Functional and Timing Models for a Programmable Hardware Accelerator

by

Matthew Fox

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2015

Certified by . . . .
Joel Emer
Professor of the Practice
Thesis Supervisor

Certified by . . . .
Vivienne Sze
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Prof. Albert Meyer
Chairman, Masters of Engineering Thesis Committee

Functional and Timing Models for a Programmable Hardware Accelerator

by

Matthew Fox

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Classes of problems that exhibit high levels of data reuse have heavy overhead costs due to the necessity of repeated memory accesses. A proposed programmable hardware accelerator using a spatial architecture paired with triggered instructions could solve this class of problems more efficiently. Performance models are useful for evaluating the performance of different applications and design alternatives. This project develops a functional model for this hardware accelerator as well as a simple timing model to allow for examination of the performance of this proposed architecture. Furthermore, convolution, pooling, and non-linearity are implemented as programs on this accelerator in order to show functionality and flexibility present within the system. This project shows the feasibility of a spatial architecture powered by triggered instructions and provides the framework for future testing and prototyping for this problem space.

Thesis Supervisor: Joel Emer
Title: Professor of the Practice

Thesis Supervisor: Vivienne Sze
Title: Assistant Professor

Acknowledgments

I would like to thank my family and friends for the continued support in helping me to complete this project when I was often overwhelmed, tired, or found a roadblock. I felt constantly pushed and supported throughout this experience because of the support system that I have found at MIT.

Finally, I would like to thank the members of my group that made the project possible. My co-supervisors Professor Joel Emer and Professor Vivienne Sze provided endless assistance in problem understanding, design decisions, and implementation that allowed me to complete this project. Their advice throughout this experience has not only pushed me in my work on this thesis, but also has provided me with useful best practices and approaches to hard problems that I will inevitably use often in my future career.

Contents

1 Introduction **11**

1.1 Motivations for Spatial Architectures 11

1.2 Current Solutions 13

1.3 Triggered Instruction Hardware Accelerator 13

1.4 Project Motivation 14

1.5 Thesis Chapters 15

2 System Overview **17**

2.1 Compute Modules 17

2.2 Interconnection 18

3 The Functional Model **19**

3.1 Compute Modules 19

3.1.1 Memory Streamer 20

3.1.2 Multiplier 22

3.1.3 FIFO 22

3.1.4 TIA Processing Element 23

3.2 Latency-Insensitive Channels 28

3.2.1 Multichannels 29

3.3 Static and Dynamic Instructions 29

3.3.1 Normal Module Instructions 29

3.3.2 TIA Instructions 30

4 The Timing Model **33**
4.1 Configuration . . . 33
4.2 Running the Functional Model . . . 34
4.3 Cycle Times . . . 34

5 Implemented Examples **37**
5.1 Nonlinearity . . . 37
5.2 Pooling . . . 38
5.2.1 Maximum . . . 39
5.2.2 Average . . . 39
5.3 Convolution . . . 43

6 Future Work and Discussion **47**
6.1 Results . . . 47
6.2 Main Challenges . . . 48
6.3 Future Work . . . 48
6.4 Impact . . . 49

A Nonlinearity Implementation **51**

B Maximum Implementation **55**

C Average Implementation **59**
C.1 Subtracting Divider Implementation . . . 59
C.2 Shifting Divider Implementation . . . 64

D Convolution Implementation **73**

List of Figures

1-1 Convolution Sliding Window 12

3-1 Example Functional Model Configuration 19

3-2 Memory Streamer Block Diagram 21

3-3 Multiplier Block Diagram 22

3-4 FIFO Block Diagram 23

3-5 TIA Processing Element Block Diagram 24

3-6 Methods of the TIA module 26

4-1 Cycle Time CSV File Example 35

4-2 Example Timing Model Output 35

5-1 Example Output from Nonlinearity 38

5-2 Plot of Divider Results 42

5-3 Tail Growth of Shifting Divider 43

5-4 Graph of Filter Size vs. Cycle Counts 45

Chapter 1

Introduction

As in any hardware accelerator, the idea for this design is to decrease cost, whether speed, space, or otherwise, by applying hardware that is specifically designed for a set of problems. The main vision for this particular hardware accelerator is to leverage sub problem independence and data reuse in algorithms well fit to spatial architectures.

1.1 Motivations for Spatial Architectures

When working with hardware accelerators, it is important that the architecture is well matched to the problem set. As an example, convolutional neural nets fall into this category of problems. Convolutional neural nets are extremely powerful in today's world due to their application in important problems. One example in which convolutional neural nets (CNNs) have been used successfully is in classifying high-resolution images with very low error rates [3]. Furthermore, CNNs have been used to filter images very quickly. Let's look at how this applies to our spatial architecture.

In order to filter an image, one will first grab the data for the filter and then the first block of data for the image. After applying the filter to this block of the image data, the filter will then be applied to a new block that encompasses much of the same data as the first, which can be seen in Figure 1-1. This process is repeated until the filter has been applied to the entire image, using the filter as a sliding window

across the image as seen in the figure.

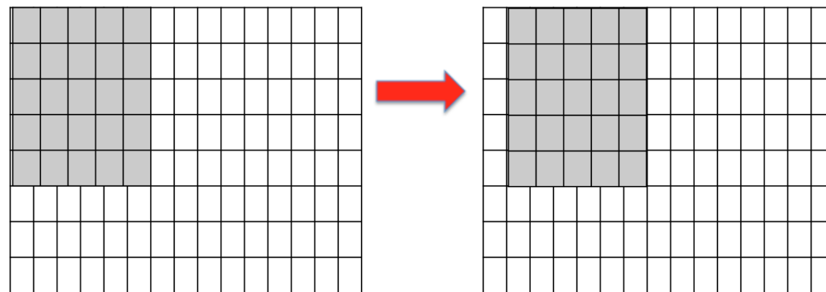


Figure 1-1: The sliding window above portrays how the filter (gray) applies to many of the same pixels of a given image in subsequent convolution steps.

In order to do this, we normally have to have separate memory accesses for each time we need a certain block of pixel data for the image. However, we already have used this data before in the last convolution step, so we are wasting cycles grabbing the same image data multiple times.

In a normal pipelined processor, or even an out-of-order superscalar processor, we would be unable to amortize this cost of constantly retrieving data. This is simply one use case for the hardware accelerator that we are developing. There exist many applications in which data is reused and must be constantly retrieved from memory. One may wonder why this problem has not been tackled before, and it has in such systems as Transport-Triggered Architectures [6], AsAP (Asynchronous Array of Simple Processors) [8], and an extended version of AsAP with 167 processing elements [7]. However, we will show that this architecture is more efficient in space and time than those currently in existence while maintaining configurability necessary for a general accelerator.

You may be familiar with this idea of hardware acceleration in relation to vectorizable algorithms that see large efficiency boosts when run on a vector engine. These boosts are seen because the problem type is very fit the the hardware that is used to run it. A vector engine is able to process problems simultaneously when presented with a proper vectorized structure as is possible in a verctorizable algorithm. In the same way, workloads that are well suited to spatial programming can see significant

boosts when run on an enabling architecture [1]. Many papers, such as the ones listed above, have shown that there is promise in this area and this project aims to help evaluate just how effective these hardware accelerators can become.

1.2 Current Solutions

One way that people currently approach speeding up solutions to this class of problems is by using FPGAs. One nice aspect of FPGAs is that they include a similar configurable aspect that we have included in our design for the hardware accelerator that will be described later. The problem with using FPGAs to solve this set of problems is that their lookup table based data paths cannot reach the compute density of either a microprocessor or a SIMD engine [1].

The next major class of proposed solutions use what is known as a spatial architecture. A spatial architecture uses an array of processing elements instead of a single processor with centralized control. By making very simple and small processing elements, one can use on the order of hundreds of these processing elements in a reasonable configuration. The spatial array is also highly connected so that data can be passed between the processing elements in order to leverage data reuse and share results among processing elements.

Some current proposed solutions use a similar coarse-grained ALU-style data path that achieves very good compute density such as [4] and [5]. However, all previously proposed architectures use some form of program counter centric control, which can be shown to introduce several inefficiencies when capturing ALU control patterns for highly connected spatial architectures [1].

1.3 Triggered Instruction Hardware Accelerator

As introduced at ISCA in 2013, the TIA (Triggered Instruction Architecture) Hardware Accelerator improves on the state of the art. As described in [1], this design gets high compute density while efficiently capturing ALU control patterns for

highly connected spatial architectures. Furthermore, it has been shown through run-throughs that this architecture can achieve 8x greater area-normalized performance over general-purpose processors as well as a 2x speedup over program-counter(PC) based spatial designs [1]. Some algorithms show even greater performance gains. Notably, running edit-distance on this platform shows 50x runtime performance improvement and 100x reduction in energy compared to a normal x86 processor [2].

These massive improvements are due to the spatial architecture design that allows data to flow between many processing elements instead of constantly fetched from the memory controller. Furthermore, this design benefits greatly from triggered-instructions, which remove the program counter completely and allow the processing elements to have state as in a finite-state machine, transitioning smoothly without any branch or jump instructions - one of the biggest improvements from PC-based control [1]. TIA instructions are able to do this by utilizing a concept known as guards. A guard is a condition on the processor state that will allow a given instruction to trigger. Each instruction defines a group of guards, often on the predicate state of the processor, which are checked by the scheduler before an instruction is chosen. From the set of instructions that can be run at a given time, the processing elements uses a simple algorithm to chose which instruction or instructions to launch. These instructions, in turn, can update the state of the processing element and launch other instructions. These improvements alongside other optimizations have proven that this design has great promise as the future in spatial architecture.

1.4 Project Motivation

At this point, it seems clear that more research into this programmable hardware accelerator should be pursued due to the theoretical results that we talked about above. Building a functional model - the task of this thesis - is the next step towards producing an end product that can achieve the performance improvements on this class of problems. We must evaluate that these gains for convolutional neural nets can be realized and assess design alternatives for the hardware accelerator.

Furthermore, this thesis also encompasses a simple timing model - a driver that controls and connects the functional blocks of the spatial architecture. In this project, we built a separate functional and timing model, which provides benefits over a single monolithic model that controls both timing and functionality. The main benefit is that the architecture of the blocks - the functionality present in the functional model - does not change much over time. However, the timing model is extremely mercurial in the fact that it is common to want a change in construction or configuration for a single hardware model. By separating the functional and timing models, we allow for many timing models to match to a single functional model. This allows the functional model of the hardware accelerator to persist much longer than would otherwise be possible. This is the standard for conventional system architecture, and the challenge will be to apply it for a hardware accelerator.

1.5 Thesis Chapters

Chapter two describes the overall design of the hardware accelerator.

Chapter three describes the design of the functional model for the hardware accelerator, as well as the implementation details necessary for running the functional model.

Chapter four describes the design of the timing model and the key implementation decisions there.

Chapter five describes the implemented examples that show the flexibility and functionality of the functional and timing models.

Chapter six describes the conclusions and the future work planned for this project.

Chapter 2

System Overview

Now that the premise of a triggered-instruction architecture driven hardware accelerator has been supported, let us take a look into the specifics of the system for which the performance models were built. We will start by taking a brief look at the compute modules, and then move towards the interconnections between the modules themselves.

2.1 Compute Modules

Within the design of this hardware accelerator, there are currently four compute modules - the memory streamer, the multiplier, the FIFO, and the TIA processing element. However, both performance models and the system design allows for a larger library of compute modules should the need ever arise. Let's take a quick look at each of the modules.

The memory streamer module is used as a way to send data from a memory source to other compute modules using a regular access cycle. Furthermore, one can also use the memory streamer to write data results back from a compute module into a memory source.

The multiplier module is used as an external compute unit in order to keep the processing element small in size and simple in function. Any multiplications that must occur in the system for a given algorithm must pass through an instance of this

module.

The FIFO module is the traditional first-in first-out buffer used in many systems. In this hardware accelerator, it is often used to temporarily store data that must be recirculated through a processing element.

The most complex and arguably most important compute module is the TIA processing element module. The TIA module is intended as the computation centers of the hardware accelerator. In the accelerator, there are on the order of hundreds of these modules that are highly connected to pass data between the modules in order to leverage high data reuse properties of certain algorithms. This module contains a block of instructions, a scheduler that triggers these instructions, and an ALU that executes and complete each instruction.

2.2 Interconnection

As stated above, one of the most novel and powerful aspects of this hardware accelerator design is the connectivity of the compute modules. The connectivity of the compute modules allows data to pass between processing elements, which cuts down on necessary shared memory accesses, improving the speed of running certain algorithms. This means that we need to design a mesh-like network between modules that allows for high connectivity. This network is composed of single channels that are implemented as latency-insensitive channels.

Latency-insensitive channels provide two guarantees. Firstly, at least one message must be able to be in flight at any given time, and, secondly, messages are delivered in a first-in, first-out manner. Connecting compute modules with these channels allows us to give network creation more flexibility by keeping the guarantees minimal.

Chapter 3

The Functional Model

Figure 3-1 is an example of a connected configuration of the functional model. The modules and connections that go into such a configuration will be explained below.

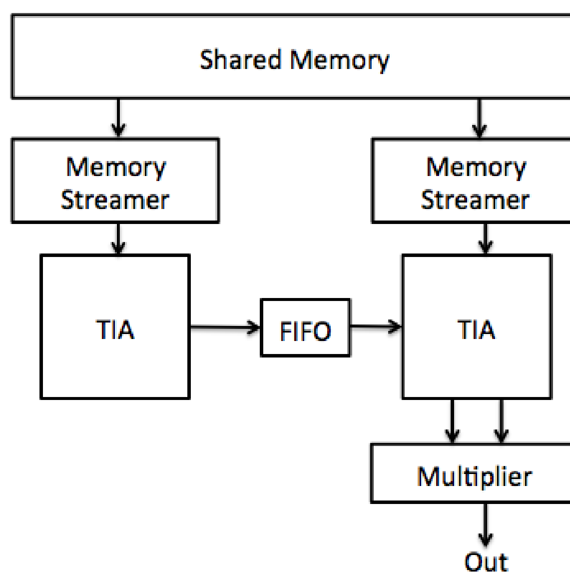


Figure 3-1: Above is a block diagram for an example configuration of the functional model. The user can connect the compute modules in any way that they would like and have any number of compute modules and interconnections.

3.1 Compute Modules

The functional model is broken down into the following four compute modules:

- Memory Streamer
- Multiplier
- FIFO
- TIA Processing Element

All of these modules inherit from a base class that defines three useful methods across all modules. We designed the base class so that the timing model has a consistent and simple interface with which to interact with the functional model. The following methods are available across all compute modules:

1. Schedule Method: The schedule method decides which instruction can run for a given module, and returns a dynamic version of this instruction to the timing model.
2. Execute Method: As the name suggests, this method computes the result of the instruction that it was passed. The results are stored in a vector that is returned to the timing model as part of the dynamic instruction. In this stage, the results of the instruction have not been made visible to the architectural state of the module.
3. Commit Method: This method frees any resources, updates any module state, and outputs the uncommitted result.

Each module described below has been carefully split into these three methods such that the interface is quite simple.

3.1.1 Memory Streamer

Firstly, let's look at the memory streamer. The memory streamer is intended to constantly send data from a centralized shared memory through its output channel to be used by other blocks. In order to perform this action, it takes a base address, a stride between addresses to be sent, and an end address in its constructor. These

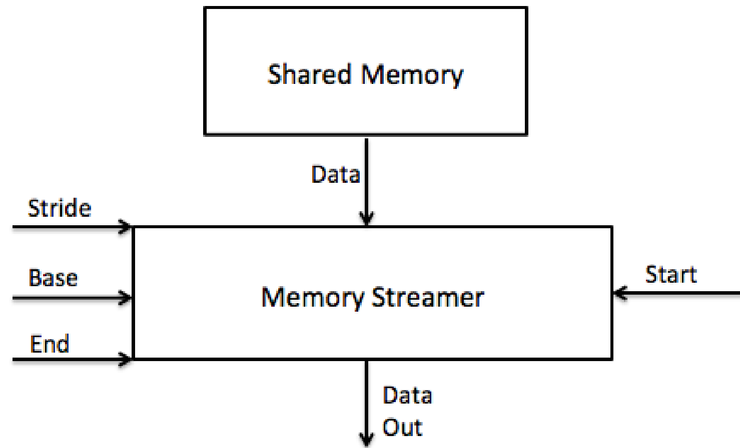


Figure 3-2: Above is the block diagram for the memory streamer configured in read mode. The write mode would be identical except the Data and Data Out channels would push data in the opposite direction.

can be updated over time and serve as state for the memory streamer. The memory streamer will be ready to execute operations whenever its output channel is not full and its current address is not greater than its end address. Finally, this module allows the timing model to update the contents of memory so that we can load any instructions or data into the memory directly from the timing model. This module is very simple both in function and construction.

The implementation of the memory streamer includes six input channels. Three of these channels allow for updating the state variables of the memory streamer to access a new area of memory - stride, base address, and end address. These internal variables are updated when all three channels have data so that the updates are affected together. There is also a channel that is designated to send a start or stop signal to the streamer to initiate or pause a stream. Finally, we have one channel each for data coming in and data going out. The data-in channel is used when the streamer is configured in write mode to load the correct data into the memory. The data-out channel is used when the streamer is configured in the read mode to send the data from memory out to another module.

In order to maintain flexibility in the timing model, the streamer is passed a

pointer to memory that is instantiated in the timing model. This allows the timing model to set up one or many memory arrays that can be written or read.

3.1.2 Multiplier

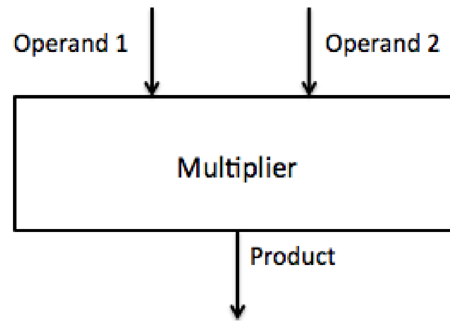


Figure 3-3: Above is the block diagram for the multiplier module. It takes two arguments as operands and returns a product.

Next, let's look at the multiplier. The multiplier takes two operands as input and outputs the product. Because the processing elements are intended to be as simple as possible, they do not have the ability to multiply operands, which is why the multiplier module is necessary. The multiplier module's schedule method checks if both operands are available in the input channels and that its output channel is not full. The execute method computes the multiplication of the operands and stores the result in a dynamic instruction. Finally, the commit method sends this product to the output channel.

3.1.3 FIFO

The next compute module is the FIFO. The FIFO is intended to be a storage device that works in a first-in first-out (FIFO) manner to store messages throughout the system. The schedule method checks whether we have data incoming that we need to store and room to store it. The schedule method also checks if we have data and room in the output channel to send it forward. The execute method grabs the data from the input channel on a push (adding data to the FIFO) or grabs the data from

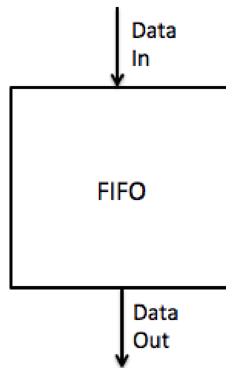


Figure 3-4: Above is the block diagram for the FIFO module.

the storage buffer on a pop (sending data from the FIFO). This data is returned with the dynamic instruction along with the proper tag for the data. The commit method updates the FIFO's pointers to indicate how the data in the circular buffer has been changed as well as update the channels with the proper data.

When any FIFO is initialized, it takes as an argument the size of the allocated memory space so that it will know when to wrap around the data storage using a circular buffer.

In addition to the normal schedule, execute, and commit methods present in all modules, the FIFO also includes helper methods that can check if the FIFO is full or empty. These methods simply compare the read and write pointers into the circular buffer and see if the parity of the number of loops that each pointer has made is equal. For example, the buffer is empty if the read and write pointers are equal and each has looped the same number of times. If the pointers are equal but the loop counters are not, then the FIFO is full.

3.1.4 TIA Processing Element

Finally, we will take a look at the TIA processing element module. This compute module is by far the most complicated and intricate of all the modules. This is the module that will allow for the major gains that we have seen in the TIA spatial

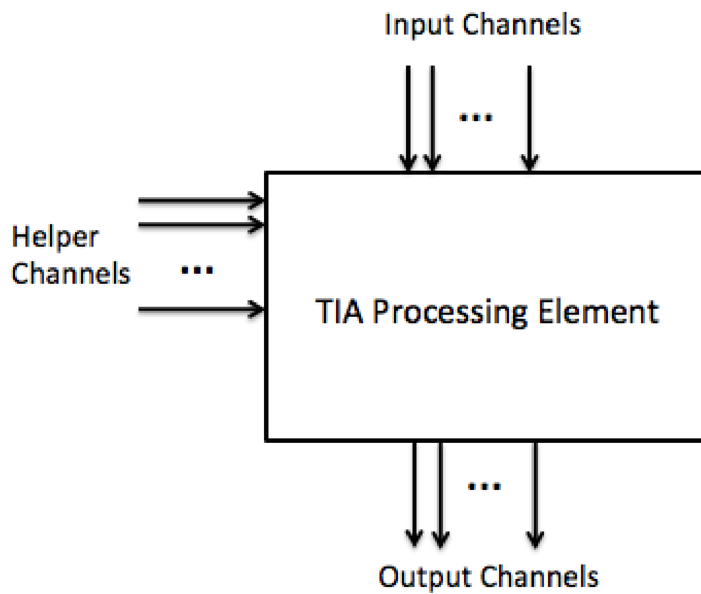


Figure 3-5: Above is the block diagram for the TIA processing element module. There can be a variable number of helper, input, and output channels as defined when instantiating of the object.

architecture.

Constructor

Let's start by taking a look at the constructor for the TIA processing element. The first argument into the TIA module is a group of helper channels. These channels are in place to load instructions into the instruction memory of the TIA module, edit these instructions during runtime, and complete any other module updates that are not handled with the normal instruction set. The second argument is a group of input channels. These are used as possible inputs to instructions that are run by the module. There may be any number of input channels depending on the initialized value. The third and last channel argument is the group of output channels. These channels connect any output of the TIA module to other modules in the system whether they are additional TIA modules or otherwise. Again, there can be a variable number of output channels that is decided when an object is initialized in the timing module. Each of these channel arguments is passed into the constructor as a member of the

Multichannel class that is described in Section 3.2.1.

Registers

The last two arguments into the constructor of the TIA processing element are the number of predicate registers and the number of data registers. As described in the original ISCA paper on this topic, predicate registers contain boolean values that describe the current state of the TIA module. Triggered instructions can check these as guards to see if they can run at a given time. Data registers are the classic processor construct to store small amounts of local memory within the processing element.

Both the predicate registers and the data registers have their own class with associated methods to simplify the interface between the registers and the TIA module. Each class provides methods to add registers, get the value of a given register, set the value of a given register, set the state of the register file as a whole, get the number of registers, and get the state of the register file. These methods have proved very useful in creating a clean and readable interface into these register files.

Finally in the constructor, we initialize the scoreboards that we will use to manage resource conflicts. In the TIA module, we create scoreboards for all possible input elements - the predicate registers, the input channels, and the data registers. A bit is set to true when any given register or channel is in use and then reset upon commit of the instruction that uses the resource.

Figure 3-6 shows the different methods available for the timing model on the TIA module. Each of these stages will be described in detail below.

Schedule Method

Now, let's take a look at the schedule method of the TIA module. The first action that can be completed by the schedule method is to add an instruction to the instruction memory. In order to do this, a method is called within the TIA module that adds the instruction. This method checks that the added instruction is legal in that all input sources and output destinations are within the range of valid resources given the construction of the object.

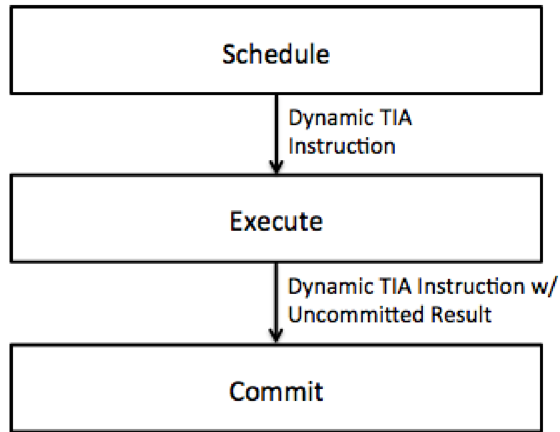


Figure 3-6: Above depicts the methods for the TIA module along with the data types that are passed between the methods.

Next, the schedule method will call another helper method to see which instructions in the instruction memory can trigger given the processor state. In order to check this, the module check to see if all input channels have valid data, any input flags are correctly set, the predicate state is correct for the instruction, the output channels can accept an output, and none of the used resources are currently being used as stated by the scoreboards. This method returns a vector of indexes into the instruction memory so that the schedule method can continue to reference the correct instructions.

Assuming at least one instruction can trigger, the schedule method calls a final helper method to trigger a list of instructions. In the current implementation, we simply choose to launch the first instruction that can be triggered. However, in future implementations, one could decide to launch multiple instructions at once. The trigger method will add all used resources to the scoreboard and return a dynamic instruction to the timing model. If no instructions can trigger given the state of the processor, a NOP is inserted and returned to the timing model.

Execute Method

Now we move onto the execute method of the TIA module. The first check is that the instruction is a valid instruction and not a NOP. If we are dealing with a NOP, then

we will simply return it to the timing model. In the case that we are dealing with a valid instruction, we will get the static instruction from the instruction memory in order to have access to important fields within the instruction.

The next operation involves getting arguments and placing them in a vector for easy access throughout the method. Using the input fields of the static instruction, we store the arguments and any associated tags in vectors. This will allow us to treat all arguments in the same way when we compute the result of the ALU command.

After all arguments have been gathered, it is time to do the computation specified by the ALU command. Using a simple switch statement, all command calculations are implemented cleanly. The result or results get stored in a vector of uncommitted results that get added to the dynamic instruction and passed back to the timing model.

Given the result of the ALU, we are able to decide what exact predicate updates will take place. Given the static predicate updates, we can decide which updates to complete based on the low order bit, the high order bit, a comparison against zero, or any case. These updates are passed along to the dynamic instruction to affect the predicate state in the commit stage. Finally, the dynamic instruction along with the uncommitted results and valid predicate updates are passed along to the timing model to be used for the commit stage.

Commit Method

In the commit stage of the TIA module, all of the results of the execute stage affect the state of the TIA module. The uncommitted results are written to any output channels or data registers as specified by the static instruction. Any input channel that is slated to be dequeued is handled in this stage. All predicate updates are also put into effect. Finally, any resources that have been in use for the TIA module are released from the scoreboards.

Much care has been put into the design of the TIA module to ensure that all implemented instructions and data constructs are as flexible as possible and that adding new ALU instructions would be as simple and learnable as possible.

3.2 Latency-Insensitive Channels

Functionality for the connecting the compute blocks is the next and final portion of the functional model. In our design, the network is key in allowing the model to pass data throughout the system efficiently and generally. In order to keep the network as simple and general as possible, we have chosen to implement latency insensitive channels as a means of building a network.

Latency insensitive channels are connections from one block to another that follow a set of simple rules. The first rule is that every channel must be able to have at least one message in-flight at any given time. In our implementation, the total number of messages possible in flight is parameterized when the object is instantiated. The second and final rule is that messages must be delivered in a first-in first-out method. Other than these two promises, the channels may behave in any way.

To build the functional model, we have allowed for two properties - 1. Compute blocks must be able to tell if a channel can accept another message and if there is a message that can currently be received. 2. The timing model must be able to set when a message has reached the end of the channel and if a channel has room to receive more messages. In order to allow for these properties, we have created internal variables that set if a channel can accept or release new messages. Compute modules have access to these variables through public methods, but cannot update the variables. The timing model has access to methods to update the state of the channel at the beginning of every update loop.

As a final note, the channels carry packets that include two fields - the data field and the tag field. The data field of a packet is simply an integer value. The tag field is a single bit that accompanies the integer and can signal the processor of any information that needs to accompany the integer. An example of a use for the tag would be to notify the TIA module that there is no more data for a given operation.

3.2.1 Multichannels

Multichannels are a very simple class that provide a powerful tool for interacting with Latency-Insensitive channels. The purpose of Multichannels is to bundle together channels to pass them into and between modules.

Methods on Multichannels include adding a channel to an internal vector and getting the total number of channels. The TIA module can directly access the vector within the Multichannel, so we can use the indexing operator into the vector to access the channel objects directly.

3.3 Static and Dynamic Instructions

One of the key features of this functional model is the ability to store instructions in a concise yet variable way. Now, we take a look at two different types of instructions - normal module instructions and TIA instructions - to discuss the design decisions made in storing and communicating instruction information. For each of these types of instructions, we have both the static and dynamic varieties. The static version of either instruction contains the necessary information for an instruction to launch and begin running. The dynamic version of an instruction is used when the instruction is in-flight so that the module can store important information in a well-defined construct that travels with the instruction information.

3.3.1 Normal Module Instructions

The first type of instruction is quite simple in nature. All compute modules except the TIA module use the normal instruction class in operation. The normal static instruction is simply an ALU-type command that is defined in an enumerated type. The following are the possible ALU-type commands that are implemented along with the compute module that uses them:

- FIFO: NOP, Push, Pop
- Memory Streamer: NOP, Start, Stop, Read Stream, Write Stream

- Multiplier: NOP, Multiply
- TIA Module: NOP, Increment, Decrement, Add, Subtract, Less than (or equal), Greater than (or equal), Pass value through, Pass through two values, Dequeue, Shift left/right

The dynamic version of the normal instruction allows dynamic information to be stored alongside the ALU-type command. We have created a class that includes the ALU-type command along with a vector for the uncommitted results of any option. For interacting with this class, there are public methods to get or set the uncommitted result as well as get or set the ALU-type instruction that accompanies the dynamic instruction.

3.3.2 TIA Instructions

The TIA module is the only module to break away from the normal instruction type because the needs of the TIA's instruction class far outnumber the needs of any other module's instruction type. This section will focus on describing all of the data that is stored in an instruction and the use of each type of data. For each internal variable that is discussed, there are proper public methods for getting and setting the values of this variable. To avoid redundancy, these methods are not mentioned in the following discussions.

The first internal variable to a TIA instruction is the ALU command that will be used in the execute stage. The possible ALU commands that are implemented are listed in section 3.3.1, though the process of adding more ALU commands is quite simple and described in the documentation.

The next internal variable is the predicate state required for the TIA instruction. This construct is a vector of integer, boolean pairs which state the index of the predicate register and the value of the register that is required for the instruction to trigger. These are easily associated with a basic guard on an instruction or a state in a state machine.

Flags on input channels are another internal variable stored with any TIA instruction. These are simply the tags for the next packet to be fetched from any input channel. Again, we have built a similar structure using a vector of integer, boolean pairs which state the index of the input channel and whether the tag should be a 1 or a 0. One good use of this would be to launch different instruction based upon if we are at the end of a data string or still processing data.

The next construct for the TIA module is the input vector. This vector contains pairs of two integers. The first integer determines the source of input - channel, data register, or predicate register state. The second integer is an index into each of these constructs. When checking if a TIA instruction can trigger, we use the first integer to determine which scoreboard to check and the second to index into the scoreboard and see if the resource is free.

Similar to the input vector is the output vector. This vector contains pairs of a boolean and an integer. The boolean determines if the output should go to a channel or a data register, and the integer determines the index into either the channel vector or the data register file.

Our next internal variable is the dequeue list. When running an instruction, input channels are not automatically dequeued when the data is grabbed. Instead, the instruction explicitly lists which channels it should dequeue upon completion. In practice, this will likely be a subset of the channels used for input, but the flexibility is currently provided for any instruction to dequeue any channel or even dequeue a single channel multiple times.

The possible predicate updates are also included in the static instruction. These updates are stored as three pieces of information: a predicate update type, the index into the predicate register file, and the default update value. The predicate update type is an enumerated type that has four values: ANY, ZERO, LOB (low order bit), and HOB (high order bit). ANY means that the predicate update should follow the default value. ZERO means that the predicate should update to true when the ALU result is zero. LOB means that the predicate should update to true when the ALU result's low order bit is 1. HOB means that the predicate should update to true when

the ALU result's high order bit is 1. If ZERO, LOB, or HOB do not update that predicate to true, then it is updated to false.

The final construct stored in a static TIA instruction is the data register update list. This allows any instruction to write constants into data registers. The information is stored in a vector that contains the data register's index and the value to write to that register. A common use case for this construct is to zero out a register before using it as a local variable.

The dynamic version of the TIA instruction is similar to the static instruction in many ways. One of the key design decisions made about the dynamic instruction was to not include a static instruction as part of the dynamic instruction, but instead, to refer to the static instruction by using an index into the TIA module's instruction memory. By taking this approach, we have to communicate less data between the functional and timing models about an in-flight instruction.

This decision to use an index instead of an internal static instruction opens the opportunity that an instruction could be updated by a helper channel during the lifetime of a dynamic instruction. We decided that this occurrence would be rare but must be considered by the instruction designer/compiler when deciding to update instruction memory banks.

Other than a pointer to the static instruction, the dynamic instruction also holds a vector of uncommitted results that is used to pass the ALU results from the execute stage to the commit stage. Furthermore, the predicate updates that were compiled in the execute stage using the output of the ALU are included in a vector to allow the commit stage to update the predicate registers upon completion.

Using the dynamic instructions that pair with static instructions allows for the necessary flexibility in results as well as the organized structure to maintain simplicity across both the functional and timing models.

Chapter 4

The Timing Model

The timing model built for this thesis is intended to be simple in nature. The major emphasis of this thesis was building the functional model, and we created the timing model as a wrapper to provide to us a handle to work with the functional model and to show its flexibility. We use the timing model to complete three tasks - 1. Initialize and program all blocks for a given configuration. 2. Run the schedule, execute, and commit methods of all compute blocks. 3. Compute the cycle count for any output of any program. The timing model uses a mapping between instruction and cycle counts in order to measure how long each update loop, and therefore program, takes. The intent of building this timing model was generalization - we would like it to cleanly run all configurations and run through all methods for compute blocks in the given configuration.

4.1 Configuration

The first step of this timing model was to handle a designed configuration. This meant that the timing model must connect all of the instance of functional modules together via a network of latency insensitive channels. Furthermore, it configures and programs all functional modules. This entails such tasks as loading data into the main memory, giving instructions to the TIA blocks, giving initial parameters to the memory streamer module, and other module configurations.

The configuration step of the timing model is one of the areas for future research. we believe that it will be necessary to create a compiler-type program to port higher-level code into TIA instructions. This is discussed more in Section 6.2. To see some of the examples of TIA instructions that have been implemented in example timing models, take a look at Appendix A, B, C or D.

4.2 Running the Functional Model

In this timing model, there is a clear analogy to a normal pipelined processor. In a pipelined processor, you will often have 5 stages - fetch, decode, execute, memory, and write back. Just as we have described above, the timing model will be responsible for running these sorts of stages for the compute blocks - our schedule, execute, and commit stages. In order to run all modules in this fashion, we simply create a vector of all objects of each module and then call the schedule, execute, and commit methods on each of these modules.

4.3 Cycle Times

One of the more interesting aspects of the timing model is the calculation of cycle times. Because specific cycle counts vary within a module , we allow the user to define cycle counts for the schedule and commit methods as well as for every ALU-type command in the execute stage. This is important because we now have a dynamic total time spent in one update loop that we must calculate while running the timing model.

For a user to update these cycle counts, we have provided a CSV (comma separated values) file that is processed by the timing model each time it is run in order to update the internal vectors that hold the cycle counts. An example of this CSV file can be seen in Figure 4-1.

During an update loop, we must calculate the most cycles that any given module takes in order to see what we must add to the total cycle time. In order to do this, we

	Schedule	Execute	Commit
TIA_NOP	2	1	1
TIA_PlusPlus	2	2	1
TIA_MinusMinus	2	3	1
TIA_Add	2	2	1
TIA_Subtract	2	2	1
TIA_LessThan	2	1	1
TIA_GreaterThan	2	1	1
TIA_LessThanEqual	2	1	1
TIA_GreaterThanEqual	2	1	1
TIA_PassThrough	2	1	1
TIA_PassThrough2	2	2	1
TIA_Deq	2	1	1
Multiplier_NOP	2	1	1
Multiplier_Multiply	2	8	1
MemStream_NOP	2	1	1
MemStream_Start	2	2	1
MemStream_Stop	2	2	1
MemStream_Read	2	7	1
MemStream_Write	2	9	1
Fifo_NOP	2	1	1
Fifo_Push	2	3	1
Fifo_Pop	2	5	1
TIA_ShiftRIght	2	2	1
TIA_ShiftLeft	2	2	1

Figure 4-1: Above depicts an example CSV file that would be used to calculate the number of cycles that are used for any instruction on any module.

first calculate the overhead for each type of module, and then add in the maximum cycle count for the ALU-type instructions run on any instance of that module type. After running all methods on every type of module, we take the maximum of the maximum cycle counts per module to get an overall maximum cycle count. This is then added to the global cycle time.

The cycle times are reported back to the user when any output appears from the system. An example of the output of running the timing model can be seen in Figure 4-2. In a future version with longer tests and more complex instruction control, these counts will be written to a log file and stored for future use.

```
Sum: 30000000
Count: 5
Output: 6000000
Number of Cycles: 2079
```

Figure 4-2: The above screenshot shows an example output of running the timing model under a given configuration.

Chapter 5

Implemented Examples

In order to show that the functional and timing models both work properly and provide the necessary constructs needed for this hardware accelerator, we have implemented four units of code using different instances of the timing model. Hopefully these examples prove instructive and show the flexibility and power of this project.

Furthermore, we chose to implement functions for nonlinearity, pooling, and convolution because these form the basis of convolutional neural nets. We believe that these three examples will provide a useful range of functionality and prove helpful for future endeavors

5.1 Nonlinearity

Nonlinearity was one of the simpler and easier functions to implement in a timing model. Our version of nonlinearity involves checking if an input value is greater or less than zero and then scaling that value by different predetermined factors depending on the result. To build this example, we simply used a single TIA module with two input channels and two output channels and a single multiplier. One of the input channels is used to stream the multiplying factors to the TIA module, and the other input streams the data to be scaled. Both of the outputs from the TIA module were sent as inputs to the multiplier that calculated the product and produced our end result.

In working with the TIA module to code this program, we first loaded the factors into local data registers. Using the Data Register Update portion of the TIA instruction, we then concurrently set another data register to 0 for our comparisons. When data came in on the data input channel, we first compared it to the data register containing 0, then changed the predicate in such a way to launch either an instruction to pass through the input data and the subzero factor or the input data and the above zero factor. The predicate registers would then be reset to the original comparison state for the next input data. The factor and our last input data would enter the multiplier and eventually output the result.

In the currently implemented example, handling each piece of data takes approximately 26 cycles. An example output can be seen in Figure 5-1.

```
Output: 0
Number of Cycles: 28
Output: 0
Number of Cycles: 44
Output: 0
Number of Cycles: 60
Output: 2
Number of Cycles: 76
Output: 4
Number of Cycles: 92
Output: 6
Number of Cycles: 108
```

Figure 5-1: An example output from running the nonlinearity program as currently implemented.

For a complete description of the TIA code used to implement nonlinearity on our functional model, see Appendix A.

5.2 Pooling

Pooling was the next step in implementation for our system. Because there are many important tasks that fall under the heading of pooling, we decided to implement two

different pooling functions to continue to demonstrate the flexibility of the model - average and maximum.

5.2.1 Maximum

The module configuration for the maximum function is quite simple; we use a TIA module with a single input and output channel. The input channel streams data with a tag set to zero until we are done with our data set. After the data is all through the module, we receive an input with tag set to 1, and we output the result.

The first instruction in our maximum program sets a local data register to the first input value. This will ensure that we have a valid output even upon a single data point. Next, we get into a loop that checks if the next input value is greater than the current maximum. If it is, we pass the input value through to the data register. If not, we will dequeue the input channel and look to the next value. Finally, when we see the input tag of 1, we pass through the value in our current maximum to the output channel.

In the current implementation, finding the maximum of a seven element list takes 64 cycles. Each additional element added to the list takes 6 cycles for comparisons and possible replacement.

For a complete description of the TIA code used to implement the maximum function on our functional model, see Appendix B.

5.2.2 Average

The averaging function was a bit more complicated to implement. Two different versions of the function have been implemented in order to explore some of the performance testing ability of the timing model. A full description of the analysis between the two implementations of the averaging function can be seen in Section 6.1.

The configuration for the two averaging functions are slightly different. For the first implementation, we use a single TIA module with a single input channel as a data stream, and a single output channel. For the second implementation, we use two

TIA modules that are connected by two channels. The first TIA module has a single input channel as the data stream, and the second TIA module has a single output channel for the final result. This function was broken into two modules in order to allow for pipelining the addition of the data stream and the division of the sum and count.

For both implementations, the first part of the code on the first TIA module is the same. The idea is that this TIA module will get all input data and add the data points while keeping track of the number of data points. To do this, we first set our data registers for both the running sum and the counter to zero. Then, for each input, if the tag is still zero, we will add the value to our running sum. This then changes the predicate to launch an increment operation that updates the counter. When we encounter a tag equal to one, we know that the data for the set is done, and we pass through both the running sum and the counter to the next TIA module.

The task of the second TIA module is to compute the quotient of the total sum divided by the counter. However, in the TIA module's ALU, we do not have a divider. We also have decided not to build a separate module such as the multiplier, because we do not believe that division will be sufficiently common with a need towards efficiency to justify dedicated hardware. In this light, we look at two different division implementations.

Subtracting Divider

The first method for division is simply to subtract the counter from the total sum until the difference drops below zero. We doing the subtraction operation, check if the result is still greater than zero, and then increment the running quotient until we drop below zero. The correct quotient rounded down to the nearest integer is then complete and output through the output channel. The code to complete this portion of the division also resides on the first and only TIA module for this implementation.

An example of how this divider works is shown here. Imagine your data points are 1, 2, 3, 4, and 5. First, we add the data points to get a running sum of 15 and a count of 5 data values. Now, we will subtract 5 from 15 and update our running sum

to 10. Because 10 is greater than or equal to 0, we make our running quotient 1. We will then increment our running quotient twice more when we update our running sum to 5 and 0 respectively. Finally, when we attempt to subtract the count one last time, we see that -5 is smaller than 0, and return our running quotient or 3 as our average.

Shifting Divider

The second method for division is an attempt to shorten the length of the division operation. The idea behind the optimization is to use shifts to subtract larger multiples of the count than we could with direct subtraction of the count. In order to do this method, we have to put a ceiling on the number of data points that could go into an average. We put this limit on the total count because we must know how many bits we can shift our count to the left before we overflow the integer. After deciding on this maximum number of bits, we shift the count left by the remaining bits and subtract this from the total sum. In our implemented example, we decided upon 16 bits as our maximum space taken by the count, so we begin by shifting our count to the left by 16. If the result is greater than zero, then we add one shifted to the left by the same amount to the running quotient. No matter the result, we then try to shift the count by one fewer bits and do the subtraction. When the number of bits we shift by drops below zero, the floor of the correct quotient will be present, and we output this to the output channel.

Using the same example from the subtracting divider, we will go into the divide step with a running sum of 15 and a count of 5. We will then attempt to subtract $2^{16} * 5$ from 15. Obviously, this will be quite negative, so we will reduce the exponent over 2, and try again. We first succeed in subtracting $2^1 * 5$ or 10 from 15. This results in us adding 2^1 to our running quotient to get a current result of 2. We also succeed in subtracting $2^0 * 5$ or 5 from the remainder of 5. This results in an addition of 1 to our running quotient to get a final result of 3. This shows the power of the shifting divider in that we can add powers of two at a time to the running quotient to arrive at a results much more swiftly.

Results

To see the performance gains of the shifting divider, we ran many tests on different averages to see how the two implementations compare. There is slightly more overhead with the shifting divider, so at very small averages, the subtracting divider takes fewer cycles. However, once the average is larger than 25, the shifting divider begins to take fewer cycles, as you can see in Figure 5-2. If the average gets very large, the shifting divider can have extreme benefits over the subtracting divider. In one example where the average was 100,000, the shifting divider took four orders of magnitude fewer cycles to get the correct answer.

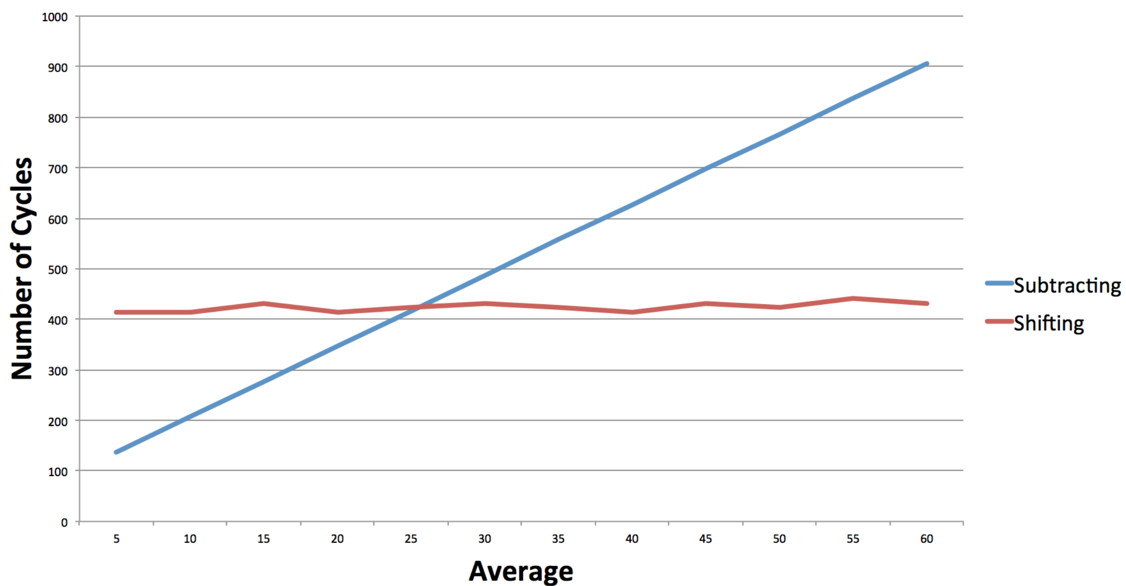


Figure 5-2: Above shows how the subtracting divider quickly surpasses the cycle count of the shifting divider. This graph only shows a very small portion of the domain, and the shifting divider will eventually grow linearly due to a maximum of 16 shifted bits, but will still grow significantly less quickly than the subtracting divider.

Just by looking at Figure 5-2, one can see that it can be beneficial to use the subtracting divider if you expect your average to be below 25. Any average above 25 would benefit from using the shifting divider.

One could also imagine an implementation where the count would first be shifted to the right until the result is zero in order to determine the number of bits in the count and correctly shift this to the left for the subtracting steps. This implementa-

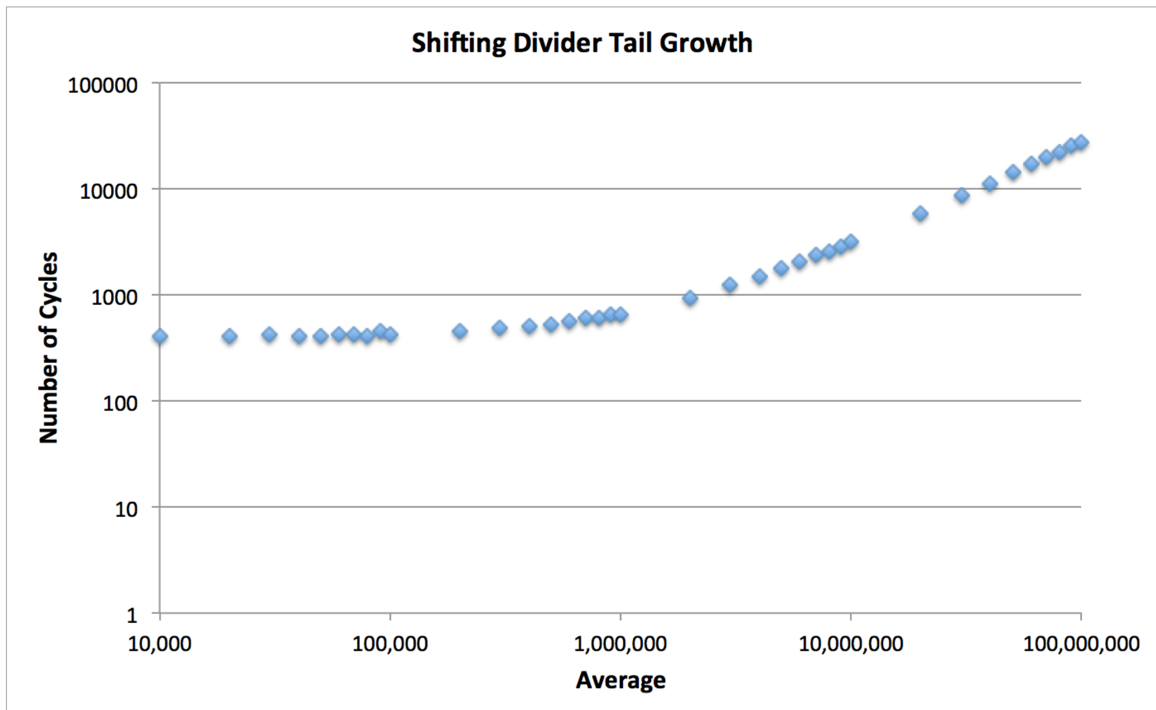


Figure 5-3: The above graph shows how the implemented shifting divider has quick growth in the cycle count as the average gets very large. Note that both axes use logarithmic scale.

tion would remove the ramp up in cycle counts that is seen as the average becomes larger than can fit in 16 bits. Due to time constraints, this example has not been implemented but is planned for the near future.

For a complete description of the TIA code used to implement the two different averaging functions on our functional model, see Appendix C.

5.3 Convolution

Convolution is the final program that has been implementing using our two models. The module configuration for convolution is a bit more complicated than any of the other programs and can be seen in Figure 1 of Appendix D, where the TIA code for this operation can also be found. For Convolution we need a TIA module with five inputs and five outputs, two FIFOs, and a multiplier module.

The use case that we explore the convolution example is applying a filter to an

image. We have equal size FIFOs that hold the image and filter data that is being used for the current convolution step. The multiplier is used to calculate the product of individual image and filter pixel data that is then sent back to the TIA module to sum and return as part of the output. Finally, we have input streams for both the image and filter data that come directly into the TIA module.

The first step in the convolution is to zero out all of the data registers that we will use. Next, we load the full filter into the filter FIFO by passing the data through the TIA module and outputting on the channel that loads the filter FIFO. Both FIFOs must be sized to be the same size as the filter so that the correct sliding window of the image is reached. We then load the image FIFO in a similar manner until it is full.

The next stage involves grabbing one piece of data from each the filter and image FIFOs and passing them along to the multiplier. The result of this multiply then enters the TIA module and is added into a running sum for this convolution step. When we have successfully run through the entire filter, we will return the running sum that is given from the multiplier module and continue onto another step. To correctly reset for the next step, we zero the running sum from our data register, discard one data point from the image FIFO, and load one new data point to the end of the image FIFO. We continue to run this algorithm until we no longer have data from the image.

Figure 5-4 shows some of the results that we observed from running this example. These tests were run on a one dimensional image of size 20. As you increase the size of the filter, the number of cycles increases because the number of operations also increase. If one were to continue to increase the size of the filter, we would see a decreasing number of cycles eventually. However, in reality, the images are much larger than the size of the filter, and this illustration would not be found in applications.

Furthermore, we can examine some of the gains that we see in doing a convolution on this hardware accelerator rather than in a stand alone processor. In our implemented example, we only access shared memory once for each value of the filter and

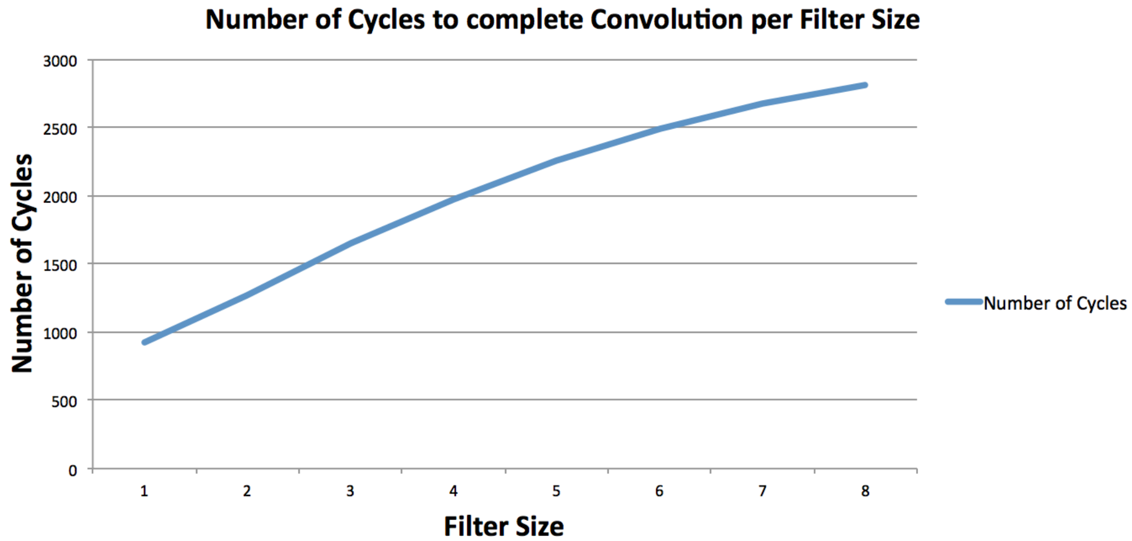


Figure 5-4: Above shows how the cycle count of the complete convolution operation increases as you increase the size of the filter due to increasing the total number of operations.

the image, so the total number of access is on the order of filter plus image size. In a stand alone processor, even assuming we could keep the entire filter in local memory, we would need to access the image data on the order of the image size times the filter size. In this example for a filter size of 5, this is the difference between total memory accesses of 25 for our TIA hardware accelerator and 80 for a stand alone processor.

Chapter 6

Future Work and Discussion

At the conclusion of this project, the functional and timing models for the described hardware accelerator have been fully implemented and delivered. We take this section to take a step back, look at the results of the work, discuss the main challenges faced in the project, consider the future work for this project, and analyze the impact that this project delivers.

6.1 Results

The main results of this thesis are in the functionality and future impact that this framework makes possible. The possibility of testing configurations and obtaining valuable metrics has been demonstrated through the implemented examples and the depth of the module design.

The implemented examples show that the functional and timing models paired together are powerful to enact useful and varied programs. Due to the flexibility present in the modules, the network connecting the modules, and the instructions that control these modules, we are able to create interesting and compelling examples that can be used as models for future programs.

6.2 Main Challenges

The main challenges that were faced throughout this project mainly stemmed from creating a software representation and design of hardware modules that would realistically map to and be informative for the end hardware design. Some of the specific challenges that were faced in this vein are listed below:

1. Designing a General Module Structure - One main goal in our modular design was to have as much in common between all the modules as possible. We had to carefully design a general module that allowed us enough flexibility for all possible modules while providing a powerful structure to leverage generality in the timing model.

2. Simplifying the Timing Model - Another key goal for this project was to design a simple, understandable timing model that could treat all implementations of the programmed array as a black box. In order to do this, we had to push some configuration steps up to the setup stage for a given architecture as well as carefully design the behavior of the modules to allow for simple execution without compromising flexibility.

3. Correctly Managing Hardware Resources - A key component of this design was correctness in allowing triggered instructions to run. This involved making sure that we kept track of all components of the architecture that were in use through scoreboarding. Deciding which components to scoreboard and correctly updating these scoreboards were some of the main challenges in this project.

6.3 Future Work

There are two types of future work that are slated for this project. The first portion of future work is to extend the timing model and the second is to utilize the functional and timing model pair to test configurations of specific hardware architectures.

There are a few areas that we would like to improve and extend the timing model. Firstly, connecting modules together is currently handled by manually adding a channel as an input or output to each module. This is practice for the size of examples that we have worked with to this point but will become impractical or tedious when implementing examples using on the order of hundreds of modules. Due to this constraint, we are looking to add a configuration engine that will take some sort of netlist and expand the representation to a given configuration.

Furthermore, for a given algorithm, the designer must carefully convert the higher-level code into valid TIA instructions. This has proven to take a fair amount of time and can be difficult to debug. Therefore, we believe that a future project to create a compiler for the triggered instruction architecture will assist designers greatly in speed and correctness of tests.

Finally, this project, from the outset, has been intended as a tool to test out different configurations of the hardware accelerator that is one step removed from theory and a few steps away from laying out a circuit. By using these models, we hope to identify key problems that would be much more costly to find in a later round of design. Furthermore, by working closely with the timing model and changing configurations often, we can find optimal architectures for many problem sets in order to truly tailor an accelerator to solve a given problem.

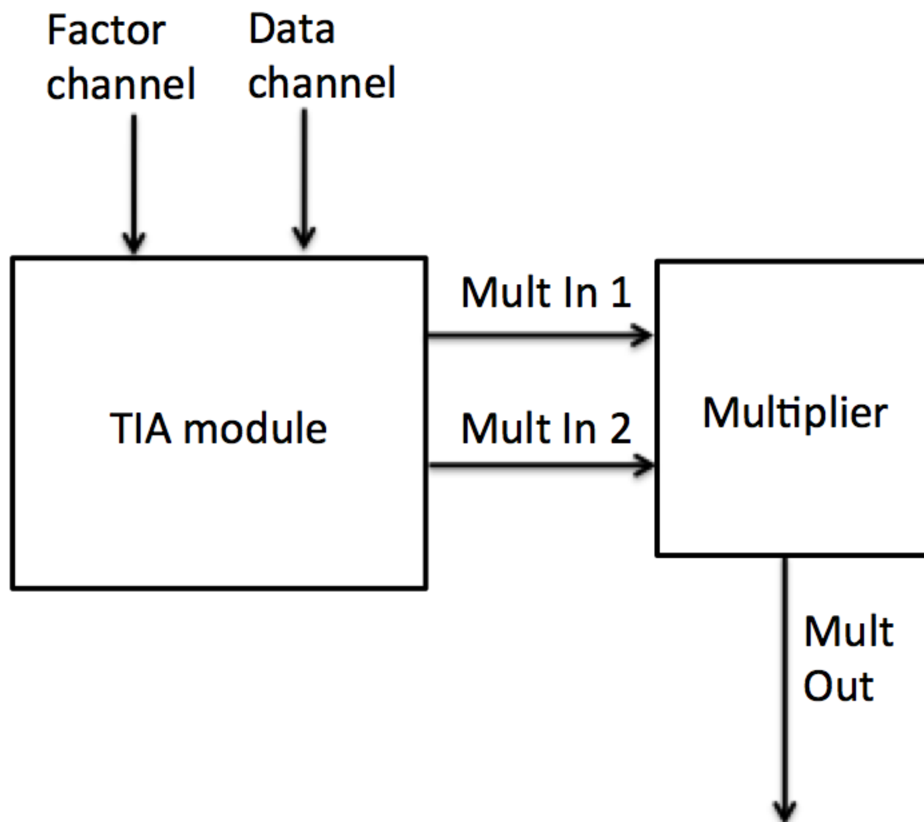
6.4 Impact

The creation of the functional and timing models for this type of programmable hardware accelerator is both novel and useful. The design decisions that accompany developing these ultra-configurable functional and timing models have made this project both highly educational and extremely enjoyable. The deliverables from this project will assist in testing and learning much about both this accelerator and future proposed accelerators in this space.

Appendix A

Nonlinearity Implementation

In order to take a deeper look into the implementation of the nonlinearity function, let's break down each of the TIA instructions that went into creating the program.



This graphic shows the architecture used for this example. There is a single TIA

module and a single Multiplier module. The TIA module uses 2 data registers and 3 predicate registers.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#1	**Predicate State:**	FALSE	FALSE	FALSE
	Predicate Update:	TRUE		
	Input Source:	Factor Channel		
	Output Location:	Data Reg 0		
	Dequeue List:	Factor Channel		

The first instruction for our nonlinearity program takes the factor that will multiply all data inputted beneath zero and puts the factor into data register 0. It then updates the predicate registers correctly to launch the next instruction.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#2	**Predicate State:**	TRUE	FALSE	FALSE
	Predicate Update:		TRUE	
	Input Source:	Factor Channel		
	Output Location:	Data Reg 1		
	Dequeue List:	Factor Channel		

The second instruction takes the factor that will multiply all data inputted above zero and puts the factor into data register 1. The predicate update then readies the system for a data stream.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#3	**Predicate State:**	TRUE	TRUE	FALSE
	Predicate Update:	TRUE on 0		TRUE on 0
	Input Source:	Data Channel		
	Output Location:	None		
	Dequeue List:	None		

The third instruction takes the next available value in the data stream and checks if this value is less than zero. If it is, it will direct the TIA module to the factor for negative inputs. If it isn't, it will direct the module to the factor for positive inputs.

		Pred Reg 0	**Pred Reg 1**	**Pred Reg 2**
#4	**Predicate State:**	TRUE	TRUE	TRUE
	Predicate Update:			FALSE
	Input Source:	Data Channel		
	Output Location:	Mult In 1	Mult In 2	
	Dequeue List:	Data Channel		

The fourth instruction passes along a data value that is greater than zero along with the correct factor to the multiplier module. The output of this module will be the correctly scaled result.

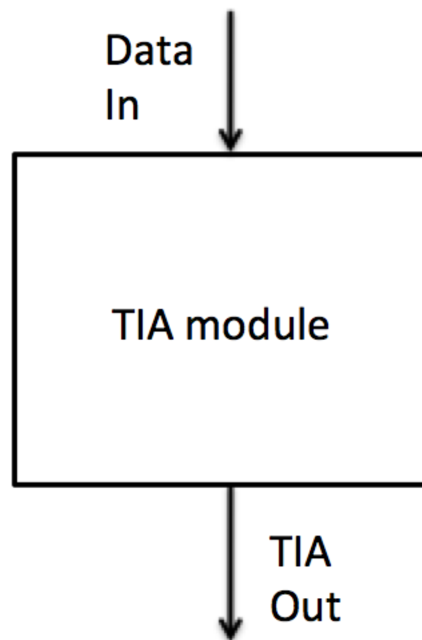
		Pred Reg 0	**Pred Reg 1**	**Pred Reg 2**
#5	**Predicate State:**	FALSE	TRUE	FALSE
	Predicate Update:	TRUE		
	Input Source:	Data Channel		
	Output Location:	Mult In 1	Mult In 2	
	Dequeue List:	Data Channel		

The fifth instruction is the same as the fourth but used for negative numbers.

Appendix B

Maximum Implementation

In order to take a deeper look into the implementation of the maximum function, we'll first take a look at the configuration, and then at the TIA instructions used to run the code.



This graphic shows the architecture used for this example. This configuration is extremely simple and has only a single TIA module with one input and one output

channel. The TIA module uses 1 data register and 3 predicate registers.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#1	Predicate State:	FALSE	FALSE	FALSE
	Predicate Update:	TRUE		
	Input Source:	Data In		
	Output Location:	Data Reg 0		
	Dequeue List:	Data In		
	ALU Operation:	Pass_Through		

The first instruction for the maximum function simply pushes the first data value into the data register that will hold the current maximum. This allows us to use this as our first max value and work from here to get a running maximum.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#2	Predicate State:	TRUE	FALSE	FALSE
	Predicate Update:		FALSE on 0	TRUE
	Input Source:	Data In	Data Reg 0	
	Output Location:	N/A		
	Dequeue List:	N/A		
	ALU Operation:	Greater Than		

The second instruction takes the next available data point and checks if it is greater than our current maximum value. If it is, then we will replace the current max with this data point. If not, we will simply dequeue this data point from our input stream. This instruction is only launched if there is no flag set on the incoming data saying that the computation should end.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#3	**Predicate State:**	TRUE	FALSE	TRUE
	Predicate Update:			FALSE
	Input Source:	Data In		
	Output Location:	Data Reg 0		
	Dequeue List:	Data In		
	ALU Operation:	Pass_Through		

The third instruction replaces the current maximum value with the next value in the data stream. This instruction is launched if the most next data value was deemed to be greater than the current max.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#4	**Predicate State:**	TRUE	TRUE	TRUE
	Predicate Update:		FALSE	FALSE
	Input Source:	Data In		
	Output Location:	N/A		
	Dequeue List:	Data In		
	ALU Operation:	DEQ		

The fourth instruction tosses the data value in the data stream because it was not larger than the current maximum value.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#4	**Predicate State:**	TRUE	FALSE	FALSE
	Predicate Update:	FALSE		
	Input Source:	Data Reg 0		
	Output Location:	TIA Out		
	Dequeue List:	Data In		
	ALU Operation:	Pass_through		

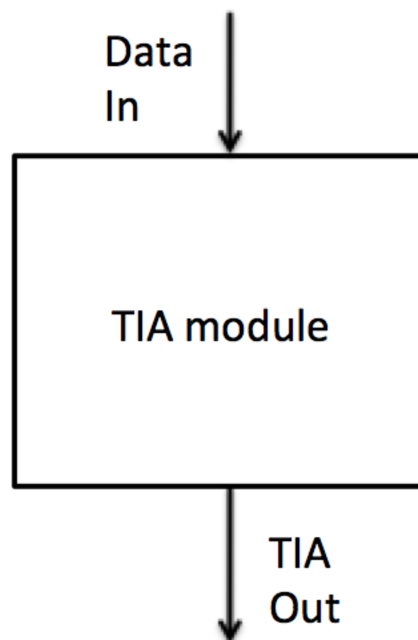
The fifth and final instruction is launched on the same predicate as the greater than instruction but when the flag is set on the input channel. This signifies that all points in the data set have been processed. Therefore, the current maximum value is the correct maximum and is passed along to the output channel.

Appendix C

Average Implementation

Let's take a closer look at the configuration and implementation of the two averaging functions. We'll begin with the subtracting divider and then take a look at the shifting divider.

C.1 Subtracting Divider Implementation



The average function that is implemented with the subtracting divider is implemented on with a single TIA module using 3 predicate registers and 4 data registers. Let's take a look at the instructions.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	
#1	**Predicate State:**	FALSE	FALSE	FALSE	
	Predicate Update:	TRUE			
	Input Source:	N/A			
	Output Location:	N/A			
	Dequeue List:	N/A			
	ALU Operation:	DEQ			
		Data Reg 0	**Data Reg 1**	**Data Reg 2**	**Data Reg 3**
	Direct Reg Update:	0	0	0	0

The first instruction simply loads zero into all of data registers that we will be using for our averaging function.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#2	**Predicate State:**	TRUE	FALSE	FALSE
	Predicate Update:		TRUE	
	Input Source:	Data Channel	Data Reg 0	
	Output Location:	Data Reg 0		
	Dequeue List:	Data Channel		
	ALU Operation:	ADD		

The second instruction starts a running addition that adds the newest piece of data from the data channel into the running sum stored in data register 0.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#3	**Predicate State:**	TRUE	TRUE	FALSE
	Predicate Update:		FALSE	
	Input Source:	Data Reg 1		
	Output Location:	Data Reg 1		
	Dequeue List:	N/A		
	ALU Operation:	PlusPlus		

The third instruction keeps the counter in line with the running sum by incrementing the counter on each new data point after it is added to the running sum.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#4	**Predicate State:**	TRUE	FALSE	FALSE
	Predicate Update:	FALSE	TRUE	
	Input Source:	Data Channel	Data Reg 0	
	Output Location:	Data Reg 0		
	Dequeue List:	Data Channel		
	ALU Operation:	ADD		

The fourth instruction has the same purpose as the second instruction, but is fired when the last data value is entered. This can be known by looking for a set input flag that comes with the last data value. This, along with the next instruction, will transfer control to the divide step.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#5	Predicate State:	FALSE	TRUE	FALSE
	Predicate Update:		FALSE	TRUE
	Input Source:	Data Reg 1		
	Output Location:	Data Reg 1		
	Dequeue List:	N/A		
	ALU Operation:	PlusPlus		

The fifth instruction does the final increment for the counter and passes control to the division step.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#6	Predicate State:	FALSE	FALSE	TRUE
	Predicate Update:	TRUE		
	Input Source:	Data Reg 0	Data Reg 1	
	Output Location:	Data Reg 0		
	Dequeue List:	N/A		
	ALU Operation:	SUBTRACT		

The sixth instruction subtracts the counter from the running sum to see if we can increment the running quotient. This subtraction occurs until the running sum drops below zero, at which point, we know that we can no longer divide the sum by the counter.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#7	**Predicate State:**	TRUE	FALSE	TRUE
	Predicate Update:	TRUE on 0	TRUE	
	Input Source:	Data Reg 0	Data Reg 3	
	Output Location:	N/A		
	Dequeue List:	N/A		
	ALU Operation:	Greater Than	Or Equal	

The seventh instruction does this check to ensure that the running sum has not dropped below zero. If it hasn't, it passes control to the eighth instruction to increment the running quotient. If we are below zero, it passes control to the ninth function to return the solution.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#8	**Predicate State:**	FALSE	TRUE	TRUE
	Predicate Update:	FALSE	FALSE	
	Input Source:	Data Reg 2		
	Output Location:	Data Reg 2		
	Dequeue List:	N/A		
	ALU Operation:	Plus Plus		

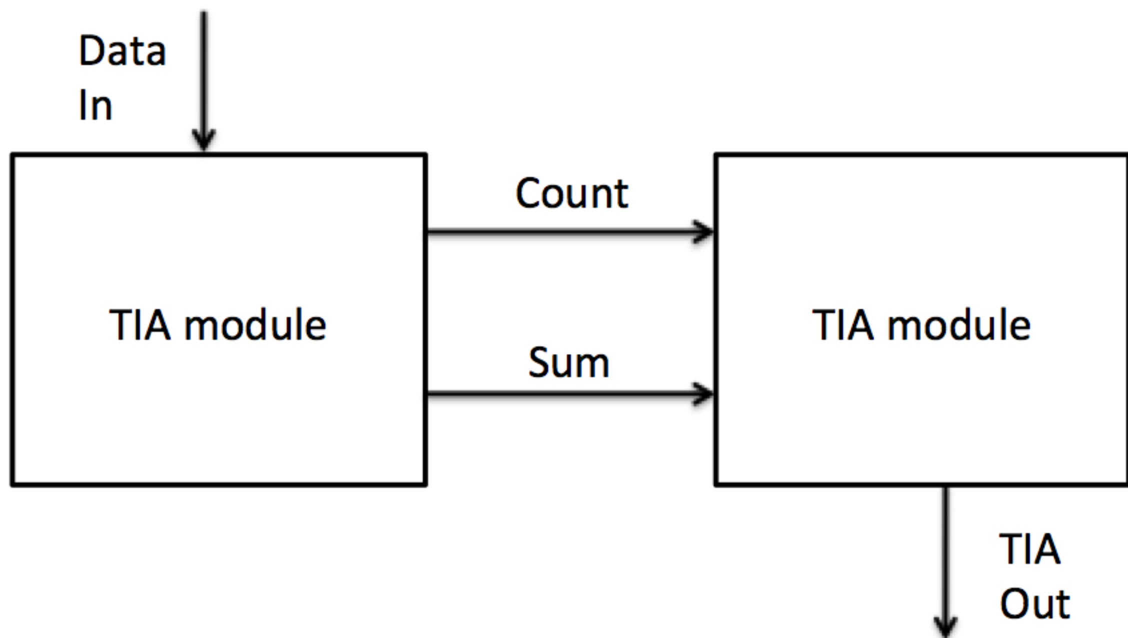
As stated above, the eighth instruction increments the running quotient and sends control back to the sixth instruction to attempt another subtraction step.

		Pred Reg 0	Pred Reg 1	Pred Reg 2
#9	**Predicate State:**	TRUE	TRUE	TRUE
	Predicate Update:	FALSE	FALSE	FALSE
	Input Source:	Data Reg 2		
	Output Location:	TIA Out		
	Dequeue List:	N/A		
	ALU Operation:	Pass Through		

Finally, the ninth instruction returns the running quotient as the floor of the correct quotient.

C.2 Shifting Divider Implementation

Now, we will take a look at the shifting divider implementation. Below is the configuration associated with this implementation.



This configuration is very similar to the original but it now split into two TIA modules. The first module computes the count and sum and then sends the two values to the second TIA module. The program on the first TIA in the shifting divider implementation is the same as the first five instructions of the subtracting divider implementation, along with a single instruction that passes the count and sum to the second TIA module. For the sake of brevity, we will not repeat these instructions and move straight to the instruction set on the second TIA module.

		Pred Reg 0	**Pred Reg 1**	**Pred Reg 2**	**Pred Reg 3**	**Pred Reg 4**
#1	**Predicate State:**	FALSE	FALSE	FALSE	FALSE	N/A
	Predicate Update:	TRUE				
	Input Source:	Sum	Count			
	Output Location:	Data Reg 0	Data Reg 1			
	Dequeue List:	Sum	Count			
	ALU Operation:	Pass Through 2				
		Data Reg 2	**Data Reg 3**	**Data Reg 4**	**Data Reg 5**	**Data Reg 6**
	Direct Reg Update:	1 << 16	0	16	0	1

The first instruction loads the sum into data register 0, the counter into data register 1, 1 shifted to the left by 16 into data register 2, 0 into data register 3, 16 into data register 4, 0 into data register 5, and 1 into data register 6. Data register 4 is used as the multiplicative factor that we will add to our running quotient. Data register 5 acts as this running quotient. Data register 4 is used for multiple purposes, but begins as a way to shift the count to the left by 16 bits. Data register 5 allows us to reference 0, and data register 6 allows us to reference 1 to do shifts to the right.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#2	Predicate State:	TRUE	FALSE	FALSE	FALSE	N/A
	Predicate Update:		TRUE			
	Input Source:	Data Reg 1	Data Reg 6			
	Output Location:	Data Reg 1				
	Dequeue List:	N/A				
	ALU Operation:	Shift Left				

The second instruction shifts the count over by 16 bits. Because we constrict the count to a total of 16 bits, we know we won't be losing any of these with this shift. This gives us a much larger factor to try to subtract from the running sum so that we don't have to calculate the running quotient one factor at a time.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#3	Predicate State:	TRUE	TRUE	FALSE	FALSE	N/A
	Predicate Update:		FALSE	TRUE		
	Input Source:	Data Reg 0	Data Reg 1			
	Output Location:	Data Reg 4				
	Dequeue List:	N/A				
	ALU Operation:	Subtract				

The third instruction tries to subtract this larger factor of the counter from the running sum and stores it in a temporary variable.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#4	**Predicate State:**	TRUE	FALSE	TRUE	FALSE	N/A
	Predicate Update:		TRUE		TRUE on 0	
	Input Source:	Data Reg 4	Data Reg 5			
	Output Location:	N/A				
	Dequeue List:	N/A				
	ALU Operation:	Greater Than	Or Equal			

The fourth instruction checks to see if this result is larger than zero, which would mean we should try to subtract this factor again. If it is smaller than zero, we will want to shift the counter to the right by 1 bit and try a subtraction operation again.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#5	**Predicate State:**	TRUE	TRUE	TRUE	FALSE	FALSE
	Predicate Update:					TRUE
	Input Source:	Data Reg 2	Data Reg 3			
	Output Location:	Data Reg 3				
	Dequeue List:	N/A				
	ALU Operation:	ADD				

The fifth instruction adds the multiplicative factor to the running quotient if the subtraction step did not drop the running sum below zero.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#6	**Predicate State:**	TRUE	TRUE	TRUE	FALSE	TRUE
	Predicate Update:			FALSE		FALSE
	Input Source:	Data Reg 4				
	Output Location:	Data Reg 0				
	Dequeue List:	N/A				
	ALU Operation:	Pass Through				

The sixth instruction moves the temporary difference into the running sum because the subtraction operation passed our test and was valid.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#7	**Predicate State:**	TRUE	TRUE	TRUE	TRUE	N/A
	Predicate Update:	FALSE		FALSE	FALSE	
	Input Source:	Data Reg 1	Data Reg 6			
	Output Location:	Data Reg 1				
	Dequeue List:	N/A				
	ALU Operation:	Shift Right				

The seventh instruction is the beginning of the rest of the shifting operation. We shift the count to the right by 1 in order to subtract a smaller factor from the running sum.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#8	**Predicate State:**	FALSE	TRUE	FALSE	FALSE	N/A
	Predicate Update:			TRUE	TRUE on 0	
	Input Source:	Data Reg 2	Data Reg 6			
	Output Location:	Data Reg 2				
	Dequeue List:	N/A				
	ALU Operation:	Shift Right				

The eighth instruction shifts the multiplicative factor to the right by 1 as well and ensures that it is still larger than 0. If it is not, then we have a valid quotient which we can output. If it is still larger than 0, then we will continue to try and subtract smaller factors of the counter.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#9	**Predicate State:**	FALSE	TRUE	TRUE	FALSE	N/A
	Predicate Update:			FALSE	TRUE	
	Input Source:	Data Reg 0	Data Reg 1			
	Output Location:	Data Reg 4				
	Dequeue List:	N/A				
	ALU Operation:	Subtract				

The ninth instruction does the same subtraction step as the third instruction, but in our new control loop.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#10	Predicate State:	FALSE	TRUE	FALSE	TRUE	N/A
	Predicate Update:	TRUE		TRUE on 0		
	Input Source:	Data Reg 4	Data Reg 5			
	Output Location:	N/A				
	Dequeue List:	N/A				
	ALU Operation:	Greater Than	or Equal			

The tenth instruction does the same check to see if the subtraction produced a subzero output as was checked in the fourth instruction. Again, this is duplicated because we are in a different control loop.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#11	Predicate State:	TRUE	TRUE	FALSE	TRUE	FALSE
	Predicate Update:					TRUE
	Input Source:	DataReg 2	Data Reg 3			
	Output Location:	Data Reg 3				
	Dequeue List:	N/A				
	ALU Operation:	Add				

The eleventh instruction is a mirror of the fifth instruction in adding the multiplicative factor to our running quotient.

		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#12	**Predicate State:**	TRUE	TRUE	FALSE	TRUE	TRUE
	Predicate Update:			TRUE		FALSE
	Input Source:	Data Reg 4				
	Output Location:	Data Reg 0				
	Dequeue List:	N/A				
	ALU Operation:	Pass Through				

The twelfth instruction is similar to the sixth instruction as it pushes the temporary sum into the running sum. However, the control is immediately pushed to the seventh instruction to shift down as it is impossible to have multiple iterations of this factor succeed at this step.

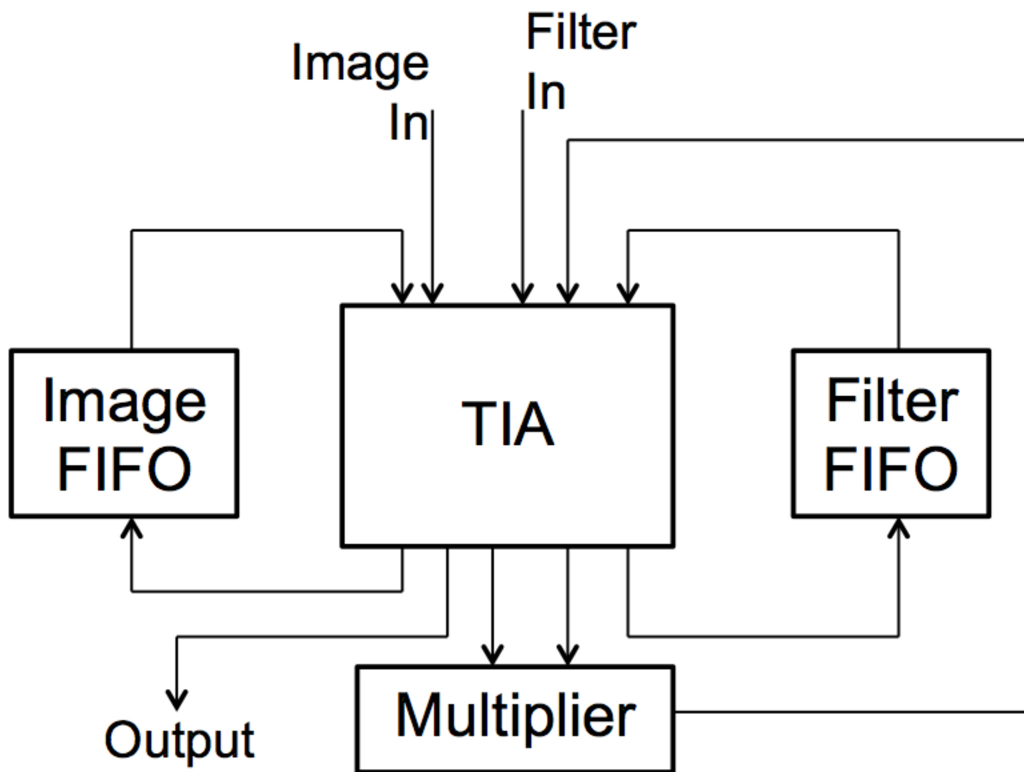
		Pred Reg 0	Pred Reg 1	Pred Reg 2	Pred Reg 3	Pred Reg 4
#13	**Predicate State:**	FALSE	TRUE	TRUE	TRUE	N/A
	Predicate Update:		FALSE	FALSE	FALSE	
	Input Source:	Data Reg 3				
	Output Location:	TIA Out				
	Dequeue List:	N/A				
	ALU Operation:	Pass Through				

Our final instruction is reached when we have shifted the counter all the way back to its original position. At this point we have a valid quotient and output it on the output channel.

Appendix D

Convolution Implementation

The final implemented example is convolution. First, let's take a look at the configuration that is slightly more complex that the other implemented examples' configurations.



This configuration has a single TIA module with two predicate registers and one data register, a FIFO that recirculates the image and a FIFO that recirculates the Filter, and a multiplier module that is looped back into the TIA module. The instruction set below should elucidate why this structure fits the form for convolution. This implementation uses a slightly different mode of ordering instructions. Because instructions that come first have higher priority, we can use instruction ordering as a proxy for cutting down on the number of predicate registers needed.

		Pred Reg 0	**Pred Reg 1**
#1	**Predicate State:**	N/A	N/A
	Predicate Update:		
	Input Source:	Mult Out	Data Reg 0
	Output Location:	Data Reg 0	
	Dequeue List:	Mult Out	
	ALU Operation:	ADD	

The first instruction here handles when a multiplication for a convolution step has been completed that is not the last multiplication in the convolution step. The results is added back to the running sum.

		Pred Reg 0	**Pred Reg 1**
#2	**Predicate State:**	N/A	N/A
	Predicate Update:		
	Input Source:	Mult Out	Data Reg 0
	Output Location:	Out Channel	
	Dequeue List:	Mult Out	
	ALU Operation:	ADD	
		Data Reg 0	
	Direct Reg Update:	0	

The second instruction handles the same case as above, except when the multiplication is the last step of the convolution step. The product is added to the running sum, the running sum is output on the output channel, and the running sum is overwritten with zero.

		Pred Reg 0	Pred Reg 1
#3	**Predicate State:**	FALSE	TRUE
	Predicate Update:		FALSE
	Input Source:	Image In	
	Output Location:	Image FIFO	
	Dequeue List:	Image In	
	ALU Operation:	Pass Through	

The third instruction is intended to load the image data into the image FIFO. This takes data from the Image In channel and forwards it to the FIFO.

		Pred Reg 0	Pred Reg 1
#4	**Predicate State:**	N/A	N/A
	Predicate Update:		TRUE
	Input Source:	Filter In	
	Output Location:	Filter FIFO	
	Dequeue List:	Filter In	
	ALU Operation:	Pass Through	

The fourth instruction is similar to the third instruction, but loads the filter FIFO.

		Pred Reg 0	Pred Reg 1
#5	**Predicate State:**	FALSE	FALSE
	Predicate Update:		
	Input Source:	Image FIFO	Filter FIFO
	Output Location:	Mult In 1	Mult In 2
	Dequeue List:	Image FIFO	Filter FIFO
	ALU Operation:	Pass Through 2	

The fifth instruction is intended to pass the two arguments - one data point from the filter and one from the image - to the multiplier. This instruction is launched when we are not on the last multiplication of a convolution step.

		Pred Reg 0	Pred Reg 1
#6	**Predicate State:**	FALSE	FALSE
	Predicate Update:	TRUE	TRUE
	Input Source:	Image FIFO	Filter FIFO
	Output Location:	Mult In 1	Mult In 2
	Dequeue List:	Image FIFO	Filter FIFO
	ALU Operation:	Pass Through 2	

The sixth instruction is very similar to the fifth instruction but is launched on the last multiplication of the convolution step. This instruction then launches the seventh instruction.

		Pred Reg 0	**Pred Reg 1**
#7	**Predicate State:**	TRUE	TRUE
	Predicate Update:	FALSE	TRUE
	Input Source:	Image FIFO	
	Output Location:	N/A	
	Dequeue List:	Image FIFO	
	ALU Operation:	DEQ	

The seventh and last instruction dequeues one of the data points from the image FIFO so that we can replace it with the next data point using the third instruction.

Bibliography

[1] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, C. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, J. Emer. Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures. In Proceedings of the 40th International Symposium on Computer Architecture (ISCA), pages 142-153, Jun. 2013.

[2] J. Tithi, N. Crago, J. Emer. Exploiting Spatial Architectures for Edit Distance Algorithms. In Performance Analysis of Systems and Software (ISPASS), pages 23-34, Mar. 2014.

[3] A. Krizhevsky, I. Sutskever and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks, Proc. Neural Information and Processing Systems, 2012.

[4] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. Computer, 37(7):44âĂŞ55, July 2004.

[5] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In Proceedings of 17th International Conference on High Performance Computer Architecture (HPCA), 2011.

[6] J. Hoogerbrugge and H. Corporaal. Transport-Triggering vs. Operation-Triggering. In Lecture Notes in Computer Science 786, Compiler Construction, pages 435âĂŞ449. Springer-Verlag, 1994.

[7] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas. A 167-Processor Computational Platform in 65 nm CMOS. IEEE Journal of Solid-State Circuits, 44(4):1130âĂŞ1144, April 2009.

[8] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas. An Asynchronous Array of Simple Processors for DSP Applications. In Solid-State Circuits Conference (ISSCC), Digest of Technical Papers, pages 1696âĂŞ1705, Feb. 2006.