle is an intravector operation, and defined by:

$$ (\forall i : 0 \leq i < P : t[i] := s[p[i]]) $$

where $t$ is the target vector, $s$ the source vector, and $p$ the pattern vector. The shuffle copies
or moves data within a vector. It can do all permutations on a vector, but also copy data from one element throughout the whole vector for example, and any combination of these two types of operations.

2.1.2 Implementation

In essence, the shuffle consists of $16P$ multiplexer trees, each having $\frac{1}{2}P$ LUTs and $P - 1$ MUX components. To analyse the growth of the shuffle unit in relation to $P$, these are the number of LUTs used for 1 bit wide registers, so that each vector is $P$ bits wide (see figure 2.1 for graph):

<table>
<thead>
<tr>
<th>$P$</th>
<th>#LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>144</td>
</tr>
<tr>
<td>32</td>
<td>544</td>
</tr>
<tr>
<td>64</td>
<td>2112</td>
</tr>
</tbody>
</table>

![Graph showing the number of LUTs vs. $P$](image)

Figure 2.1: Shuffle 1 bit wide

2.1.3 Optimization

To speed up the shuffle, we can chop the multiplexing stage in pieces. An $N : 1$ multiplexer has $S = \log N$ stages, so when we chop the shuffle in $M$ and $S - M$ layers, the first stage will have $2^{S-M}$ outputs. For instance when $M$ is half of $S$ (assume $S$ multiple of two) then
there will be $\sqrt{N}$ outputs. This results in $2^{S-M} X : 1$ multiplexers and one $Y : 1$ multiplexer, with $X = \frac{N}{2^{S-M}}$, and $Y = 2^{S-M}$. This will increase the number of flip-flops used, by $2^{S-M}$ multiplied by register width but can result in a fewer number of layers in a cycle and thus decrease the path delay.

In figure 2.2 the scaling of $P$ in relation to number of MHz with various number of stages can be seen.

![Figure 2.2: Shuffle scaling of $P$ in relation to MHz](image)

After $P = 16$, the speed for one stage decreases significantly. This corresponds with our architecture overview notes in section 1.2, as $P = 16$ means that we need to build 33:1 multiplexers for the “1 stage” case, as $1 + 2 \cdot 16 = 33$. The source of the extra “plus one” is the broadcast input.

Adding a stage causes speed to remain at the same level as one stage less when doubling $P$ once. This pattern is broken by 3 stages at $P = 32$ and beyond, due to sheer size of needed logic blocks. Routing delay becomes the dominant factor in speed.

### 2.2 Register file

Now we will look at the register file for a processor. As our goal is a load/store architecture, the register file is of great importance. We want to have 32 registers, each $16P$ bits wide, so it will be complex and large compared to a scalar register file.

The Xilinx Virtex-4 FPGA provides several types of components that might be used to implement a register file. Regular flip-flops is one option, but since some of the LUTs can
also be used as distributed RAM, see [8] page 164, we first tried implementing it using that technology. The advantage is that it stores 16 bits in the space normally occupied by one LUT, and provides multiplexing to access one of these bits. Additionally a dual-port version is available, that allows two read ports and one write port. The 32 registers are then mapped to $2P$ of these distributed RAM components.

![Diagram of register file with multiple read and write ports]

Figure 2.3: Register file with multiple read and write ports

However, as we want to work towards a VLIW design, the register file will need to have multiple read and write ports, which is illustrated in figure 2.3. Multiple read ports can be achieved by copying, but supporting multiple write ports is difficult, as these components have only one write port. Possible solutions are to use double clock rate for these components compared to the rest of the design; however this is hard to implement, and does not scale any further. Another solution might be restrictions of certain units to write to one or the other set of 16 registers, but this is very much contrary to the idea of a register file, and also complicates ISA design: it will be irregular.

Keeping these drawbacks in mind, we choose to implement the register file using flip-flops, where we can arbitrate the write access manually giving rise to multiple write ports. Multiple read ports will result in a greater fanout of each flip-flop.

### 2.3 Clock frequency goal: 200 MHz

We want to achieve a high clock frequency for the processor, therefore we try to set a reasonable goal. The intrinsic, highest clock frequency of a Xilinx Virtex-4 FPGA is 500 MHz. However, we can note that this is for the highest speed grade, the lowest speed grade reaches 400 MHz. Furthermore we can note from our shuffle experiment that even a $P = 4$ shuffle, which is smallest tested shuffle, only reaches 300 MHz. We must conclude that although the DSP blocks can run at a high frequency, the regular LUT logic is not capable of performing useful functionality at such a high frequency. Therefore we halve the clock frequency, add some margin, and as such reach our goal frequency of 200 MHz.
2.4 Conclusion

In this chapter we looked at implementation of the shuffle instruction as a separate unit. We do this because the shuffle instruction implementation size depends on $P$. The amount of needed logic is $O(P^2)$. Speed can be improved by pipelining the instruction, but this requires more flip-flops. From $P = 32$ onward, using 2 stages is to be advised, 3 stages is noise: it yields no gain, for at least upto $P = 64$.

We also looked at the register file implementation where we would prefer to use distributed RAM to keep the number of used slices low, but due to wanting a VLIW ISA architecture, we choose to use flip-flops for the register file implementation.

Using the information we learned during these experiments, we choose to use VLIW in our ISA design, a load/store architecture where the memory to register file path is completely separate from the bypasses of the register files between the other two units. The DSP components allow us to implement a fast MAC unit. Most processor designs have an ALU unit, but we do not expect to need it for the algorithms we want to run, so we do not look into implementing such a unit. The other main vector unit will be the shuffle unit, of which implementation issues have been discussed in this chapter. The next chapter will describe the ISA.
Chapter 3

Instruction Set Architecture

In this chapter we describe the instruction set architecture for the processor. We will describe the choices regarding which instructions were implemented and which ones were not, and why. We describe which data types are supported by the set of instructions, the state of the processor, and the instructions as operations on the state. The vector width is represented using parameter $P$.

3.1 Introduction

We choose not to implement an existing ISA, but instead to design a new one. The reason for this choice is due to our processor being implemented on an FPGA, so we are limited in the type and number of instructions we can map efficiently onto an FPGA design. Some operations are supported by the components present in the FPGA, like hardware multipliers and adders, that we want to use to achieve a high clock frequency, but these do not support saturation and scaling for instance, while some instruction sets may allow saturation or scaling after adding or multiplying. Implementing support for these variants would take down the clock speed. Another reason is our focus on scalability, instead of on particular instruction details.

The processor has a load/store architecture so all arithmetic instructions are on registers, and never on memory directly. There will also not be any memory-to-memory instructions for this reason. The load/store instructions are aligned to simplify implementation. The instruction set is constructed from the point of view of the algorithms we want to implement, and is not intended to be complete. Do not expect support for conditional branches, interrupts, and exceptions.

To access memory the address calculation unit (ACU) is to be used, which has 16 pointers, to calculate pointers that are used to index into memory. Modulo addressing of loading and storing is supported by loading the pointer window size and end. Pointer addresses are vector pointers, in other words, addresses point to $P$ words of data. The processor contains 32
(vector) registers in its register file.

The multiplier broadcast register is used to multiply a vector with a constant. The shuffle register to shuffle a constant into a vector. These broadcast registers are vectors because loading/storing is vector-based.

### 3.2 Data types

In this section we describe the types the processor uses that are exposed through the ISA.

- fixed-point signed 1.15 number: range $-1 \text{ to } 1 - 2^{-15}$
- byte, 8 bits wide
- code address: integer, range 0 to 511
- data address: integer, range 0 to 8191
- external address: integer, range 0 to $2^{24} - 1$
- shuffle pattern: integer with two extra bits at most significant end, for details see 3.11.1.
- hardware loop: a record:
  - count: integer, range 0 to 2047
  - start: code address
  - end: code address

### 3.3 State

The state of the processor will be used to describe the instructions as operations on the state. We will give a name to each part of the state, so that we can refer to it in the instruction specification.

The processor state consists of:

- the program counter, $pc$ of type code address
- 32 vector registers, $R[0]..R[31]$ of type $P$ size vector of type fixed-point
- 16 pointer registers, $P[0]..P[15]$ of type data address
- 16 pointer window size registers, $B[0]..B[15]$ of type data address
- 16 pointer end registers, $E[0]..E[15]$ of type data address
• 16 internal pointer registers, \( I[0].I[15] \) of type data address
• 16 internal pointer window size registers, \( C[0].C[15] \) of type data address
• 16 internal pointer end registers, \( F[0].F[15] \) of type data address
• 16 external pointer registers, \( X[0].X[15] \) of type external address
• 16 pattern registers, \( T[0].T[15] \) of type \( P \) size vector of type shuffle pattern
  
  • a vector accumulator register, \( a[0].a[P - 1] \) of type fixed-point
  
  • a vector multiplier broadcast register, \( vmbc[0].vmbc[P - 1] \) of type fixed-point
  
  • a vector shuffle broadcast register, \( vsbc[0].vsbc[P - 1] \) of type fixed-point
  
  • a color buffer, \( c[0].c[3P] \) of type byte
  
  • a scalar/vector conversion multiplexer control variable, \( s \) with range 0..\( \frac{1}{4}P - 1 \)
  
  • current hardware loop registers, \( hwloop \) of type hardware loop
  
  • hardware loop stack of size 4, \( loopst[0].loopst[3] \) of type hardware loop

### 3.4 Memory

Furthermore we define \( 8192P \) words of data memory, \( mem[0..8191P] \) of type fixed-point. We do not consider this to be part of the processor state, but we will refer to it in the description of some instructions.

### 3.5 I/O

The processor can perform I/O on an external memory unit using the following signals:

• a read/write data bus, \( data[0..7] \) of type byte

• an address line, \( addr \) of type external address, to which the processor writes

• a write enable line, also written to by the processor

### 3.6 Instruction encoding

Due to the processor having a VLIW architecture, instructions have separate, independent parts; furthermore, we have chosen to have one issue slot per unit. In other words, there is
<table>
<thead>
<tr>
<th>LD/ST</th>
<th>EXT</th>
<th>JMP</th>
<th>MAC</th>
<th>SHF</th>
<th>RT S</th>
<th>RT M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.1: Instruction encoding load/store unit**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Ext Ptr Reg</th>
<th>Int Ptr Reg</th>
<th>Buffer opcode</th>
<th>Incr Ptr</th>
<th>Int Ptr Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ext Ptr Immediate</td>
</tr>
</tbody>
</table>

**Figure 3.2: Instruction encoding external communication**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Relative Jump Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Loop start offset</td>
</tr>
<tr>
<td></td>
<td>Loop end offset</td>
</tr>
<tr>
<td></td>
<td>Loop count</td>
</tr>
</tbody>
</table>

**Figure 3.3: Instruction encoding flow control unit**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Dest Reg</th>
<th>Src 1 Reg</th>
<th>Src 2 Reg</th>
</tr>
</thead>
</table>

**Figure 3.4: Instruction encoding MAC unit**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Dest Reg</th>
<th>Source Reg</th>
<th>Pattern Reg</th>
</tr>
</thead>
</table>

**Figure 3.5: Instruction encoding shuffle unit**
a one-to-one mapping from issue slots to VLIW functional units. Figures 3.1 to 3.6 give a  
graphical representation of the instruction encoding.

We use a virtual register $m[0]..m[P]$ for specification purposes, used as means of commu-  
nication by the load/store and mac units.

### 3.7 Load/store unit

The load/store unit has the following instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDV</td>
<td>Load register</td>
</tr>
<tr>
<td>STV</td>
<td>Store register</td>
</tr>
<tr>
<td>LDP</td>
<td>Load pointer (with immediate)</td>
</tr>
<tr>
<td>LDT</td>
<td>Load pattern</td>
</tr>
<tr>
<td>LDM</td>
<td>Load multiplier broadcast vector</td>
</tr>
<tr>
<td>LDS</td>
<td>Load shuffle broadcast vector</td>
</tr>
<tr>
<td>LDB</td>
<td>Load pointer window size</td>
</tr>
<tr>
<td>LDE</td>
<td>Load pointer window end</td>
</tr>
<tr>
<td>LDVM</td>
<td>Load register (modulo pointer operation)</td>
</tr>
<tr>
<td>STVM</td>
<td>Store register (modulo pointer operation)</td>
</tr>
<tr>
<td>LDTM</td>
<td>Load pattern (modulo pointer operation)</td>
</tr>
<tr>
<td>LDMM</td>
<td>Load multiplier broadcast vector (modulo pointer operation)</td>
</tr>
<tr>
<td>LDSM</td>
<td>Load shuffle broadcast vector (modulo pointer operation)</td>
</tr>
<tr>
<td>MOV</td>
<td>Move (copy) register</td>
</tr>
</tbody>
</table>

**LDV**

Format: $\text{LDV Rx, Py, z}$

Formally: $(\forall i: 0 \leq i < P : R[x][i] \leftarrow \text{mem}[P[y] \cdot P + i]); P[y] \leftarrow P[y] + z$

Load a vector from memory at an aligned address contained in pointer register $y$ to vector  
register $x$. Afterwards update the pointer to increase its address by $z$ (5 bits signed). This is  
a latency 1 instruction, and exists only for specification purposes.
STV

Format: STV Rx, P y, z

Formally: $(\forall i: 0 \leq i < P : \text{mem}[P[y] \cdot P + i] \leftarrow R[x][i]); P[y] \leftarrow P[y] + z$

Store the vector in register $x$ to memory at an aligned address pointed to by pointer register $y$. Afterwards update the pointer to increase its address by $z$ (5 bits signed). This is a latency 1 instruction, and exists only for specification purposes.

LDP

Format: LDP Px, y

Formally: $P[x] \leftarrow y$

Load pointer register $x$ with value $y$.

LDT

Format: LDT Tx, P y, z

Formally: $(\forall i: 0 \leq i < P : T[x][i] \leftarrow \text{mem}[P[y] \cdot P + i]); P[y] \leftarrow P[y] + z$

Load shuffle pattern from memory at address contained in pointer register $y$ to pattern register $x$. Afterwards update the pointer to increase its address by $z$ (5 bits signed). For shuffle pattern specification see 3.11.1. This is a latency 1 instruction, and exists only for specification purposes.

LDM

Format: LDM Px, y

Formally: $(\forall i: 0 \leq i < P : \text{vmbc}[i] \leftarrow \text{mem}[P[x] \cdot P + i]); P[x] \leftarrow P[x] + y$

Load multiplier broadcast vector register from memory at address contained in pointer register $x$. Afterwards update the pointer to increase its address by $y$ (5 bits signed). This is a latency 1 instruction, and exists only for specification purposes.

LDS

Format: LDS Px, y
Formally: $(\forall i : 0 \leq i < P : vsbc[i] \leftarrow mem[P[x] \cdot P + i]); P[x] \leftarrow P[x] + y$

Load shuffle broadcast vector register from memory at address contained in pointer register $x$. Afterwards update the pointer to increase its address by $y$ (5 bits signed). This is a latency 1 instruction, and exists only for specification purposes.

**LDB**

Format: LDB Bx, y

Formally: $B[x] \leftarrow y$

Load pointer window size register $x$ with the value $y$.

**LDE**

Format: LDE Ex, y

Formally: $E[x] \leftarrow y$

Load pointer window ending address register $x$ with the value $y$.

**LDVM**

Format: LDVM Rx, Py, z

Formally: $(\forall i : 0 \leq i < P : R[x][i] \leftarrow mem[P[y] \cdot P + i]); P[y] \leftarrow P[y] + z; \text{if } P[y] > E[y] \Rightarrow P[y] \leftarrow P[y] - B[y] \times i$

Load a vector from memory at an aligned address contained in pointer register $y$ to vector register $x$. Afterwards update the pointer to increase its address by $z$ (5 bits signed).

**STVM**

Format: STVM Rx, Py, z

Formally: $(\forall i : 0 \leq i < P : mem[P[y] \cdot P + i] \leftarrow R(x[i])); P[y] \leftarrow P[y] + z; \text{if } P[y] > E[y] \Rightarrow P[y] \leftarrow P[y] - B[y] \times i$

Store the vector in register $x$ to memory at an aligned address pointed to by pointer register $y$. Afterwards update the pointer to increase its address by $z$ (5 bits signed).
LDTM

Format: LDTM T x, Py, z

Formally: $(\forall i : 0 \leq i < P : T[x][i] \leftarrow \text{mem}(P[y] \cdot P + i));\; P[y] \leftarrow P[y] + z;\; \text{if } P[y] > E[y] \Rightarrow P[y] \leftarrow P[y] - B[y]\text{fi}$

Load shuffle pattern from memory at address contained in pointer register $y$ to pattern register $x$. Afterwards update the pointer to increase its address by $z$ (5 bits signed). For shuffle pattern specification see 3.11.1.

LDMM

Format: LDMM P x, y

Formally: $(\forall i : 0 \leq i < P : \text{vmbc}[i] \leftarrow \text{mem}(P[x] \cdot P + i));\; P[x] \leftarrow P[x] + y;\; \text{if } P[x] > E[x] \Rightarrow P[x] \leftarrow P[x] - B[x]\text{fi}$

Load multiplier broadcast vector register from memory at address contained in pointer register $x$. Afterwards update the pointer to increase its address by $y$ (5 bits signed).

LDSM

Format: LDSM P x, y

Formally: $(\forall i : 0 \leq i < P : \text{vsbc}[i] \leftarrow \text{mem}(P[x] \cdot P + i));\; P[x] \leftarrow P[x] + y;\; \text{if } P[x] > E[x] \Rightarrow P[x] \leftarrow P[x] - B[x]\text{fi}$

Load shuffle broadcast vector register from memory at address contained in pointer register $x$. Afterwards update the pointer to increase its address by $y$ (5 bits signed).

MOV

Format: MOV Rx, Ry

Formally: $(\forall i : 0 \leq i < P : R[x][i] \leftarrow R[y][i])$

Copy contents of register $y$ into register $x$. 

23
3.8 External I/O unit

The external communication unit supports the following instructions:

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELI</td>
</tr>
<tr>
<td>EIB</td>
</tr>
<tr>
<td>EIE</td>
</tr>
<tr>
<td>ELX</td>
</tr>
<tr>
<td>ELD</td>
</tr>
<tr>
<td>ELB</td>
</tr>
<tr>
<td>ELC</td>
</tr>
<tr>
<td>EST</td>
</tr>
<tr>
<td>ESB</td>
</tr>
<tr>
<td>ESC</td>
</tr>
</tbody>
</table>

ELI

Format: ELI lx, y

Formally: \( I[x] \leftarrow y \)

Load internal pointer register \( x \) with the value \( y \).

EIB

Format: EIB Cx, y

Formally: \( C[x] \leftarrow y \)

Load internal pointer window size register \( x \) with the value \( y \).

EIE

Format: EIE Fx, y

Formally: \( F[x] \leftarrow y \)

Load internal pointer window ending address register \( x \) with the value \( y \).
ELX

Format: \texttt{ELX Xx, y}

Formally: $X[x] \leftarrow y$

Load external pointer register $x$ with the value $y$.

ELD

Format: \texttt{ELD lx, xy}

Formally: $addr \leftarrow X[y]; (\forall i: 0 \leq i < 4: \text{mem}[I[x] \cdot P + 4s + i] \leftarrow (\text{data}[2i], \text{data}[2i + 1]));

s \leftarrow s + 1 \text{ mod } \frac{1}{4}P$

Load data from external source at address $X[y]$ into internal vector memory at address $I[x]$, at part $s$ of the vector.

ELB

Format: \texttt{ELB x}

Formally: $addr \leftarrow X[x]; (\forall i: 0 \leq i < 8: \text{c}[3P - 8 + i] \leftarrow \text{data}[i]); (\forall i: 0 \leq i < 3P - 8: \text{c}[i] \leftarrow \text{c}[i + 8])$

Load data from external source into the color buffer, and shifts the old data.

ELC

Format: \texttt{ELC lx}

Formally: $(\forall i: 0 \leq i < P: \text{mem}[I[x] \cdot P + i] \leftarrow \text{c}[3i]); (\forall i: 0 \leq i < P: \text{c}[3i] \leftarrow \text{c}[3i + 1]; \text{c}[3i + 1] \leftarrow \text{c}[3i + 2])$

Load one color component from the color buffer into data memory, widening the data from byte to fixed-point width (using zero-padding). Each triplet of bytes is shifted anti-clockwise.

EST

Format: \texttt{EST Xx, ly}

25
Formally: $addr \leftarrow X[x]; (\forall i : 0 \leq i < 4 : (data[2i], data[2i + 1]) \leftarrow mem[I[y] \cdot P + 4s + 3i]); s \leftarrow s + 1 \mod \frac{1}{4}P$

Store data to external memory at address $X[y]$ from internal vector memory at address $I[x]$, from part $s$ of the vector.

**ESB**

Format: **ESB Xx**

Formally: $addr \leftarrow X[x]; (\forall i : 0 \leq i < 8 : data[i] \leftarrow c[i]); (\forall i : 0 \leq i < 3P - 8 : c[i] \Leftarrow c[i + 8])$

Store data to external memory from the color buffer, and shifts the old data.

**ESC**

Format: **ESC Ix**

Formally: $(\forall i : 0 \leq i < P : c[3i] \leftarrow mem[I[x] \cdot P + i]); (\forall i : 0 \leq i < 3P - i : c[3i] \leftarrow c[3i + 1]; c[3i + 1] \leftarrow c[3i + 2])$

Store one color component to the color buffer from data memory, saturating the data fixed-point width to a byte using three bits. Each triplet of bytes is shifted anti-clockwise.

### 3.9 Flow control unit

The flow control unit supports the following instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JR</strong></td>
<td>Jumps to a specified label</td>
</tr>
<tr>
<td><strong>DOI</strong></td>
<td>Specify a hardware loop</td>
</tr>
</tbody>
</table>

**JR**

Format: **JR** label

Formally: $pc \leftarrow pc + 2 + \text{offset(label)}$

Continue execution at instruction specified by label. The actual opcode contains a relative 7-bits offset. There are two branch delay slots after the jump instruction, which is implementation dependent.
DOI

Format: DOI \(x, label\_start, label\_end\)

Formally: push\((\text{loopst}, hw\text{loop})\); \(hw\text{loop} \leftarrow (x-1, pc + label\_start, pc + label\_end)\)

Schedule a hardware loop to be started at instruction specified by label \(label\_start\), ending at and including instruction specified by label \(label\_end\). The loop will execute \(x\) times, \(0 < x\), so \(x - 1\) “repeats” are to be counted. The instruction cannot operate as a jump beyond the end label. The loop start and end addresses cannot be the same instruction, i.e. the loop needs to consist of at least two instructions. The loop start and end addresses must be at least 3 instructions later than the DOI instruction.

3.10 MAC unit

The MAC unit supports the following instructions:

There are only two instruction formats: one for the version of each instruction taking a constant and one for the version taking vectors only. The first we call the “broadcast” version, and are prefixed with a \(B\), the other is prefixed with a \(V\). The accelerator register of the MAC unit operates with higher precision hence rounding is useful. Rounding means that 1 is added to the result in case the most significant bit to the least significant side of the result is 1.

VADD

Format: \text{VADD} Rx, Ry, Rz

Formally: \((\forall i : 0 \leq i < P : a[i] \leftarrow R[y][i] + R[z][i]; R[x][i] \leftarrow a[i])\)

Add the registers \(y\) and \(z\) and store the result in register \(x\).

BADD

Format: \text{BADD} Rx, Ry

Formally: \((\forall i : 0 \leq i < P : a[i] \leftarrow R[y][i] + \text{vmbc}[0]; R[x][i] \leftarrow a[i])\)

Add the register \(y\) and element zero of the multiplier broadcast vector register, and store the result in register \(x\).
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADD</td>
<td>Adds two vectors</td>
</tr>
<tr>
<td>BADD</td>
<td>Adds a constant to a vector</td>
</tr>
<tr>
<td>VSUB</td>
<td>Subtracts a vector from another</td>
</tr>
<tr>
<td>BSUB</td>
<td>Subtracts a constant from a vector</td>
</tr>
<tr>
<td>VMUL</td>
<td>Multiplies two vectors</td>
</tr>
<tr>
<td>BMUL</td>
<td>Multiplies a vector with a constant</td>
</tr>
<tr>
<td>VMAD</td>
<td>Multiplies two vectors and adds another</td>
</tr>
<tr>
<td>BMAD</td>
<td>Multiplies a vector with a constant and adds another vector</td>
</tr>
<tr>
<td>VMSB</td>
<td>Subtracts two multiplied vectors from a vector</td>
</tr>
<tr>
<td>BMADR</td>
<td>Multiplies two vectors, adds another, and rounds the result</td>
</tr>
<tr>
<td>BMADR</td>
<td>Multiplies a vector with a constant, adds another vector, and rounds the result</td>
</tr>
<tr>
<td>VMSBR</td>
<td>Subtracts two multiplied vectors from a vector, and rounds the result</td>
</tr>
<tr>
<td>BMADR</td>
<td>Subtracts a vector that is multiplied by a constant from a vector, and rounds the result</td>
</tr>
<tr>
<td>VMACR</td>
<td>Multiplies two vectors, adds to accumulator, and rounds the result</td>
</tr>
<tr>
<td>BMACR</td>
<td>Multiplies a vector with a constant, adds to accumulator, and rounds the result</td>
</tr>
<tr>
<td>VMDCR</td>
<td>Subtracts two multiplied vectors from the accumulator, and rounds the result</td>
</tr>
<tr>
<td>BMDCR</td>
<td>Subtracts a vector that is multiplied by a constant from the accumulator, and rounds the result</td>
</tr>
</tbody>
</table>
**VSUB**

Format: `VSUB Rx, Ry, Rz`

Formally: $(\forall i : 0 \leq i < P : a[i] \leftarrow R[y][i] - R[z][i]; R[x][i] \leftarrow a[i])$

Subtract register $z$ from register $y$ and store the result in register $x$.

**BSUB**

Format: `BSUB Rx, Ry`

Formally: $(\forall i : 0 \leq i < P : a[i] \leftarrow R[y][i] - vmbc[0]; R[x][i] \leftarrow a[i])$

Subtract element zero of the multiplier broadcast vector register from register $y$, and store the result in register $x$.

**VMUL**

Format: `VMUL Rx, Ry, Rz`

Formally: $(\forall i : 0 \leq i < P : a[i] \leftarrow R[y][i] \times R[z][i]; R[x][i] \leftarrow a[i])$

Multiply the registers $y$ and $z$, and store the result in register $x$.

**BMUL**

Format: `BMUL Rx, Ry`

Formally: $(\forall i : 0 \leq i < P : a[i] \leftarrow R[y][i] \times vmbc[0]; R[x][i] \leftarrow a[i])$

Multiply the register $y$ and element zero of the multiplier broadcast vector register, and store the result in register $x$.

**VMAD**

Format: `VMAD Rx, Ry, Rz, Rm`

Formally: $(\forall i : 0 \leq i < P : a[i] \leftarrow R[m][i] + (R[y][i] \times R[z][i]); R[x][i] \leftarrow a[i])$

Multiply the registers $y$ and $z$, add register $m$ to it, store the final result in the accumulator, and copy that to register $x$. 
BMAD

Format: BMAD Rx, Ry, Rm

Formally: \((\forall i : 0 \leq i < P : a[i] \leftarrow R[m][i] + (R[y][i] * vmbc[0]); R[x][i] \leftarrow a[i])\)

Multiply the register \(y\) and element zero of the multiplier broadcast vector register, add register \(m\) to it, store the final result in the accumulator, and copy that to register \(x\).

VMSB

Format: VMSB Rx, Ry, Rz, Rm

Formally: \((\forall i : 0 \leq i < P : a[i] \leftarrow R[m][i] - (R[y][i] * R[z][i]); R[x][i] \leftarrow a[i])\)

Multiply the registers \(y\) and \(z\), and subtract it from register \(m\). Store the final result in the accumulator and copy it to register \(x\).

BMSB

Format: BMSB Rx, Ry

Formally: \((\forall i : 0 \leq i < P : a[i] \leftarrow R[m][i] - (R[y][i] * vmbc[0]); R[x][i] \leftarrow a[i])\)

Multiply the register \(y\) and element zero of the multiplier broadcast vector register, and subtract it from register \(m\). Store the final result in the accumulator, and copy it to register \(x\).

VMAC

Format: VMAC Rx, Ry, Rz

Formally: \((\forall i : 0 \leq i < P : a[i] \leftarrow a[i] + (R[y][i] * R[z][i]); R[x][i] \leftarrow a[i])\)

Multiply the registers \(y\) and \(z\), and add it to the accumulator. Store the final result in register \(x\).

BMAC

Format: BMAC Rx, Ry

Formally: \((\forall i : 0 \leq i < P : a[i] \leftarrow a[i] + (R[y][i] * vmbc[0]); R[x][i] \leftarrow a[i])\)
Multiply the register $y$ and element zero of the multiplier broadcast vector register, and add it to the accumulator. Store the final result in register $x$.

**VMDC**

Format: `VMDC Rx, Ry, Rz`

Formally: $(\forall i : 0 \leq i < P : a[i] \leftarrow a[i] - (R[y][i] \cdot R[z][i]); R[x][i] \leftarrow a[i])$

Multiply the registers $y$ and $z$, and subtracts it from the accumulator. Store the final result in register $x$.

**BMDC**

Format: `BMDC Rx, Ry`

Formally: $(\forall i : 0 \leq i < P : a[i] \leftarrow a[i] - (R[y][i] \cdot vmbc[0]); R[x][i] \leftarrow a[i])$

Multiply the register $y$ and element zero of the multiplier broadcast vector register, and subtracts it from the accumulator. Store the final result in register $x$.

**VMULR**

Format: `VMULR Rx, Ry, Rz`

Formally: $(\forall i : 0 \leq i < P : a[i] \leftarrow \text{round}(R[y][i] \cdot R[z][i])); R[x][i] \leftarrow a[i])$

Multiply the registers $y$ and $z$, and rounds the result. Store the result in register $x$.

**BMULR**

Format: `BMULR Rx, Ry`

Formally: $(\forall i : 0 \leq i < P : a[i] \leftarrow \text{round}(R[y][i] \cdot vmbc[0])); R[x][i] \leftarrow a[i])$

Multiply the register $y$ and element zero of the multiplier broadcast vector register, and rounds the result. Store the result in register $x$.

**VMSBR**

Format: `VMSBR Rx, Ry, Rz, Rm`
Formally: \( (\forall i : 0 \leq i < P : a[i] \leftarrow \text{round}(R[m][i] - (R[y][i] \times R[z][i])); R[x][i] \leftarrow a[i]) \)

Multiply the registers \( y \) and \( z \), and subtract it from register \( m \). Round and store the final result in the accumulator and copy it to register \( x \).

**BMSBR**

Format: \( \text{BMSBR Rx, Ry} \)

Formally: \( (\forall i : 0 \leq i < P : a[i] \leftarrow \text{round}(R[m][i] - (R[y][i] \times \text{vmbc}[0])); R[x][i] \leftarrow a[i]) \)

Multiply the register \( y \) and element zero of the multiplier broadcast vector register, and subtract it from register \( m \). Round and store the final result in the accumulator, and copy it to register \( x \).

**VMACR**

Format: \( \text{VMACR Rx, Ry, Rz} \)

Formally: \( (\forall i : 0 \leq i < P : a[i] \leftarrow \text{round}(a[i] + (R[y][i] \times R[z][i])); R[x][i] \leftarrow a[i]) \)

Multiply the registers \( y \) and \( z \), add it to the accumulator, and round the accumulator. Store the final result in register \( x \).

**BMACR**

Format: \( \text{BMACR Rx, Ry} \)

Formally: \( (\forall i : 0 \leq i < P : a[i] \leftarrow \text{round}(a[i] + (R[y][i] \times \text{vmbc}[0])); R[x][i] \leftarrow a[i]) \)

Multiply the register \( y \) and element zero of the multiplier broadcast vector register, add it to the accumulator, and round the accumulator. Store the final result in register \( x \).

**VMDCR**

Format: \( \text{VMDCR Rx, Ry, Rz} \)

Formally: \( (\forall i : 0 \leq i < P : a[i] \leftarrow \text{round}(a[i] - (R[y][i] \times R[z][i])); R[x][i] \leftarrow a[i]) \)

Multiply the registers \( y \) and \( z \), substract it from the accumulator, and round the accumulator. Store the final result in register \( x \).
BMDCR

Format: BMDCR Rx, Ry

Formally: \((\forall i: 0 \leq i < P : a[i] \leftarrow \text{round}(a[i] - (R[y][i] \ast vmbc[0])); R[x][i] \leftarrow a[i])\)

Multiply the register \(y\) and element zero of the multiplier broadcast vector register, substract it from the accumulator, and round the accumulator. Store the final result in register \(x\).

3.11 Shuffle unit

There is only one kind of instruction for the shuffle unit, the shuffle instruction:

\[
\text{SHF} \ Rx, \ Ry, \ Tz
\]

It will shuffle the vector contained in register \(y\) according to shuffle pattern in pattern register \(z\), and store the result in register \(x\).

3.11.1 Pattern specification

The shuffle pattern is layed out in a vector. (For an explanation of the shuffle operation see 2.1. In each element in this vector, the least significant \(2 + \log_2 P\) bits are used, say \(p_1, \ldots, p_m\) where \(p_1\) is the most significant bit. The least significant bits, \(p_3, \ldots, p_m\) specify the source position. \(p_2\) specifies whether to copy from element 0 of the shuffle broadcast register, if so, then \(p_3, \ldots, p_m\) ought to be zero, otherwise the result is undefined. \(p_1\) is a mask bit, if enabled, then the value in this position in the target vector is retained, otherwise the corresponding value from the source register is copied.

3.12 Rotate multiplier broadcast unit

Formally: \((\forall i: 0 \leq i < P: vmbc[i] \leftarrow vmbc[i + 1 \mod P])\)

This unit rotates the multiplier broadcast vector register one position backward, and has only one instruction taking no parameters: \(\text{RMB}\).

3.13 Rotate shuffle broadcast unit

Formally: \((\forall i: 0 \leq i < P: vsbc[i] \leftarrow vsbc[i + 1 \mod P])\)

33
This unit rotates the shuffle broadcast vector register one position backward, and has only one instruction taking no parameters: RSB.

### 3.14 Assembler

To facilitate writing algorithms for our vector processor, we implement an assembler that translates instructions in written form to the equivalent opcodes.

Now we will describe the syntax used. Lines that are empty, and the part of a line after a hash (#) sign or double slash (//) token is ignored, and can be used for comments. Instructions for the individual units forming one complete VLIW instruction have to be on one line, and are separated by a semicolon (;). Lines that end with a colon indicate a label name, that can be used in jump instructions. Note that the assembler does not check for duplicate label names.

Output of the assembler will be one instruction per line, encoded in hexadecimal format, 16 characters. Using the command line option “-r”, the output will be formatted in the initialization format for block RAM in VHDL.

### 3.15 Allowed Parallelism

All combinations of above instructions are allowed except the ones listed below:

- the VMAD, VMADR, VMSB, VMSBR, BMAD, BMADR, BMSB, and BMSBR instructions cannot be used together with any of the load/store instructions. When encoding the instructions, the \( m \) register is loaded by means of the register file read port of the load/store unit by use of the LDA instruction whose only parameter is \( R_m \);

- issuing a rotate shuffle broadcast instruction three cycles after a load shuffle broadcast instruction is forbidden;

- issuing a rotate multiplier broadcast instruction three cycles after a load multiplier broadcast instruction is forbidden;

- issuing an external load one cycle after an external store is forbidden.

### 3.16 Instruction latency

We will now list the latencies of instructions that are sharing data (the number of cycles such that the modification of the source instruction is just observed by the destination instruction):
<table>
<thead>
<tr>
<th>Resource</th>
<th>Source</th>
<th>Destination</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer regs</td>
<td>LDP</td>
<td>LDM, LDS, LDV, STV</td>
<td>4</td>
</tr>
<tr>
<td>Ptr range regs</td>
<td>LDB, LDE</td>
<td>LDM, LDS, LDV, STV</td>
<td>2</td>
</tr>
<tr>
<td>Pointer regs</td>
<td>ELI</td>
<td>ELD, EST, ELC, ESC</td>
<td>4</td>
</tr>
<tr>
<td>Ptr range regs</td>
<td>EIB, EIE</td>
<td>ELD, EST, ELC, ESC</td>
<td>2</td>
</tr>
<tr>
<td>Pointer regs</td>
<td>ELX</td>
<td>ELD, EST, ELB, ESB</td>
<td>3</td>
</tr>
<tr>
<td>Data memory</td>
<td>ELD, ELC</td>
<td>LDM, LDS, LDV</td>
<td>1</td>
</tr>
<tr>
<td>Data memory</td>
<td>STV</td>
<td>EST, ESC</td>
<td>1</td>
</tr>
<tr>
<td>Mult bcast reg</td>
<td>LDM</td>
<td>MAC</td>
<td>4</td>
</tr>
<tr>
<td>Mult bcast reg</td>
<td>LDM</td>
<td>RMB</td>
<td>4</td>
</tr>
<tr>
<td>Mult bcast reg</td>
<td>RMB</td>
<td>MAC</td>
<td>1</td>
</tr>
<tr>
<td>Shuffle bcast reg</td>
<td>LDS</td>
<td>RSB</td>
<td>4</td>
</tr>
<tr>
<td>Shuffle bcast reg</td>
<td>LDS</td>
<td>SHF</td>
<td>3</td>
</tr>
<tr>
<td>Shuffle bcast reg</td>
<td>RSB</td>
<td>SHF</td>
<td>0</td>
</tr>
<tr>
<td>Pattern regs</td>
<td>LDT</td>
<td>SHF</td>
<td>3</td>
</tr>
<tr>
<td>Register file</td>
<td>LDV, MOV</td>
<td>all MAC, SHF</td>
<td>4</td>
</tr>
<tr>
<td>Register file</td>
<td>SHF</td>
<td>SHF</td>
<td>1</td>
</tr>
<tr>
<td>Register file</td>
<td>SHF</td>
<td>all MAC</td>
<td>2</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC</td>
<td>VMAC, BMAC, VMDC, BMDC</td>
<td>1</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC</td>
<td>VMACR, BMACR, VMDCR, BMDCR</td>
<td>1</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC</td>
<td>VADD, BADD, VSUB, BSUB</td>
<td>1</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC</td>
<td>VMUL, BMUL</td>
<td>2</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC</td>
<td>VMULR, BMULR</td>
<td>2</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC</td>
<td>VMAD, BMAD, VMSB, BMSB</td>
<td>2</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC</td>
<td>VMADR, BMADR, VMSBR, BMSBR</td>
<td>2</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC</td>
<td>STV</td>
<td>5</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC, SHF</td>
<td>MOV</td>
<td>3</td>
</tr>
<tr>
<td>Register file</td>
<td>all MAC</td>
<td>STV</td>
<td>5</td>
</tr>
<tr>
<td>Register file</td>
<td>SHF</td>
<td>STV</td>
<td>4</td>
</tr>
</tbody>
</table>

### 3.17 Common concepts left out

There is a set of common concepts that almost all ISAs implement or provide. Our ISA includes key elements from vector processing, in particular related to DSP applications. However, we do not provide facilities for interrupts, exceptions, and conditional branches.

### 3.18 Conclusion

In this chapter we described the chosen ISA for our processor. We chose to design our own ISA, so we can focus on the instructions we need for implementation of algorithms, so one should not expect a full generic processor; and due to the nature of FPGAs, we cannot map all instructions as easily as in regular chip design. In the next chapter we will discuss the implementation of this ISA.
Chapter 4

Implementation

In this chapter we discuss the implementation details of the instruction set architecture on
the vector processor. We will look at pipelining, instruction decoding, and bypasses. An
overview of the processor architecture can be seen in figure 4.1.

4.1 General implementation approach

To increase clock speed in general, advantage is taken of the fact that instructions contain
smaller “pieces of work” that can be done in parallel; this is done by using pipelining. It is
used to introduce parallelism in instruction processing, and is explained well in [1], appendix
A, with a good overview in figure A.18.

Implementation choices include no stalling, and related, issuing one instruction per clock cycle.
When a processor stalls, it does not do any useful work, not stalling means the programmer
can choose to perform other calculations. Issuing one instruction per clock cycle means we
will not have paths covering more than one cycle; they will need to be pipelined.

Our processor has 7 stages, see figure 4.2; 1 instruction fetch stage, 3 instruction decoding
stages, 2 execute stages, and a writeback stage. Loading and storing is completely separate
from the other execution paths and units in the processor, so there is no “mem” stage in our
processor.

Shared resources:

- register file, has 4 read ports and 3 write ports
- multiplier broadcast register, has 1 read and 1 write port
- shuffle broadcast register, has 1 read and 1 write port
- data (vector) memory, has 2 read/write ports

36
Figure 4.1: Vector processor architecture
Figure 4.2: Pipeline overview
- external I/O data bus

Resource usage by functional units:

<table>
<thead>
<tr>
<th>Functional unit</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>1 read port on pointer registers</td>
</tr>
<tr>
<td>RF (LD/ST)</td>
<td>1 read port on the register file</td>
</tr>
<tr>
<td>RF (MAC)</td>
<td>2 read ports on the register file</td>
</tr>
<tr>
<td>RF (SHF)</td>
<td>1 read port on the register file</td>
</tr>
<tr>
<td>WB (LD/ST)</td>
<td>1 write port on the register file</td>
</tr>
<tr>
<td>WB (MAC)</td>
<td>1 write port on the register file</td>
</tr>
<tr>
<td>WB (SHF)</td>
<td>1 write port on the register file</td>
</tr>
<tr>
<td>LD/ST</td>
<td>1 read/write port on data memory</td>
</tr>
<tr>
<td>BDC (LD/ST)</td>
<td>1 write port on broadcast registers</td>
</tr>
<tr>
<td>RT M</td>
<td>1 read and 1 write port on multiplier broadcast register</td>
</tr>
<tr>
<td>RT S</td>
<td>1 read and 1 write port on shuffle broadcast register</td>
</tr>
<tr>
<td>Ext LD</td>
<td>1 read/write port on data memory</td>
</tr>
<tr>
<td>Ext LD</td>
<td>external I/O data bus</td>
</tr>
<tr>
<td>Ext ST</td>
<td>external I/O data bus</td>
</tr>
</tbody>
</table>

Using the diagram we can explain parallelism exceptions: issuing an external load instruction directly after an external store instruction is forbidden due to conflict on the shared data bus. Also, one cannot rotate a broadcast register 3 cycles after issuing a load broadcast register instruction, due to sharing one write port on a broadcast register.

The next sections will each discuss one part of the pipeline.

### 4.2 Fetching

The first pipeline stage fetches one instruction per clock cycle from code memory at the address pointed to by $pc$, into an instruction buffer.

### 4.3 Instruction decoding

Instruction decoding consists of three stages. The following paragraphs will each discuss one of them.

The first stage controls the $pc$, which contains the address of the next instruction to be fetched. In case of a jump, the $pc$ is updated with the relative address provided, otherwise the current hardware loop is checked, if any, or else the $pc$ is incremented by one. Due to the $pc$ being written this cycle, and the code memory being a pipelined synchronized read memory block, there are two unconditional branch delay slots. This stage also detects low latency usage of
registers: registers used shortly after they are written to, where bypassing the register file will be needed.

The second stage calculates pointer updates; see next subsection. This stage also determines for each unit whether it will be writing its output to the register file. This information will be used in the next stage to determine from what source unit the register file will be written. Furthermore, this stage fetches for each unit the requested registers from the register file, see also section 5.1.2. It also copies various values vector wide: the operational mode of the MAC unit, and the input multiplexers of the MAC and shuffle units, see also section 5.5.

The register file is implemented in flip-flops due to the requirement of having multiple read and write ports. On the other hand, the pointer registers are implemented in a series of distributed RAM, which each can contain 16 addressable bits. Only one read and write port were required for the pointer registers, so distributed RAM is a good choice for them.

The third stage determines for each register file location what the source unit will be, if it is written to at all. It does this by using the information from the second stage that states for each register whether a particular unit will be writing to it. Also in this the stage the inputs of the executional units are filled, depending on what bypasses are enabled; which is calculated in the first stage and distributed in the second stage.

4.4 Pointer updates

The ACU, address calculation unit, contains a set of pointer registers, that will be used to load data from the memory banks into the register file. Now we will explain the issues encountered when implementing it.

We want to be able to manipulate the pointer registers. We have chosen to only allow them to be loaded with an immediate value, or updated, after loading, with an increment value. We have also introduced the capability for them to be “modulo” pointers, as the EVP[6] has. We define a “window” to be a range of addresses that bound the value of a pointer. So, after incrementing a pointer outside a “window”, the window size will be subtracted from it. While implementing this feature we notice that doing an addition, a comparison, and a substract in one clock cycle results in a long logic path, and it being the critical path. As we want to focus on vector issues, we choose to allow a higher latency such that the incremented pointer value is available in the next cycle, but the modulo “corrected” value, takes two cycles to become available.

Modulo pointers we use in our FIR filter implementation, and we can remark that we only need it every loop of $N$, depth of filter, cycles. A similar things holds for the usage on the ripple demo. One clock cycle higher latency for the modulo operation on the pointer is thus not a problem in this scenario.
4.5 Execution

In this section we describe execution details of the various units with respect to pipelining. Execution consists of two pipeline stages.

The load/store unit gets fed the address to read in the third instruction decode stage, the data will be available in the second execution stage due to using pipeline buffers in the memory component.

The flow control unit operates entirely in the instruction decoding stages, and is described in the instruction decoding section; see above.

The MAC unit executes the requested operation in the first execution stage. In the second execution stage it copies the output value into another set of flip-flops that serve as the input to the register file to decrease spatial dependency between the MAC unit and the register file.

The shuffle unit has a predefined variable number of execution stages: one, two, or three. This means that the writeback stage is shifted backwards, neutral, or forward one stage respectively. Instruction decoding will take into account what number of stages are used, to fit the writeback stage after the variable number of execution stages. As a result, the latency of the shuffle is also variable; it is exactly one more than the number of execution stages chosen.

The rotate units, for the shuffle broadcast and the multiplier broadcast vectors, are rotated in the third instruction decoding stage. Note however that they are loaded in the writeback stage like the register file is, so issuing a load broadcast instruction three instructions before a rotate broadcast instruction for the same unit is forbidden.

4.5.1 Bypasses

Pipelining increases clock speed, because it decreases the amount of sequential work to be done each clock cycle, thus decreasing the critical path delay. However, it introduces latency for instructions operating on the register file, because execution units read from, and write to the register file with several clock cycles in between. Bypasses are used to solve this problem; see also [1] figure A.23. This section will explain the details for our processor.

In general, one bypasses the register file by reading from the appropriate unit at the input location of a unit, instead of using the pipeline buffer for this unit as source, which has the register file as its source usually. However, since routing delay is critical on an FPGA, and bypassing has large routing delay (2 nanoseconds is not unusual), we choose to do the multiplexing in a separate, prior stage.

We choose not the implement bypasses for the load/store unit. Also the rotate broadcast vector units do not need bypasses because they only serve as input to the MAC and shuffle units, and we do not support writing their value into the register file. That leaves bypasses between
the shuffle and MAC unit, which are implemented. However, due to further optimizing to reduce spatial constraints, more bypasses are introduced, see section 5.2.

4.5.2 Masked bypass

Due to the shuffle unit being able to do a partial write to the register file, thus the result register being “masked”, a problem occurs when implementing the bypass straightforwardly. Since only part of the target register is updated, we only have partial data in the bypass too, we do not have the complete result vector.

We make the observation that the unit wanting to read from the bypass will have to do a register file read anyway in case the bypass is not active. Combining this with the fact that the register file or a later bypass will still contain the previous value of the target register for the shuffle operation, making the choice to read from the shuffle bypass conditional by using its output mask will be a correct one.

4.6 Write back

In the write back pipeline stage, the output of the various units is written into the register file. In the instruction decoding stages we determine for each register what the unit will be to be read from, if any. This means that for each register we can have a multiplexer tree instead of linear cascade, which decreases the path delay, because LUTs and MUXes can be tied together efficiently, without any routing delay; a cascade ties LUTs to other LUTs sequentially, requiring interslice connections.

4.7 Fixed point format

Adding two fixed point numbers is the same operation as adding two integers, on bit level. Multiplying two fixed point numbers is different, however, it needs a shift to correct its output. This happens because multiplying two 1.15 fixed point numbers leads to a 2.30 fixed point number, not a 1.31; supposing we are using 32 bit integer arithmetic. Therefore the results needs to be shifted one to the left to get a 1.31 fixed point number, and then we do rounding, saturation, truncation as we wish on the most significant 16 bits.

Note that multiplying the most negative number with itself produces the value 1, which is out of range. Instead the result is again the most negative number. One normally expects saturation to occur, since the result of multiplying two most negative numbers is a number that cannot be represented in this format, and thus the result to be saturated to the nearest value. However, for speed and simplicity we have chosen not to correct for this. We also remark that in most cases, for example when multiplying with a constant, we know this situation cannot occur.
The DSP48 block in the Virtex-4 is an integer unit. For fixed point calculation, we need to connect it in such a way to be able to do both additions and multiplications. However, the DSP48 block does not support shifting the multiplier output one to the left, so we choose to provide operands for the addition operation shifted one bit to the right, and define the accumulator (the “P” register) to be shifted one bit to the right, and connect the signals appropriately. This has no consequences for arithmetic precision, since we only handle this fixed point format in operands.
Chapter 5

Optimization

In this chapter we describe the actions taken to optimize our implementation for clock speed. When implementing the ISA in VHDL, we take care to design our processor such that it is able to achieve possible clock frequency, given prior experience and common knowledge. However, when we go about synthesizing our design, translating it to FPGA technology, we look at the results and see room for improvement. This is what we will call optimization and describe in this chapter: an iterative process, in which we tweak the implementation, synthesize, analyze the results, and start over again.

Optimizing means we need to look at the critical path of the design, and try to make it shorter by pipelining, reducing the load, or other methods depending on how a component contributes to the critical path. We will look at the various functional units separately.

5.1 Instruction decoding

In instruction decoding we transform instructions into operations the functional units can handle. The instruction set specifies operation types, but not necessarily in the format the functional units accept them. Therefore we separate this translation, so that some of it is done in an earlier stage. Particularly important is to minimize routing delay: the next section will discuss practical FPGA spatial dependencies.

5.1.1 Lowering spatial dependencies

Another problem we run into is that the instruction decoding output is contained in a set of flip flops located closely together. However the information in the flip-flops is needed vector-wide, which is spread across the FPGA. As $P$ increases, the load on these flip-flops will increase, and routing will be increasingly difficult. A typical result is that in a post-routing diagram all components have been mapped tightly together. $P$ does not scale further than 4,
otherwise clock speed decreases linearly with $P$.

We solve this problem by the introduction of an extra instruction decode stage in between bypass calculation stage and the stage that performs the bypass. This extra stage multiplies the instruction decode outputs vector-wide, so that those flip-flops can be mapped near to the element in the vector unit where it is used as input. The result post-routing is more space between the components, also when $P$ scales up to 16.

5.1.2 Register fetching

In the last decoding stage, we retrieve values from bypasses or the register file for each unit’s inputs. In this section we will discuss needed optimizations in this area.

Retrieving a value from the register file means a 32:1 MUX, since we have 32 registers. If we then add to this the routing from bypasses, we introduce spatial dependencies between the register file and the pipeline registers we want to bypass from resulting in a large routing delay.

To reduce this routing delay and the spatial dependencies at the same time we need to break up the register fetching and bypass multiplexing. This results in the register fetching being done in the second decoding stage, and then using this value in the bypass multiplexing in the third decoding stage.

Now we face a new problem, which is increased latency, by one, from all register storing to all register fetching instructions. Essentially the problem is we do not have that particular register (the one that has been written to) at that point in time: it is present in the register file, but we will need to wait one cycle before we can read it. We can solve this by copying all units’ outputs one cycle into a next stage, into an extra “writeback” pipeline register. In the first instruction decode stage we add logic to detect usage of the register with exactly this latency, and read from the extra writeback pipeline register if this is the case.

5.2 MAC unit

We will look at how to resolve critical paths related to the MAC unit in this section. The MAC unit output buffer has a high output load (or “fanout”), which is caused by it writing to the register file and via bypasses into the shuffle and MAC units’ input buffers.

As stated above, there is a high load, to be more precise it is the number of registers plus the number of bypasses. The effect is that every vector element unit’s output is too tightly connected with the register file creating routing problems. Observe that this is independent of $P$, since the load is not dependent on $P$.

We solve this by introducing an extra stage in between the output of the MAC unit and the writing to the register file. This causes the load of the output of each MAC unit to drop to
the number of bypasses plus one, and the load caused by the register file is moved to a next stage of flip-flops. This solves the problem well because it breaks the dependency of the MAC units and the register file, the intermediary stage can be placed near to the register file.

Hereby we introduce another problem: the latency of the MAC instruction increases by one. We overcome this problem by adding a bypass from the MAC output to all units inputs; this amounts to only the MAC and shuffle unit, however. As this reintroduces the dependency problem, we use two sets of flip-flops in the intermediary stage: one set that is connected to the register file and another set that is connected to the various bypasses.

5.3 Shuffle

Now we will focus on optimizations applied to the shuffle unit. We started off by implementing the shuffle unit separately; for the results of that work, see appendix 2.1. In this section we will look at integration and optimization issues of the shuffle unit in our processor.

5.3.1 Multiplexed pipeline

Putting the shuffle unit into a vector processor context, we will introduce bypasses like with the other units. For a latency one bypass, we need to implement this bypass within this pipeline stage. We can make the observation that we can implement this as if the shuffle unit were $2P$ wide instead of $P$: with one bit we choose between this pipeline’s stage input register and the pipeline output register.

The advantage of the previous insight is that on FPGA level, we can make one large $2P : 1$ multiplexer, instead of $2P : 1$ multiplexers that are later combined using extra LUTs. Routing delay between LUTs and MUXes is zero, up to 32:1 for Virtex-4 devices, so one big multiplexer tree may result in higher clock frequency than two separate ones.

5.4 Location constraints

The placing of the components generated by the synthesis is normally done completely by the place and route tools. It is sometimes beneficial to guide this process, and now we will look at how we can use it to optimize our processor.

When placing components, we can see the placer uses guidelines and heuristics; the components need to be close together for low routing delay, but if they are placed too close together, the routing becomes more difficult instead, due to proximity to interslice lanes. Our shuffle unit is different to the MAC unit, in that it depends on values vectorwide, while all vector elements in the MAC unit are independent of each other. Looking at post place and routed designs, we see that the shuffle is placed tightly around the MAC units. This “pushes away”
the bypass multiplexer for the input of the MAC unit; to such a degree, that it becomes troublesome, as that bypass multiplexer depends (indirectly, see 5.1.2), on the register file, but also on MAC and shuffle units’ outputs. As the bypass multiplexer for the MAC input is placed further away from the MAC unit, the critical path length increases at twice that rate.

Using location constraints[11], we can force the shuffle unit to be placed at some distance of the MAC unit, allowing the input multiplexer to be placed near to the MAC unit. Location constraints are specified in a file with a ucf extension, and one can specify using wildcards what components must be limited to what region. Example location constraints:

```
INST "id1_instr*" LOC=SLICE_X176Y0:SLICE_X231Y383;
INST "lreg_*" LOC=SLICE_X170Y196:SLICE_X189Y228;
INST "pc_*" LOC=SLICE_X170Y196:SLICE_X189Y228;
INST "id2_*" LOC=SLICE_X116Y0:SLICE_X231Y383;
INST "id3_ptr_*" LOC=SLICE_X116Y0:SLICE_X231Y383;
INST "mem_patdest_*" LOC=SLICE_X186Y0:SLICE_X231Y383;
INST "mem_pat_*" LOC=SLICE_X186Y0:SLICE_X231Y383;
INST "mem_out_*" LOC=SLICE_X186Y0:SLICE_X231Y383;
INST "regfile_*" LOC=SLICE_X28Y0:SLICE_X135Y383;
INST "shuffle_out_*" LOC=SLICE_X24Y0:SLICE_X27Y383;
INST "multbdc_0*" LOC=SLICE_X24Y0:SLICE_X29Y383;
INST "shf1_*" LOC=SLICE_X24Y0:SLICE_X79Y383;
INST "*mem*RAMB16*" LOC=RAMB16_X3Y0:RAMB16_X6Y47;
INST "mac.0.DSP48_inst" LOC=DSP48_X0Y1;
INST "mac.0.mult_*_result" LOC=SLICE_X14Y0:SLICE_X23Y23;
INST "mac_rp1_0*" LOC=SLICE_X28Y0:SLICE_X51Y23;
INST "mac_rp2_0*" LOC=SLICE_X28Y0:SLICE_X51Y23;
```

Here we separate instruction decoding ("idX_*") and the register file from the MAC unit bypass multiplexers ("mac_rpX_0*"). The DSP48 blocks are located between SLICE_X23* and SLICE_X24*. By forcing these pieces of logic to be placed apart, the bypass multiplexers can be placed close to the DSP blocks; which is needed as it is one of the critical paths as we indicated above.

The layout of our processor for \( P = 8 \) without using location constraints is shown in figure 5.1. By contrast, using location constraints on the layout, results in a layout as can be seen in figure 5.2. Additionally, in that figure, the location of some key units is highlighted.

Implementing this method, we confirm that the input multiplexer for the MAC unit is now placed close enough to it, so that it is no longer the critical path. In the case of a particular version of our processor, with \( P = 8 \), without location constraints the design runs at 67.8MHz; using location constraints, it runs at 176MHz. Note that we used normal effort placing and routing, although there are also high effort settings. The higher effort settings may decrease the gap, but they take a lot longer to complete. A fast design iteration cycle is preferred, and location constraints can effectively hint the placer to make significantly better choices.
Figure 5.1: Unconstrained layout result on FPGA for $P = 8$
Figure 5.2: Constrained layout result on FPGA for $P = 8$
5.5 Bypass

The bypass can also be optimized. In the plain implementation, we have a list of flags representing a set of locations we want to bypass from; when a flag is set, we must bypass from that location. We also have a specific order in which we check these flags, as we must bypass from the lowest stage, which is the latest point in time we have written to that register.

Synthesizing this design we get a linear array of multiplexers, usually LUTs, in the reverse order of our list above: last multiplexer checks first flag, if true then read from that bypass location, otherwise read another multiplexer. This implementation is not ideal, the path length is linear in the number of choices to make, while a multiplexer tree in general grows logarithmically to the number of choices. Added to that, as already said, LUTs plus MUXes form an efficient multiplexer tree without intertree delay, up to 32:1 multiplexers.

\[
\begin{align*}
&\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\text{d} \\
\text{e}
\end{array} \\
&\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\text{d} \\
\text{e}
\end{array}
\end{align*}
\]

Figure 5.3: Bypass optimization

To transform this linear if-else-if style into a tree (visualized in figure 5.3), we can draw a tree with number of leaves equal to number of inputs, and place at each leaf an input, sorted according to the list from left to right. The non-leaf nodes will contain a condition each, whether to choose the left branch, which is the disjunction of the conditions for all leaves on the left branch.

5.6 Conclusion

In this chapter we looked at optimizing the implementation outlined in the previous chapter for speed. Most notably, we find that complex paths need to be broken down into simpler paths, for example separating register fetching. Dependencies between units need to be minimized as much as possible, otherwise we get into spatial mapping problems, for example when the units grow bigger as $P$ increases, we need to introduce extra flip-flops in front to spread the load, and an extra stage behind to decrease register file dependencies.

In the next chapter we will look at implementation of some algorithms on our processor to prove its capabilities.
Chapter 6

DSP Algorithms

Several algorithms were implemented to test and show the vector processor’s capabilities.

6.1 FIR filter

In this section we explain the implementation of a FIR filter on the vector processor. Xilinx has examples in [9], chapter 3 to 6.

6.1.1 Specification

The FIR filter is defined by:

\[ y_n = \sum_{i=0}^{N-1} x_{n-i} h_i \]  

(6.1)

where \( y_n \) are the output samples, \( x_n \) are the input samples, \( h_i \) are the coefficients, and \( N \) is the number of coefficients or number of taps.

6.1.2 Implementation

There are several possible implementations of this filter, but most desirable is one where \( N \) and \( P \) are independent. We chose an approach where every \( N \) cycles, \( P \) outputs are generated, also known as "outer-loop parallelism". Every vector element contains the partial output result for that position. As a consequence, in cycle \( i \), \( 0 < i < N \), coefficient \( i \) needs to be broadcasted to all vector element units. Also, the data being loaded into a vector register,
needs to shifted by one position, and have one new data element inserted at the end. This
will be accomplished by the shuffle unit in combination with a shuffle broadcast register.

In every loop iteration, which will consist of \( N \) instructions, we need to do:

- \( \left\lceil \frac{N}{P} \right\rceil \) LDM and LDS instructions
- 1 STV instruction

This means that \( 2 \left\lceil \frac{N}{P} \right\rceil + 1 \leq N \) has to hold, for the algorithm to be schedulable; we keep the
multiplier occupied 100\% of the time.

Regarding \( N \) and \( P \), there are two cases: \( N \) is a multiple of \( P \), or it is not. The first case is
the easiest one; then the loading of the shuffle and broadcast registers and the storing of the
result vector can be scheduled easily \( \frac{N}{P} \) times within the main loop. In the second case, when
\( N \) is not a multiple of \( P \), we choose to make the number of instructions a multiple of \( P \) by
repeating the loop body \( \gcd(P,N) \) times so that we have \( \text{lcm}(P,N) \) instructions.

See appendix A.1, page 70, for the assembly code of an example FIR implementation for
\( N=6, P=4 \). Using our method, the loop body consists of \( \text{lcm}(6,4)=12 \) instructions.

6.1.3 Conclusion

The aim of this section was to implement a FIR filter on our vector processor. We described a
method to implement such a FIR filter with \( N \) and \( P \) independent, using so called outer-loop
parallelism, meaning that every \( N \) cycles we compute \( P \) outputs. Furthermore, our processor
provides instructions in such a way that we can keep the multiplier occupied 100\% of the
time in the main loop. We provided an example implementation for \( N=6, P=4 \) that has
12 instructions in its main loop and therefore provides \( 2P \) outputs every loop iteration.

6.2 FFT

Another algorithm our vector processor can run is an FFT. We base our implementation
largely on the results in [13]. In that application note, a self-sorting 64-point FFT is derived
and implemented on the EVP in approximately 50 cycles. In this section we will look at
adapting it to run on our processor. In particular, due to the EVP having more instructions
and features than our processor, we need to see what is used by the FFT implementation.

6.2.1 Implementation, \( P=8 \)

Our processor does not handle complex numbers in its instruction set. Instructions on complex
numbers on the EVP treat the vector as having 8 16-bit complex numbers, with the real and
imaginary parts alternating within the vector. We will implement complex multiplication and addition “in software” using existing instructions, and will separate the real and complex parts into separate vectors; a complex vector will be a pair of vectors, one containing the real parts and the other containing the imaginary parts. The complex multiplication $z \leftarrow x \cdot y$, where we map $x$ to $(a, b)$, $y$ to $(c, d)$, and $z$ to $(e, f)$ registers, translates into:

\[
\begin{align*}
\text{MUL} & \text{ Re, Ra, Rc} \\
\text{MDC} & \text{ Re, Rb, Rd} \\
\text{MUL} & \text{ Rf, Rb, Rc} \\
\text{MAC} & \text{ Rf, Ra, Rd}
\end{align*}
\]

Furthermore, there is a special shuffle instruction in the EVP for butterfly shuffles with a parameterized butterfly size, see page 352 of [6]. On the EVP the register mask and shuffle pattern are separate, in our processor they are combined into the shuffle pattern. We will load the needed butterfly patterns, with the needed masked variations, from data memory as needed.

We start out with a $P = 8$ FFT implementation, and later create a $P = 16$ implementation. This is due to the fact that on the EVP complex numbers’ real and imaginary parts are stored side by side in one vector, so operations on complex numbers on the EVP, and therefore also in the application note’s algorithm, work on 8 numbers at a time. The EVP algorithm uses $x_\rightarrow$, $w_\rightarrow$, and $v_\rightarrow$ variables, which we map onto registers. In the resulting assembly code we document what variable names are mapped onto what registers.

Another detail to implement is software pipelining. The algorithm basically is a loop, that loads, processes, and stores one 64-length vector. The load instructions have some latency, and processing also has some latency with respect to storing. Note though that as the algorithm is loading the final bits of data we can start processing the first loaded ones; the same holds for the last stage of the loop, where we can start to store in parallel to processing the final bits.

We want to waste as few cycles as possible not processing, so we copy some of the first load instructions that are not accompanied by a processing instruction to the end of the loop, before we start storing data. However, we will overwrite some registers too soon, before they have been processed and stored. We solve this by copying that many processing instructions as well, which introduces gaps for the load/store unit, so that we can complete the cycle and fill the unit with loading and storing instructions.

The resulting assembly code for the $P = 8$ variant is listed in appendix A.2.1.

### 6.2.2 Implementation, $P=16$

Now we will look at modifying the created $P = 8$ implementation into a $P = 16$ one.

As the application note at the end of section 2.1 in [13] says: “[…] a doubling of the vector size
would mean that already in stage 3, intra-vector re-ordering (shuffling), is required”. Since the butterfly pattern grain size is halved each stage, renaming of registers is not sufficient anymore, we need to shuffle the contents. We add the two required extra shuffle patterns to our collection of needed patterns, and start shuffling vectors in stage 3 instead of stage 4.

The resulting assembly code for the $P = 16$ implementation is listed in appendix A.2.2.

### 6.2.3 Results

Number of cycles in main loop:

<table>
<thead>
<tr>
<th>Processor</th>
<th>$P$</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVP</td>
<td>16</td>
<td>49</td>
</tr>
<tr>
<td>VPF</td>
<td>16</td>
<td>88</td>
</tr>
<tr>
<td>VPF</td>
<td>8</td>
<td>176</td>
</tr>
</tbody>
</table>

The reference EVP implementation has 51 cycles, 49 cycles with software pipelining, for the variant without saturation and scaling; which most closely matches our implementation, since our processor does not have saturation or scaling.

### 6.2.4 Conclusion

In this section we looked at implementing a 64-point self-sorting FFT, with help from an application note [13]. We created a $P = 8$ and a $P = 16$ implementation, due to the EVP implementation operating on 8 complex numbers each cycle.

From the results we can see that doubling the vector width, indeed halves the number of cycles for the main loop.

Comparing our implementation to the EVP one, we can see that our implementation needs 39 cycles more, so it is approximately 44 percent slower. This is directly due to the fact that the EVP has a VMAC unit and a VALU unit, while we only have a VMAC unit. As the algorithm consists of mainly butterflies, see figure 9 on page 12 of [13], that have one subtraction, one addition and one multiplication that takes two cycles, most of the additions and subtractions for a vector can be scheduled in parallel with the multiplication of their previous vector, almost halving the number of needed cycles.

### 6.3 Conclusion

In this chapter we looked at implementing two standard DSP algorithms on our processor, a FIR filter and a 64-point self-sorting FFT. We can conclude that both are implementable in an efficient form, although the FFT could use an additional ALU unit like the EVP has, to
parallelize most of the additions and subtractions with the multiply accumulate operations. However, seen the Virtex-4 architecture, fast addition is possible using the DSP blocks, but one will get a multiplication capability along with it, anyway; reducing multiplier efficiency again.

In the next chapter we will discuss another algorithm we looked at, but since that resulted in adding instructions, it will be discussed in a separate chapter.
Chapter 7

Ripple

In this chapter we explain the implementation of a “ripple” demo on the processor. The ripple demo ignited additional research, in particular in the area of smart scatter/gather, and memory collision avoidance.

7.1 Introduction

The ripple demo behaviour can be described as follows. One has a pool containing water, where the floor consists of a picture. The demo then simulates what happens when pebbles are thrown into the pool, with the viewer looking straight from above. The movement of the water will cause the picture to be distorted, and the waves will be shaded so as to make them more visible.

Figure 7.1: Waves cause visual distortion of the image on the “floor”

Distortions are caused by the waves on the surface, see figure 7.1. We model the height, by keeping two buffers of the height of the water for a pixel, one buffer containing the height in the last frame, and one of the frame before [14]. We calculate the new height by using
the second last value as a change velocity. Furthermore, for the distortion vector for a pixel horizontally or vertically, we use the difference between the two neighbour pixels horizontally or vertically, respectively. This results in having a maximum distortion distance, and a search area for every pixel.

Details are listed in the pseudocode below. buffer1 and buffer2 are the height buffers; they have a border of one pixel on all sides, which will contain nothing but zero.

for every position $(x,y)$ in the image:
  
  $\text{buffer2}[x][y] = ((\text{buffer1}[x-1][y-1] + \text{buffer1}[x+1][y-1] + \text{buffer1}[x-1][y] + \text{buffer1}[x+1][y] + \text{buffer1}[x-1][y+1] + \text{buffer1}[x+1][y+1] + \text{buffer1}[x][y-1] + \text{buffer1}[x][y+1]) \div 4) - \text{buffer2}[x][y]$;
  
  $\text{buffer2}[x][y] = \text{buffer2}[x][y] - (\text{buffer2}[x][y] \div \text{damping}$ factor$)$;

for every pixel $(x,y)$ in the image:
  
  $\text{offset}_x = \text{buffer2}[x-1][y] - \text{buffer2}[x+1][y]$;
  
  $\text{offset}_y = \text{buffer2}[x][y-1] - \text{buffer2}[x][y+1]$;
  
  $\text{shading} = (\text{offset}_x + \text{offset}_y) \div \text{scaling factor}$;

  $\text{src}_x = x + \text{offset}_x$;
  
  $\text{src}_y = y + \text{offset}_y$;
  
  $\text{check.bounds}(\text{src}_x, \text{src}_y)$;

  $\text{pix} = \text{texture}[\text{src}_x, \text{src}_y]$;
  
  $\text{pix} = \text{pix} + \text{shading}$;
  
  $\text{plot pixel at } (x,y) \text{ with color } \text{pix}$

$\text{swap } \text{buffer1} \text{ and } \text{buffer2}$

When we vectorise this algorithm we notice that the distortion vectors for a vector of pixels of which the background is to be fetched will collide when using the simple, straightforward pixels-to-bank mapping. There are two more or less independent problems when the current ISA is to be used (illustrated in figure 7.2):

1. we cannot guarantee that all banks will have to fetch data from the same address;

2. a collision means that some banks need to fetch multiple addresses, and other banks none.

We discuss three approaches to solve this collision problem: the first is to make $P$ copies, another to implement a smarter load instruction, the third is avoiding collision using a different vectorization approach. Each will be discussed in its section following.
7.2 Approach 1: $P$ copies

One way to solve the collisions generated by the distortion vectors is to make $P$ copies into memory, so that all banks have all of the pixels, and as such, can read any requested pixel. This solves both identified subproblems in the previous section at once.

The number of cycles needed for the inner loop is \( \frac{(8+3P+12)\text{fps pixels}}{P} \). The copying of the pixels vector-wide will already consume half of the cycles when $P$ is 8, worsening as $P$ gets larger.

We can conclude using Amdahl’s law that this approach will not provide significant vector speedup.

7.3 Approach 2: Smart gather instruction

Now we will describe an alternative to solve the collision problem outlined in the introduction. We add an instruction that can specify a different address per bank, commonly called “gather”. This solves the first subproblem of the introduction section. To solve the second, we smarten the instruction to iteratively fetch addresses that do not conflict. That means it figures out every cycle what addresses can be fetched that have not been fetched yet, and fetches those. Additionally it constructs a shuffle pattern to move the data from a bank to the correct “lane”, the position in the vector that requested this data. Section 7.5 will describe the added instructions in detail.
7.4 Approach 3: Splitting the screen

A third approach that might be used is splitting the screen in \( P \) areas, and processing one pixel per area per clock cycle. An advantage is that we no longer have collisions for competing memory banks, since the distance between the pixels being processed in two areas are further apart than the search areas for those pixels. Disadvantages are that we will need more memory on the FPGA, as the search area for a pixel is not shared with adjacent pixels, and another is that the memory layout is different from the usual cyclic mapping onto vectors: the first \( P \) pixels of the image for instance are all needed in the first area, so need to be loaded into the first bank, not distributed across the banks. This can be done by loading \( P \) lines, and then using scatter/gather to move the pixels into the desired locations.

This approach is promising performance wise, but we did not look into actually implementing it due to its memory requirements being unreasonably high for the Virtex-4 FPGA.

7.5 Gather implementation

We want to implement approach 2, therefore we add the smart gather instructions and will now specify the details of those. Besides the smart gather instruction, we add another one: a wait instruction, that will stall the processor until the gathering has been completed. Since the number of collisions is unknown in advance, we can wait for the completion this way, and avoid needing to insert \( P \) NOP instructions to wait for the worst case, which is \( P \) accesses in the same bank. Using both instructions will reduce the above stated \( 3P \) to \( 3x \), where \( x \) is the average number of needed fetches.

The next subsections will discuss the instructions in detail.

7.5.1 Gather state

We extend the state with a boolean \( smg\_todo \), which is true if and only if there still are addresses to be fetched for the SMG instruction.

SMG

Format: `SMG Rd, Ra`

Formally: \( smg\_todo \leftarrow true; \left( \forall i : 0 \leq i < P : R[d][i] \leftarrow mem[R[a][i]] \right); smg\_todo \leftarrow false \)

This instruction is part of the load/store unit. It will load a vector of data from memory addressed by a vector of addresses contained in a vector register \( R[a] \). Every \( R[a][i] \) is interpreted as a 16-bit memory address. When two addresses “collide”, due to being located on
the same memory bank, the processor will iteratively load addresses until the full vector has been loaded. The instruction generates shuffle patterns which it feeds to the shuffle unit.

SGW

Format: SGW

Formally: if $smg \_ todo$ then $pc \leftarrow pc + 1$

This instruction is part of the flow control unit. We extend the flow control unit to be able to stall, that means that the $pc$ keeps its value while $smg \_ todo$ is true, and we feed NOPs into the processor for as long as the smart gather fetching has not completed yet. A consequence is that instructions remaining in the pipeline are executed, i.e. this instruction has an implementation determined latency before the processor will stall.

### 7.5.2 Gather related latency

<table>
<thead>
<tr>
<th>Resource</th>
<th>Source</th>
<th>Destination</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data memory</td>
<td>STV</td>
<td>SMG</td>
<td>1</td>
</tr>
<tr>
<td>$smg _ todo$</td>
<td>SMG</td>
<td>SGW</td>
<td>3</td>
</tr>
<tr>
<td>Pipeline flush</td>
<td>SGW</td>
<td>stall</td>
<td>2</td>
</tr>
<tr>
<td>Register file</td>
<td>SMG</td>
<td>all MAC,SHF,STV</td>
<td>8+load cycles</td>
</tr>
</tbody>
</table>

### 7.5.3 Allowed parallelism

We allow all combinations of instructions, except those already stated in the ISA chapter plus the following ($l$ designates the number of loads needed to fully load the requested vector of addresses):

- no LDV or STV instructions allowed in the $2..2 + l$ instructions following the SMG instruction;

- no SHF instructions allowed in the $6..6 + l$ instructions following the SMG instruction.

### 7.5.4 Pipeline with gather

A diagram is provided in figure 7.3 that gives an overview of the extension of the instruction format timing with functional units for the gather instruction. Execution of the gather consists of six stages, with the extension to the pipeline shown in figure 7.4:
1. construction of $P$ trees of booleans of height $\log P$, with each node determining whether for a bank, the left subtree contains the address to be fetched; also construct a shuffle pattern that will shuffle the data vector loaded from the memory banks into the result vector according to the requested addresses;

2. execution of $P$ multiplexer trees of addresses, with height $1 + \log P$, each node having input of the corresponding node in the tree of the previous stage, copying the address to be fetched from a lane to a bank;

3. load the constructed vector of addresses;

4. propagation of data to shuffle unit (2 stages);

5. shuffle the loaded data into result vector.

![Figure 7.3: Instruction format and timing gather extension](image)

### 7.6 Gather collision measurements

To get a feeling for the performance, we want to know the number of fetches we need to do to complete a gather. We simulate a run of our ripple demo in an application written in C, once where the water surface is mostly flat, or “quiet” (see figure 7.9) and another where the water surface has a lot of waves, being “stormy” water (see figure 7.5); as the collision pattern can differ. We want to know the percentage of the total set of gathers that can be successfully completed within $n$ fetches, $1 \leq n \leq P$. Also will we simulate gathering a random vector of addresses (see figure 7.7), so we can compare our ripple gather behaviour to random gather behaviour.

As we want to decrease the number of needed fetches, we add a variant that checks in case of a collision, whether the addresses are the same: we can do one fetch and copy the data to both locations, instead of needing two fetches. The results for this variant are incorporated in the graphs as the “AC” variant, which is short for address check.
Figure 7.4: Pipeline gather extension
Figure 7.5: Percentage fetchable given a maximum number of cycles to gather, in stormy water

Figure 7.6: Expected number of cycles needed to complete a gather
Figure 7.7: Percentage fetchable given a maximum number of cycles to gather, random addresses

Figure 7.8: Percentage fetchable given a maximum number of cycles to gather, for P=64
Figure 7.9: Percentage fetchable given a maximum number of cycles to gather, in quiet water

Figure 7.10: Percentage fetchable given a maximum number of cycles to gather, for $P=8$
Figure 7.11: Percentage fetchable given a maximum number of cycles to gather, for $P=16$

Figure 7.12: Percentage fetchable given a maximum number of cycles to gather, for $P=32$
Particularly figure 7.8 makes clear that gather behaviour in our ripple demo is better compared to the random case: we need to wait at most 2 cycles. This is good enough, and so we will not implement the address check variant.

7.7 Conclusion

In this chapter we looked at implementing a ripple demo. The demo is implementable in several ways, of which we describe two, with adding a new instruction being the more efficient one. Inherently the demo exhibits better collision patterns on the memory banks in the addresses it wants to gather when compared to random access patterns. We use the upper bound on the needed number of fetches found in experiments for the implementation of our algorithm.
Chapter 8

Conclusion and evaluation

Our aim was to explore scalability and performance issues when designing and implementing a vector processor on an FPGA. We did this by designing a custom ISA, and implementing it on the Xilinx Virtex-4 FPGA. A complete machine was not our goal: instead, we focused on the vector aspects of a vector processor, and so there is little to no support for a scalar path and unit, nor for a conditional branch. By taking a set of algorithms to test the processor with, the ISA was designed to support these algorithms.

With regard to scalability we note that $P = 8$ was the variant we worked on day-to-day, and therefore fairly well optimized; for greater $P$ the tools have significantly longer run time. The processor runs at 200 MHz for $P = 4$ and $P = 8$; at $P = 16$ we achieve 120 MHz, and $P = 32$ will run at only 30 MHz (estimated) at this moment. For the $P = 16$ case we observe a critical path in the shuffle to MAC bypass, and in smart gather stall control. We expect that by modifying the ISA in the becoming available of the output value of the shuffle to the MAC unit one cycle later, and similar for stall control, $P = 16$ can also run near to 200MHz. In the $P = 32$ case the FPGA is too small for the same general layout to fit that was also used for $P = 16$ and below.

We can conclude:

- it is feasible to implement a 200MHz vector processor on an FPGA, at least on the Xilinx Virtex-4, when choosing the ISA carefully;

- one should expect higher latencies for instructions, due to physical routing limitations; nevertheless, high throughput, meaning issuing at least one instruction per cycle, is feasible;

- complex instructions need to be pipelined such that they use one DSP block, or have at most two LUTs in any short path, or one on long paths;

- implementing a general orthogonal instruction set is hard due to dependencies, special cases (an ALU for instance consists of a lot of operations to be implemented), and missing features of native blocks (for example saturation on DSP blocks);
location constraints can increase performance by a factor of 3, see section 5.4, as it can help by forcing pipeline registers to be located closer together so that more LUT logic can be placed in between; especially a vector processor implementation is a complex, but more or less regular structure, where this regularity can be enforced using location constraints.

When designing an ISA that is to be run on FPGAs, in comparison to silicon, the following items will need attention:

- allowing higher latencies of functional unit output values which are to be used in other functional units, i.e. the out value will only become available a couple of instructions later;
- type and number of variants of instructions, saturating of MAC output for instance is difficult;
- especially allow higher latencies for functional units of different operation with respect to intervector versus intravector.

8.1 Evaluation

Looking back, we can identify some issues we might have solved or handled differently if this project were repeated. In hindsight we can remark the following items:

- for the DOI instruction we require that the first instruction of the loop referred to is at least three instructions later. This is due to the pipeline depth, and we could have compensated for this fact by subtracting two from the loop start offset while decoding the instruction. This does not hold for the loop end address though, since the instructions have already been fetched at the time the loop end address is decoded, while these might have been after the loop, and as such should not have been fetched;
- debugging the functionality, in particular the pipeline, timing, and interaction is challenging;
- debugging and optimizing simultaneously is even more challenging;
- the assembler needs to take care of mapping variables to address ranges, such that one can work with variables instead of with the addresses themselves.

69
Appendix A

Code

A.1 FIR filter code

```plaintext
# FIR filter for P=4, N=6

# R0 - 0
# R1 - sample vector
# R2 - result vector
# T1 - rotate backward broadcast
# P1 - sample pointer
# P2 - coeff pointer
# P3 - zero/pattern pointer
# P4 - dest pointer

# load pointer
LDP P3, 0
LDB R3, 0
LDE E3, 1
LDP P1, 2
LDB R1, 2
LDE E1, 5
LDP P2, 6
LDB R2, 6
LDE E2, 7
LDV R0, P3, 1
LUT T1, P3, 0
LDV R1, P1, 1
LDP P4, 8
LDM P2, 1
LDS P1, 1
LIM P2, 1; NOP; D01 4, loopstart, loopend
NOC ; NOP; NOP ; NOP ; SHF R1, R1, T1; RMB
# first iteration, without save
NOC ; NOP; NOP ; EMUL R2, R1; SHF R1, R1, T1; NOP; RSB
LDS P1, 1; NOP; NOP ; EMAC R2, R1; SHF R1, R1, T1; RMB; RSB
LIM P2, 1; NOP; NOP ; EMAC R2, R1; SHF R1, R1, T1; RMB; RSB
NOC ; NOP; NOP ; EMAC R2, R1; SHF R1, R1, T1; RMB; NOP
LIM P2, 1; NOP; NOP ; EMAC R2, R1; SHF R1, R1, T1; NOP; RSB
LDS P1, 1; NOP; NOP ; EMAC R2, R1; SHF R1, R1, T1; RMB; RSB
# main loop
loopstart:
NOC ; NOP; NOP ; EMUL R3, R1; SHF R1, R1, T1; NOP; RSB
NOC ; NOP; NOP ; EMAC R3, R1; SHF R1, R1, T1; RMB; NOP
```

70
A.2 FFT code

A.2.1 Vector width 8

```c
# FFT filter for P=8, M=64
# complex multiply: e + f*i := (a + b*i) * (c + d*i)
# NQP; NQP; NQP; VMUL Re, Ra, Rc
# NQP; NQP; NQP; VMDC Re, Rd, Rd
# NQP; NQP; NQP; VMUL Rf, Ra, Rd
# complex addition: e + f*i := (a + b*i) + (c + d*i)
# NQP; NQP; VADD Re, Ra, Rc
# complex subtraction: e + f*i := (a + b*i) - (c + d*i)
# NQP; NQP; VSUB Re, Ra, Rc

# R0-R7: v0-v7, real part vector 0-7
# R8-R15: v0-v7, imaginary part vector 0-7
# R16-R19: v0-vt3, real part twiddle vector 0-3
# R20-R23: v0-vt3, imaginary part twiddle vector 0-3
# R24: w, real part vector M_T
# R25: w, imaginary part vector M_T
# R26: vS, temporary shuffle vector S_T

# P0: constant data pointer (shuffles + twiddles)
# P1: input sample pointer
# P2: output sample pointer
# T0: shuffle pattern "plain copy"
# T1: shuffle pattern stage 4, first half enabled, butterfly 64 bits
# T2: shuffle pattern stage 4, second half enabled, butterfly 64 bits
# T3: shuffle pattern stage 5, first half enabled, butterfly 32 bits
# T4: shuffle pattern stage 5, second half enabled, butterfly 32 bits
# T5: shuffle pattern stage 6, first half enabled, butterfly 16 bits
# T6: shuffle pattern stage 6, second half enabled, butterfly 16 bits

# memory layout
# address P0: T0, T1, T2, T3, T4, T5, T6
# twiddle 1c, 1d, 1a, 1b, 2a, 2b, 3, 4, 5
# |
# |---------loop--------|

```
LDE EO, 16
LDP P1, 26
LDP P2, 58
LDT T0, PO, 1
LDT T1, PO, 1
LDT T2, PO, 1
LDT T3, PO, 1
LDT T4, PO, 1
LDT T5, PO, 1
LDT T6, PO, 1
LDS R18, PO, 1
LDS R22, PO, 1
LDS R19, PO, 1
LDS R23, PO, 1
LDS R16, PO, 1
LDS R20, PO, 1
LDS R17, PO, 1
LDS R21, PO, 1

### Load input data ###
LDS R0, P1, 8
LDS R4, P1, -7
LDS R8, P1, 8
LDS R12, P1, -7
LDS R1, P1, 8; NOP; DQI 32, loopbegin, loopend

### Stage 1 ###
# vN := v0 - v4
NOP ; NOP; NOP; VSUB R24, R0, R4
NOP ; NOP; NOP; VADD R0, R0, R4
# v0 := v0 + v4
NOP ; NOP; NOP; VSUB R26, R8, R12
LDN R5, P1, -7 ; NOP; NOP; VADD R8, R8, R12
# v4 := vM * vT0
LDN R9, P1, 8 ; NOP; NOP; VMUL R4, R24, R16
LDN R13, P1, -7; NOP; NOP; VMDC R4, R25, R20
NOP ; NOP; NOP; VMUL R12, R26, R16
loopbegin:
NOP ; NOP; NOP; VMAC R12, R24, R20
# vM := v1 - v6
NOP ; NOP; NOP; VSUB R24, R1, R5
NOP ; NOP; NOP; VSUB R26, R9, R13
# v0 := v1 + v6
LDN R2, P1, 8 ; NOP; NOP; VADD R1, R1, R5
LDN R6, P1, -7 ; NOP; NOP; VADD R9, R9, R13
# v6 := vM * vT1
LDN R10, P1, 8 ; NOP; NOP; VMUL R5, R24, R17
LDN R14, P1, -7; NOP; NOP; VMDC R5, R25, R21
NOP ; NOP; NOP; VMUL R13, R26, R17
NOP ; NOP; NOP; VMAC R13, R24, R21
# vM := v2 - v6
NOP ; NOP; NOP; VSUB R24, R2, R6
NOP ; NOP; NOP; VSUB R26, R10, R14
# v0 := v2 + v6
LDN R3, P1, 8 ; NOP; NOP; VADD R2, R2, R6
LDN R7, P1, -7 ; NOP; NOP; VADD R10, R10, R14
# v6 := vM * vT2
LDN R11, P1, 8 ; NOP; NOP; VMUL R6, R24, R18
LDN R15, P1, 1 ; NOP; NOP; VMDC R6, R25, R22
NOP ; NOP; NOP; VMUL R14, R26, R18
NOP ; NOP; NOP; VMAC R14, R24, R22
# vM := v3 - v7
NOP ; NOP; NOP; VSUB R24, R3, R7
NOP ; NOP; NOP; VSUB R26, R11, R15
# v0 := v3 + v7
NOP ; NOP; NOP; VADD R3, R3, R7
```c
NQP ; NQP; NQP; VADD R11, R11, R15
# v7 := vM * v13
LDV R16, P0, 1 ; NQP; NQP; VML R7, R24, R19
LDV R20, P0, 1 ; NQP; NQP; VMDC R7, R25, R23
LDV R17, P0, 1 ; NQP; NQP; VML R15, R26, R19
LDV R21, P0, 1 ; NQP; NQP; VMAC R15, R24, R23

### Stage 2 ###
# vM := w0 - w4  \{ w0 = v0, w4 = v2 \}
NQP ; NQP; NQP; VSUB R24, R0, R2
NQP ; NQP; NQP; VSUB R25, R6, R10
# w0 := v0 + w4
NQP ; NQP; NQP; VADD R0, R0, R2
NQP ; NQP; NQP; VADD R8, R8, R10
# w4 := vM * v0
NQP ; NQP; NQP; VML R2, R24, R16
NQP ; NQP; NQP; VMDC R2, R25, R20
NQP ; NQP; NQP; VML R10, R25, R16
NQP ; NQP; NQP; VMAC R10, R24, R20
# vM := w1 - w6  \{ w1 = v1, w6 = v3 \}
NQP ; NQP; NQP; VSUB R24, R1, R3
NQP ; NQP; NQP; VSUB R25, R9, R11
# w0 := w1 + w6
NQP ; NQP; NQP; VADD R1, R1, R3
NQP ; NQP; NQP; VADD R9, R9, R11
# w5 := vM * v11
NQP ; NQP; NQP; VML R3, R24, R17
NQP ; NQP; NQP; VMDC R3, R25, R21
NQP ; NQP; NQP; VML R11, R25, R17
NQP ; NQP; NQP; VMAC R11, R24, R21
# vM := w2 - w6  \{ w2 = v4, w6 = v6 \}
NQP ; NQP; NQP; VSUB R24, R4, R6
NQP ; NQP; NQP; VSUB R25, R12, R14
# w2 := w2 + w6
NQP ; NQP; NQP; VADD R4, R4, R6
NQP ; NQP; NQP; VADD R12, R12, R14
# w6 := vM * v10
NQP ; NQP; NQP; VML R6, R24, R16
NQP ; NQP; NQP; VMDC R6, R25, R20
NQP ; NQP; NQP; VML R14, R25, R16
NQP ; NQP; NQP; VMAC R14, R24, R20
# vM := w3 - w7  \{ w3 = v5, w7 = v7 \}
NQP ; NQP; NQP; VSUB R24, R5, R7
NQP ; NQP; NQP; VSUB R25, R13, R15
# w3 := w3 + w7
NQP ; NQP; NQP; VADD R5, R5, R7
NQP ; NQP; NQP; VADD R13, R13, R15
# w7 := vM * v11
NQP ; NQP; NQP; VML R7, R24, R17
NQP ; NQP; NQP; VMDC R7, R25, R21
LDV R16, P0, 1 ; NQP; NQP; VML R15, R26, R17
LDV R20, P0, 1 ; NQP; NQP; VMAC R15, R24, R21

### Stage 3 ###
# vM := x0 - x4  \{ x0 = v0 = v0, x4 = v1 = v1 \}
NQP ; NQP; NQP; VSUB R24, R0, R1
NQP ; NQP; NQP; VSUB R25, R6, R9
# x0 := x0 + x4
NQP ; NQP; NQP; VADD R0, R0, R1
NQP ; NQP; NQP; VADD R8, R8, R9
# x4 := vM * v10
NQP ; NQP; NQP; VML R1, R24, R16
NQP ; NQP; NQP; VMDC R1, R25, R20
NQP ; NQP; NQP; VML R9, R25, R16
NQP ; NQP; NQP; VMAC R9, R24, R20
# vM := x1 - x6  \{ x1 = v4 = v2, x6 = v5 = v3 \}
```
NQP ; NQP; NQP; VSUB R24, R2, R3
NQP ; NQP; NQP; VSUB R25, R10, R11
# x1 := x1 + x6
NQP ; NQP; NQP; VADD R2, R2, R3
NQP ; NQP; NQP; VADD R10, R10, R11
# x5 := v0 * v0
NQP ; NQP; NQP; VMUL R3, R24, R16
NQP ; NQP; NQP; VMUL R3, R25, R20
NQP ; NQP; NQP; VMUL R11, R25, R16
NQP ; NQP; NQP; VMAC R11, R24, R20
# x2 := x2 - x6 { x2 = x2 - v4, x6 = v3 = v5 }
NQP ; NQP; NQP; VSUB R24, R4, R5
NQP ; NQP; NQP; VSUB R25, R12, R13
# x2 := x2 + x6
NQP ; NQP; NQP; VADD R4, R4, R5
NQP ; NQP; NQP; VADD R12, R12, R13
# x6 := v0 * v0
NQP ; NQP; NQP; VMUL R5, R24, R16
NQP ; NQP; NQP; VMDC R5, R25, R20
NQP ; NQP; NQP; VMUL R13, R25, R16
NQP ; NQP; NQP; VMAC R13, R24, R20
# x3 := x3 + x7 { x3 = x3 + v6, x7 = v7 = v7 }
NQP ; NQP; NQP; VSUB R24, R6, R7
NQP ; NQP; NQP; VSUB R25, R14, R15
# x7 := v0 * v0
NQP ; NQP; NQP; VMUL R7, R24, R16
NQP ; NQP; NQP; VMDC R7, R25, R20
NQP ; NQP; NQP; VMUL R15, R25, R16
NQP ; NQP; NQP; VMAC R15, R24, R20
LDV R16, PO, 1 ; NQP; NQP; VMUL R15, R25, R16
LDV R20, PO, 1 ; NQP; NQP; VMAC R15, R24, R20
### Stage 4 ###
# x0 := x0 - x4 { x0 = x0 - v0, x4 = x1 = v1 }
NQP ; NQP; NQP; VSUB R24, R0, R1
NQP ; NQP; NQP; VSUB R25, R8, R9
# x0 := x0 + x4
NQP ; NQP; NQP; VADD R0, R0, R1
NQP ; NQP; NQP; VADD R8, R8, R9
# x4 := v0 * v0
NQP ; NQP; NQP; VMUL R1, R24, R16
NQP ; NQP; NQP; VMDC R1, R25, R20
# x1 := x1 + x5
NQP ; NQP; NQP; VSUB R24, R2, R3
NQP ; NQP; NQP; VSUB R25, R10, R11
# x5 := v0 * v0
NQP ; NQP; NQP; VMUL R3, R24, R16
NQP ; NQP; NQP; VMUL R3, R25, R20
# x2 := x2 - x6 { x2 = x2 - v4, x6 = v3 = v5 }
NQP ; NQP; NQP; VSUB R24, R4, R5
NQP ; NQP; NQP; VSUB R25, R12, R13
# x2 := x2 + x6
NQP ; NQP; NQP; VADD R4, R4, R5
NQP ; NQP; NQP; VADD R12, R12, R13
# x6 := v0 * v0
NQP ; NQP; NQP; VMUL R5, R24, R16
NQP ; NQP; NQP; VMDC R5, R25, R20
NQP ; NQP; NQP; VMUL R13, R25, R16
Vgx
"µ¢hg£bodp{£ij ¹{£boXhgoXhXxboŠkm£ijghX#qrhXx^qrº
Vgx
"µ¢hg£^q{p{£bŽ ¹{£^q—hgbjghXxbzŠkm£bŽXhXŽghXxb"gº
NQP ; NQP; NQP; VMAC R13, R24, R20
# v6 := x3 - x7  \{ x3 = v6 = v6, x7 = v7 = v7 \}
NQP ; NQP; NQP; VSUB R24, R6, R7
NQP ; NQP; NQP; VSUB R26, R14, R15
# x3 := x3 + x7
NQP ; NQP; NQP; VADD R6, R6, R7 ; SHIF R26, R2, T0
NQP ; NQP; NQP; VADD R14, R14, R15 ; SHIF R2, R0, T3
# x7 := v6 * v70
NQP ; NQP; NQP; VMUL R7, R24, R16 ; SHIF R0, R26, T4
NQP ; NQP; NQP; VMDC R7, R25, R20 ; SHIF R26, R10, T0
# { use twiddle vector 2, so we can reload vector 1 for next cycle earlier }
LDV R17, PO, 1 ; NQP; NQP; VMAC R15, R26, R16 ; SHIF R10, R8, T3
LDV R21, PO, 1 ; NQP; NQP; VMAC R15, R24, R20 ; SHIF R8, R26, T4

### Stage 5 ###

# v6 := w0 - w4  \{ w0 = v0, w4 = v2 \}
NQP ; NQP; NQP; VSUB R24, R0, R2 ; SHIF R26, R3, T0
NQP ; NQP; NQP; VSUB R26, R8, R10 ; SHIF R3, R1, T3
# w0 := w0 + w4
NQP ; NQP; NQP; VADD R0, R0, R2 ; SHIF R1, R26, T4
NQP ; NQP; NQP; VADD R8, R8, R10 ; SHIF R26, R11, T0
# w4 := v6 * v0
NQP ; NQP; NQP; VMUL R2, R24, R17 ; SHIF R11, R9, T3
NQP ; NQP; NQP; VMDC R2, R25, R21 ; SHIF R9, R26, T4
NQP ; NQP; NQP; VMUL R10, R25, R21 ; SHIF R36, R6, T0
NQP ; NQP; NQP; VMAC R10, R24, R21 ; SHIF R6, R4, T3
# v6 := v1 - w6  \{ w1 = v1, w6 = v3 \}
NQP ; NQP; NQP; VSUB R24, R1, R3 ; SHIF R4, R26, T4
NQP ; NQP; NQP; VSUB R26, R9, R11 ; SHIF R26, R14, T0
# w0 := v1 + w6
NQP ; NQP; NQP; VADD R1, R1, R3 ; SHIF R14, R12, T3
NQP ; NQP; NQP; VADD R9, R9, R11 ; SHIF R12, R26, T4
# w6 := v6 * v1
NQP ; NQP; NQP; VMUL R3, R24, R17 ; SHIF R36, R7, T0
NQP ; NQP; NQP; VMDC R3, R25, R21 ; SHIF R7, R5, T3
NQP ; NQP; NQP; VMUL R11, R26, R17 ; SHIF R5, R26, T4
NQP ; NQP; NQP; VMAC R11, R24, R21 ; SHIF R26, R15, T0
# w2 := w2 - w6  \{ w2 = v4, w6 = v5 \}
NQP ; NQP; NQP; VSUB R24, R4, R6 ; SHIF R15, R13, T3
NQP ; NQP; NQP; VSUB R26, R12, R14 ; SHIF R13, R26, T4
# w2 := w2 + w6, shuffling for stage 6 starts
NQP ; NQP; NQP; VADD R4, R4, R6 ; SHIF R36, R4, T0
NQP ; NQP; NQP; VADD R12, R12, R14 ; SHIF R4, R0, T5
# w6 := v6 * v0
NQP ; NQP; NQP; VMUL R6, R24, R17 ; SHIF R0, R26, T6
NQP ; NQP; NQP; VMDC R6, R25, R21 ; SHIF R26, R12, T0
NQP ; NQP; NQP; VMUL R14, R26, R17 ; SHIF R12, R8, T5
NQP ; NQP; NQP; VMAC R14, R24, R21 ; SHIF R8, R26, T6
# v6 := w3 - w7  \{ w3 = v5, w7 = v7 \}
NQP ; NQP; NQP; VSUB R24, R5, R7 ; SHIF R26, R5, T0
LDV PO, 20 ; NQP; NQP; VSUB R26, R13, R15 ; SHIF R5, R1, T5
# w3 := w3 + w7
NQP ; NQP; NQP; VADD R5, R5, R7 ; SHIF R1, R26, T6
# { refresh first two twiddle vectors }
LDV R16, PO, 1 ; NQP; NQP; VADD R13, R13, R15 ; SHIF R26, R13, T0
# w7 := w7 + w1
LDV R20, PO, 1 ; NQP; NQP; VMUL R7, R24, R17 ; SHIF R13, R9, T5
LDV R17, PO, 1 ; NQP; NQP; VMDC R7, R25, R21 ; SHIF R9, R26, T6
LDV R21, PO, 1 ; NQP; NQP; VMUL R15, R26, R17 ; SHIF R36, R6, T0
# { start loading input data for next cycle }
LDV R0, PI, 8 ; NQP; NQP; VMAC R15, R24, R21 ; SHIF R6, R2, T5

### Stage 6 ###

# v6 := v0 - w4
LDV R4, PI, -7 ; NQP; NQP; VSUB R24, R0, R4 ; SHIF R2, R26, T6
LDV R8, PI, 8 ; NQP; NQP; VSUB R24, R8, R12 ; SHIF R26, R14, T0

75
A.2.2 Vector width 16

# FFT for P=16, N=64

# complex multiply: e + f*i := (a + b*i) * (c + d*i)
# NOP; VMUL Re, Ra, Rc
# NOP; VMDC Re, Rb, Rd
# NOP; VMUL Rf, Rb, Rc
# NOP; VNAC Rf, Ra, Rd
# complex addition: e + f*i := (a + b*i) + (c + d*i)
# NOP; VADD Re, Ra, Rc
# NOP; VADD Rf, Rb, Rd
# complex subtraction: e + f*i := (a + b*i) - (c + d*i)
# NOP; VSUB Re, Ra, Rc
# NOP; VSUB Rf, Rb, Rd

# R0-R3: v0-v3, real part vector 0-3
# R4-R7: v0-v3, imaginary part vector 0-3
# R8: v0, real part twiddle vector 0
# R10: v0, imaginary part twiddle vector 0
# R12: v0, real part vector M_T
# R13: v0, imaginary part vector M_T
# R14: v0, temporary shuffle vector S_T

# P0: constant data pointer (shuffles + twiddles)
# P1: input sample pointer
# P2: output sample pointer
# T1: shuffle pattern 1, stage 3, first half enabled, butterfly 128 bits
# T2: shuffle pattern 2, stage 3, second half enabled, butterfly 128 bits
# T1: shuffle pattern 3, stage 4, first half enabled, butterfly 64 bits
# T2: shuffle pattern 4, stage 4, second half enabled, butterfly 64 bits
# T1: shuffle pattern 5, stage 5, first half enabled, butterfly 128+32 bits
# T2: shuffle pattern 6, stage 5, second half enabled, butterfly 128+32 bits
# T1: shuffle pattern 7, stage 6, first half enabled, butterfly 16 bits
# T2: shuffle pattern 8, stage 6, second half enabled, butterfly 16 bits

# memory layout
# address PO: twiddle vectors 1-5, shuffle patterns 1-8
# twiddle 1a, 1b, 2, T1, T2, 3, T3, T4, 4, T5, T6, 5, T7, T8
# |

### Load constants ###
LDI P0, 0
LDI R0, 0
LDE EO, 14
LDI P1, 16
LDI P2, 58

### Load input data ###
LDV R0, P1, 4
LDV R2, P1, -3
LDV R4, P1, 4
LDV R6, P1, -3
LDV R1, P1, 4
LDV R3, P1, -3
LDV R5, P1, 4
LDV R8, PO, 1
# 32 is number of input vectors to process
LDV R10, PO, 1 ; NOP; DII 32, loopbegin, loopend

### Stage 1 ###
# vM := v0 - v2
NOP ; NOP; NOP; VSUB R12, R0, R2
NOP ; NOP; NOP; VADD R0, RO, R2
# v0 := v0 + v2
NOP ; NOP; NOP; VSUB R13, R4, R6
NOP ; NOP; NOP; VADD R4, R4, R6
# v2 := vM * v0
LDV R5, P1, 4 ; NOP; NOP; VMUL R2, R12, R8
LDV R7, P1, 1 ; NOP; NOP; VMDC R2, R13, R10
NOP ; NOP; NOP; VMAC R6, R12, R10
loopbegin:
NOP ; NOP; NOP; VMAC R6, R12, R10
# vM := v2 - v6
LDV R8, PO, 1 ; NOP; NOP; VSUB R12, R1, R3
LDV R10, PO, 1 ; NOP; NOP; VSUB R13, R5, R7
# v0 := v2 + v6
NOP ; NOP; NOP; VADD R1, R1, R3
NOP ; NOP; NOP; VADD R5, R5, R7
# v6 := vM * v12
NOP ; NOP; NOP; VMUL R3, R12, R8
NOP ; NOP; NOP; VMDC R3, R13, R10
NOP ; NOP; NOP; VMUL R7, R13, R8
NOP ; NOP; NOP; VMAC R7, R12, R10

### Stage 2 ###
# vM := v0 - v2 { w0 = v0, w2 = v1 }
LDV R8, PO, 1 ; NOP; NOP; VSUB R12, R0, R1
LDV R10, PO, 1 ; NOP; NOP; VSUB R13, R4, R5
# w0 := w0 + v2
NOP ; NOP; NOP; VADD R0, RO, R1
NOP ; NOP; NOP; VADD R4, R4, R5
# w2 := vM * v0
LDT T1, PO, 1 ; NOP; NOP; VMUL R1, R12, R8
LDT T2, PO, 1 ; NOP; NOP; VMDC R1, R13, R10
NOP ; NOP; NOP; VMUL R5, R5, R5
NOP ; NOP; NOP; VMAC R5, R12, R10
# vM := w1 - w0 { w1 = v2, w3 = v3 }
NP    ; NOP; NOP; VSUB R12, R2, R3
NP    ; NOP; NOP; VSUB R13, R6, R7
# w1 := w1 + w3
MOV R14, R1    ; NOP; NOP; VADD R2, R2, R3
NOP    ; NOP; NOP; VADD R6, R6, R7
# w3 := w1 * v0
MOV R14, R5    ; NOP; NOP; VMUL R3, R12, R8    ; SHF R1, R0, T1
NOP    ; NOP; NOP; VMDC R3, R13, R10    ; SHF R0, R14, T2
MOV R14, R3    ; NOP; NOP; VMUL R7, R13, R8    ; SHF R5, R4, T1
LDV R8, PO, 1  ; NOP; NOP; VMAC R7, R12, R10    ; SHF R4, R14, T2
### Stage 3 ###
# v0 := w0 - w2
MOV R14, R7    ; NOP; NOP; VSUB R12, R0, R1    ; SHF R3, R2, T1
LDV R10, PO, 1 ; NOP; NOP; VSUB R13, R4, R5    ; SHF R2, R14, T2
# w0 := w0 + w2
NOP    ; NOP; NOP; VADD R0, R0, R1    ; SHF R7, R6, T1
NOP    ; NOP; NOP; VADD R4, R4, R5    ; SHF R6, R14, T2
# w2 := w1 * v0
LDT T1, PO, 1  ; NOP; NOP; VMUL R1, R12, R8
LDT T2, PO, 1  ; NOP; NOP; VMDC R1, R13, R10
NOP    ; NOP; NOP; VMUL R5, R13, R8
NOP    ; NOP; NOP; VMAC R5, R12, R10
# v1 := w1 - w3
NOP    ; NOP; NOP; VSUB R12, R2, R3
NOP    ; NOP; NOP; VADD R6, R6, R7
# w3 := w1 + w3
MOV R14, R5    ; NOP; NOP; VMUL R3, R12, R8    ; SHF R1, R0, T1
NOP    ; NOP; NOP; VMDC R3, R13, R10    ; SHF R0, R14, T2
MOV R14, R3    ; NOP; NOP; VMUL R7, R13, R8    ; SHF R5, R4, T1
LDV R8, PO, 1  ; NOP; NOP; VMAC R7, R12, R10    ; SHF R4, R14, T2
### Stage 4 ###
# v0 := w0 - w2
MOV R14, R7    ; NOP; NOP; VSUB R12, R0, R1    ; SHF R3, R2, T1
LDV R10, PO, 1 ; NOP; NOP; VSUB R13, R4, R5    ; SHF R2, R14, T2
# w0 := w0 + w2
NOP    ; NOP; NOP; VADD R0, R0, R1    ; SHF R7, R6, T1
NOP    ; NOP; NOP; VADD R4, R4, R5    ; SHF R6, R14, T2
LDT T1, PO, 1  ; NOP; NOP; VMUL R1, R12, R8
LDT T2, PO, 1  ; NOP; NOP; VMDC R1, R13, R10
NOP    ; NOP; NOP; VMUL R5, R13, R8
NOP    ; NOP; NOP; VMAC R5, R12, R10
# v1 := w1 - w3
NOP    ; NOP; NOP; VSUB R12, R2, R3
NOP    ; NOP; NOP; VADD R6, R6, R7
# w1 := w1 + w3
MOV R14, R5    ; NOP; NOP; VMUL R3, R12, R8    ; SHF R1, R0, T1
NOP    ; NOP; NOP; VMDC R3, R13, R10    ; SHF R0, R14, T2
MOV R14, R3    ; NOP; NOP; VMUL R7, R13, R8    ; SHF R5, R4, T1
LDV R8, PO, 1  ; NOP; NOP; VMAC R7, R12, R10    ; SHF R4, R14, T2
### Stage 5 ###
# v0 := w0 - w2
MOV R14, R7    ; NOP; NOP; VSUB R12, R0, R1    ; SHF R3, R2, T1
LDV R10, PO, 1 ; NOP; NOP; VSUB R13, R4, R5    ; SHF R2, R14, T2
# w0 := w0 + w2
NOP    ; NOP; NOP; VADD R0, R0, R1    ; SHF R7, R6, T1
NOP    ; NOP; NOP; VADD R4, R4, R5    ; SHF R6, R14, T2
78
A.3 Ripple code

# Ripple demo for vector processor
# note: registers and pointer names need to be translated
# numbers before assembling using e.g. a sed script

$ rplpp: prevlineprevpixel
$ rplcp: prevlinecurrpixel
$ riplp: prevlinenextpixel
$ riplv: prevlinenextvector
$ ripn2: prevlinenext2vector
$ rclpp: currlineprevpixel
$ rclcp: currlinecurrpixel
$ rclmp: currlinenextpixel
$ rclnv: currlinenextvector
$ rcln2: currlinenext2vector
$ rnlpp: nextlineprevpixel
$ rnlcp: nextlinecurrpixel
$ rnlmp: nextlinenextpixel
$ rnlnv: nextlinenextvector
$ rnln2: nextlinenext2vector

v2 := v3 * v0
LD T1, P0, 1  ; NDP; NOP; VMUL R1, R12, R8
LD T2, P0, 1  ; NDP; NOP; VMDC R1, R13, R10
NOP  ; NDP; NOP; VMUL R5, R13, R8
NOP  ; NDP; NOP; VMAC R5, R12, R10
w3 = w1 - w3  (w1 = v2, w3 = v3)
NOP  ; NDP; NOP; VSUB R12, R2, R3
MOV R14, R1  ; NDP; NOP; VSUB R13, R6, R7
w1 := w1  + w3
MOV R14, R5  ; NDP; NOP; VADD R2, R2, R3
LD R8, P0, 1  ; NDP; NOP; VADD R6, R6, R7
w3 := w0  + v3
LD R10, P0, 1  ; NDP; NOP; VMUL R3, R12, R8 ; SHF R1, R0, T1
MOV R14, R3  ; NDP; NOP; VMDC R3, R13, R10 ; SHF R0, R14, T2
MOV R14, R7  ; NDP; NOP; VMUL R7, R13, R8 ; SHF R5, R4, T1
{ start loading input data for next cycle }
LD R0, P1, 4  ; NDP; NOP; VMAC R7, R12, R10 ; SHF R4, R14, T2
### Stage 6 ###
# v4 := v0  - v4
LD R2, P1, -3  ; NDP; NOP; VSUB R12, R0, R2 ; SHF R3, R2, T1
LD R4, P1, 4  ; NDP; NOP; VSUB R12, R4, R6 ; SHF R2, R14, T2
w0 := w0  + v4
LD R6, P1, -3  ; NDP; NOP; VADD R12, R0, R2 ; SHF R7, R6, T1
LD R1, P1, 4  ; NDP; NOP; VADD R12, R4, R6 ; SHF R6, R14, T2
v4 := v2  - v6
LD R3, P1, -3  ; NDP; NOP; VSUB R12, R1, R3
STV R12, P2, 4  ; NDP; NOP; VSUB R12, R5, R7
w2 := w0  + v6
STV R12, P2, -3; NDP; NOP; VADD R12, R1, R3
STV R12, P2, 4  ; NDP; NOP; VADD R14, R5, R7
# store vectors, start next cycle
STV R12, P2, -3; NDP; NOP; VSUB R12, R0, R2
STV R12, P2, 4  ; NDP; NOP; VADD R0, R0, R2
STV R12, P2, -3; NDP; NOP; VSUB R13, R4, R6
STV R12, P2, 4  ; NDP; NOP; VADD R4, R4, R6
LD R6, P1, 4  ; NDP; NOP; VMUL R2, R12, R8
LD R7, P1, 1  ; NDP; NOP; VMUL R6, R13, R8
loopend:
STV R14, P2, 1  ; NDP; NOP; VMUL R6, R13, R8

79
# Roldbuf2: old buf2
# Rnewbuf2: new buf2
# Pplbuf1: prevline buf1
# Pclbuf1: currline buf1
# Pnlbuf1: nextline buf1
# Pclbuf2: currline buf2
# Pconst: constants
# Tlast_to_front: pattern last to front
# Tfirst_pl_to_back: pattern first p-1 to back
# Tlast_pl_to_front: pattern last p-1 to front
# Tfirst_to_back: pattern first to back
# Ibufsrc: buf1 source address
# Ibuf2src: buf2 source address
# Ibuf2out: buf2 output address
# Xbufsrc: buf1 source address
# Xbuf2src: buf2 source address
# Xbuf2out: buf2 output address
# internal memory:
# 0..519: buf1; (Ibufsrc-Pplbuf1,Pclbuf1,Pnlbuf1)
# 520..649: buf2; (Ibuf2src-Pclbuf2,Ibuf2out-Pclbuf2)
# 4098: zero
# 4093: broadcast multipliers height loop (1/4, 63/64)
# 4094: broadcast multipliers bitmap loop (1/8192, 1/8192+linewidth, P, 8*linewidth,
#   MEM_GREEN, MEM_BLUE)
# 4095: broadcast multipliers bitmap loop (3/4, 16, 3/4, 16, 3/4, 16)
# external memory:
# 0..1: init pos
# 0, 1, 2, 3, 4, 5, 6, 7 = 1,13076,262149,393223
# 2..3: shuffle pattern Tlast_to_front
# 7,16,16,16,16,16,16,16 = 468766,1048592,1048592
# 4..5: shuffle pattern Tfirst_pl_to_back
# 16, 0, 1, 2, 3, 4, 5, 6 = 1048592,1048592,1048592
# 6..7: shuffle pattern Tlast_pl_to_front
# 1, 2, 3, 4, 5, 6, 7,16 = 65538,196612,32768
# 8..9: shuffle pattern Tfirst_to_back
# 16,16,16,16,16,16,16, 0 = 1048592,1048592,1048592
# 10..11: broadcast multipliers height loop
# 8192,32266,8192,32266,8192,32266,8192,32266 =
# 538903168,538903168,538903168,538903168
# 12..13: broadcast multipliers 1 bitmap loop
# 4,4096,8,8192,8192,8192,0,0 = 266240,532940,536879104,0
# 14..15: broadcast multipliers 2 bitmap loop
# 24576,16,24576,16,24576,16,0,0,0 = 1610612752,1610612752,1610612752
# 16..200215: height buffer 1 (((1024+2*P)*(768+2))/4 = 200200)
# 200216..400415: height buffer 2
# 400416..695327: background bitmap (3+1024=768/3)
# 695328..990239: output bitmap

LDB Bplbuf1, 520 ; ELI Iconst, 4088
LDB Eclbuf1, 520 ; ELX Xconst, 0
LDB Bnbuf1, 520 ; EIB Bbufsrc, 520
LDP Pconst, 4088 ; EIE Ebufsrc, 519
LDE Eplbuf1, 519 ; EIB Bbuf2src, 130
LDE Enbuf1, 519 ; ELD Iconst, Xconst, 1
LDB Eclbuf2, 130 ; ELI Iconst, Xconst, 1
LDE Eclbuf2, 649 ; ELI Iconst, Xconst, 1
NQP ; ELD Iconst, Xconst, 1
NQP ; ELI Iconst, Xconst, 1
NQP ; ELD Iconst, Xconst, 1
NQP ; ELD Iconst, Xconst, 1
NQP ; ELI Iconst, Xconst, 1
NQP ; ELI Iconst, Xconst, 1
NQP ; ELD Iconst, Xconst, 1
NQP ; ELI Xconst, Xconst, 1
# process height buffer
procevenframe:
NQP; ELX Xheight, 200216; D01 2, procoddframe, procframeend
NQP; ELX Xbuf2src, 200216; JR procframe
NQP; ELX Xbuf2out, 200476
NQP; ELX Xbuf1src, 16
procoddframe:
NQP; ELX Xbuf1src, 200216
NQP; ELX Xbuf2src, 16
NQP; ELX Xbuf2out, 276
NQP; ELX Xheight, 16
procframe:
LDP Pconst, 4093; ELI Dbuf1src, 0; D01 768, procheightbuffer, procheightbufferend
LDP Ppbuf1, 0; ELI Xbuf2src, 620; D01 134, procheight_firstbuf2, procheight_firstbuf2end
LDP Pclbuf1, 130; ELI Xbuf2out, 620; D01 393, procheight_firstbuf1, procheight_firstbuf1end
LDP Pnlbuf1, 260
LDP Pclbuf2, 521
LIM Pconst, 1
procheight_firstbuf1:
NQP; ELI Dbuf1src, Xbuf1src, 1
procheight_firstbuf1end:
NQP; ELI Dbuf1src, Xbuf1src, 1
procheight_firstbuf2:
NQP; ELI Xbuf2src, Xbuf2src, 1
procheight_firstbuf2end:
NQP; ELI Xbuf2src, Xbuf2src, 1

# prepare processing a line of height buffer
LDP Rplcp, Ppbuf1, 1; NQP; JR procheightlinefirst
LDP Rclcp, Pclbuf1, 1
LDP Rnlcp, Pnlbuf1, 1
procheightbuffer:
$ copy next2vector to curr, fetched in previous line
MOV Rplcp, RpIn2
MOV Rclcp, Rcln2
MOV Rnlcp, Rnln2
procheightlinefirst:
#   skip frame
LDP Rplnv, Ppbuf1, 1; ELI Dbuf1src, Xbuf1src, 1; D01 128, procheightlinestart, procheightlineend
LDP Rpln2, Ppbuf1, 1; ELI Dbuf1src, Xbuf1src, 1
LDP Rclnv, Pclbuf1, 1; ELI Xbuf2src, Xbuf2src, 1
LDP Rcln2, Pclbuf1, 1; ELI Xbuf2out, Xbuf2out, 1
LDP Rnlnv, Pnlbuf1, 1; ELI Xbuf2out, Xbuf2out, 1
LDP Rnln2, Pnlbuf1, 1; ELI Xbuf2out, Xbuf2out, 1
# inner loop of processing height buffer
# previous line
procheightlinestart:
LDP Roldbuf2, Pclbuf2, 0; NQP; NQP; NQP; SHF Rplpp, Rplcp, Tlast_to_front
MOV Rplcp, Rplnv ; NQP; NQP; NQP; SHF Rplpp, Rplnv, Tfirst_pl_to_back
MOV Rplnv, Rpln2 ; NQP; NQP; SHF Rplpp, Rplnv, Last_pl_to_front
LDP Rpln2, Ppbuf1, 1 ; NQP; NQP; SHF Rplpp, Rpln2, Tfirst_to_back
# current line
NQP; ELI Dbuf1src, Xbuf1src, 1; NQP; NQP; SHF Rclpp, Rclcp, Tlast_to_front
MOV Rclcp, Rclnv ; ELI Dbuf1src, Xbuf1src, 1; NQP; NQP; SHF Rclpp, Rclnv, Tfirst_pl_to_back
MOV Rclnv, Rcln2 ; ELI Xbuf2src, Xbuf2src, 1; NQP; NQP; SHF Rclpp, Rclnv, Last_pl_to_front
LDP Rcln2, Pclbuf1, 1; ELI Xbuf2src, Xbuf2src, 1; NQP; NQP; SHF Rclpp, Rcln2, Tfirst_to_back
# next line

81
NOP
MOV Rn1cp, Rninv
LDW Rn1in, Rn12
LDW Rn1in, Rn12, Tlast_to_front
# calculation
NOP; NOP; NOP; NOP; SHF Rn1pp, Rn1cp, Tlast_to_front
NOP; NOP; NOP; NOP; SHF Rn1pp, Rninv, Tfirst_pl_to_back
NOP; NOP; NOP; NOP; SHF Rn1pp, Rn1cp, Tlast_pl_to_front
NOP; NOP; NOP; NOP; SHF Rn1pp, Rn1cp, Tfirst_pl_to_back

# store
NOP
NOP
NOP
STV Roldbuf2, Pclbuf2, 1
NOP
NOP; BST Xbuf2out, Ibuf2out, 1
procheightlineend:
NOP; BST Xbuf2out, Ibuf2out, 1

# skip frame of zeroes
NOP
LDW Roldbuf2, Pclbuf2, 1; ELD Ibufisrc, Xbufisrc, 1
LDW Roldbuf2, Pclbuf2, 1; ELD Ibufisrc, Xbufisrc, 1
NOP; ELD Ibuf2src, Xbuf2src, 1
NOP; ELD Ibuf2src, Xbuf2src, 1
NOP; ELD Ibuf2out, Xbuf2out, 1
procheightbufferend:
NOP; ELD Ibuf2out, Xbuf2out, 1

# Rplcp: prevlinecurrpixel
# Rclpp: currlineprevpixel
# Rc1cp: currlinecurrpixel
# Rclpp: currlineprevpixel
# Rclpp: currlineprevpixel
# Rclpp: currlineprevpixel
# Rclpp: currlineprevpixel
# Rplcp: nextlinecurrpixel
# Roffset: Xoffset
# Ryoffset: Yoffset
# Rshade: shade
# Rgather: srcGather
# Rpos: pos in buffer
# Roffset: 0,1,2,...,P-1
# Rbackclr: backcolor
# Plheight: -> Rplcp-alc; prevline height: 3072..3691 (4*(1024/P-2))
# Pclheight: -> Rplcp-alc; currline height
# Flheight: -> Rplcp-alc; nextline height
# Poutred: <- Rbackclr; result red
# Poutgreen: <- Rbackclr; result green
# Poutblue: <- Rbackclr; result blue
# Ibackred: -> Rbackclr; back red: 0..1023 (8*1024/P)
# Ibackgreen: -> Rbackclr; back green: 1024..2047
# Ibackblue: -> Rbackclr; back blue: 2048..3071
# Ioutred: <- Poutred; result red: 3692
# Ioutgreen: <- Poutgreen; result green: 3693

82
probitmap:
LDW PFillHeight, 3072; EIW EHeight, 520
LDW PChHeight, 3202; EIW EHeight, 3591
LDW PLineHeight, 3332; ELI IHeight, 3072
LDW PLineWidth, 520; ELI IBackRed, 0
LDW PLineHeight, 3694; EIW EBackRed, 1023
LDW PChHeight, 520; EIW EBackGreen, 1024
LDW EChHeight, 3694; EIW EBackGreen, 1024
LDW EnHeight, 520; EIW EBackBlack, 1024
LDW EnHeight, 3694; EIW EBackGreen, 2047
LDW POutRed, 3692; EIW EBackBlue, 2048
LDW POutGreen, 3693; EIW EBackBlack, 1024
LDW POutBlue, 3694; EIW EBackBlue, 3071
LDW PConst, 1; ELX XBack, 400416
MIV Rpos, RInitPos; ELX OutBitmap, 696328
NOP; ELI IOutRed, 3692
NOP; EIW IOutRed, 1
NOP; EIW IOutGreen, 3692
NOP; ELI IOutGreen, 3693
NOP; EIW IOutGreen, 1
NOP; EIW IOutGreen, 3693; DOJ 96, probitmapbuffer, probitmapbufferend
NOP; EIW IOutBlue, 3694; DOJ 393, probitmap_firstlines, probitmap_firstlinesend
NOP; EIW IOutBlue, 1
NOP; EIW IOutBlue, 3694
probitmap_firstlinesend:
NOP; ELI IHeight, XHeight, 1
NOP; ELI IHeight, XHeight, 1
NOP; ELB Xback, 1
NOP; ELB Xback, 1
NOP; ELB Xback, 1
NOP; ELB Xback, 1
NOP; ELB Xback, 1
probitmap_firstlines:
NOP; LDW TheHeight, XHeight, 1
MIV Rpos, RInitPos; DOJ 8, probitmaplineblockstart, probitmaplineblockend
NOP; PLoad first height vectors
LDW R1cp, PFillHeight, 1; NOP; JR probitmaplinefirst
LDW RChp, PChHeight, 1; NOP; DOJ 8, probitmaplineblockstart, probitmaplineblockend
LDW RAnp, PLineHeight, 1
MIV Rpos, RInitPos; DOJ 128, probitmaplinestart, probitmaplineend
probitmaplinefirst:
LDW RChp, PChHeight, 1; ELI IHeight, XHeight, 1; DOJ 128, probitmaplinestart, probitmaplineend
MIV Rpos, RInitPos; ELI IHeight, XHeight, 1
NOP; inner loop of processing height buffer
MIV Rpos, RInitPos; DOJ 8, probitmaplineblockstart, probitmaplineblockend
LDW R1cp, PFillHeight, 1; ELB XBack, 1
MIV Rpos, RInitPos; SHF Rlcpp, Rlcp, TIastToFront
MIV Rlcp, RChp; ELB XBack, 1
MIV Rlcp, RChp; ELB XBack, 1
MIV Rlcp, RChp; ELB XBack, 1
MIV Rlcp, RChp; ELB XBack, 1
MIV Rlcp, RChp; ELB XBack, 1
MIV Rlcp, RChp; ELB XBack, 1
MIV Rlcp, RChp; ELB XBack, 1
MIV Rlcp, RChp; ELB XBack, 1
MIV Rlcp, RLine, TIastPlToBack
NQP ; ELC Ibackblue, 1
NQP
NQP
NQP
# XOffset=(Array[-i]-Array[i])
NQP ; ELD Iheight, Xheight, 1; NQP; VSUB Roffset, Rclpp, Rclnp
# YOffset=(Array[SCREEN_W]-Array[SCREEN_W])
NQP ; ELD Iheight, Xheight, 1; NQP; VSUB Ryoffset, Rplcp, Rnlcp
# shade = (Xoffset + YOffset) * 2
NQP ; NQP; NQP; ADD Rshade, Ryoffset, Roffset
NQP ; NQP; NQP; ADD Rshade, Rshade, Rshade
# gatherPos = pos + (Xoffset >> 13)
LDA Rpos ; NQP; NQP; BMAC Rgather, Roffset; NQP; RMB
# gatherPos = (YOffset >> 13) + linewidth
NQP ; NQP; NQP; BMAC Rgather, Ryoffset; NQP; RMB
# pos += P
NQP ; NQP; NQP; ADD Rpos, Rpos ; NQP; RMB
# bound checking
NQP ; NQP; NQP; VSBC Rnewbuf2, Rgather, Rgather
NQP ; NQP; NQP; VSUB Rgather, Rgather, Rnewbuf2
NQP ; NQP; NQP; BSUB Rnewbuf2, Rgather
NQP
NQP
NQP ; NQP; NQP; BMAC Rgather, Rgather; NQP; RMB
NQP
NQP
NQP ; NQP; NQP; BMAC Rgather, Rgather; NQP; RMB
# offset to red to blue pixels
LIM Pconst, 1
SOG Rbackred, Rgather
NQP ; NQP; NQP; BMAC Rbackred, Rbackred ; NQP; RMB
NQP
NQP
# offset to blue to green pixels
NQP
SOG Rbackgreen, Rgather
NQP
NQP
SOG Rbackblue, Rgather
NQP
NQP
NQP
# red = shade + red * 3/4
LDA Rshade; NQP; NQP; BMAC Rbackred, Rbackred ; NQP; RMB
NQP
NQP
# green = shade + green * 3/4
LDA Rshade; NQP; NQP; BMAC Rbackgreen, Rbackgreen; NQP; RMB
NQP ; NQP; NQP; BMAC Rbackgreen, Rbackgreen; NQP; RMB
NQP
# blue = shade + blue * 3/4
LDA Rshade; NQP; NQP ; BMAC Rbackblue, Rbackblue ; NQP; RMB
STV Rbackred, Poutred, O; NQP; NQP; BMAC Rbackblue, Rbackblue; NQP; RMB
LIM Pconst, 1
STV Rbackgreen, Poutgreen, 0
NQP
# store to buffer and external
NQP ; ESC Ioutred, 1
STV Rbackblue, Poutblue, O; ESC Ioutgreen, 1
NQP ; ESC Ioutblue, 1
NQP; ESB Ioutbitmap, 1
NQP; ESB Ioutbitmap, 1
NQP; ESB Ioutbitmap, 1
probitmaplineend:
NQP
LDV Rplcp, Pplheight, 1; ELD Iheight, Xheight, 1

84
procbitmaplineblockend:
LDW Rn1cp, Pnlheight, i; ELD Iheight, Xheight, i
procbitmapbufferend:
NOP

procframeend:
NOP
NOP; NOP; JR procevenframe
NOP
NOP

85
Bibliography


[12] Adrian Hey, “The FFT Demystified”
    http://www.epTools.com/tn/T0001/INDEX.HTM
    General FFT links page: http://www.fftw.org/links.html


[14] Hugo Elias, “2D water”
    http://freespace.virgin.net/hugo.elias/graphics/x_water.htm