Design and synthesis of a high-performance, hyper-programmable DSP on an FPGA

Stephen Nichols

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Design and Synthesis of a High-Performance, Hyper-Programmable DSP on an FPGA

by

Stephen W. Nichols

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Assistant Professor Dr. Andres Kwasinski
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
April 2011

Approved By:

Dr. Andres Kwasinski
Assistant Professor, Department of Computer Engineering
Primary Adviser

Dr. Marcin Lukowiak
Assistant Professor, Department of Computer Engineering

Dr. Antonio F. Mondragon
Assistant Professor, Department of Computer Engineering Technology
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Design and Synthesis of a High-Performance, Hyper-Programmable DSP on an FPGA

I, Stephen W. Nichols, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or in part.

Stephen W. Nichols

Date
Abstract

In the field of high performance digital signal processing, DSPs and FPGAs provide the most flexibility. Due to the extensive customization available on FPGAs, DSP algorithm implementation on an FPGA exhibits an increased development time over programming a processor. Because of this, traditional DSPs typically yield a faster time to market than an FPGA design. However, it is often desirable to have the ASIC-like performance that is attainable through the additional customization and parallel computation available through an FPGA. This can be achieved through the class of processors known as hyper-programmable DSPs.

A hyper-programmable DSP is a DSP in which multiple aspects of the architecture are programmable. This thesis contributes such a DSP, targeted for high-performance and realized in hardware using an FPGA. The design consists of both a scalar datapath and a vector datapath capable of parallel operations, both of which are extensively customizable. To aid in the design of the datapaths, graphical tools are introduced as an efficient way to modify the design. A tool was also created to supply a graphical interface to help write instructions for the vector datapath. Additionally, an adaptive assembler was created to convert assembly programs to machine code for any datapath design.

The resulting design was synthesized for a Cyclone III FPGA. The synthesis resulted in a design capable of running at 135MHz with 61% of the logic used by processing elements. Benchmarks were run on the design to evaluate its performance. The benchmarks showed similar performance between the proposed design and commercial DSPs for the simple benchmarks but significant improvement for the more complex ones.
Contents

Abstract ........................................................................................................ iii

Glossary ........................................................................................................ xi

1 Introduction .............................................................................................. 1
  1.1 Motivation ............................................................................................ 1
  1.2 Contribution ....................................................................................... 3
  1.3 Thesis Organization ........................................................................... 3

2 Related Work ........................................................................................... 4
  2.1 Hyper-Programmable Processors ....................................................... 4
    2.1.1 Altera Nios II ............................................................................... 4
    2.1.2 EPIC ............................................................................................ 6
    2.1.3 CUSTARD .................................................................................. 8
  2.2 Transport Triggered Architecture ...................................................... 10
    2.2.1 FFTTA ...................................................................................... 10
    2.2.2 TTA-based Co-design Environment .......................................... 11
  2.3 NoC-based Datapaths ......................................................................... 12
  2.4 Commercial DSPs ............................................................................. 13
    2.4.1 Analog Devices TigerSHARC .................................................... 14
    2.4.2 Texas Instruments TMS320C6746 ............................................. 15

3 Design ...................................................................................................... 17
  3.1 Vector Processor .................................................................................. 18
    3.1.1 Vector Datapath .......................................................................... 19
    3.1.2 Status Table .............................................................................. 26
    3.1.3 Operand Control ....................................................................... 28
  3.2 Scalar Processor .................................................................................. 29
    3.2.1 Instruction Fetch ........................................................................ 30
# List of Figures

1.1 DSP implementation spectrum ................................................. 2

2.1 Block diagram of the Nios II architecture [7] .............................. 5
2.2 Block diagram of C2H accelerated Nios II [3] ............................. 6
2.3 Organization of EPIC architecture [14] .................................... 7
2.4 Organization of CUSTARD architecture [16] ............................. 8
2.5 Organization of a TTA [15] .................................................. 10
2.6 Block diagram of the FFTTA architecture [23] ......................... 11
2.7 TCE’s graphical processor designer tool [26] ............................. 12
2.8 Block diagram of the IPNoSys architecture [19] ....................... 13
2.9 Block diagram of the TigerSHARC architecture [8] ................. 14
2.10 Block diagram of the TMS320C6746 DSP architecture [31] .... 15

3.1 Design overview ............................................................ 17
3.2 Vector processor overview .................................................. 18
3.3 Example vector datapath .................................................... 19
3.4 Bus definition ............................................................... 20
3.5 Bus flow-control bits ....................................................... 20
3.6 Header word definitions .................................................. 20
3.7 Wrapper takes a packet (top) and simplifies it (bottom) for a processing element ......................................................... 22
3.8 Picking off local opcode bits .............................................. 23
3.9 Picking off local parameter words ........................................ 23
3.10 Bidirectional bus register ................................................ 25
3.11 Vector datapath status table .......................................... 26
3.12 Timing diagram of a status table entry .................................. 27
3.13 Overview of operand control module ................................ 28
3.14 Scalar processor overview ............................................... 29
3.15 Scalar processor pipeline flow ........................................ 30
3.16 Instruction fetch design .................................................. 31
3.17 Instruction decode design
3.18 Scalar opcode format
3.19 Scalar processing element - adder
3.20 Scalar processing element - vector datapath control

4.1 Design flow
4.2 Screenshot of Vector Datapath Designer tool
4.3 Generics in FFT element
4.4 Screenshot of Scalar Datapath Designer tool
4.5 Entity declaration for adder element
4.6 Screenshot of Vector Datapath Programmer tool
4.7 FFT opcode definitions
4.8 Assembly from example vector instruction
4.9 Screenshot of Scalar Assembler tool
4.10 Example assembly program

5.1 Overview of the test setup
5.2 Overview of the simulation environment
5.3 Overview of the hardware test environment
5.4 Example vector datapath
5.5 Band-pass and band-stop windows

6.1 Logic elements (left) and RAM (right) used by the design
6.2 Logic elements (left) and RAM (right) used by the vector processor
6.3 Logic elements (left) and RAM (right) used by the scalar processor
6.4 Vector Search instruction
6.5 Assembly listing for 2k vector search
6.6 Execution time for 2048 element search benchmark
6.7 Vector Multiply instruction 1
6.8 Vector Multiply instruction 2
6.9 Execution time for 2048 element vector multiply benchmark
6.10 Accuracy of 2048 element vector multiply
6.11 FFT instruction
6.12 Execution time for 2048 point FFT benchmark
6.13 Execution time for 1024 point FFT benchmark
6.14 Execution time for 256 point FFT benchmark
6.15 Accuracy of 2048 point FFT ........................................ 72
6.16 Overlap-save filtering, \( y(n) = x(n) * h(n) \) ........................... 73
6.17 FIR instruction 1 ..................................................... 73
6.18 FIR instruction 2 ..................................................... 74
6.19 FIR instruction 3 ..................................................... 74
6.20 FIR instruction 4 ..................................................... 75
6.21 Execution time for \( N=10000, M=10 \) FIR benchmark ........ 76
6.22 Execution time for \( N=10000, M=100 \) FIR benchmark ....... 77
6.23 Execution time for \( N=10000, M=512 \) FIR benchmark ...... 77
6.24 Accuracy of \( N=10000, M=100 \) FIR ................................. 78
6.25 Power consumption of each DSP .................................. 79

A.1 Header word definitions ............................................. 88
List of Tables

2.1 Nios II 256-point FFT performance [3] . . . . . . . . . . . . . . . . . . . . 6
3.1 List of port signals for VHDL processing element cores . . . . . . . . . . 24
3.2 Port signals for VHDL scalar processing elements . . . . . . . . . . . . . 33
4.1 Outline of Vector Datapath Designer VHDL generation . . . . . . . . . . 38
4.2 Additional port signals for VHDL scalar instruction fetch modules . . . . 40
4.3 Outline of Scalar Datapath Designer VHDL generation . . . . . . . . . . 42
4.4 Vector Datapath Programmer display directives . . . . . . . . . . . . . . 46
5.1 UART to DSP interface commands . . . . . . . . . . . . . . . . . . . . . 54
5.2 Elements in the example scalar datapath . . . . . . . . . . . . . . . . . . 57
6.1 Resource usage of synthesized design . . . . . . . . . . . . . . . . . . . . 59
6.2 Set of benchmark programs . . . . . . . . . . . . . . . . . . . . . . . . . 62
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit.</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit.</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing.</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor.</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform.</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out.</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response.</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array.</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit.</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output.</td>
</tr>
<tr>
<td>MATLAB</td>
<td>MATrix LABoratory.</td>
</tr>
<tr>
<td>MEX</td>
<td>Matlab EXecutable.</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip.</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation.</td>
</tr>
<tr>
<td>OFDM</td>
<td>Orthogonal Frequency-Division Multiplexing.</td>
</tr>
<tr>
<td>PLL</td>
<td>Phase-Locked Loop.</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory.</td>
</tr>
<tr>
<td>SDR</td>
<td>Software Defined Radio.</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data.</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to Noise Ratio.</td>
</tr>
<tr>
<td>TTA</td>
<td>Transport Triggered Architecture.</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

In the field of high performance digital signal processing (DSP), digital signal processors (DSPs) and field programmable gate arrays (FPGAs) provide the most flexibility. Due to the extensive customization available on FPGAs, DSP algorithm implementation on an FPGA exhibits an increased development time over programming a DSP. Because of this, traditional DSPs typically have a faster time to market than an FPGA design. However, it is often desirable to have the increased performance that is attainable through the additional customization and parallel computation available through an application-specific integrated circuit (ASIC) or FPGA. It would therefore be advantageous to introduce a new class of processor called hyper-programmable DSPs. A traditional spectrum of DSPs consisting of four categories was given in [27]. Figure 1.1 depicts this spectrum with the addition of hyper-programmable DSPs. The vertical axis in the diagram depicts the range of flexibility achievable through software. As programmability increases, cost and development time decrease. The horizontal axis shows the range of specialization of each technology. As specialization increases, performance and efficiency also increase. By comparing these aspects, the basic tradeoffs between the DSPs in the spectrum can be seen. [27]
The term “hyper-programmable” has been used to describe an architecture where multiple aspects of the architecture are programmable, rather than just supporting a programming model [12]. In other words, a hyper-programmable processor’s datapath is customizable, although not necessarily at run-time. Such an architecture can be realized through the use of a customizable processor programmed on an FPGA.

One DSP application that requires high performance, flexibility, and low power usage is software-defined radio (SDR). Although standard DSP processors are flexible, they often are unable to achieve high performance with low power dissipation [13]. ASIC processors do not provide the flexibility required to support variations of implemented wireless algorithms as is expected from SDRs [13]. A hyper-programmable DSP, on the other hand, can be customized to meet these requirements. Additionally, the ability to quickly reconfigure this type of processor makes it an excellent fit for the ever-changing domain of wireless communication.
1.2 Contribution

This thesis contributes a high-performance, hyper-programmable DSP design realized in hardware using an FPGA. To achieve high performance, the design consists of both a scalar datapath and vector datapath capable of parallel operations. Both datapaths are extensively customizable at compile-time, thus providing hyper-programmability. To aid in the design of the datapaths, graphical tools were developed as an efficient way to modify the design. A tool was also created to supply a graphical interface to help write instructions for the vector datapath. Additionally, an adaptive assembler was created to convert assembly programs to machine code for any datapath design.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 lays the groundwork for the thesis by discussing other processor designs related to the proposed DSP. Chapter 3 goes into detail on the design and architecture of the proposed processor. The supporting toolset consisting of datapath design tools, a graphical programming tool, and an assembler is presented in Chapter 4. Chapter 5 provides details about both the simulation and hardware testbench environments. The results of the design synthesis along with performance of a set of benchmark programs are given and discussed in Chapter 6. Chapter 7 concludes this thesis with suggestions for future research and development.
Chapter 2

Related Work

One realization of a hyper-programmable processor is through the use of a customizable soft processor core on an FPGA. Examples of both commercial and academic implementations of this are presented in the following section. TTA (Transport Triggered Architecture) processors are then discussed as an architecture that can provide a foundation for hyper-programmability. The IPNoSys architecture, which uses packets to encapsulate instructions and data to be processed in an NoC (Network-on-Chip) datapath, is then presented. Lastly, an overview of commercial high-performance and low-power DSPs is given.

2.1 Hyper-Programmable Processors

2.1.1 Altera Nios II

The Nios II is a 32-bit general-purpose RISC processor core designed by Altera for use in FPGA designs [7]. A block diagram of the processor is depicted in Figure 2.1. The gray sections of the figure represent optional modules while the white sections represent required ones.

As shown in the figure, there are a variety of optional modules to choose from when customizing the Nios II. A number of the options are used to enhance memory operations through caches and memory management modules. If context switching is important, shadow registers can be added to accelerate the operation. The primary way to improve
performance, however, is through the use of custom instructions. The Nios II architecture supports up to 256 user-defined custom instructions. The logic for the custom instructions connects directly to the processor’s ALU which allows hardware operations to be accessed and used exactly like native instructions. [7]

In addition to the custom instructions, unlimited memory mapped hardware accelerators can be added to the system. Accelerators can directly access memory and other system resources which yields high bandwidth. Hardware accelerators can be created manually using VHDL or automatically using the Nios II C-to-Hardware Acceleration (C2H) Compiler. [7]

The C2H Compiler takes specified portions of the C code to be run on the Nios II and creates logic that accelerates software algorithms [5]. This allows the performance of Nios II programs to be significantly improved without any manual logic design. An example of using the C2H Compiler to speed up an FFT (Fast Fourier Transform) algorithm is given in
The resulting hardware accelerator is depicted in Figure 2.2. As shown, the accelerator logic has direct memory access to achieve higher performance.

![Diagram of C2H accelerated Nios II](image)

To demonstrate the performance speedup, results from both the original and optimized designs running at 100MHz were given in [3]. The benchmark computed the FFT algorithm on 256 points of real data. The results from both designs are shown in Table 2.1. From these results, it can be seen that the C2H compiler can provide a significant performance improvement while foregoing much additional development time.

<table>
<thead>
<tr>
<th></th>
<th>Cycles</th>
<th>Execution Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Only</td>
<td>87,767</td>
<td>877.675 µs</td>
<td>1</td>
</tr>
<tr>
<td>Hardware Accelerated</td>
<td>5,272</td>
<td>52.719 µs</td>
<td>16.65</td>
</tr>
</tbody>
</table>

Table 2.1: Nios II 256-point FFT performance [3]

### 2.1.2 EPIC

The Explicitly Parallel Instruction Computing (EPIC) processor is an implementation of a hyper-programmable processor designed for academic exploration of various performance/area tradeoffs. The design supports precompilation customization of the following parameters [14]:

- [3]
- number of ALU units
- number of general purpose registers
- number of predicate registers
- number of branch target registers
- number of registers each instruction can use
- number of instructions per issue
- width of datapath and registers
- functionality of ALU.

Figure 2.3: Organization of EPIC architecture [14]

As shown in 2.3, the datapath implementation is split into two pipeline stages. In the first stage the Fetch/Decode/Issue Unit decodes and issues multiple instructions per cycle.
Two operand registers for each instruction are fetched from the general purpose register file. The second stage consists of a collection of execution and control units: arithmetic and logic units (ALUs), a load/store unit (LSU), a comparison unit (CMPU), a branch unit (BRU), and a branch target register (BTR). Because multiple instructions are issued simultaneously, several of these units can be used concurrently. The outputs of the computational units are connected to the Write Back Unit to handle writing each result back to the register file. [14]

EPIC programs are written in C and then compiled using a custom compiler. Along with C code, information regarding specific processor customizations is fed as input to the compiler. In addition to performing machine independent optimizations, the compiler statically schedules instructions by performing dependence analysis and resource conflict avoidance. This moves the complexity of controlling the usage of each resource from hardware to the compiler. [14]

2.1.3 CUSTARD

CUSTARD (CUStomizable Threaded Architecture) is a hyper-programmable processor that supports multiple hardware threads and automatic instruction set customization. The
processor design, shown in Figure 2.4, is based on a four-stage pipelined architecture. By providing multiple banks of registers, the register file adds support for multiple contexts within the same processor hardware. Each context stores state information of registers, the stack, and the program counter for each thread. Because the context switch happens at the hardware level, multiple threads can be run concurrently with little to no overhead. Additionally, context switches can be used to hide latency which normally causes a stall.  

As a hyper-programmable processor, CUSTARD supports customizations at time of synthesis of a number of parameters  

- number of threads  
- threading type (block or interleaved)  
- custom instructions  
- forwarding and interlock architecture  
- number of registers  
- number of register file ports  
- register bitwidth.

CUSTARD programs are written in C and compiled using an optimizing compiler. In addition to producing optimized machine code, the compiler also generates custom instructions using a technique dubbed “Similar Sub-Instructions.” This technique finds instruction datapaths that can be reused across similar portions of code. These datapaths are then added as custom execution units to the processor in addition to updating the decoding logic to map new instructions to opcodes reserved for custom instructions.
2.2 Transport Triggered Architecture

One architecture of interest is the Transport Triggered Architecture (TTA). A TTA is programmed by describing the transport of data between function units rather than just operations of function units. The general organization of a TTA is shown in Figure 2.5. As depicted in the diagram, TTAs are organized as a set of functional units (FUs) and register units (RUs). Each of the function units performs an operation as a consequence of data being transported to one of its input ports. The result from the computation is then stored on a register connected to its output port. The interconnection network provides the means to transport data among functional and register units. The interconnect consists of one or more busses whose operation is controlled by instruction words. Since instructions only specify where to move data from and to, the entire instruction set consists of a single instruction: a move instruction. Because of its simple, modular design, a TTA provides an excellent candidate for a hyper-programmable processor design. [15]

![Figure 2.5: Organization of a TTA](image)

2.2.1 FFTTA

One example of a customized TTA is the FFTTA processor which customizes a TTA to efficiently perform the FFT algorithm. The general organization of the design is shown in Figure 2.6. The interconnect network for the design consists of 18 busses used to provide a variety of connections between the 8 functional units and 11 register units. The functional
units in the design include a complex adder, a complex multiplier, a data address generator, a coefficient generator, a real adder, a load-store unit, a comparator unit, and a control unit. The register units provide a total of twenty-three 32-bit general purpose registers. The interconnection among function and register units was minimized on each bus to provide an optimized architecture for the specific application. This results in reduced area and power consumption without sacrificing any performance. [23]

Figure 2.6: Block diagram of the FFTTA architecture [23]

2.2.2 TTA-based Co-design Environment

At Tampere University of Technology, TTAs have been the focus of a number of research projects since 2003 [26]. Researchers at the university developed and maintain a toolset for designing application specific processors based on the TTA architecture. This toolset is called the TTA-based Co-design Environment (TCE). TCE provides a complete co-design flow from C programs down to synthesizable VHDL and parallel program binaries. The tools allow processor customization of the register files, function units, and the interconnection network. TCE includes graphical customization tools, processor generator, C compiler, and a simulator. A screenshot of the processor generator is shown in Figure [2.7] [22]
2.3 NoC-based Datapaths

Networks-on-Chip (NoC) enable parallel communication between cores using messages encapsulated in packets. Typically an NoC design consists of a set of routers interconnected with other routers and end nodes such as processor cores. Normally the routers are only responsible for the data transmission. However, by adding processing elements to the routers, the network becomes a datapath. The advantage of this architecture is that it creates a modular and thus scalable design. [19]

The IPNoSys architecture is an example of a processor using an NoC-based datapath. An overview of the design is shown in Figure 2.8. As shown in the diagram, the processor consists of four memory banks (Mem), four memory access units (MAU), and sixteen routing and processing units (RPU). [19]

Instructions and operands are read from memory and encapsulated in packets by an MAU and injected into the datapath. Each packet is made up of multiple message words consisting of header words, instruction words, operand words, and a terminator word. In addition to producing packets, MAUs can also write result words from processed packets.
back into memory. [19]

The RPU’s are made up of a packet buffer, a crossbar switch, an ALU, and control logic. As an RPU receives a packet, it places it into a packet buffer. If the packet has any instruction words, the first instruction in the packet is then executed in the ALU. The resulting value from the operation is then inserted into the packet as an operand for a future instruction. The current instruction and associated operand words are then removed from the packet. The packet is then routed to one of the neighboring RPU’s where the next instruction can be processed. This results in a packet flowing from RPU to RPU until the final result is computed and written back to memory. [19]

2.4 Commercial DSPs

Since there are many DSPs commercially available targeting general purpose signal processing, it makes sense to detail and compare a few. Two high-end DSPs with floating-point support were selected to represent the following classes:
• Performance optimized DSPs
• Power optimized DSPs.

2.4.1 Analog Devices TigerSHARC

The TigerSHARC DSP is described by Analog Devices as an “ultrahigh performance, static superscalar processor optimized for large signal processing tasks and communications infrastructure” [8]. A block diagram of the architecture is shown in Figure 2.9.

![Figure 2.9: Block diagram of the TigerSHARC architecture][8]

The design features dual computational blocks that support IEEE 32-bit single-precision floating-point, extended-precision 40-bit floating point, and 8-, 16-, 32-, and 64-bit fixed-point processing. Each computational block is composed of 32-bit registers, a multiplier, an ALU, a shifter, and a Viterbi decoder. Compute instructions can be independently issued to two modules within a compute block in a single cycle. This means that between the two computational blocks, up to four instructions can be issued per cycle. When using
fixed-point arithmetic, SIMD (Single Instruction, Multiple Data) instructions can be used to operate on eight 8-bit, four 16-bit, or two 32-bit words simultaneously. \[8\]

### 2.4.2 Texas Instruments TMS320C6746

The TMS320C6746 is a low-power DSP with support for floating-point computations. A block diagram of the DSP core is shown in Figure [2.10](#).

![Figure 2.10: Block diagram of the TMS320C6746 DSP architecture](#)

The design features dual datapaths each with an ALU (.L), a shifter unit (.S), a multiply unit (.M), and a data unit (.D). The ALU unit is capable of up to four single precision additions every cycle or four double precision additions every two cycles. The shifter unit provides support for two floating point reciprocal calculations per cycle. The multiplier can perform two single precision multiplications per cycle or two double precision multiplications every four cycles. The data unit provides the means to load data from memory to the
register file and store results from the register file back into memory. [29]
Chapter 3

Design

The primary design objectives for the architecture are hyper-programmability and DSP performance. To achieve hyper-programmability, the architecture must support the addition and modification of computational blocks. To achieve high performance, the framework around the computational blocks must be capable of processing instructions and moving data quickly.

![Figure 3.1: Design overview](image)

The processor design is broken up into two main parts as shown in Figure 3.1. The vector datapath operates on blocks of data and is responsible for performing the bulk of
the computations. The scalar processor sequences instructions to the vector processor, handles external communication, and performs non-vector operations. The scalar and vector processors are discussed in detail in the following sections.

3.1 Vector Processor

In order to achieve the goal of high-performance, computation must dominate instruction overhead. Because many DSP algorithms work on blocks of data at a time [33], the overhead of instructions can be reduced by processing vectors up to thousands of data elements long. An overview of the vector processor design is shown in Figure 3.2. Vector instructions are streamed by the scalar processor into the operand control modules. These modules then create and transmit a packet containing instruction information and vector data into the vector datapath. The vector datapath receives these packets and processes them to produce the computational result. The resulting vector data is then written back into a memory bank. The status table keeps track of packets as they enter and exit the datapath, providing useful information for scheduling upcoming instructions.

Figure 3.2: Vector processor overview
3.1.1 Vector Datapath

The vector datapath consists of three types of components: routing elements, processing elements, and interconnects. Routing elements (depicted in Figure 3.3 as white polygons) are used to direct packets through the datapath. Examples of routing elements include inputs, multiplexers, demultiplexers, crossbar switches, forks, and outputs. Processing elements perform the computations and thus are the fundamental building blocks of the datapath. Examples of processing elements are shown in Figure 3.3 as gray polygons. Interconnects represent data busses and are used to connect elements together. The interconnects are depicted as lines in Figure 3.3 connecting the routing elements and processing elements.

Interconnects

Interconnects are used to transport packets containing instruction and vector data. This happens on a 40-bit bus which is broken up into control and data sections as shown in Figure 3.4. The bottom four bits are dedicated to controlling the flow of data across the bus. The protocol used is a derivative of the Avalon Streaming Interface [6]. As shown in Figure 3.5, one bit is used for reverse flow control and three bits are used for forward flow control. If the “hold” flag is raised, the flow of data across the bus must immediately halt.
until the flag is lowered. The “start of packet” flag is pulsed high during the same cycle as
the first valid word of the packet. Likewise, the “end of packet” flag is pulsed high during
the same cycle as the last valid word of the packet. The “data valid” flag is raised for every
word in the packet. Forward flow can be halted by lowering the “data valid” flag during
a packet’s transmission. The data portion was selected to be 36-bits wide since this is the
maximum width to which a Cyclone III block [RAM] can be configured [4].

Packets are made up of header words and vector data. Header words are used to store
opcode information, routing information, and writeback information. Each type of header
has a 4-bit header ID code associated with it as shown in Figure 3.6. The first word in a
packet must always be a header word. The last header word in the packet must have the ID
code 0xF indicating that the remaining words in the packet are all data. A bit definition of
each header word type is given in Figure A.1.

As shown in Figure 3.6, the chosen datatype is complex floating point. Since the data-
path targets DSP computations, the data must be complex, thus having both a real and
imaginary part. Floating point was selected in order to achieve a high dynamic range while maintaining consistent precision. Although 32-bit floating point has become the single precision standard, using fewer bits can still provide both sufficient precision and dynamic range [17]. The real and imaginary parts of the floating point number are fractional numbers with one sign bit and fourteen precision bits. The exponent is a signed 6-bit number, thus resulting in a dynamic range of \( 2^6 \times 20\log_{10}(2) \approx 385dB \) [17].

**Processing Elements**

Processing elements can perform a variety of computations on vector data including

- Vector addition
- Windowing
- Complex conjugation
- Element-wise multiplication
- Dot product
- Fourier transform

Each element has one or more operand inputs and generates a result on a single bus. Processing elements can have parameterizable operations which are controlled by the opcode section of the incoming packet. To maintain the desired performance, processing elements must be capable of consuming and producing a new vector element every clock cycle. Some processing elements, such as the FFT, might be too complex to complete in a single clock cycle and thus need to be broken up into multiple pipeline stages.

In order to provide a simple interface for processing element development, a wrapper is placed around each processing element core. This wrapper parses the incoming packet collecting the information pertinent to the local element. It then passes just the data along
Figure 3.7: Wrapper takes a packet (top) and simplifies it (bottom) for a processing element with adjusted flow-control signals to the processing element as shown in Figure 3.7. Header words are delayed and used by the wrapper to reconstruct the outgoing packet.

The wrapper is also responsible for presenting opcode bits and parameter words to the processing element. Each processing element must predefine the number of opcode bits it expects. When an instruction is issued, these opcode bits are concatenated to form one or more header words with the opcode. When a wrapper module receives the packet, it takes the lowest bits and shifts the opcode words to remove the local opcode and prepare the packet for the next element. An example of this is shown in Figure 3.8.

In addition to receiving opcode bits, processing elements can also receive parameter words. Parameter words are used for runtime parameters since opcode bits only provide compile-time instruction. The wrapper uses bits in the opcode to enable or disable the reception of parameter words. When enabled, the element can update any of its local parameters. If disabled, the parameter value passed to the processing element core will
remains unchanged since the wrapper registers all incoming parameters. After collecting local opcodes, the wrapper erases any parameters locally used from the packet as shown in Figure 3.9. The port signals provided to a processing element by the wrapper are shown in Table 3.1.

Since many processing elements operate on multiple input data vectors simultaneously, it is important that the data sections of incoming packets are aligned in time. For example, if a processing element begins to receive data for operand X but not for operand Y, then it must stall the data on the X bus until data is also available on the Y bus. The wrapper handles this by raising the reverse flow-control hold flag to pause the X bus until Y data is ready. This acts to align the data portion of packets across all the processing element...
<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>in std_logic</td>
<td>Provides the system clock signal.</td>
</tr>
<tr>
<td>reset</td>
<td>in std_logic</td>
<td>Provides a system reset signal.</td>
</tr>
<tr>
<td>opcode_NAME[i]</td>
<td>in std_logic in std_logic_vector in unsigned in signed</td>
<td>Optional. Local opcodes can be split up into multiple parts each with a unique name.</td>
</tr>
<tr>
<td>param_NAME[i]</td>
<td>in std_logic_vector in unsigned in signed</td>
<td>Optional. Any number of parameters can be received.</td>
</tr>
<tr>
<td>ctrl</td>
<td>in vdpStreamFwdCtrl</td>
<td>Flow-control flag from incoming packets is provided as shown in Figure [3.7].</td>
</tr>
<tr>
<td>hold</td>
<td>in std_logic</td>
<td>A hold flag is provided for reverse-flow control. If raised, all processing must immediately halt.</td>
</tr>
<tr>
<td>in[i].data</td>
<td>in vdpData</td>
<td>There is one incoming vector data port for each incoming operand packet.</td>
</tr>
<tr>
<td>in[i].hold</td>
<td>out std_logic</td>
<td>Optional. Internal processing can halt incoming data by asserting this flag until it is ready for more data.</td>
</tr>
<tr>
<td>out0.data</td>
<td>out vdpData</td>
<td>Optional. The resulting data vector is passed out through this port.</td>
</tr>
<tr>
<td>out0.ctrl</td>
<td>out vdpStreamFwdCtrl</td>
<td>Optional. This port provides output flow-control signals. This is only needed if internal processing latency is variable.</td>
</tr>
</tbody>
</table>

Table 3.1: List of port signals for VHDL processing element cores

In order to achieve high performance, the wrapper must add a pipeline delay for forward data, forward control bits, and reverse control bits. As shown in Figure [3.10], logic must be in place to account for the reverse hold condition in addition to forward and reverse registers. This is achieved by providing the ability to output delayed forward data to account for the edge conditions of reverse flow-control.
Routing Elements

Routing elements control the flow of packets through the datapath. With only a few simple elements, basic networks can be constructed in the datapath.

The first type of routing element is a multiplexer. This element combines multiple busses into a single bus by time multiplexing packets. The multiplexer selects busses of incoming packets in a “first-come, first-served” fashion. The other busses entering the multiplexer are then paused using reverse flow-control hold flags until the first packet’s transfer is complete. For example, if a packet is flowing through input B and a packet arrives at input A, then the bus at input A will be stalled. Once packet B has finished flowing through the multiplexer, the flow of packet A can be resumed.

Demultiplexers have a single input bus and multiple outputs. Opcode bits are picked off of the incoming packet to determine which output to route the packet down. If the selected output has a reverse flow-control halt, the incoming packet to the demultiplexer is also halted.

A fork element takes a single input and produces two outputs. Each output path can be enabled or disabled via opcode bits picked off by the fork element. Since header data often needs to differ between different paths, header data for a forked path is encapsulated using special headers. As packets are transported out through a fork port, the header encapsulation is removed, thus producing standard packets.

A crossbar element has both multiple inputs and multiple outputs. Each input uses opcode bits to select a single output to which to route each packet. If the “multipath” option is enabled, each input can route any given packet to more than one output at once.
Crossbar elements are composite elements and thus are made up of a group of multiplexer, demultiplexer, and fork elements.

### 3.1.2 Status Table

The status table is part of the vector datapath control logic and is used to monitor packets entering and exiting the vector datapath. The table has sixteen entries each with packet ID and status fields as shown in Figure 3.11. Since the architecture of the Cyclone III allows for efficient implementation of four-to-one multiplexers [4], the number of entries in the table was chosen to be a power of four. Limiting the table to four entries was deemed too small since it is possible to have more than four packets in the datapath simultaneously. Stepping up to sixteen entries allows for a sufficiently large number of packets to be tracked simultaneously.

| Packet ID | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid | eop | sop | valid |
|   | entry | 0 |   | entry | 1 |   | entry | 2 |   | entry | 3 |   | entry | 4 |   | entry | 5 |   | entry | 6 |   | entry | 7 |   | entry | 8 |   | entry | 9 |   | entry | 10 |   | entry | 11 |   | entry | 12 |   | entry | 13 |   | entry | 14 |   | entry | 15 |

Figure 3.11: Vector datapath status table

The lowest four bits of the packet ID determine in which table entry a packet is recorded. As a packet enters the datapath, its packet ID is detected and the corresponding entry is
cleared and then set with the new ID. The “sop entering” bit is also latched indicating that the “start of packet” flag has been seen entering the datapath. The “eop entering” bit is later raised at the end of the packet. The “valid entering” is only lowered when the packet flow is halted due to forward or reverse flow-control. The packet status bits are also captured as the packet leaves the datapath to produce the “sop exiting,” “eop exiting,” and “valid exiting” fields. A timing diagram depicting a status table entry for a packet flowing into and then out of the datapath is shown in Figure 3.12.

Figure 3.12: Timing diagram of a status table entry

Fields from the status table can be read at any time by each operand control module in addition to the scalar processor. This information can then be used for scheduling, data dependencies, and debug purposes.
3.1.3 Operand Control

The operand control modules are used to build the packets that enter the datapath. As shown in Figure 3.13, each control module has an instruction FIFO that gets filled with instruction packets generated by the scalar processor. The FIFOs are sized to be 256 words deep since that is the largest size that fits in a single Cyclone III block RAM \( [4] \). The instruction packets queuing in the FIFO have memory read information along with header data for the packet that will enter the datapath.

![Figure 3.13: Overview of operand control module](image)

Instruction packets always start with a “read info” header word. As shown in Figure A.1, this header contains the address and length of the vector data to be read from the memory bank. After “read info” is registered, the operand control module proceeds to copy the packet header data into the datapath while producing matching flow-control signals. Once the header has finished, vector data is read from the memory bank and placed in the packet as it enters the datapath. Upon completion of the packet output, the module will then process the next instruction from the instruction FIFO.

Another feature of the operand control module is its control packet output based on vector datapath status table entries. By using “read delay” headers (shown in Figure A.1), the scalar processor can force the operand control module to delay the start of any packet. There are two modes for a read delay. In the first mode, the operand control waits until the “end of packet exiting” flag is raised for the specified packet ID. The second mode begins outputting data as soon the “start of packet exiting” flag is raised for the specified packet.
ID. The data output is then paused whenever the “valid exiting” flag is lowered. These modes allow data dependencies to be handled while maintaining high performance.

### 3.2 Scalar Processor

The architecture selected for the scalar processor design is that of a TTA. The simple modular design of a TTA is ideal for achieving the flexibility needed for hyper-programmability with little to no loss in performance as the design is scaled [15].

![Figure 3.14: Scalar processor overview](image)

An overview of the scalar processor design is shown in Figure 3.14. Scalar instructions reside in the scalar instruction memory and are accessed by the instruction fetch module. The instruction data is then parsed by the instruction decode module. The processing elements then execute the operation indicated by the instruction.

As shown in Figure 3.15, the pipeline has three stages: instruction fetch, instruction
decode, and execute. Because of the latency associated with high-speed registered block RAMs, the instruction fetch stage has a latency of two cycles. An instruction is issued every other cycle to prevent data-dependency hazards.

![Scalar processor pipeline flow](image)

**Figure 3.15: Scalar processor pipeline flow**

### 3.2.1 Instruction Fetch

A block diagram depicting the design of the instruction fetch module is shown in Figure 3.16. The design consists of two simple parts: the program counter and the associated control logic. Under normal operation, the program counter increments upon every tick from the control logic. If a jump instruction is issued, the program counter is changed to that location. The program counter output is connected to the instruction memory address port and thus controls the order of instruction execution.

The control logic in the instruction fetch module is minimal since it only needs to perform three simple tasks. The first task is to produce a tick pulse every other cycle. The module must also provide an instruction valid flag to indicate when data from the instruction memory read should be processed by the instruction decode module. The flag is only raised every other clock cycle to match the instruction fetch. If a jump occurs, the instruction valid flag is immediately lowered in order to discard the next instruction before it enters the decode stage. The third task the control logic must perform is to provide a delayed value of the program counter that corresponds the instruction word being passed to the decode module. This is useful for relative jumps and viewing the status of program execution.
3.2.2 Instruction Decode

An overview of the instruction decode module design is shown in Figure 3.17. The module multiplexes source ports from processing elements in addition to providing them destination enable signals. A single destination enable signal is raised at a time, and only if the instruction is marked as valid. In addition to selecting between processing element sources, an immediate value can also be used as the source data.

As shown in Figure 3.18, there are two instruction modes for source data: a processing element or a 32-bit immediate value. The opcode was chosen to be large enough to provide ample room for growth in the number of processing element ports. Since there are 15 “destination select” bits, the design can support over thirty-two thousand destination ports. The “source select” portion of the instruction is 32 bits and can therefore support over four billion source ports. In order to conserve resources in smaller designs, the upper unused source and destination bits of the instruction are ignored.

Since high performance was one of the primary objectives of the processor, each output from the decode module is registered using flip-flops. This isolates any timing requirements of individual modules from each other. Another performance optimization taken was in the
multiplexer design. Instead of using a typical design, each source is run through an input enable followed by a large logical-or gate. This keeps the number of logic layers needed as a function of $\log_4(N)$, thus achieving a scalable high performance architecture. Without these optimizations, the instruction decode module would require a lower clock rate, thus limiting the maximum operating frequency of the system.

### 3.2.3 Processing Elements

Scalar processing elements perform computation, storage, and I/O operations for the scalar processor. Each element can have any number of input (destination) and output (source) ports. The port signals provided to a processing element are shown in Table 3.2. In addition
to these ports, an element can also have custom ports that interact with systems external to the scalar processor. Examples of such elements include “vector processor control” elements and register elements with direct external access.

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>in std_logic</td>
<td>Provides the system clock signal.</td>
</tr>
<tr>
<td>reset</td>
<td>in std_logic</td>
<td>Provides a system reset signal.</td>
</tr>
<tr>
<td>ena</td>
<td>in std_logic</td>
<td>Provides a local enable signal.</td>
</tr>
<tr>
<td>dst[i]</td>
<td>in boolean</td>
<td>Asserted when the port is selected as the destination and the instruction is marked as valid.</td>
</tr>
<tr>
<td>dst</td>
<td>in std_logic_vector</td>
<td>Data word transferred to a destination port. When all dst[i] ports are inactive, this port can be ignored.</td>
</tr>
<tr>
<td>src[i]</td>
<td>out std_logic_vector</td>
<td>Data output associated with this port. Data should always be present since the decode module handles the multiplexing.</td>
</tr>
</tbody>
</table>

Table 3.2: Port signals for VHDL scalar processing elements

An example of a scalar processing element is an addition module as shown in Figure 3.19. This element has two destination ports and a single source port. The first destination allows the register value to be directly set. The second destination performs an addition between incoming data and the registered value. An add occurs by writing the first operand to dst0 and then the second operand to dst1. The result could then be used through src0.

One important scalar processing element is the “vector datapath control” element. This

![Adder Module](image)

Figure 3.19: Scalar processing element - adder
Figure 3.20: Scalar processing element - vector datapath control

element provides a way for the scalar processor to send instructions to and control the vector processor. As shown in Figure 3.20, the module has sixteen destination ports, each one dedicated to transmitting a certain type of header. If the vector instruction FIFO becomes filled, a hold flag is raised to pause the scalar processor until more instructions can be received. In addition to the instruction control, the module also interfaces with the vector memory. Two destination ports are dedicated to storing write data for vector memory writes. Vector memory writes are controlled through the “Auxiliary Read/Write” instruction (shown in Figure A.1). Vector memory reads are also requested in the same fashion. Bits from the read data are then split into two source ports for the scalar processor.
Chapter 4

Toolset

In order to help with the design and programming of the vector and scalar processors, four tools were created:

- Vector Datapath Designer
- Scalar Datapath Designer
- Vector Datapath Programmer
- Scalar Assembler

Each application was developed using Microsoft Visual Basic 5.0, a language suited for rapid application development of Windows programs [25].

Figure 4.1 shows the design flow of using these tools during the creation and programming process of the system. The Vector Datapath Designer tool provides a graphical interface used to create and configure a vector datapath. The Scalar Datapath Designer tool is used to choose which scalar processing elements are used and to which addresses they connect. The Vector Datapath Programmer tool helps produce scalar assembly code used to control the vector datapath. The Scalar Assembler takes assembly code written for the scalar processor and produces machine code.
4.1 Vector Datapath Designer

4.1.1 User Interface

A screenshot of the Vector Datapath Designer tool is shown in Figure 4.2. The pane on the left portion of the interface lists routing and processing elements that can be added to the design. The routing elements are always placed on the top of the list and are hardcoded into the tool. Processing elements appear below routing elements and are dynamically loaded from a library subdirectory containing VHDL processing elements. New elements are added to the library simply by copying the VHDL source file into this directory. Since each of these files is designed following the port template shown in Table 3.1, the tool is able to automatically use them in datapath designs. Any port not defined in Table 3.1 would be routed through the top level vector processor file to allow for communication with other systems such as an external RAM.

The pane spanning the majority of the window displays a graphical representation of the datapath. Instances of routing and processing elements are added to the design by dragging them from the library list and dropping them onto this pane. Elements can later be repositioned by dragging them around within this pane. An element can also be resized by clicking and dragging the bottom right corner of the element. An element can be removed by selecting it and then pressing the delete key.

The input and output ports of each element are respectively drawn as dots on the left
Elements are connected together by using an interconnect between an output port of one element and an input port of another. This is done by clicking on one port and dragging over to the port to which it should connect. An interconnect running between the two ports will then appear in the design.

Figure 4.2: Screenshot of Vector Datapath Designer tool

Clicking on an element in the diagram will select it, a state denoted by red highlighting. Properties of a selected element show up in the pane on the bottom of the interface. The minimum set of properties allows numerical repositioning and resizing of the element. Additionally, any build time parameters for the element (VHDL generics) appear up here. In Figure 4.2, for example, the FFT element is selected and the property “max_size” is set to 2048. The textual description provided above each property in the interface is obtained from the comment line beside the port declaration in the VHDL source. Similarly, the default value for the property is also taken from the VHDL declaration. A code snippet with the VHDL generic definition of the FFT element is shown in Figure 4.3.
Figure 4.3: Generics in FFT element

### 4.1.2 Datapath Generation

Datapath designs can be saved using the save command under the file menu. The output file is a fully synthesizable VHDL file. This is created using a combination of datapath design, element properties, the routing element library, and the processing element library.

A basic outline of a generated file is shown in Table 4.1.

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header Comments</td>
<td>Comments listing the creation data and basic description of the file</td>
</tr>
<tr>
<td>Configuration Data</td>
<td>Comments stating configuration parameters used for the datapath design</td>
</tr>
<tr>
<td>Package Imports</td>
<td>Loads locally used VHDL packages</td>
</tr>
<tr>
<td>Entity Declaration</td>
<td>Defines ports entering and leaving the datapath. Also includes custom ports in processing elements</td>
</tr>
<tr>
<td>Component Declarations</td>
<td>Component declarations for each of the processing elements</td>
</tr>
<tr>
<td>Signal Declarations</td>
<td>Declarations for all the signals used in the file</td>
</tr>
<tr>
<td>Component Instantiations</td>
<td>Instantiations of each element, wrapper, and status table</td>
</tr>
</tbody>
</table>

Table 4.1: Outline of Vector Datapath Designer VHDL generation

The first two sections generated consist solely of VHDL comments. The first section is a header that lists basic file information such as name, data created, and a brief description stating the file generation tool used. The second section lists the name, diagram location, and properties of every element and interconnect. This information allows the Vector Datapath Designer tool to later open the design for additional design.

Directly following the comments are the VHDL package imports. Both the standard logic and numeric packages are included here to allow elements to use std_logic, std_logic_vector, signed, and unsigned data types. The custom package “vdpDatatypes_pkg” which defines
custom data types used by processing elements and interconnects is also included. Declarations for routing elements, the wrapper module, and the status table module are imported by including the ‘vdpDeclarations_pkg’ package.

The entity declaration contains a number of port declarations. Each path into and out of the datapath has both a forward control/data port and a reverse control port. Additionally, any port in a processing element instantiation that is not listed in Table 3.1 gets routed to a port in this entity declaration. The output from the vector status table is also included in the port list so it can be used by operand control modules and the scalar processor.

The component declarations section contains a definition of the generic and port lists for each processing element used in the design. Each declaration is generated from the entity declarations in the VHDL source of each element. Since routing elements are declared in one of the package inclusions, they do not need to be declared here.

The signal declaration section contains the signals used by the component instantiations. This includes a signal for every interconnect and signals for connecting processing element wrappers to processing elements.

The main portion of the datapath file is the component instantiation section. Each routing and processing element used in the design is instantiated here. Any parameters for a processing element are also defined here through a generic map. For each processing element there is also a wrapper module instantiation to provide the opcode bits, parameters, and control signals associated with the element. Another module which is instantiated is the status table. Each input and output port of the datapath is connected to the module to produce the status of packets flowing in and out of the datapath.

### 4.2 Scalar Datapath Designer

#### 4.2.1 User Interface

A screenshot of the Scalar Datapath Designer tool is shown in Figure 4.4. Like the Vector Datapath Designer tool, it also has a library subdirectory from which processing elements
are loaded and placed in the pane on the left side of the window. Since the VHDL source for each processing element follows the port template shown in Table 3.2, the Scalar Datapath Design tool is able to automatically load and understand the files. In addition to processing elements, instruction fetch modules also appear in the library list. The only characteristic distinguishing them from processing elements is that they include the ports shown in Table 4.2 in addition to the ports listed in Table 3.2.

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr</td>
<td>in std_logic_vector</td>
<td>Instruction data that was fetched.</td>
</tr>
<tr>
<td>instr_valid</td>
<td>in std_logic</td>
<td>Indicates instruction data is valid.</td>
</tr>
</tbody>
</table>

Table 4.2: Additional port signals for VHDL scalar instruction fetch modules

Elements are added to the design by dragging them from the library list pane into the
spreadsheet section which occupies the main section of the interface. The spreadsheet displays the assigned address, assembly mnemonic, and description of each destination and source port. The mnemonic and descriptions for each port are inferred from comments following each port declaration in the VHDL source. A code snippet of the entity declaration for an adder processing element is shown in Figure 4.5. The mnemonic immediately follows the declaration while the description is surrounded by parentheses.

```vhdl
entity sdpAdd is
generic
  adderBits : integer := 32 --Number of bits to add;
end;
port
  clk : in std_logic; --System clock
  reset : in std_logic; --System reset
  ena : in std_logic; --System enable
  dst0 : in boolean; --x (set x)
  dst1 : in boolean; --add (adds value to x)
  dst : in std_logic_vector(31 downto 0);
  src0 : out std_logic_vector(31 downto 0) --x (x+=y)
end sdpAdd;
```

Figure 4.5: Entity declaration for adder element

Clicking on a processing element in the spreadsheet selects the element and allows its parameters to be modified. Parameters appear in the bottom pane of the interface. The first property listed allows the name of the element to be changed. This is useful for later since the name is directly referenced within assembly code. The next two parameters allow the destination and source address offsets to be defined. These values rarely need to be modified since they are automatically assigned upon adding the element. Lastly, any element defined compile-time properties (VHDL generics) are also listed here. As with the Vector Datapath Designer tool, the text description is retrieved from the comment beside the VHDL generic definition.
4.2.2 Datapath Generation

Datapath designs can be saved by using the save command under the file menu. As with the Vector Datapath Designer tool, the output is a synthesizable VHDL file. The output generation process uses the datapath design in combination with data parsed from the VHDL sources of used processing elements. The resulting VHDL file is a synthesizable VHDL design of a TTA processor. A basic outline of a generated file is shown in Table 4.3.

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header Comments</td>
<td>Comments listing the creation data and basic description of the file</td>
</tr>
<tr>
<td>Configuration Data</td>
<td>Comments stating configuration parameters used for the datapath design</td>
</tr>
<tr>
<td>Package Imports</td>
<td>Loads locally used VHDL packages</td>
</tr>
<tr>
<td>Entity Declaration</td>
<td>Defines processing element ports that connect to other modules</td>
</tr>
<tr>
<td>Component Declarations</td>
<td>Component declarations for each of the processing elements</td>
</tr>
<tr>
<td>Signal Declarations</td>
<td>Declarations for all the signals used in the file</td>
</tr>
<tr>
<td>Instruction Decode</td>
<td>Passes data from a source port to a destination port based on an instruction word</td>
</tr>
<tr>
<td>Component Instantiations</td>
<td>Instantiations of each processing element</td>
</tr>
</tbody>
</table>

Table 4.3: Outline of Scalar Datapath Designer VHDL generation

The header comments section is used to list the filename, creation date, and a brief description declaring the Scalar Datapath Designer as the generator. The configuration data section is also comprised of VHDL comment lines. It lists the names and properties of each processing element in the design. This allows the design to be easily parsed so it can later be reopened for future design changes.

As with the vector datapath source, packages to provide std_logic, std_logic_vector, signed, unsigned, and custom data types are included in the package import section. The section following is the entity declaration section. The minimal ports listed here are the system clock, reset, and enable signals. Every other signal is defined by a processing element and then simply passed through the entity to the next higher level of VHDL. Examples of
such signals include vector instructions and external access to registers.

The signal definition section primarily defines signals used to connect processing elements to the instruction decode logic. For each processing element source port, a 32-bit signal is required to connect it to the instruction decode logic. Each destination port only requires a single bit value which acts as a port enable line. Additionally, a single 32-bit signal is needed to pass the multiplexed source data from the instruction decode logic to each processing element. A 48-bit signal is needed along with a 1-bit signal for the instruction data and data valid ports from the instruction fetch module.

The basic design of the instruction decode logic was shown in Figure 3.17. Each of the source signals along with the potential immediate value are multiplexed together to form the source operand. The address for each port of the multiplexer is defined by the source address parameter shown in the design spreadsheet. To reduce unnecessary logic usage, only enough address bits are used to satisfy the largest address. This is also true for the destination logic. A decoder is used to take the destination address extracted from the instruction and create destination enable signals. A maximum of one destination enable is active at a time. As depicted in the design figure, all signals produced by the instruction decode logic are registered, meaning that all of the logic is placed inside of a clock triggered process.

The last section instantiates each of the processing elements added to the design. Source and destination ports are connected to signals used by the instruction decode logic. Any ports not defined in Table 3.2 are connected to replicas of the ports found in the entity section discussed above. In addition to port connections, generic parameters are also defined here for each module that requires them.

### 4.3 Vector Datapath Programmer

The Vector Datapath Programmer tool provides a graphical interface to aid in producing instructions to control the vector processor. A screenshot of the program with an example.
A datapath loaded is shown in Figure 4.6. A datapath VHDL file generated by the Vector Datapath Designer tool can be opened by using the “Open vdp” button in the upper-left portion of the screen.

![Figure 4.6: Screenshot of Vector Datapath Programmer tool](image)

When a design is loaded, the graphical depiction of the datapath is entirely grayed out by default which is essentially a vector NOP instruction. In order to create a more complex instruction, one or more input ports must be enabled. This is toggled either through a parameter of the input port or by simply clicking on the interconnect associated with the port. Similarly, demultiplexer and crossbar output path selection is selected either through element parameters or by clicking on the associated interconnect. Each output path of a fork element is also enabled or disabled through parameters or by clicking on the associated interconnects.

As in the Vector Datapath Design tool, the parameter pane is located at the bottom portion of the window. These parameters determine what the opcode bits and parameter
words associated with the processing element are set to by the instruction. In Figure 4.6, the settings for the FFT processing element are shown.

The way that opcodes appear in the settings pane is determined by parsing the comment line directly beside the port definition in the VHDL source of the processing element. There are three ways an opcode input can be displayed: a checkbox, a text input box, or option boxes. The checkbox method can be used when the opcode is a single bit in length. Checking the box sets this bit to “1” while unchecking it clears it to “0”. The textbox method is used when the opcode is at least two bits long. Both positive and negative decimal number can be entered. Hexadecimal and binary number can also be entered by appending the prefixes “0x” and “0b” respectively. The option box method displays multiple option boxes of which only one selection can be made, each of which corresponds to a value for the opcode.

A code snippet from the entity declaration of the FFT processing element is shown below as an example. As seen in the code, each port has a default value assigned to it. This value is used as the default for the opcode by the Vector Datapath Programmer tool. The comment next to each port can be used to determine both the layout in the settings pane and the representation of the element in the datapath. In the code snippet shown in Figure 4.7, the opcodes “opcode_mode” and “opcode_order” both use the option box display method while “opcode_size” uses a textbox.

```
opcode_mode : in std_logic := '0'; --|Mode: {FFT | iFFT}
opcode_size : in unsigned(3 downto 0) := "1010"; --log2(fft size) @2^x
opcode_order: in std_logic := '0'; --Output order: {Linear: Bins / arranged from 0 to N-1 | Bit Reversed: Bins arranged in bit-/reversed order} @//{ |Bit Rev}
```

Figure 4.7: FFT opcode definitions

The format of a comment which sets up the textbox input method can be as simple as the comment equating to a brief description of the opcode. The format when using an option box display consists of a description followed by a list of options. This list is defined
within a set of curly brackets and separated by the vertical bar character. An example of this can be seen in the code snippet above where the description is “Mode” and the options are listed as “FFT” and “iFFT.”

In addition to defining how opcode parameters are entered, directives can also be added to the comment to define what information is printed on the processing element representation in the datapath diagram. This is done by using the @ symbol followed by the mode to display in. For example, in the code snippet above “opcode_size” uses the directive “@2^x” to cause the value to be displayed as a power of two in the diagram as shown in Figure 4.6. Available display directives are listed in Table 4.4. One thing to note is that “@/” can be used instead of “@” to prevent the name from opcode name being displayed.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@0b</td>
<td>Displays value in binary</td>
</tr>
<tr>
<td>@0x</td>
<td>Displays value in hexadecimal</td>
</tr>
<tr>
<td>@2^x</td>
<td>Displays two raised to the power of the value</td>
</tr>
<tr>
<td>@{OptionA</td>
<td>OptionB</td>
</tr>
</tbody>
</table>

Table 4.4: Vector Datapath Programmer display directives

With the exception of in and out ports, element settings are used to define opcode bits and parameter words. Two of the settings for in ports are the definition of the packet’s payload length and the memory bank starting read address. A field also exists to define an ID for the packet. The out port allows the memory bank write address to be set along with a secondary way to set the packet’s ID.

Once an instruction design is complete, the “Copy asm to clipboard” button can be used to create scalar assembly that will send that instruction to the vector processor. The assembly automatically generated to produce the instruction shown in Figure 4.6 is shown in Figure 4.8. When this code is run by the scalar processor, the instructions will be sent to the operand control modules to be executed in the vector datapath.
4.4 Scalar Assembler

The Scalar Assembler tool takes as input a scalar datapath generated by the Scalar Datapath Designer tool along with an assembly code file. It then assembles the code and generates an executable that is runnable by the associated scalar processor. A screenshot of the Scalar Assembler program is shown in Figure 4.9.

Figure 4.9: Screenshot of Scalar Assembler tool

Under the open menu there are two options: one to open the scalar datapath and the
other to open the assembly source file. Once these files are loaded, the code can be compiled by running the compile command under the compile menu. The resulting machine code is then displayed in the window in hexadecimal. This executable can then be saved into a hex file which can later be loaded into the processor’s instruction memory.

The format for TTA assembly code is always a move instruction represented by an equals sign. Each line takes the form “A.dstX = B.srcY” where A is the destination element, dstX is the destination port, B is the source element and srcY is the source port. Shorthand form can sometimes be used where dstX and/or srcY are omitted and the assembler will infer the first available port as the one to use. If using an immediate as a source, the instruction would then take the form “A.dstX = I’ where I is an immediate value.

Any line can be marked with a label to aid with writing jump instructions. Whenever a label is used as the source operand, the immediate value will be set to the difference between the label and current line. The example assembly program shown in Figure 4.10 makes use of labels for looping and jumping over instructions.

; A simple test program

cnt0 = 5 ; Loop for 5 iterations
loop1:
    R0 = cnt0 ; Output the count
    PC.cmp = cnt0 ; Count = 0?
    cnt0.dec = cnt0 ; Decrement loop counter
    PC.rjz0 = loop1 ; If loop > 0 goto loop1

    R1 = PC.PC ; Output the PC
    PC = laterdown

    R0 = -123 ; Here is something that won’t be executed

laterdown:
    R2 = 0xABCDEFFF ; Output 0xABCDEFFF
    PC.rjmp = 0 ; Halt

Figure 4.10: Example assembly program

When the compile command is run, the Scalar Assembler parses the assembly code
matching elements and ports for the source and destinations with the definition of the data-path provided. Any immediate values are converted to 32-bit hexadecimal numbers to fit the instruction format. If an element name or port cannot be found, an error message is displayed on the screen stating the problem.
Chapter 5

Test Setup

In order to test the design, both simulation and hardware based systems were created. A block diagram of this is depicted in Figure 5.1. As shown, tests are set up and controlled through a MATLAB interface. This provides an excellent environment to set up test vectors and compare resulting outputs against expected results.

Figure 5.1: Overview of the test setup
Both the simulation driver (simDriver) and the hardware driver (dspDriver) were designed to use the same interface in MATLAB. This allows switching between simulation and hardware to be as simple as changing which driver is being used.

5.1 Simulation Environment

The simulation driver is used as an interface between MATLAB and ModelSim. Data is transferred to and from the ModelSim testbench using files. As shown in Figure 5.2, four files are passed to ModelSim. After the simulation completes, three files are returned to MATLAB.

![Figure 5.2: Overview of the simulation environment](image)

The first file written by the simulation driver and loaded by the testbench is used to initialize the scalar processor’s program memory. This file contains a hexadecimal number on each line to represent the memory contents. It is identical to the “hex” file outputted by the Scalar Assembler tool. The second file is used to initialize registers in the scalar processor. This provides a way to parameterize a program without recompiling it. The
“RAM Initialization” file is used to load input vector data into the system. To decrease simulation time, input memory data is broken up into blocks of data that each have an address and length definition associated with them. The “RAM Output Select” file is used to define which blocks of memory to output. The format of this file is similar to the “RAM Initialization” file, but it only contains addresses and lengths of blocks, not data.

The testbench uses the textio package to read in the files. Non-synthesizable VHDL is used to copy the input data into the scalar processor and RAM instantiations. Once everything is loaded, the testbench enables the DSP. The program is then run by the processor. The testbench monitors the program counter, and when the count reaches the last line of the program, the processor is halted. The output files are then generated by the testbench.

The first output file contains the final value of the registers. These values can be used to relay scalar results such as execution time back to MATLAB. The “RAM Output” file contains RAM contents read after completion of program execution. To reduce simulation time, only sections of RAM listed by the “RAM Output Select” file are saved. The last file written is the program counter. While the design is simulating, the MATLAB driver monitors this file. Once a value is written to it, this signals MATLAB that the simulation is complete and that the results can be read and processed.

5.2 Hardware Environment

The DSP driver is used as an interface between MATLAB and the FPGA. Data is transferred to and from the FPGA using an RS232 serial connection at 115200 baud. In order to achieve quick binary communication, a C-based MEX program was created to directly interface to the RS232 port. The DSP driver uses this to transfer data and commands. A MAX232 IC [28] is used to convert from RS232 voltage levels down to a voltage level compatible with the FPGA I/O pins.

An Altera Cyclone III Development Kit (DK-DEV-3C120N) was used as the hardware
platform. The FPGA included on this board is the Cyclone III EP3C120. This FPGA provides 119K logic elements, 3.8 Mbits of block memory, 288 18-bit multipliers, and 4 PLLs. One of the internal PLLs was used to multiply the 50MHz clock signal provided by the development board up to the operating frequency of the processor. An operating frequency of 135MHz was chosen by rounding down to the nearest 5MHz after taking the worst-case maximum frequency at $85^\circ C$ as reported by Altera’s TimeQuest timing analyzer. This results in configuring the PLL with a frequency multiplication/division factor of 27/10.

An overview of the FPGA test configuration is depicted in Figure 5.3. As shown, only two I/O pins are used: UART receive and transmit. These pins are connected to a module used to perform the low-level UART communication. This module then provides byte-level communication to the “UART to DSP Interface” module. This interface has a state machine that inputs commands, executes them, and provides a return value. A list of commands it supports is shown in Table 5.1.

Figure 5.3: Overview of the hardware test environment

Each command in the table has a 1-byte code associated with it. The “UART to DSP Interface” module first waits for a command code to be received. If the command has
a data portion associated with it, the module then receives that. The command is then executed and the return value is transmitted. For commands without an associated return, an acknowledgement byte is sent back instead. Since the DSP driver always waits for a response after sending a command, this keeps communication synchronized and prevents any buffer overflows.

<table>
<thead>
<tr>
<th>Command</th>
<th>Data</th>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>1 byte: Ack</td>
<td></td>
<td>Resets the DSP.</td>
</tr>
<tr>
<td>Set Address</td>
<td>4 bytes: Address</td>
<td>1 byte: Ack</td>
<td>Sets the address counter internal to the interface.</td>
</tr>
<tr>
<td>Write Instr Mem</td>
<td>6 bytes: 48-bit word</td>
<td>1 byte: Ack</td>
<td>Writes a word to the scalar instruction memory.</td>
</tr>
<tr>
<td>Write RAM</td>
<td>5 bytes: 36-bit word</td>
<td>1 byte: Ack</td>
<td>Writes a word to RAM.</td>
</tr>
<tr>
<td>Read RAM</td>
<td>5 bytes: 36-bit word</td>
<td></td>
<td>Reads a word from RAM.</td>
</tr>
<tr>
<td>Enable DSP</td>
<td>1 byte: Ack</td>
<td></td>
<td>Enables the DSP.</td>
</tr>
<tr>
<td>Disable DSP</td>
<td>1 byte: Ack</td>
<td></td>
<td>Disables the DSP.</td>
</tr>
<tr>
<td>Read PC</td>
<td>4 bytes: 32-bit word</td>
<td></td>
<td>Reads the scalar program counter.</td>
</tr>
<tr>
<td>Read Register</td>
<td>4 bytes: 32-bit word</td>
<td></td>
<td>Reads a register value from the scalar processor.</td>
</tr>
<tr>
<td>Write Register</td>
<td>4 bytes: Value</td>
<td>1 byte: Ack</td>
<td>Writes a register value to the scalar processor.</td>
</tr>
</tbody>
</table>

Table 5.1: UART to DSP interface commands

Since the DSP driver uses the same interface as the simulation driver, a single test script can be used to run a program in simulation and hardware. When a program is run in hardware, the first step taken by the script is to disable and reset the DSP. Next the program, vector data, and initial register values are transferred to the FPGA. The DSP is then enabled resulting in the program to be executed. After pausing for a few seconds to allow the program execution to complete, the program counter, register values, and vector results are read out of the FPGA. The results can then be analyzed in MATLAB.
5.3 Example Vector Datapath

The Vector Datapath Designer tool was used to create an example vector datapath capable of performing basic DSP operations. As shown in Figure 5.4, the datapath has three input ports and three output ports. Vector data enters the datapath through the “Data Import” element. This element is used to read vector data directly from the system RAM. A parameter word is used to define the starting address of each read. The length of each read is matched to the length of the packet data as defined by the packet entering the element.

![Diagram of Example Vector Datapath]

Figure 5.4: Example vector datapath

The corollary to the import module is the export module. As with the import, the address is defined by a parameter word. By default the length to write is defined by the packet length. An opcode mode also exists that allows a select portion of the packet data to be written to RAM. Two parameter words are associated with this mode to specify the segment start and end offsets.

The Complex Multiply element is used to perform an element-by-element complex multiply. Each input of the module can optionally be conjugated, which is controlled using opcode bits. By default neither input is conjugated.

The FFT element is used to compute the discrete Fourier transform up to 2048 points in length. This element provides the majority of the computational power for this particular
datapath design. An opcode bit is used to define whether to perform the forward or inverse transform. The size of the FFT being computed is also defined using opcode bits. This allows transforms smaller than 2048 to be computed. The output order of the data words is also definable via an opcode bit. Since the FFT algorithm naturally results in data that is bit-reversed in order, the element will by default reorder the output to a linear order. The output can also be explicitly chosen to be generated in bit-reversed order to reduce the latency of the computation output.

The window generator element is used to create windows and mask data. The element has an option to select between band-pass and band-stop windowing. Two parameter words are used to define the two edge of the window as shown in Figure 5.5. An opcode bit is used to select whether the window is directly outputted as the resulting data or if it is immediately used to mask the incoming data.

![Figure 5.5: Band-pass and band-stop windows](image)

The statistic element is used to find the maximum element in a vector. Since complex data is used, the maximum magnitude of each element is compared. The module finds the index of the maximum element, the value of that element, and the magnitude squared of that element. Any combination of these resulting values can be included in the module’s output vector.

Each of the input and output ports are connected to a memory bank and operand control. Each memory bank is sized to 2048 words in length. These memory banks essentially act as registers for the vector datapath.
5.4 Example Scalar Datapath

The Scalar Datapath Designer tool was used to create an example scalar datapath. A list of elements added to the datapath is shown in Table 5.2. As seen in the table, sixteen processing elements were used in total. The instruction fetch module was named “PC” since it includes the program counter. The module provides instructions to support jumps and conditional jumps. A “compare to zero” instruction is included to support conditional jumps. To support more conditional jumps, a comparison module (named “cmp”) was added to the datapath. This module compares two values and outputs “1” if the condition is true and “0” if the condition is false. Both signed and unsigned comparisons are supported.

<table>
<thead>
<tr>
<th>Name</th>
<th>Dst Address</th>
<th>Src Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0x0000-0x0003</td>
<td>0x0000</td>
<td>Reads instructions from program memory and controls jumps</td>
</tr>
<tr>
<td>cmp</td>
<td>0x0004-0x0005</td>
<td>0x0001-0x0005</td>
<td>Allows for signed and unsigned comparisons</td>
</tr>
<tr>
<td>R0</td>
<td>0x0006</td>
<td>0x0006</td>
<td>Register 0</td>
</tr>
<tr>
<td>R1</td>
<td>0x0007</td>
<td>0x0007</td>
<td>Register 1</td>
</tr>
<tr>
<td>R2</td>
<td>0x0008</td>
<td>0x0008</td>
<td>Register 2</td>
</tr>
<tr>
<td>R3</td>
<td>0x0009</td>
<td>0x0009</td>
<td>Register 3</td>
</tr>
<tr>
<td>cnt0</td>
<td>0x000A</td>
<td>0x000A</td>
<td>32-bit decrement counter</td>
</tr>
<tr>
<td>sub</td>
<td>0x000B-0x000C</td>
<td>0x000B</td>
<td>Performs 32-bit subtraction</td>
</tr>
<tr>
<td>adder0</td>
<td>0x000D-0x000E</td>
<td>0x000C</td>
<td>Performs 24-bit addition</td>
</tr>
<tr>
<td>adder1</td>
<td>0x000F-0x0010</td>
<td>0x000D</td>
<td>Performs 24-bit addition</td>
</tr>
<tr>
<td>adder2</td>
<td>0x0011-0x0012</td>
<td>0x000E</td>
<td>Performs 32-bit addition</td>
</tr>
<tr>
<td>timer0</td>
<td>0x0013</td>
<td>0x000F</td>
<td>Resettable free-running counter</td>
</tr>
<tr>
<td>vdpStatus</td>
<td>0x0014-0x0015</td>
<td>0x0010-0x0014</td>
<td>Retrieves status from the vector datapath</td>
</tr>
<tr>
<td>vdp0</td>
<td>0x0016-0x0027</td>
<td>0x0015-0x0016</td>
<td>Sends instruction words to the vector operand control for path 0</td>
</tr>
<tr>
<td>vdp1</td>
<td>0x0028-0x0039</td>
<td>0x0017-0x0018</td>
<td>Sends instruction words to the vector operand control for path 1</td>
</tr>
<tr>
<td>vdp2</td>
<td>0x003A-0x004B</td>
<td>0x0019-0x001A</td>
<td>Sends instruction words to the vector operand control for path 2</td>
</tr>
</tbody>
</table>

Table 5.2: Elements in the example scalar datapath

Four 32-bit registers were included in the design. These are used to input parameters,
hold internal values, and output return values. Each register can be directly accessed by the controlling entity. Since processing elements hold their output states until a new input is presented, few registers are required in a TTA.

A decrementer, a subtractor, and three adders were included in the design. The decrementer can be set to any value and the output will be that value minus one. This is useful as a loop counter. The subtractor takes two input values and outputs the difference of those inputs. The adders take two inputs and output the sum of those inputs. Multiple adders were included in order to allow multiple incrementers to be used by a program without having to explicitly use registers.

The timer module is used to measure time in units of clock cycles. The count value can be set to any value. It then proceeds to increment upwards once per clock cycle. The count can then be read at any time. The timer was included to provide a way to measure the execution time of a program. This is done by clearing the count as the first instruction of a program and then copying the final count to a register as the last instruction of the program (excluding halt).

There are four modules included for interaction with the vector datapath. The vector datapath status module is used to see the status for any packet ID. The module takes an input of a packet ID and then provides an output for “sop seen entering,” “eop seen entering,” “sop seen exiting,” and “eop seen exiting.” Additionally, a “running” flag is generated. This flag is raised whenever any packet has the “sop seen entering” flag raised but not the “eop seen exiting” flag. These values can be used to conditionally wait until an event has occurred in the vector datapath. The module also allows the status table to be cleared to start with an empty table for a new program execution. In addition to the vector datapath status module, three vector datapath control modules are included, one for each operand control module in the vector processor.
Chapter 6

Results

6.1 Synthesis

The design detailed in Section 5.2 consisting of a DSP, RAM, and debug UART was synthesized using Altera Quartus II version 9.0. The resulting resource utilization of the design is shown in Table 6.1. The “Total Usage” column shows the resources used in total by the entire synthesized design. The “DSP Usage” column shows the portion utilized specifically by the scalar and vector datapaths. To give perspective on how much of the FPGA is used, the “Total Available” column shows the total resources offered by the Cyclone III EP3C120.

<table>
<thead>
<tr>
<th></th>
<th>Total Usage</th>
<th>DSP Usage</th>
<th>Total Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Elements</td>
<td>19,820</td>
<td>18,990</td>
<td>119,088</td>
</tr>
<tr>
<td>Memory Blocks</td>
<td>220</td>
<td>76</td>
<td>432</td>
</tr>
<tr>
<td>Multipliers</td>
<td>42</td>
<td>42</td>
<td>288</td>
</tr>
</tbody>
</table>

Table 6.1: Resource usage of synthesized design

A logic element is the smallest unit of logic in the Cyclone III architecture. It consists of a four-input lookup-table, a register, and output logic. A memory block provides up to 9kbits of storage. Each block supports dual-port operation with port widths up to 36-bits wide. The multiplier blocks in the Cyclone III can be configured either as a single 18bit multiplier or as two 9bit multipliers. Each multiplier supports both signed and unsigned operation. [4]
A breakdown of how many logic elements and RAM blocks used by each portion of the design is shown in Figure 6.1. As seen, the vector processor portion of the DSP accounts for a majority of the logic used. This makes sense since it is responsible for the majority of the computational work. Similarly, the vector portion uses significantly more RAM than the scalar portion. This is because the vector processor operates on large amounts of data at a time.

![Figure 6.1: Logic elements (left) and RAM (right) used by the design](image)

To expand upon Figure 6.1, a breakdown of the resources used exclusively by the vector processor portion of the design is shown in Figure 6.2. As shown, the processing elements in the vector datapath account for 61% of the logic used. This means that 39% of the logic acts as overhead to support the computational logic. In comparison, 71% of the logic in the MIPS architecture is support overhead for its datapath [21]. In addition to logic, 47% of the RAM is used by the processing element and 53% as support overhead. Although not shown in Figure 6.2, 100% of the multipliers in the design are used by processing elements.

A breakdown of the resources used by the scalar processor portion of the design is shown in Figure 6.3. As shown, 64% of the logic used is by the processing elements, which leaves 36% of the logic as support overhead. The only RAM used in the scalar
Figure 6.2: Logic elements (left) and RAM (right) used by the vector processor

processor is for instruction memory. Thus 100% of the RAM is overhead used by support portions of the processor.

Figure 6.3: Logic elements (left) and RAM (right) used by the scalar processor
6.2 Benchmarks

A set of benchmark programs was written to test and evaluate the system. A number of benchmarks mappable to the example vector datapath were selected from the industry standard BDTI DSP Kernel Benchmarks [10]. Each benchmark performs an operation fundamental to DSP applications. Table 6.2 lists these benchmarks along with example applications of the DSP operation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Applications</th>
</tr>
</thead>
</table>
| Vector Search | • Signal detection  
               | • Automatic gain control  
               | • Error control coding   |
| Vector Multiply | • Windowing       
               | • Mixing            
               | • Computing inner products |
| FFT          | • OFDM             
               | • Radar             
               | • Sonar             
               | • Spectral analysis |
| FIR          | • Channel equalization 
               | • Spectrum shaping  
               | • Speech processing  
               | • Convolution       |

Table 6.2: Set of benchmark programs

The vector search program finds the value and location of the maximum value in a vector. This operation can be used for signal detection, automatic gain control, and error control coding [10]. The vector multiply program computes the element-wise product of two vectors. This operation is used for windowing, mixing, and computing inner products. The FFT program computes the discrete Fourier transform. This computation is used in OFDM (Orthogonal Frequency-Division Multiplexing), radar, sonar, and spectral analysis [10]. The FIR program applies filters on complex data. This can be used for channel equalization, spectrum shaping, speech processing, and general convolution [10].

Each of the benchmark programs was run on the synthesized hardware and the execution time was collected. In addition to the execution time, the accuracy of each benchmark
was also analyzed. The resulting performance was compared to both the high-performance TigerSHARC DSP and the low-power TMS320C6746 DSP. Performance data for each of these DSPs was obtained from [8] and [32] respectively.

### 6.2.1 Vector Search

The vector search benchmark searches through a vector for the element with the largest magnitude. The program was written using a single vector instruction depicted in Figure 6.4. As shown, the instruction begins by passing the packet through the “data import” element to load the input vector from RAM. The address used for the RAM import resides in a scalar register set prior to program execution. The packet is then routed to the “statistics” element where the maximum is found. The output of the “statistics” element is then routed to the “data export” element where it is written back to RAM. The output address is also set by a scalar register value.

![Figure 6.4: Vector Search instruction](image)

Once the instruction was designed, the generated assembly code was copied into a program source file. Additional scalar code was added to record the execution time. The resulting program is shown in Figure 6.5. The code before the vector instruction clears the timer count and vector status module. After the instruction is issued, a while loop waits for
the vector instruction to complete. The timer value is then saved to a register. An infinite loop is used as the last instruction in order to prevent uninitialized program memory from being executed.

```
;vmax2k program
; Inputs:
; R0=Address of x
; R1=Address for results
; Outputs:
; R0=Execution cycles

;Clear timer to calculate execution time
timer0 = 0

;Clear vdpStatus
vdpStatus.clr = 1

;Import x, Export results
;Export = statistics(import)
vdp2.ReadInfo = 0x08000000
vdp2.Opcode = 0x00001FB6
vdp2.Param = R0 ;e18_dataImport.addr
vdp2.Param = R1 ;e19_dataExport.addr
vdp2.WriteInfo = 0x01000000

;Wait for all packets to finish
runningWait:
PC.cmp = vdpStatus.running
PC.rjz0 = runningWait

;Save execution time into R0
R0 = Timer0

;Halt
PC.rjmp = 0
```

Figure 6.5: Assembly listing for 2k vector search
The benchmark was run and the resulting execution time was collected and compared as shown in Figure 6.6. As depicted, the proposed design is about 16% faster than the TMS320C6746 at executing the benchmark.

![Figure 6.6: Execution time for 2048 element search benchmark](image)

To measure the accuracy of the benchmark, the SNR (Signal to Noise Ratio) of the largest reported magnitude squared value was calculated. SNR is found using the definition shown in Equation 6.1 [20]. Since MATLAB was used to create the test vector, it could also be used to produce the theoretical results sans error. The resulting SNR showed an accuracy of greater than 85dB.

\[
SNR_{dB} = 20 \log_{10} \left( \frac{\text{signal}}{\text{noise}} \right) = 20 \log_{10} \left( \frac{\text{theoretical}}{\text{theoretical} - \text{measured}} \right) \quad (6.1)
\]
6.2.2 Vector Multiply

The vector multiply program loads two vectors 2048 elements long, performs an element-wise complex multiplication, and outputs the resulting vector. Because the program requires two vectors to be loaded from RAM, two vector instructions were required in the program. The first instruction imports the first vector and stores it in memory bank 0 as shown in Figure 6.7. The second instruction imports the second vector, multiplies it by the first, and exports the result back to RAM as shown Figure 6.8.

Figure 6.7: Vector Multiply instruction 1

Figure 6.8: Vector Multiply instruction 2
The assembly output for each instruction as generated by the Vector Datapath Programmer tool was copied into the program. The resulting assembly code is shown in Appendix B.1. Again, the benchmarking operations were added before and after the vector instructions. One line of code that was added to the second instruction is the “read delay” header. This is used to delay the operand control module to output the second instruction only when the data becomes available.

The resulting performance is shown Figure 6.9. There are two results listed for the proposed design. The first value is when the time taken to load both vectors is included in the execution time. The second value is if one of the vectors was preloaded into one of the memory banks. This represents the cost of a multiplication as part of a larger algorithm.

![Figure 6.9: Execution time for 2048 element vector multiply benchmark](image)

To measure the accuracy of the calculation, the SNR was calculated for each element in the result vector. MATLAB was used to calculate the theoretical values using double precision arithmetic. The resulting SNR plot is given in Figure 6.10. As shown, the benchmark...
yields a result with greater than 80dB of precision.

![SNR plot](image)

Figure 6.10: Accuracy of 2048 element vector multiply

### 6.2.3 FFT

The FFT program loads a vector of length 2048, performs an FFT operation, and exports the result. The program only requires a single vector instruction which is shown in Figure 6.11. Since the FFT element has a processing latency of double the vector length, increased throughput is achieved by repeating the instruction to create a pipeline. One of the input parameters for the program, therefore, is the number of vectors to process. In addition to the 2048 point version of the program, variants were also created to support 1024 and 256 point FFT operations.

The assembly code for the program is shown in Appendix B.2. The vector instruction is surrounded by looping code. A decremener is used as the loop counter. It is initially set to the count provided in register 2. After the vector instruction, addresses are incremented...
and the loop count is decremented. If the count is greater than zero, then a jump instruction is used to loop back to where the vector instruction output began. Once the loop has completed, the program waits for the vector instructions to finish and saves the execution time.

Two execution times were measured for the proposed design. The first one measured the time it took from importing in the time-domain vector to exporting the frequency-domain result vector. The second measurement shows the time taken in a perfect pipeline which allows the latency to be ignored. This execution time can be approached if the FFT of many vectors is taken one after another. Compared to the previous two benchmarks, the FFT benchmark yields significant improvement in relative performance. Because this operation is more complex and processor intensive, it highlights the performance benefit of running a processor tailored for a specific application. With this benchmark a larger percentage of the vector datapath is utilized, thus resulting in higher performance.

The benchmark was run using multiple sizes for the input vectors, thus performing different sized FFT computations. The benchmark result for a 2048-point FFT is given in Figure 6.12. As shown, the proposed design is more than three times faster than the TMS320C6746 for a single FFT. When running multiple FFTs in a pipeline, the proposed design is ten times faster.
Figure 6.12: Execution time for 2048 point FFT benchmark

Figure 6.13: Execution time for 1024 point FFT benchmark
The results for running the benchmark using a 1024-point FFT are given in Figure 6.13. As shown, the proposed design is slightly slower than the TigerSHARC DSP when the latency of the FFT is included in the measurement. Measuring pipelined execution, however, yields a faster execution time than both of the other DSPs.

The FFT benchmark was run with 256-point vectors to compare it with results available from the Nios II. As shown in Figure 6.14, the proposed design significantly out-performs both the Nios II and the Nios II with C2H hardware optimizations. The pipelined performance of the proposed design is more than 450 times faster than the Nios II by itself and 27 times faster than the Nios II with C2H.

Figure 6.14: Execution time for 256 point FFT benchmark
To measure the precision of the benchmark, the SNR of the multiple result vectors was calculated and plotted as shown in Figure 6.15. This plot consists of the SNR of five 2048-point FFT calculations. As shown, the resulting SNR yields an accuracy of greater than 70dB.

![Figure 6.15: Accuracy of 2048 point FFT](image)

6.2.4 FIR

The FIR (Finite Impulse Response) filter benchmark convolves an input signal with complex filter coefficients to produce a filtered signal. The program implements filtering using the overlap-save fast convolution filtering technique as described in [11]. The basic concept behind this technique is since FIR filtering is simply the convolution in time of two vectors, multiplication is the equivalent operation in the frequency domain. One caveat with direct frequency domain multiplication is that it results in circular convolution when FIR filtering requires linear convolution. To account for this, a portion of each iteration must be
discarded. A block diagram of the algorithm is shown in Figure 6.16.

Figure 6.16: Overlap-save filtering, \( y(n) = x(n) \ast h(n) \) \[11\]

The first instruction is used to import the filter coefficients from RAM as shown in Figure 6.17. A vector of length 2048 is always imported regardless of the filter length, since it will later get masked by the following instruction. The coefficients are routed and stored to memory bank 2.

Figure 6.17: FIR instruction 1

The second instruction masks off the invalid portion of the filter coefficients and computes a 2048-point FFT as shown in Figure 6.18. The masking is done using the “window generator” element with a band-stop mask. The first edge of the mask is set to match the
number of filter coefficients, which is a parameter inputted via a scalar register. This vector is then routed to the FFT element where a 2048 point FFT operation is performed, resulting in a frequency domain filter. This result is then routed and stored to memory bank 1.

The next instruction imports a portion of the input data sequence and performs the Fourier transform as shown in Figure 6.19. After a vector of length 2048 is imported, it is passed to the FFT element where a 2048 point FFT is performed. The result is then stored in memory bank 0.

The final instruction multiplies the input data and the filter coefficients in the frequency
domain and then returns to the time domain. This instruction is shown in Figure 6.20. The input data is read from memory bank 0 since the output of the previous instruction stored it there. The filter data is read from memory bank 1 since the frequency domain coefficients were stored there by the second vector instruction. These two vectors are multiplied together and then the inverse FFT is performed. As defined by the algorithm, a portion of the result is then exported back to RAM. The segment to output is defined by passing additional parameters into the “data export” element to specify which portion of data to keep.

The resulting assembly code is shown in Appendix B.3. After clearing the timer and vector status table, the first and second vector instructions are issued. A loop is then used to repeatedly issue the third and fourth vector instructions across the entire data input sequence. During every iteration, the import and export addresses are incremented. The increment amount is sized such that a certain amount of overlap is achieved as required by the overlap-save algorithm. In addition to incrementing the addresses, the packet IDs for each packet are also incremented. This is needed since a “read delay” header is used along with each instruction issued to guarantee that data dependency is not broken. Once the loop has completed, a while loop waits for all vector instructions to complete, and then the execution time is stored into a register.
The resulting performance for a 10-tap FIR filter is given in Figure 6.21. The test vector was a complex input signal 10,000 samples long. As shown, the proposed design is able to outperform the other DSPs in this benchmark.

![Figure 6.21: Execution time for N=10000, M=10 FIR benchmark](image)

When the number of taps is increased, the performance advantage of the proposed design becomes more evident. Figure 6.22 shows the results of the benchmark with a 100-tap filter. As shown, the proposed method runs more than 13 times faster than the next fastest DSP compared. Figure 6.23 shows a comparison of performance for a 512-tap FIR filter. Here the proposed design is more than 58 times faster than the other DSPs.

Like the other benchmarks, the accuracy of the FIR filter benchmark was measured using SNR. Figure 6.24 shows the SNR across all 10,000 samples of the benchmark with a 100-tap filter. As shown, the result has better than 65dB of accuracy.
Figure 6.22: Execution time for $N=10000$, $M=100$ FIR benchmark

Figure 6.23: Execution time for $N=10000$, $M=512$ FIR benchmark
6.3 Power Consumption

Since many DSP applications occur in a mobile environment, power consumption is often an important factor in DSPs. The Cyclone III development board provides a sense resistor for the core power supplied to the FPGA. By measuring the voltage drop across the resistor, the current can be calculated using the equation given by

$$I_{DD} = \frac{V_{\text{sense}}}{R_{\text{sense}}}$$  \hspace{1cm} (6.2)

[2]. A digital multimeter was used to measure the voltage across the sense resistor while the FPGA ran the benchmark programs. The resulting current draw was then calculated to be

$$I_{DD} = \frac{3.8mV}{0.009\Omega} \approx 422mA$$  \hspace{1cm} (6.3)

Since the core voltage is 1.2V, the resulting core power consumption was calculated to be 506mW.
Analog Devices provided typical electrical conditions of the TigerSHARC in [8] and [9]. When running at 600MHz, the TigerSHARC requires a core voltage of 1.2V [8]. The current was estimated to be 3.38A during execution of the FFT benchmark at 600MHz [9]. The power consumed can therefore be calculated as 4.056W.

Typical power consumption for the TMS320C6746 was provided in [30]. When running the device at 300MHz, the core voltage of the DSP requires 1.2V. Typical operation is defined to consist of 70% CPU utilization, 50% memory utilization, and 1 I/O pin at 50% utilization of 18.75MHz. Under these conditions, the current used is estimated to be 356.84mA. This results in a total power consumption of 428.21mW. [30]

Figure 6.25: Power consumption of each DSP

Figure 6.25 depicts the power consumption of the proposal compared to the other DSPs. Even with the efficiency loss associated with FPGAs, the proposed design consumed about an eighth of the power of the TigerSHARC. When compared to the TMS320C6746 low-power DSP, the proposed design used only 18% more power.

Two main approaches to reducing energy consumption in DSPs were identified in [24]:
• Operate at lowest possible voltage and frequency

• Reduce energy waste.

Although the proposed design operates at the same voltage as the other two DSPs, it does run at a lower clock frequency. The performance decrease associated with this is then offset by exploiting concurrency through pipelining and parallelism. The second approach to saving power can be realized through an application specific design. Unlike a general purpose design, a design that is hyper-programmable can be tailored to include only the resources actually needed, thus reducing wasted energy.
Chapter 7

Conclusions

This thesis presented a design of a high-performance, hyper-programmable DSP. An overview of hyper-programmable processors, TTAs, NoC-based datapaths, and commercial DSPs was given in the Related Work section. The design of the proposed hyper-programmable architecture was then presented. This included details about the high-performance vector datapath and the controlling scalar datapath. Port definitions were given for developing custom processing elements for both the vector and scalar datapaths. The Vector Datapath Designer and Scalar Datapath Design tools were then presented as the part of the toolset supporting datapath development. The Vector Datapath Programmer and Scalar Assembler tools were then detailed as the portion of the toolset used for generating programs to run on the design. The format for assembly programs running on the scalar portion of the design was also given.

The testbench setups for both simulation and hardware verification were then presented. Example configurations were given for both the vector and scalar datapaths. The resulting design was synthesized for a Cyclone III FPGA. The synthesis resulted in a design capable of running at 135MHz with 61% of the logic elements used by processing elements. A set of benchmarking programs consisting of operations often used in DSP algorithms was then run on the design to evaluate its performance. The proposed design achieved a similar execution time to other DSPs for the simple benchmarks. However, for the more complex benchmarks, the proposed design yielded significant improvement over the other processors. In addition to execution time, accuracy was also measured for each benchmark. Plots
depicting SNR for each resulting vector were generated and shown to be higher than 65dB for every benchmark. The power used by the FPGA was measured while continuously running a benchmark. This showed a power consumption of 506mW. This value was then compared with other DSPs and shown to be better than the high-performance DSP and on par with the low-power DSP.

7.1 Future Work

There are a number of areas where further development and research could take place. One of the areas for improvement in the design is performance. Each of the processing elements developed for this thesis is capable of running at 200MHz. Some of the supporting framework (namely the vector operand control and processing element wrappers) are only rated to run at just over 135MHz. By parallelizing and pipelining the offending logic, the maximum clock rate of the design could be increased. Another potential way to increase performance would be by using a different FPGA. To achieve maximum performance, high-power FPGAs such as the Altera Stratix series or the Xilinx Virtex series could be evaluated. If maintaining low-power is important, new FPGAs using smaller process technologies such as the Cyclone IV could also be evaluated.

One simplification made in the design was to use FPGA internal memory as the system RAM. The block memory used in the design was only 32k words in size, but accounted for two-thirds of the RAM blocks used by the entire synthesized design. By replacing this memory with an external RAM such as DDR, millions of words of storage could exist without consuming unreasonable amounts of FPGA resources.

Writing programs for the design currently must be done in an assembly-like environment. One improvement would be to include a compiler supporting a high-level language such as C. This would decrease the programming development time by creating more maintainable code in an environment familiar to many engineers. Like the assembler created for this thesis, a compiler would be required to adapt to any scalar and vector datapath targeted.
Another area for future research is in combining a hyper-programmable DSP with partial reconfiguration. Partial reconfiguration allows a subsection of an FPGA to be reconfigured while the rest of the FPGA continues to run [34]. The modular nature of a hyper-programmable processor is a natural fit for partial reconfiguration since the FPGA is configured blocks at a time. If a program running on the processor was able to swap out computational elements in the datapath in real-time, an increase in performance and efficiency could be gained.
Bibliography


Appendix A

Header definitions
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Byte</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Opcode</td>
<td></td>
<td><strong>Opcode Header</strong></td>
</tr>
<tr>
<td>0x1</td>
<td>Parameter</td>
<td></td>
<td><strong>Parameter Header</strong></td>
</tr>
<tr>
<td>0x2</td>
<td>Reserved</td>
<td></td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>0x3</td>
<td>Payload Length</td>
<td></td>
<td><strong>Read Info Header</strong></td>
</tr>
<tr>
<td>0x4</td>
<td>Packet ID</td>
<td></td>
<td><strong>Read Delay</strong></td>
</tr>
<tr>
<td>0x5</td>
<td>Reserved</td>
<td></td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>0x6</td>
<td>Reserved</td>
<td></td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>0x7</td>
<td>Reserved</td>
<td></td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>0x8</td>
<td>Reserved</td>
<td></td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>0x9</td>
<td>Address</td>
<td></td>
<td><strong>Auxiliary Read/Write Header</strong></td>
</tr>
<tr>
<td>0xA</td>
<td>HeaderID 0</td>
<td></td>
<td><strong>Fork0 Header</strong></td>
</tr>
<tr>
<td>0xB</td>
<td>HeaderID 0</td>
<td></td>
<td><strong>Fork1 Header</strong></td>
</tr>
<tr>
<td>0xC</td>
<td>Fork Data</td>
<td></td>
<td><strong>Fork Data</strong></td>
</tr>
<tr>
<td>0xD</td>
<td>Packet ID</td>
<td></td>
<td><strong>Fork Data (Write Info)</strong></td>
</tr>
<tr>
<td>0xE</td>
<td>Reserved</td>
<td></td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>0xF</td>
<td>Packet ID</td>
<td></td>
<td><strong>Write Info</strong></td>
</tr>
</tbody>
</table>

Figure A.1: Header word definitions
Appendix B

Benchmark Programs
### B.1 Vector Multiply

; mult2k program
; Inputs:
; R0=Address of x
; R1=Address of y
; R2=Address for x*y
; Outputs:
; R0=Execution cycles

```
timer0 = 0 ; Clear timer to calculate execution time
vdpStatus.clr = 1 ; Clear vdpStatus

; Import x
; out0 = import
vdp2.ReadInfo = 0x08000000
vdp2.Opcode = 0x00000056
vdp2.Param = R0 ; e18_dataImport.addr
vdp2.WriteInfo = 0x01000000

; Import y, export x*y
; export = in0 * import
vdp0.ReadInfo = 0x08000000
vdp0.ReadDelay = 0x00000001
vdp0.Opcode = 0x00001D0
vdp0.Param = R2 ; e19_dataExport.addr
vdp0.WriteInfo = 0x01010000
vdp2.ReadInfo = 0x08000000
vdp2.Opcode = 0x00000006
vdp2.Param = R1 ; e18_dataImport.addr
vdp2.WriteInfo = 0x00000000

; Wait for all packets to finish
runningWait:
PC.cmp = vdpStatus.running
PC.rjz0 = runningWait

; Save execution time into R0
R0 = Timer0

; Halt
PC.rjmp = 0
```
B.2 FFT

;FFT2k program
; Inputs:
; R0=Address of x(0) (input data)
; R1=Address of X(0) (output data)
; R2=Number of 2k FFTs to compute
; Outputs:
; R0=Execution cycles

    timer0 = 0 ;Clear timer to calculate execution time
    vdpStatus.clr = 1 ;Clear vdpStatus
    adder0 = 0x01010000 ;Use adder0 to keep track of packetIDs
    adder1 = R0 ;Setup address for import
    adder2 = R1 ;Setup address for export
    cnt0 = R2 ;Setup loop counter

loop:
    ;Compute 2k FFT
    ;export = fft(import)
    vdp2.ReadInfo = 0x08000000
    vdp2.Opcode = 0x0000EACE
    vdp2.Param = adder1 ;e22_dataImport.addr
    vdp2.Param = adder2 ;e23_dataExport.addr
    vdp2.WriteInfo = adder0

    adder0.add = 0x10000 ;Next packetID
    adder1.add = 2048 ;Next address
    adder2.add = 2048 ;Next address

    ;Loop if cnt0 > 0
    PC.cmp = cnt0
    cnt0.dec = cnt0
    PC.rjz0 = loop

    ;Wait for all packets to finish
runningWait:
    PC.cmp = vdpStatus.running
    PC.rjz0 = runningWait

    R0 = Timer0 ;Save execution time into R0
    PC.rjmp = 0 ;Halt
B.3 FIR

;FIR filter program
; Inputs:
;   R0=Address of h (filter coefficients)
;   R1=Length of h
;   R2=Address of x (input data)
;   R3=Length of x
; Outputs:
;   R0=Execution cycles

;Clear timer to calculate execution time
timer0 = 0

;Clear vdpStatus
vdpStatus.clr = 1

;Setup packetID counters
adder0 = 0x01010000 ;Use adder0 to keep track of WriteInfo / packetIDs (start with ID=1)
adder1 = 0x00000001 ;Use adder1 to keep track of ReadDelay / packetIDs (start with ID=0)

;Import h to bank2
;out2 = import
vdp2.ReadInfo = 0x08000000
vdp2.Opcode = 0x00000156
vdp2.Param = R0 ;e22_dataImport.addr
vdp2.WriteInfo = adder0
adder0.add = 0x10000 ;Next packetID
adder1.add = 0x10000 ;Next packetID

;Calculate H
;out1 = fft(windowGen.bandstop(in2))
vdp2.ReadInfo = 0x08000000
vdp2.ReadDelay = adder1
vdp2.Opcode = 0x0001567D
vdp2.Param = R1 ;e13_windowGen.edge1
vdp2.Param = 2048 ;e13_windowGen.edge2
vdp2.WriteInfo = adder0
adder0.add = 0x10000 ;Next packetID
adder1.add = 0x10000 ;Next packetID

;Calculate N-M+1 and store result in R0
sub = 2049
sub.sub = R1
R0 = sub

;Setup last address (compared against)
adder2 = R2 ;Addr x
adder2.add = R3 ;Len x
R3 = adder2 ;Addr x + len x

;Setup address for import/export
adder2 = R2

loop:

;Compute 2k FFT
;out0 = fft(import)
vdp2.ReadInfo = 0x08000000
vdp2Opcode = 0x00000ACE
vdp2.Param = adder2 ;e22_dataImport.addr
vdp2.WriteInfo = adder0

adder0.add = 0x10000 ;Next packetID
adder1.add = 0x10000 ;Next packetID

sub = R1
sub.sub = 1; %length(h)-1

;Multiply by H and compute 2k iFFT
;export = ifft(in0 * in1)
vdp0.ReadInfo = 0x08000000
vdp0.ReadDelay = adder1
vdp0.Opcode = 0x0003F574
vdp0.Param = adder2 ;e23_dataExport.addr
vdp0.Param = sub ;e23_dataExport.segStart
vdp0.Param = 2048 ;e23_dataExport.segStop
vdp0.WriteInfo = adder0
vdp1.ReadInfo = 0x08000000
vdp1.ReadDelay = adder1
vdp1.Opcode = 0x00000000
vdpl.WriteInfo = 0x00000000
adder0.add = 0x10000 ;Next packetID
adder1.add = 0x10000 ;Next packetID
adder2.add = R0 ;Increment address

;Loop if adder2 < R3
cmp = R3
cmp.cmp = adder2
PC.cmp = cmp.less
PC.rjz0 = loop

;Wait for all packets to finish
runningWait:
PC.cmp = vdpStatus.running
PC.rjz0 = runningWait

;Save execution time into R0
R0 = Timer0

;Halt
PC.rjmp = 0