— NGAO —

RTC

Hardware and Algorithms

Table of Contents

List of Figures

List of Tables

1 Introduction and Organization

This work documents the design of a real time computer system capable of solving large atmospheric tomography problems for adaptive optics in under 1 msec.

Axey (Ax=y) is a scalable, programmable and configurable array of processors operating as a systolic array.

It is scalable so that a system can be easily put together (by replication of rows and columns) that will work for any sized problem, from the smallest to at least TMT sized [1].

It is programmable, so that changes in the algorithm can be loaded at run time without changing the hardware. Additionally, even the hardware is programmable in its architecture, interconnect and logic.

It is configurable so that the parameters can be easily changed during run time to adapt to changing needs for a given system: number of sub apertures, guide stars or atmospheric layers; values for $\mathbf{C_n^2}$; mixes, heights and positions of natural and laser guide stars; height and profile of the sodium layer; height of turbulent layers; etc. Indeed, the actual architecture of the cells and the array can also be easily changed.

The algorithm implemented here to solve Ax=y for x is a preconditioned iterative method and is discussed in Section 4.

An alternative algorithm for solving for x, using Cholesky decomposition has also been proposed [2]. Some initial examination shows that the architecture documented here might also be suitable for implementing that algorithm. This will be the subject of future work and is not documented here.

Section 2, System Requirements, covers the specific problem, for which we undertook the research and developed Axey. We detail our measures to determine the relative optimality of our solution.

Section 3, The Tomography Algorithm, covers the underlying algorithms for which we needed to provide a solution and the architectural implications of the algorithms along with some of the issues to be traded off.

Section 4, The Architecture,

Section 5, Implementation,

Section 6, Summary, summarizes the work to date and that following.

This document is a snapshot of work in progress. It is complete in so far as the work required for this section of the NGAO effort. It is probably overly complete in some sections regarding detailed implementation but this was left in since it is work required by the next stages of the development effort.

Some sections and diagrams of this document (indicated in red) are incomplete or missing. However, they are portions to be filled in during subsequent work.

Some table and figure references in Section 5, are incorrectly numbered and are indicated in red. In the interest of expediency, I have left those in and will correct them soon. The tables and figures are well labeled and it should be clear what the references really point to.

Much of the work in Section 5, relating to the detailed implementation on the FPGA, comes from Matt Fischler's master's thesis [3].

2 System Requirements

The AO system, of which Axey is a part:

1. Wave Front Sensors (WFSs)

   Takes digital data from the cameras and create wavefronts. These are the raw Radon projections, of the optical path delay (OPD) of the light rays, from a guide star, through the atmosphere.

2. Tomography (Axey)

   Produce a 3D tomographic estimate of the atmosphere from the wave fronts. Data from this estimate is used to control the Deformable Mirrors (DMs).

3. DM processing

   Generate actual control information for the DMs (which will correct the atmospheric distortion) from the estimated atmospheric data.

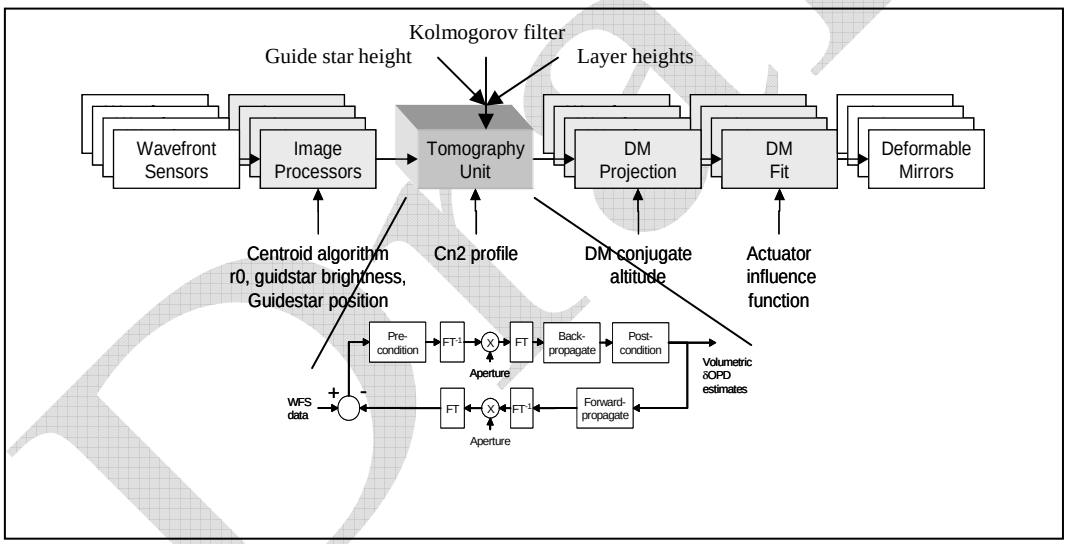

Figure 2-1 Where Does Tomography Fit in the Adaptive Optics Problem?

Show parallel and independent paths of WFSs and DMs

For each cycle, we have:

- Data I/O
- Computation

2.1 Summary of the Iterative Solution

Section needs a better name

Doesn't belong here

Thus, we now have a sequence of parallelizable steps, each with different characteristics:

- A parallelizable I/O portion, one at the beginning and one at the end of each frame
- An embarrassingly parallelizable basic tomography algorithm
- A very simple aperturing algorithm, which must unfortunately be done in the spatial domain
- A parallelizable (but with heavy busing requirements) Fourier transform required by the need to move back and forth between the spatial and Fourier domains, so that we can apply the aperture

The last three items are applied in sequence, for each iteration required to complete a frame (see _____).

Our task is to arrive at an architecture that balances the requirements and effects of each of the elements and produces an optimum result. The optimal architecture for one element will not be the optimal architecture for all the elements; nor will the optimal overall architecture likely be the optimum for most (if any) elements.

Show a small version of the overall system that this expands from

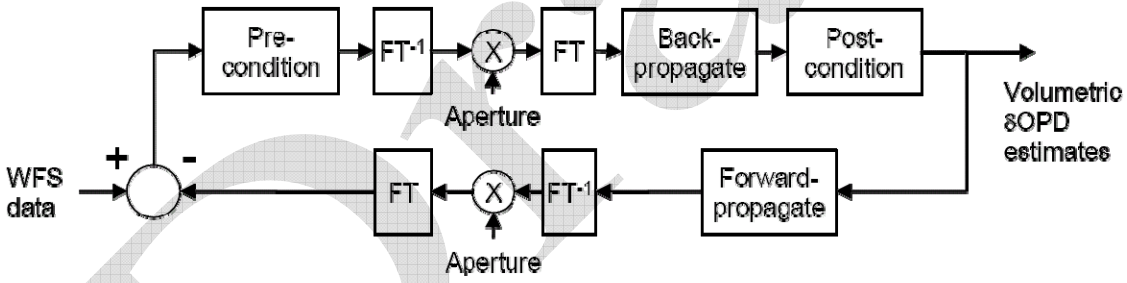

Figure 2-2 The Tomography Algorithm

To determine the optimal architecture we must understand the overall impact each element has on our overall result.

2.2 Physical Requirements

Most of the equipment needs to be mounted on or in relatively close proximity to the instrument or telescope. It must therefore take into account its effect on the instrument. The requirements of the instrument for which these operations are performed are somewhat limiting.

2.2.1 Heat and Power

The heat generated and the size and weight of the instrument must be minimized [4].

2.2.2 Size

2.2.3 Weight

2.3 Computational Requirements

2.3.1 Accuracy

2.3.2 Speed

2.3.3 Latency

2.4 I/O Rates

The data rates requirements for the system are substantial.

From WFS and to DMs

Additionally, the engine must be able to stream diagnostic data to the storage facility at ___ GB/sec without affecting its primary continuous functions of analyzing and compensating for the atmosphere.

2.4.1 Tomography

Each wave front produced would have 128 x 128 elements. Each element will be 2 bytes. A new wavefront will be received every millisecond. That is 32MB per second per WFS.

Data coming into Axey from the WFS and leaving Axey to be processed for the DMs are both substantial and almost identical at 32 MB per second

For an 8-layer MCAO system, that is 512MB per second of I/O, not counting any access required while processing the data or logging the data for analysis.

For an MOAO system, with 24 heads on the instrument, the rate would be over 1.5 GB per second

2.5 Storage Requirements and Rates

2.5.1 Configuration Storage

Since all chips are essentially identical, the storage requirements for configuration data are minimal and largely independent of the scale of Axey. The requirements amount to approximately _____ MB.

2.5.2 Operational Storage

The storage requirements for Axey itself are quite modest, amounting to only about a dozen complex registers per processor cell or perhaps a few Mega bytes for a large system.

2.5.3 Telemetry

The system requirements for storing and analyzing data observed during measurements can be many hundreds of Giga Bytes for a single night.

Complete wave front, layer and DM data sets are very large and the more frequently we log them the higher the data rate the system must support for these, exclusive of the normal operational data rates, which are already extensive.

2.5.4 Diagnostic Streams

These consist of ____

Axey must be able to stream data to the storage facility at ___ GB/sec without affecting its primary continuous functions of analyzing and compensating for the atmosphere.

2.6 Reliability

Cost of system non availability. [5]

- Basic reliability

MTBF

- Self diagnostic capability for potential failures (keeping ahead of the game)

2.7 Maintainability

Rapid diagnostic ability to identify failure location.

2.8 Ease of Use and End User Productivity

It may be a great tool, but if it is difficult to use, the productivity of the end users will suffer and the tool may not be used in many cases where it could be beneficial. One crucial factor determining the success or failure for use as a general tool will be the programming model. While massively parallel processors offer the possibility of dramatic performance gains over traditional architectures, these gains will only be realized if the programming model is user friendly and efficient.

2.9 Ease of Development

2.10 Facilities Supported

2.10.1 DM and Tip/Tilt Commands

2.10.1.1 Set any actuator on any DM or any Tip/Tilt mirror to an arbitrary voltage

2.10.1.2 Set and read the bias of any DM to an arbitrary voltage or pattern

2.10.1.3 Draw pre determined alignment patterns on any DM

2.10.1.4 Cycle (at 1 Hz) one or more actuators on any DM or Tip/Tilt mirror

2.10.1.5 Cycle all actuators on any DM in a round robin fashion

2.10.1.6 Read the current commands for any DM or Tip/Tilt mirror

2.10.1.7 Set and read the “Zero” x/y position for any tip/tilt mirror

2.10.1.8 Load a control matrix for any DM

2.10.1.9 Set and read the Open/Closed loop state for any DM

2.10.1.10 Load a flat pattern for any DM

2.10.1.11 Create a time averaged file of the DM commands for flattening and sharpening

2.10.2 Calibration Commands

2.10.2.1 Create a system matrix for any DM

2.10.2.2 Set the number of frames to use when creating the system matrix

2.10.2.3 Clear the DM of any manually set values

2.10.3 Camera Control

2.10.3.1 Set and read the camera frame rate

2.10.3.2 Set and read the camera pixel rate

2.10.3.3 Set and read the camera gain

2.10.4 WFS

2.10.4.1 Load the dark pattern

2.10.4.2 Load pixel gain pattern

2.10.4.3 Set and read the threshold level

2.10.4.4 Set and read the centroiding algorithm

2.10.4.5 Load and read the centroid weights

2.10.4.6 Load and read the reference centroids

2.10.4.7 Save the current centroid positions as an reference centroid file (averaged over a settable number of frames)

2.10.4.8 Load and read the centroid offsets

2.10.4.9 Set and read pixel offsets for the camera image

2.10.5 Reconstruction

2.10.5.1 Set the guidestar mode (NGS or LGS) for any WFS

2.10.6 Telemetry

2.10.6.1 Set and read the telemetry rate for each data stream

2.10.6.2 Set and read the data streams that are enabled for telemetry

2.10.7 Debug

2.10.7.1 Set and read debug levels (these are similar to telemetry but specific to a debug implementation)

3 The Tomography Algorithm

The tomography unit is part of a naturally highly parallel system. The inputs to this system are from multiple independent sources and its outputs go to multiple independent destinations. The processing of these inputs and output can be done in parallel. Additionally, the computations involved for each of these parallel paths can also be highly parallelized.

The tomography algorithm itself is also highly parallelizable. The result of all previous parallel processing of the inputs are combined and processed together to create an estimate of the atmosphere (structured in layers) that is consistent with the inputs from the wave front sensors.

The layer data generated is subsequently processed in parallel and used to control the DMs to correct for the atmospheric aberrations.

The majority of computational cycles and time in the system is spent in the tomography unit (see ____). Therefore, it is critical to ensure that to the maximum extent possible we the processing in this unit is efficient.

Show individual arrows

Mult data paths from mult sources Mult data paths to mult dest

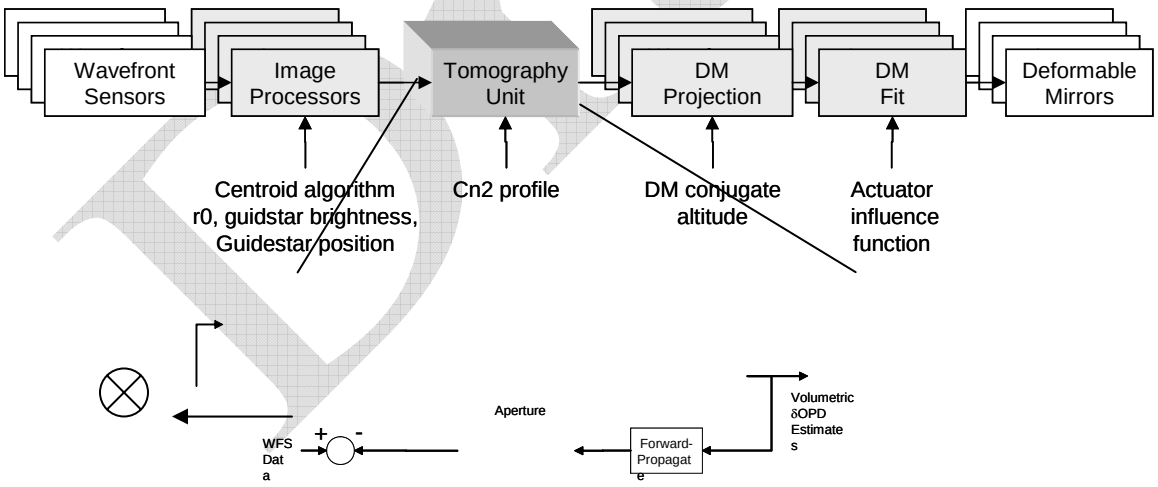

Figure 3-1 Where Does Tomography Fit in the Adaptive Optics Problem?

Our processing is no longer on parallel independent data streams, so we must find another way to speed our computations. We solve the problem by domain decomposition; dividing our data into independent regions and solving each region in parallel (see ____).

When decomposing our problem, we keep the same mathematical form for all domains. This makes it easier to parallelize the problem; it allows flexibility in scaling to different sized problems; and it aids in validating our design.

Tomography is accomplished by performing 2-D Radon transforms [6] through our estimated volume (forward projection). This is done by adding the estimated values of all voxels from each direction from each guide star and comparing the results with the actual projections of the guide stars through the real atmosphere as measured.

If our estimate of the atmosphere is inaccurate, we will see an error between the measured projections (from the WFS) and the estimated projections.

We take the errors for each sub aperture of each WFS and project it back into the estimated atmosphere (back projection). The projection is done along the path each ray took in its forward projection to the sub apertures. The errors are apportioned amongst the voxels according to the strength of each layer as determined by the state of the real atmosphere.

The projected errors from each ray that passed through a voxel are averaged and added to the current estimate of the voxel delay to get a new and more accurate estimate of the value for each voxel.

Now we have a new estimate for the atmosphere and we start the cycle again.

The following calculations are done for the various layers and guide stars in a single Fourier coefficient or sub aperture.

"L" is the specific layer involved

"G" is the specific guide star involved

It is assumed that all this pertains to the same sub aperture or Fourier coefficient in the mesh.

3.1 Forward Propagation

We trace the path of light rays as they travel through the atmosphere from a specific guide stars to a specific sub aperture.

We do this in two ways. The first is when we measure the actual atmosphere. This measurement is the line-integral of the actual phase delay of the real light from the guide star. The second is when we trace the path of the same rays through our estimated atmosphere and add up the estimated delays in each voxel.

We do this for each guide star and thus have a set of measurements of the actual atmosphere and a set of estimates for the estimated atmosphere.

$$Fwd\_Prop[G] = \sum_{L} Est\_Val[L]\, Shift[L,G]\, Scale[L]$$

The coefficient for each guide star is calculated by summing the corresponding estimated coefficient for each layer multiplied by the shift factor (complex) for that layer.

Note: the Scale[] factor is not applied for natural guide stars.

3.2 Error Calculation

Our estimated values are compared to their corresponding measured values. An error will result depending on the degree of mismatch between the actual atmosphere and our estimated atmosphere.

We need to adjust the value of the delay in the voxels in our estimated atmosphere based on this error so that they will better match the real atmosphere resulting in a lower error on the next cycle. When the RMS error has decreased to a pre-determined level, our iterations through the loop will end.

$$Adj\_Fwd\_\mathrm{Prop}[G] = IDFT\left(Fwd\_\mathrm{Prop}[G]\right)Aperture$$

Perform and IDFT on the forward propagated values for each guide star to bring it into the spatial domain. Multiply the sub aperture value by its aperture value (1 or 0).

$$Error[G] = DFT\left(Measured[G] - Adj\_Fwd\_\mathrm{Prop}[G]\right)$$

Calculate the spatial error value for each sub aperture for each guide star by subtracting the adjusted forward propagated values from the measured values. Move the spatial error values back to the Fourier space by performing a DFT on them.

$$PC\_Error[G] = Pre\_Cond\left(Error[G]\right)$$

Precondition the error values for each guide star before back propagating them.

3.2.1.1 Applying an Aperture

In the Fourier domain, we have a repeated spectrum due to sampling. The unavoidable result of this is that when we forward propagate, we end up with a wave that has values in the spatial domain that are outside of our aperture. All values outside our aperture must be zero in the spatial domain, so these values represent an error. We need to avoid back propagating this error in the next iteration.

Consequently, when we forward propagate in the Fourier domain, we must transform the resultant wave front back to the spatial domain. Then we zero out the values outside the actual aperture. This can only be done in the spatial domain.

Then we calculate the error between our actual measured value, which of course has zeros in the values outside of the aperture, and our apertured forward propagated value. Then we transform the error value back into the Fourier domain and back propagate that.

This seemingly simple Fourier/Fourier^{-1} transform pair is performed on each iteration. Unfortunately, as simple as it is to describe, it is also the single largest element in the iteration time and becomes a primary determiner of our latency, computational and busing requirements.

3.3 Back Propagation

We distribute an error value back along the paths to each guide star and distribute it to the various portions of the atmosphere through which it passes in an amount based on the layer strength, C_n^2.

$$Coeff\_Error[L] = \sum_G PC\_Error[G] C_n^2[L]$$

Calculate the new error for each coefficient in a layer by summing the back propagated errors from the same coefficient in each guide star multiplied by the C_n^2 for that layer.

Note: The C_n^2 factor is actually a conglomerate of C_n^2, gain, scaling and the averaging factor for the number of guide stars.

$$Adj\_Coeff\_Error[L] = Coeff\_Error[L] K\_Filter[L]$$

Adjust the coefficient error by multiplying it by its corresponding Kolmogorov filter value.

3.4 Filtering the Back Propagated Error

To increase the accuracy of our estimated atmosphere we can filter the errors to better fit the known characteristics of the atmospheric turbulence.

3.5 Calculating a New Estimate

The errors for each voxel from different ray directions are now averaged and added to the previous estimate to arrive at a new and more accurate estimate of each voxel's contribution to the total delay.

$$Est\_Value[L] = Adj\_Coeff\_Error[L] + Est\_Value[L]$$

Calculate the new estimated value by adding the previous estimated value to the adjusted coefficient error.

4 The Architecture

4.1 Frame Based Operation

The operation, ignoring initialization, is frame based, with a nominal frame rate of >1 KHz. At the start of each frame, data from Wave Front Sensor (WFS) cameras is loaded to the WFS. Simultaneously, data for the DMs, which was calculated in the previous frame, is loaded into the DMs.

Each WFS computes a wave front for the light from its corresponding guide star.

diagram

After the wave fronts have been processed, a consistent 3D estimate of the atmospheric volume is created by the tomography algorithm. This volume estimate is arranged by sub apertures and layers.

The estimates of the layer information are processed into wave fronts for each DM and that information is further processed into actual commands for each DM.

Tip/Tilt information is both extracted from the WFS as appropriate and recombined appropriately for use by each DM.

4.2 Data Flow

4.2.1 I/O

The solution starts and ends with I/O. How do we get the massive amount of data into our system, so we can process it; and get the equally massive amount of data out, so we can use it: all in less than one millisecond.

4.2.2 Processing the Data

Figure 4-1 shows the data flow. You can see that the problem is a parallel one. Multiple independent data paths flow from the WFSs and converge to be processed by the tomography algorithm. Multiple independent data paths flow from the tomography algorithm to be processed each by the layer and DM processing algorithms. The input and output processing is parallelizable.

The tomography algorithm is not only parallelizable, but embarrassingly parallelizable.

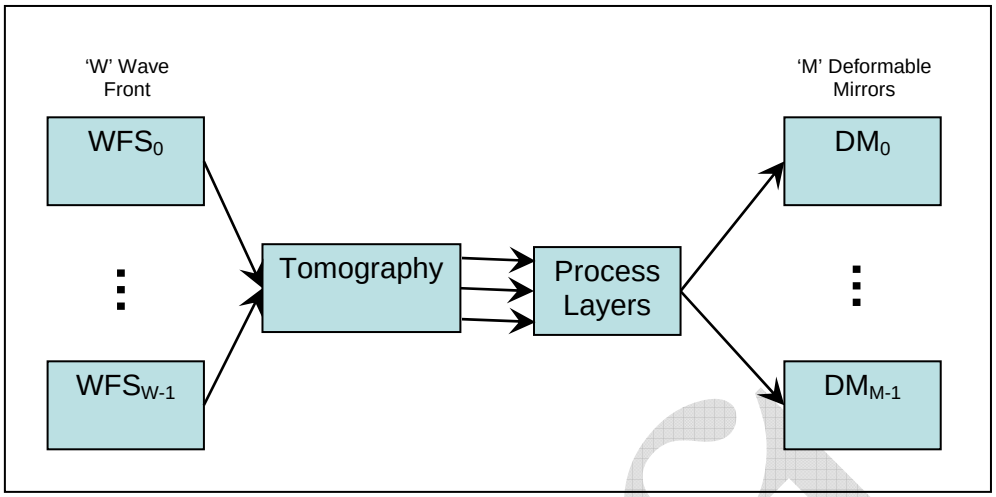

Figure 4-1 Overall data flow

Because we solve our problem in the Fourier domain, which assumes our data is replicated outside our domain, we get wrapping errors when paths are propagated through the replicated domains. This results in forward propagated values in regions outside our aperture. Before we can calculate errors to back-propagate, we must force those values to zero. If we naively back propagated these errors, the stability of our system and the accuracy of our solution would be suspect. Consequently, we must take our forward propagated Fourier domain data back to the spatial domain and reapply our aperture, forcing values outside it to zero. The augmented data flow for the Fourier domain is shown in Figure 4-2.

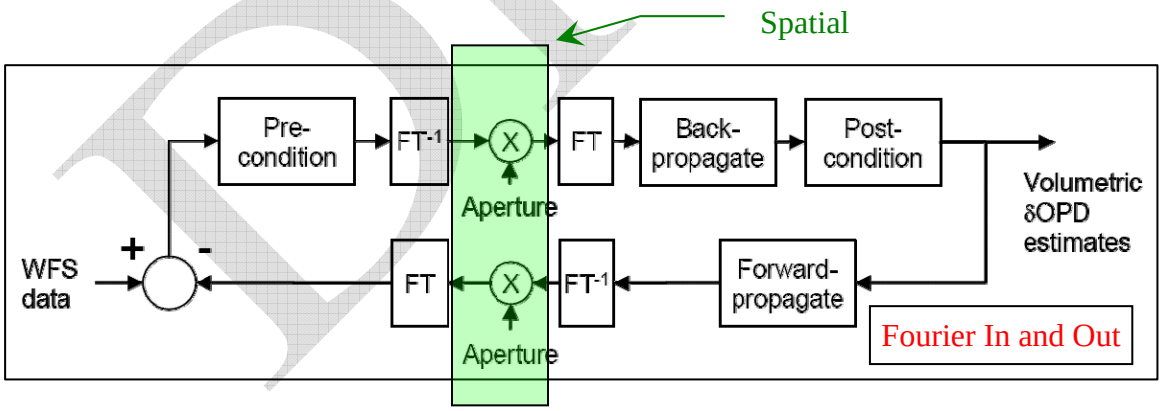

Figure 4-2 Actual Fourier Domain Data Flow with Aperturing

Move up

4.3 Macro Architecture

We divide our problem into domains and sub-domains. At the top level is the domain of the entire 3D atmospheric volume. This is further divided into layers and sub apertures[1].

For our purposes, we divide the atmosphere into columns. Each column is one sub aperture square at the ground and extends vertically through the various layers of the atmosphere. The intersection of a column and a layer defines a voxel and we assign a processor to each voxel to calculate its OPD. The voxel is the smallest sub domain we deal with.

Each integrated circuit handles sub-domain consisting of a square region of adjacent columns.

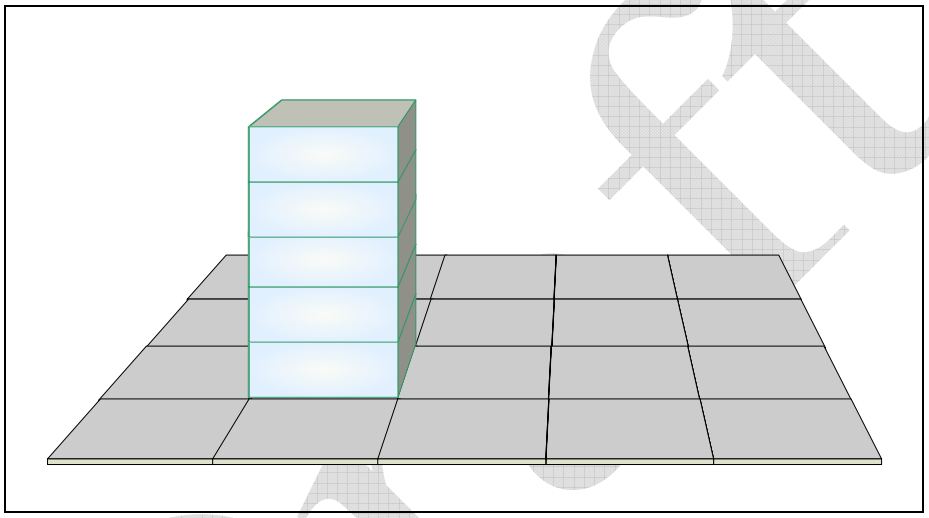

Figure 4-3 A column of combined layer and guide star processors form a sub aperture processor

4.3.1 Inter Element Communications

Figure 4-4 shows the array busing. The bandwidth across all interfaces must be the same to insure efficient operation. However, the data path width and transfer clock rate across chip and board boundaries can be different to accommodate the physical realities of the system.

[1] Note that the term "sub aperture" is normally used in a spatial context to denote a portion of the primary aperture of the telescope. Here it is also extended to denote a portion of the frequency bins across this aperture. Fourier transforms are only applied on a per layer basis so the layer information remains intact, while its sub apertures are transformed.

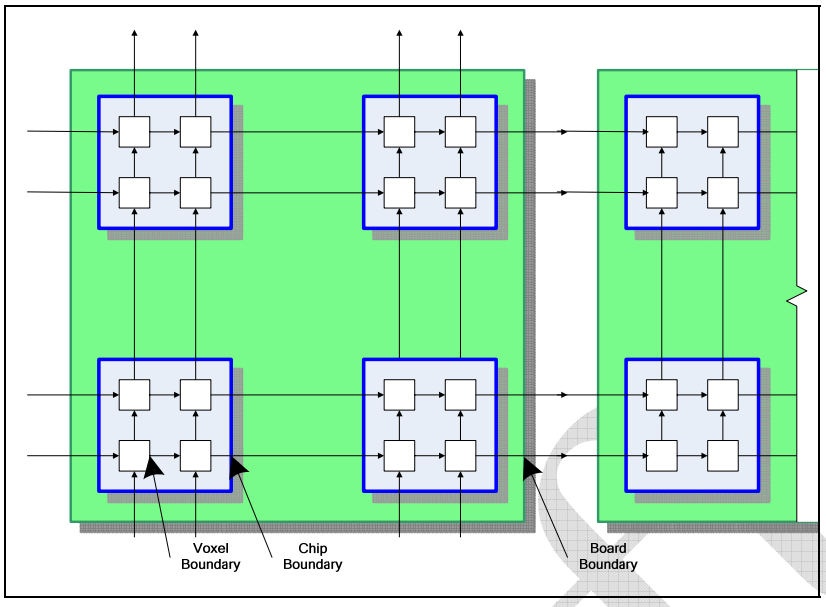

Figure 4-4 The Systolic Array, showing boundaries for different elements
Sub aperture processors (black), chips (blue), and boards (green)

Each processor for a sub aperture must talk to each orthogonal neighbor.

Each IC has $\boldsymbol{P}$ I/O pins. This gives each processor a bus size of $\frac{\boldsymbol{P}}{(4\boldsymbol{Lm})}$ I/O pins to communicate with each neighbor.

An example could be a Xilinx® Virtex-4 SX35, which has approximately 600 pins we can use for this communication. Each device handles five layers and nine sub apertures. Therefore, our bus size between adjacent chips is 10 bits. This allows eight pins for data and two for clock and control.

4.4 Micro Architecture

Control is distributed to individual chips and sub aperture processors. Both data and control are pipelined.

Given the immense memory bandwidth and compute requirements, we use a Harvard architecture (program and data memory are separate), SIMD model processor for the basic processing cell. This single common processing cell is used throughout the array. Independent communications paths for code and data and allows us to perform operations on multiple data elements on each instruction.

The system is implemented using direct communications only between the immediately adjacent cells in the x, y and z directions: four horizontally orthogonally adjacent voxels and the voxel above and below.

These sub aperture columns are arranged in a square on an x,y-grid, which corresponds to the sub apertures on the instrument. The actual number of sub apertures used is larger

than the number of sub apertures on the instrument, since we must allow for the fact that our volume will typically be a trapezoidal and be wider at the top (see _____).

A central control core is used for each chip.

Each column of processor cells has a single local state machine for control. This state machine supplies control for all the columns cells as a wide word, with some local decoding at the cell level. The column state machines are globally synchronized at the start of each frame. If a state machine inadvertently finds itself in an invalid state, a log of the event is kept and a signal is made available outside the system. System logic can determine the best action based on frequency or location of the occurrence.

Each voxel has a small amount of control logic for decode.

The optimum distribution of the control logic between elements will be addressed during a later optimization stage of development.

4.5 Arithmetic Errors and Precision

A DFT has $O(N^2)$ operations for each transform, while an FFT has O(Nlog(N)) operations. Using exact arithmetic, they have equal accuracy. However, when using finite math, the DFT has the potential to incur more round-off errors than an FFT.

Care must be taken in selection of word length to avoid this error from accumulating over many transforms. We use 18-bit words with intermediate results carried to 47-bits.

4.6 Program Flow Control

SIMD vs. MIMD

There is no need (or ability) for each voxel processor (or sub aperture processor) to have branch capability or individual register addressability within the array. All processors execute the same code on exactly the same register address in synchronism.

However, at global level, we do need to:

- Determine whether we have reached convergence across all sub apertures
- Determine what diagnostic programs to run in the time between the end of convergence and the start of a new frame
- Synchronize control logic at the start of a frame and determine when we are too close to the end of a frame to start another iteration.

The residual error between our WFS measurements and our current forward propagated values must be evaluated each iteration to determine if we have converged. To do this we need to determine the ***global*** RMS of the errors for each guide star for each sub aperture.

Each frame starts with the result of the previous frame, so in general, we will converge to a new estimate before the end of a frame. The remainder of each frame can thus be used for diagnostic and monitoring processes. Diagnostic and monitoring functions can be interrupted at the end of a frame and continued at the end of the next frame and so on until they complete

Since we must convert the forward propagated values to the spatial domain and apply the aperture to them, we calculate the error in the spatial domain. Then we transform the

error back to the Fourier domain for back propagation. In our architecture, we do this by N shifts across rows to accomplish the transform (see **Error! Reference source not found.**). During this transformation, each CP calculates the running sum-of-squares for the error. When the transform is over, if the value is below our criteria, we stop iterations and start or continue any diagnostic or monitoring processes that need to be run.

A running total of the number of iterations in the frame is also monitored and compared to a preset limit. This limit is set to guarantee that each iteration will complete before the start of the next frame. During anomalous situations, such as initial startup or an obscuration traveling rapidly across the aperture, the system may not be able to converge to our lower error limit in a single frame and must continue at the start of the next frame. We don't want the frame sync to occur in the middle of an iteration, leaving the machine in an uncertain state.

4.7 Chip issues

Sockets, cooling, by passing,

4.8 Board level issues

by passing, routing, EMI, I/O

4.9 System level issues

Inter board connection, power distribution, clock distribution, I/O, cooling, reliability, diagnostics, power, scalability

We solve our problem using domain decomposition. We have 'N' elements or voxels in our atmosphere (our domain). We have 'm' processing elements (PEs) and we divide the task of calculating the OPD for each voxel into 'm' parts with each processor handling N/m voxels.

All PEs use exactly the same algorithm and so finish in exactly the same number of clocks.

We have three basic tasks that our architecture needs to support: I/O, DFT (both highly parallelizable) and a linear solver (embarrassingly parallelizable).

No local or global memory access is required to other Processing Elements (PEs), so memory can be completely local to the PEs.

Since all PEs execute the same program, the architecture is SIMD (Single Instruction Multiple Data) and the program could be located in a single external controller [7]. However, for timing and performance reasons and in order to minimize the busing

resources used, we duplicate the control logic for each sub domain (individual FPGAs). As a side effect, this also gives us the ability to implement MIMD algorithms if needed.

The control logic for these domains is very small; and does not affect the density (cells per chip) of the implementation significantly.

5 Implementation

5.1 General Operation

The Tomography Engine is a 3D, systolic array of processor cells that calculates a 3-D estimate of the index of refraction of a volume of the atmosphere. The volume is broken into layers and each layer contains a number of voxels (volume elements that are one layer thick and one sub aperture square).

Since the calculations for the tomography are embarrassingly parallel, we dedicate a single processor for each voxel. Each processor is very simple and has enough local memory to do all its processing without needing access to global or a neighbor's memory. The nature of the parallel algorithms implemented determines the exact amount of this memory required.

The overall operation is on a frame basis, with a frame having a nominal duration of 1 msec. The operations during each frame are identical to those of all other frames.

During each frame, the engine:

- At the beginning of a frame, data is shifted into one side of the array from the WFSs. As the WFS data is shifted in, the processed data is simultaneously shifted out of the other side. Thus the data load and unload portion of a frame are completely overlapped.
- During the rest of a frame, data circulates through the array for processing, shifting from cell to cell, by rows or columns during operation. The shifting of data during operation uses the same data paths that were used for the load/unload. In this time, the iterations of the algorithm are performed until the RMS error falls below a pre-determined level.

There is no global communication between the processors. All communication is local between adjacent processors.

5.2 The systolic array

The nature of the iterative solver in the tomography engine is easily parallelized. As soon as input data become available, calculations can be performed concurrently and in-place as data flows through the system. This concept was first defined by Kung at Carnegie-Mellon University:

A systolic system is a network of processors that rhythmically compute and pass data through the system. Physiologists use the word "systole" to refer to the rhythmically recurrent contraction of the heart and arteries that pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network [8].

At first, systolic arrays were solely in the realm of single-purpose VLSI circuits. This was followed by programmable VLSI systolic arrays [9] and single-purpose FPGA systolic arrays [10]. As FPGA technology advanced and density grew, general-purpose

"reconfigurable systolic arrays" [11] could be put in single or multiple FPGA chips. The capability of each processing element in early FPGA systolic array implementations was limited to small bit-level logic. Modern FPGA chips have large distributed memory and DSP blocks that, along with greater fabric density, allow for word-level 2's complement arithmetic. The design goals for our systolic array are:

- Reconfigurable to exploit application-dependent parallelisms
- High-level-language programmable for task control and flexibility
- Scalable for easy extension to many applications
- Capable of supporting single-instruction stream, multiple-data stream (SIMD) organizations for vector operations and multiple-data stream (MIMD) organizations to exploit non homogeneous parallelism requirements [12]

Because of tasks, such as multiple 2-D DFTs per Frame, the number of compute operations drastically outnumbers the I/O operations. The system is therefore "compute-bound" [13]. The computational rate, however, is still restricted by the array's I/O operations that occur at the array boundaries. The systolic array tomography engine is composed of many FPGA chips on multiple boards.

The advantages of systolic arrays include reuse of input data, simple and repeatable processing elements, and regular data and control flow. At the same time, input data can be cycling into the array while output data is flowing out. At another point calculated data can be flowing. Each individual processing element can only grab and use the data that are presented to it from its nearest neighbors or in its local memory on every clock. This chapter will start at the lowest level of the architecture, the processing element (PE), and gradually zoom out to a system-level perspective.

5.3 The processing element (PE)

The heart of our processor is the Xilinx DSP-48 multiplier / accumulator. This is a very powerful cell, which can perform 18-bit pipelined operations at 500MHz. Each processing element uses two of these for the complex arithmetic of the Fourier transform. A single FPGA chip can have over 500 of these cells and allows us to have over 250 PEs per chip.

This is the power of the FPGA. While each chip might only be processing data at 100 MHz, each chip contains 250 processors for a combined processing capability of 25 G Operations per second.

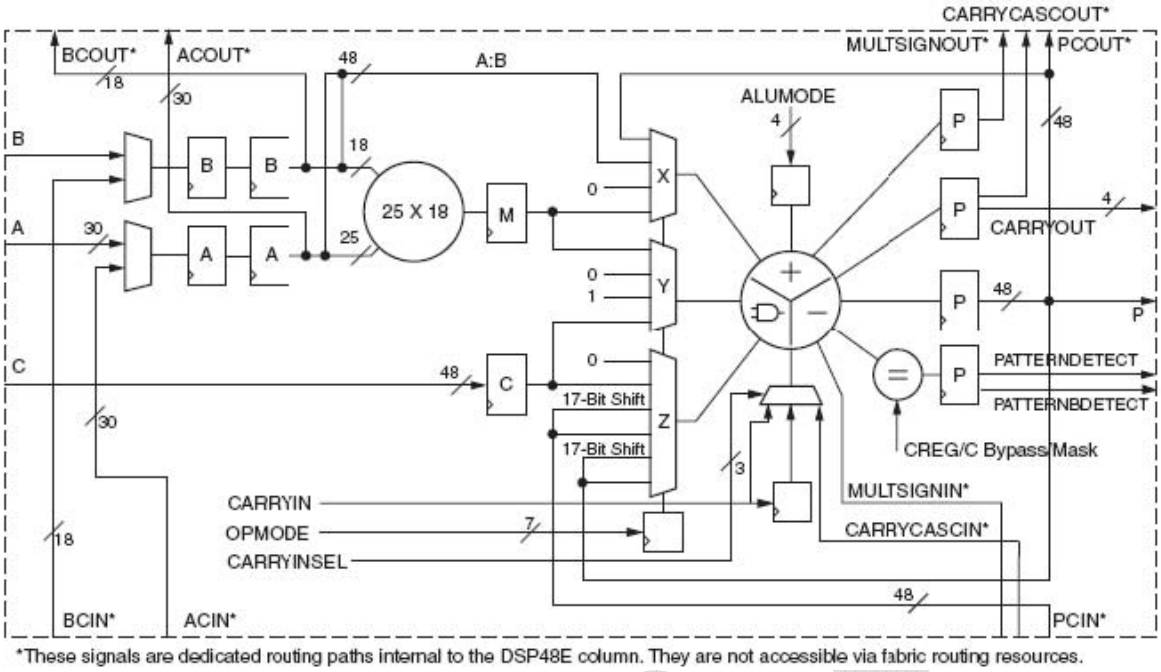

Figure 5-1 The DSP48E architecture [14]

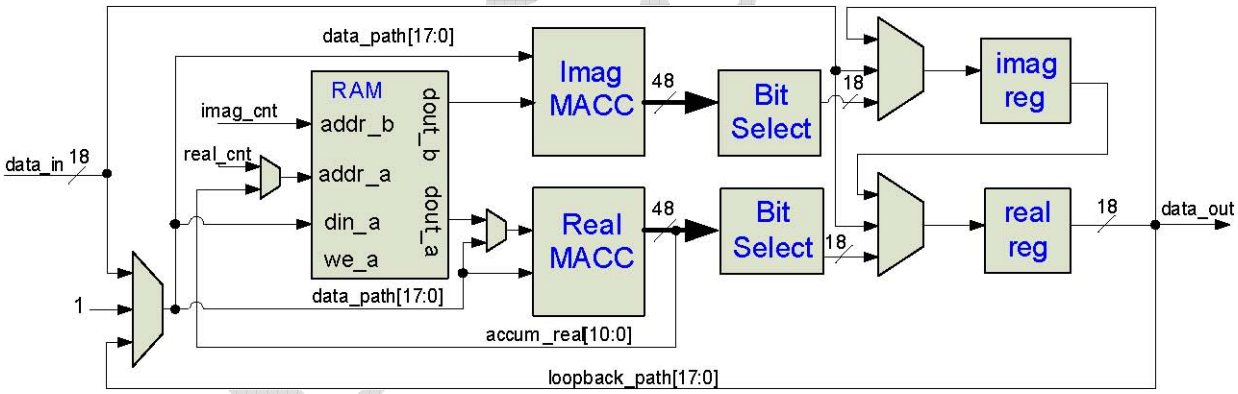

Figure 5-2 A single Processing Element (PE)

Two DSP48s are employed, one for the real and one for the imaginary part of the complex number. As shown in Figure 3.2, the PE also contains a dedicated BlockRAM memory, an18-bit register for the real part, and an18-bit register for the imaginary part. Multiplexors control whether these registers receive data from their respective MACC or if data are just snaked through them to the next PE. Note that there is only a single 18-bit input and single 18-bit output. This is because when data is flowing through the mesh, the real part is transferred on even clock edges and the imaginary part on odd edges. This is particularly important for instructions types such as the DFT where complex multiplication is performed on inputs split across two clock cycles.

5.4 PE interconnect

The switching lattice for a single 3x3 layer of the AXEY is shown in Figure 3.3. For a large telescope, this lattice could be 60x60 or larger. The individual PEs are labeled by their column and row position in the mesh. Each PE has a multiplexor on its input to route data orthogonally from a neighboring horizontal PE, vertical PE, or next layer PE. External I/O only takes place at the mesh boundary on the right and left sides. Information shifts in a circular fashion along columns, rows, or layers. All data paths are 18-bits wide to match the fixed 18-bit inputs of the DSP48E block.

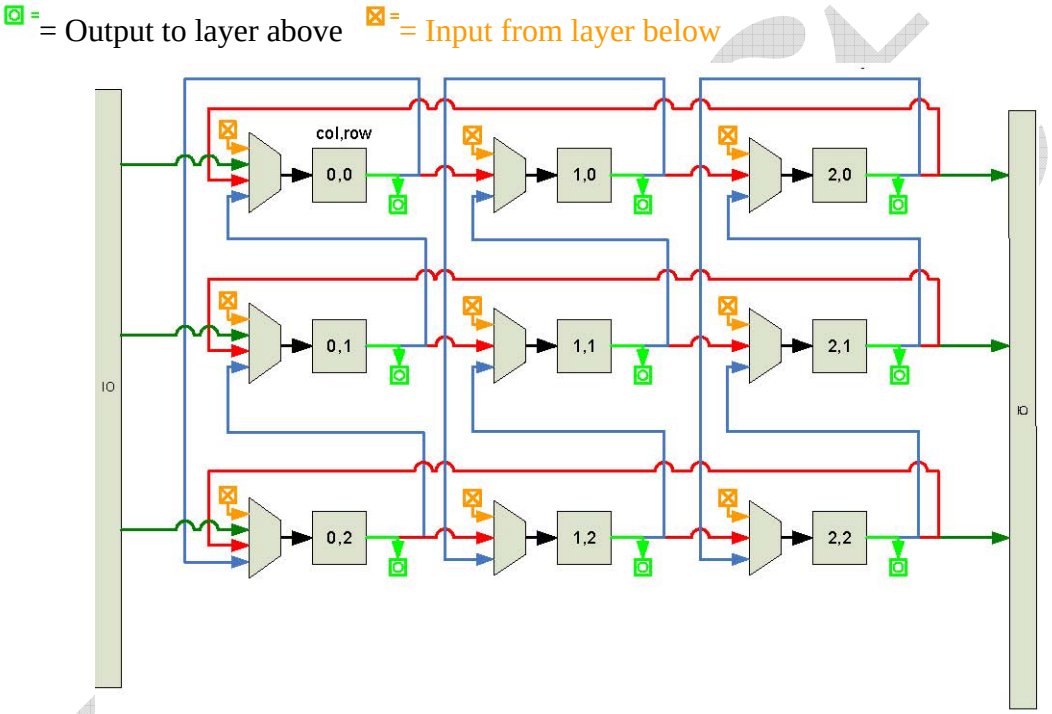

Figure 5-3 The PE lattice

FPGA Device	SX35T	SX50T	SX95T
DSP48E available	192	288	640
Maximum # PEs	192/2=96	144	320
Rows	3	5	7
Columns	3	5	7
Layers	3	3	3
# PEs used	3*3*3=27	75	147
Bits per TX/RX edge	3*3*18=162	270	378
TX/RX BW required @ 100MHz	32.4 Gbps	54 Gbps	75.6 Gbps
3.125 Gbps Transceivers (GTPs)	8	12	16
TX/RX possible BW with GTPs	25 Gbps	37.5 Gbps	50 Gbps
LVDS DDR pairs	180	240	320
TX/RX possible BW with LVDS	18 Gbps	24 Gbps	32 Gbps
Total TX/RX BW of chip	25+18=43 Gbps	61.5 Gbps	82 Gbps
TX/RX BW slack	43-32.4=10.6 Gbps	7.5 Gbps	6.4 Gbps

Table 5-1 Chip bandwidth chart

5.4.1 I/O bandwidth

In the multi-chip system, an important question is how many PEs can be partitioned for each chip. The system has I/O bandwidth requirements that outweigh all other resource requirements. In Figure 3.4, I/O requirements and resource availability for the threeVirtex5SXFPGAs are compared. In order to meet the bandwidth requirements for a reasonable number of PEs on a chip, on-chip multi-gigabit transceivers will have to be used. It is important to note from the figure that not all of the DSP48E resources were used because total chip bandwidth runs out at a certain point. In order to use additional DSP48Es, additional techniques will be employed such as fabric serialization/deserialization for the LVDS DDR pairs.

5.5 SIMD system control

Most control signals are single bit control, but some, like the address inputs to the BlockRAM, function as index counters to the stored coefficient data. In addition to fine-grained control at the PE level, control signals also manipulate the multiplexors at the switch lattice level. These control signals are global and can be issued by a single control unit or by distributed copies of the control unit. The control unit requires very few resources, so even if multiple copies are distributed, the cost is minimal. The control communication overhead of larger arrays can also be avoided with a copied control scheme.

5.5.1 Cycle accurate control sequencer (CACS)

When the tomography algorithm was mapped to hardware, we found that the array could be controlled by linear sequences of control bits, specific clock counts of idle, and minimal branching. A Cycle Accurate Control Sequencer module (CACS), shown in Figure 3.5, was architected to be a best of both worlds solution that would (1) borrow

single cycle latency benefits from finite state machines and (2) use programmability aspects of a small RISC engine. Because the CACS logic consists of only an embedded BlockRAM module and a few counters, it has both a small footprint and a fast operational speed.

The control sequence up counter acts as a program counter for the system. The following types of instructions can be issued from the control BlockRAM:

1. Real coefficient address load: Bit 22 signals the real coefficient up counter to be loaded with the lower order data bits.
2. Imaginary coefficient address load: Bit 21 signals the imaginary coefficient up counter to be loaded with the lower order data bits.
3. Control sequence address load: Bit 20 and status bits control whether or not a conditional branch is taken. If a branch is to be taken, then the control sequence up counter is loaded with the address contained in the lower order data bits.
4. Idle count: Bit23 loads adown counter with a cycle count contained in the lower order data bits. This saves program space during long instruction sequences where control bits do not have to change on every cycle. When the down counter reaches zero, the idle is finished and the control sequence up counter is re-enabled.
5. Control bus change: When the high order bits are not being used for counter loads, the low order bits can be changed cycle by cycle for the control bus registers.

Three types of low-level instructions are presented in Figure 3.6 to show how a sample control sequence in a text file is compiled by script into BlockRAM content. First the single bit outputs are defined, then an idle count command creates a pause of “cols*2-1” number of times. Note that “cols” is a variable dependant on the number of east/west columns in the systolic array. The AXEY instructions are flexible because they incorporate these variables. Single bit changes are done on the subsequent two cycles and finally the real and imaginary coefficient counters are loaded.

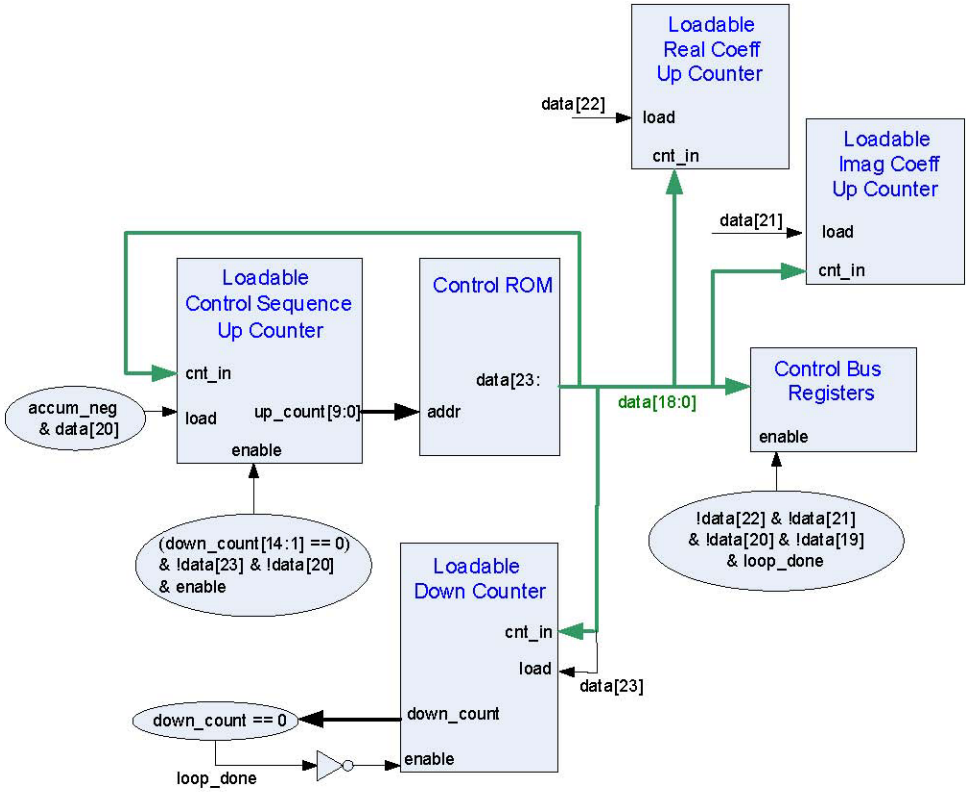

Figure 5-4 CACS Architecture

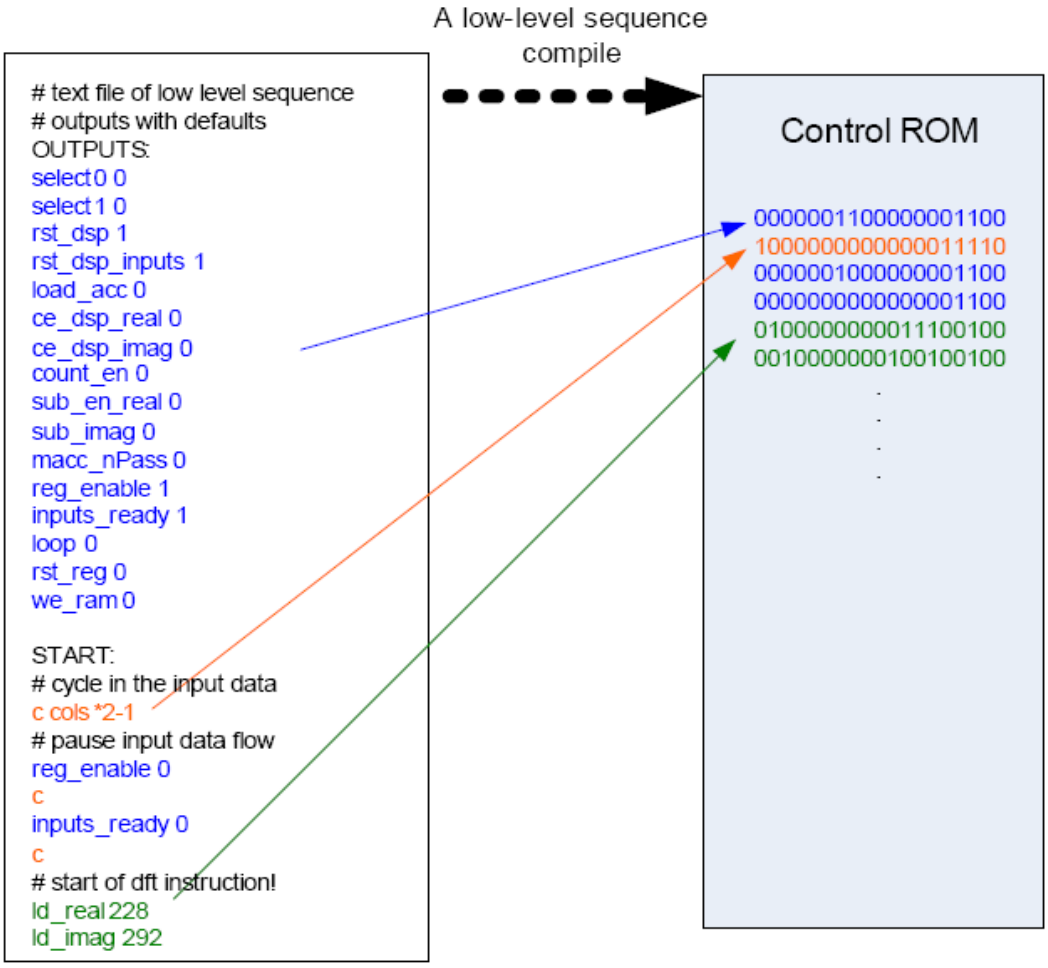

Figure 5-5 Sample CACS ROM content

5.5.2 Instruction set architecture

The instruction set was built to map the algorithms in the basic loop to the reconfigurable systolic array. First, logic was designed around the fixed Block-RAM and MACC resources with control microcode to support the sequences required by the algorithms. Out of the resulting sequences, groups of common microcode sequences were identified to form the instruction set. Instructions are simply compound sets of control sequences so new instructions are simple to add and test. As seen in Table 3.1, instructions are grouped according to which type of calculation they perform: multiple calculations, single calculations, data movement within the PE, or control counter manipulation.

Calculation Type	*Instructions*
Multiple Calculations	macc_layer, macc_gstar, dft_ns, dft_ew, square_rows, add_gstar_reals, add_reals_ns
Single Calculation	macc_loopback, add, sub
P.E. Data Movement	rd_ram, wr_ram, wr_ram_indirect, rtshift_store, noshift_store, advance_regs, refresh_regs
Counter Control	branch_if_neg, ld_ramcnt_indirect

Table 5-2 Instruction types

The instructions are shown in Table 3.2 with their respective result. Most instructions require an address upon which to act. The long sequence instructions, such as *macc layer* and *dft ew*, take a starting address and iterate through the proper number of subsequent addresses on their own.

5.5.2.1 macc layer, macc gstar, and dft ns/ew

The *macc layer*, *macc gstar*, and *dft ns/ew* instructions are all essentially complex multiply-accumulate functions. As data systolicly flow through the data registers of each PE, the complex multiply accumulate is carried out by increasing the address index for the coefficient RAM and toggling the subtract signal for the real MACC on every clock edge. This functionality is illustrated in Figure 3.7 where the active data lines are shown in red. The difference between *macc layer*, *dft ns*, and *dft ew* is only in the switch lattice, where a circular MACC is done *north/south* by *dft ns*, east/west by *dft ew*, and through the layers by *macc layer*.

The timing diagram for an in-place DFT accumulation is shown in Figure 3.8. As alternating real and complex values are shifted into the PE, labeled *N* and *n*, respectively, the corresponding coefficients, *C* and *c* from the dual-port BlockRAM are indexed. The example is for a 2x2 DFT, where the row DFT (*dft ew*) is calculated first and the column DFT (*dft ns*) is performed on those results.

Instruction	**Result**
macc layer (addr1)	Accum: (addr1)*layer1 data+(addr2)*layer2 data...
macc gstar (addr)	Accum: (addr1)*data+(addr2)* data...
Dft ns/ew (addr1)	Accum: (dft coeff1)*data1+(dft coeff2)* data2...
macc loopback (addr)	Accum: (addr)*data registers
square rows	Real Accum: (real1)2 +(imag1)2 ...
add gstar reals (addr)	Real Accum: (addr1)+(addr2)+ ...
add reals ns	Real Accum: data reg real1 + data reg real2 + ...
add (addr)	Accum: Accum+(addr)
sub (addr)	Accum: Accum-(addr)
rd ram (addr)	Accum: (addr)

Instruction	Result
Wr ram (addr)	BRAM(addr): data registers
Wr ram indirect	BRAM(Accum[10:0]): data reg real
rtshift store (addr)	data registers: (Accum[47:0]> >(addr))[17:0]
noshift store	data registers: Accum[17:0]
advance regs	data reg real: data reg imag
refresh regs	data registers: I/O
branch if neg (addr)	PC: (addr) if PE[0] Accum is negative
ld ramcnt indirect	coefficient counters: PE[0] Accum[10:0]

Table 5-3 Instructions

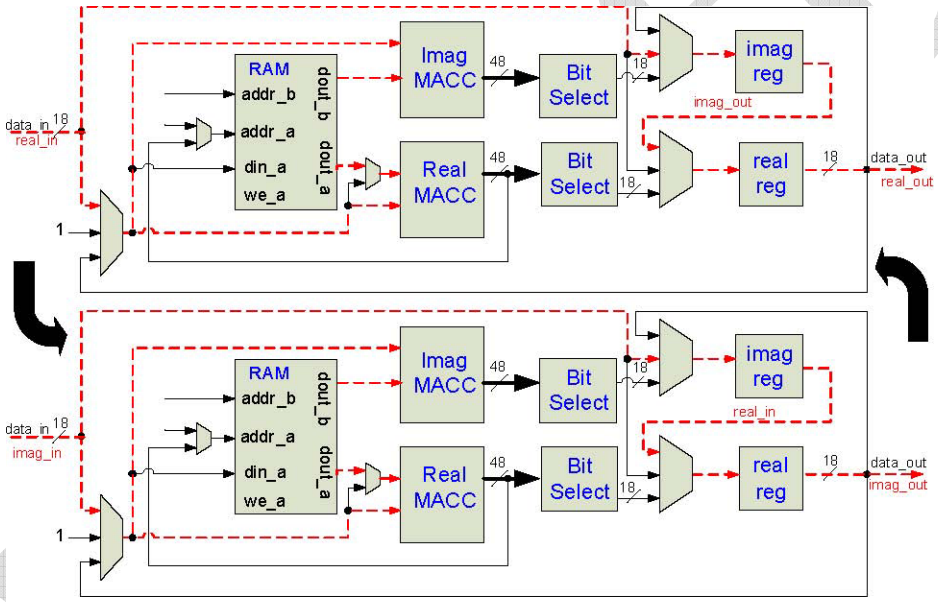

Figure 5-6 A complex MACC loop

The macc gstar instruction is performed once the error has been calculated for each guide star. A single complex data value is kept in circulation so that it can be continuously multiply-accumulated by a sequence of values in the RAM, as shown in Figure 3.9. This instruction is used for back propagation, where the C_n^2 value is circulated.

5.5.2.2 Square rows

The *square rows* instruction uses the same east/west circular shift that has been previously used for *dft ew*. The multiplexor in front of the real MACC is used to square and accumulate the incoming data stream until all columns in the row have been summed, as shown in Figure 3.10.

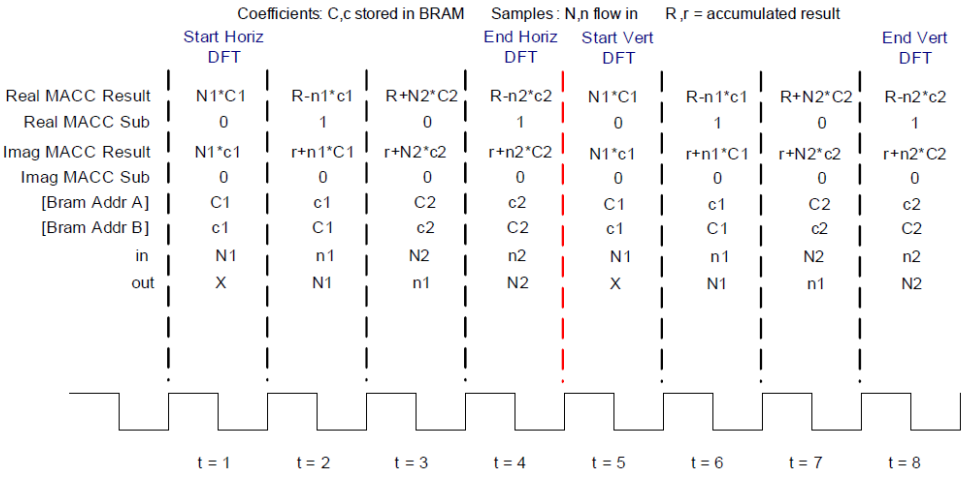

Figure 5-7 In-place DFT accumulation

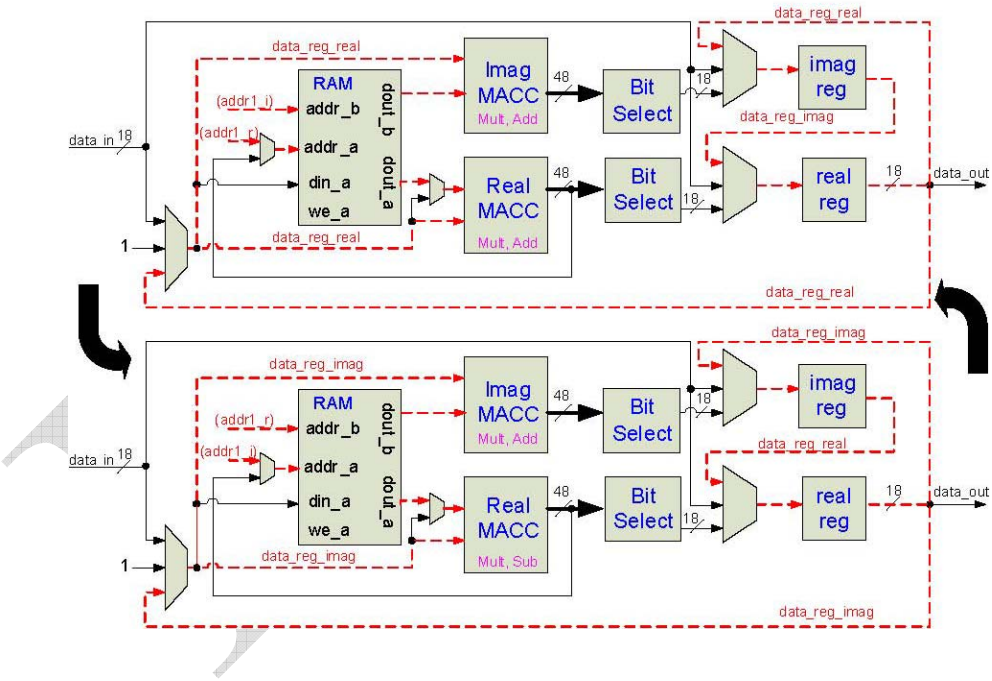

Figure 5-8 The *macc gstar* instruction

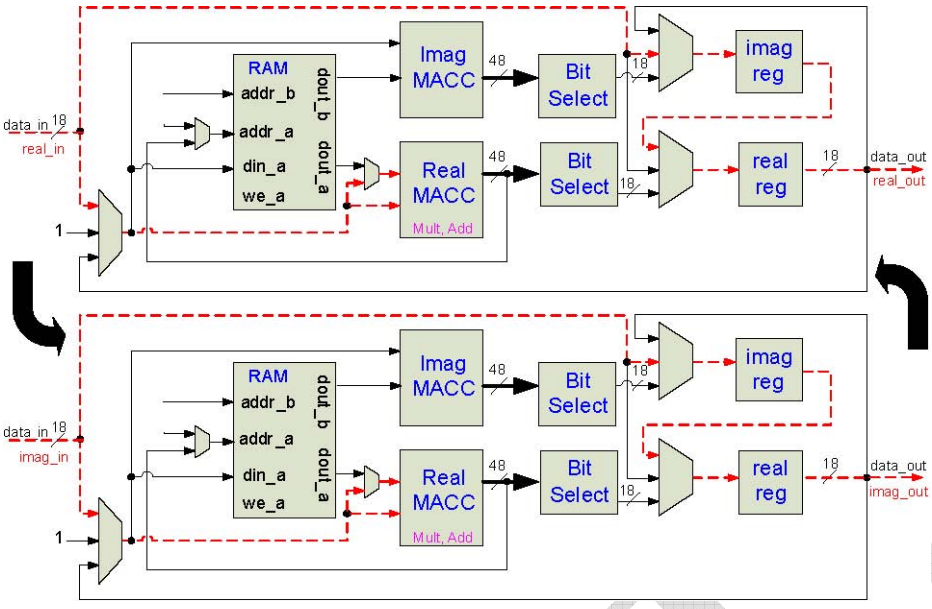

Figure 5-9 The *square_rows* instructions

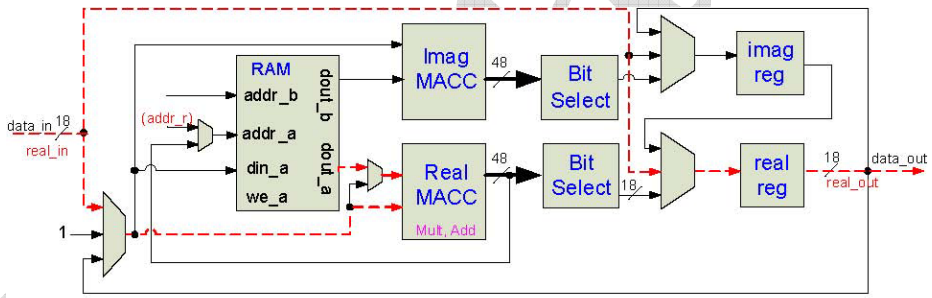

Figure 5-10 The *add_reals_ns* instructions

Multiplying by the systolicly flowing data, the multiplier instead acts upon locally loop-backed data from the PE's own real and imaginary registers, as shown in Figure 3.12.

5.5.2.3 add and sub

The add and sub instructions either perform a complex addition or a complex subtraction, respectively, of the accumulator to a value in the RAM. Because the MACC module always has to multiply by something, we feed a one constant into each multiplier's second input. The diagram is shown in Figure 3.13.

5.5.2.4 3.4.2.6 rd ram

The *rd ram* instruction first resets the contents of both MACCs. It then loads the real and imaginary counters to a specified memory location where the MACCs multiply-

accumulate the data at the given location with the constant one so that they now contain the data at that address, as shown in Figure3.14.

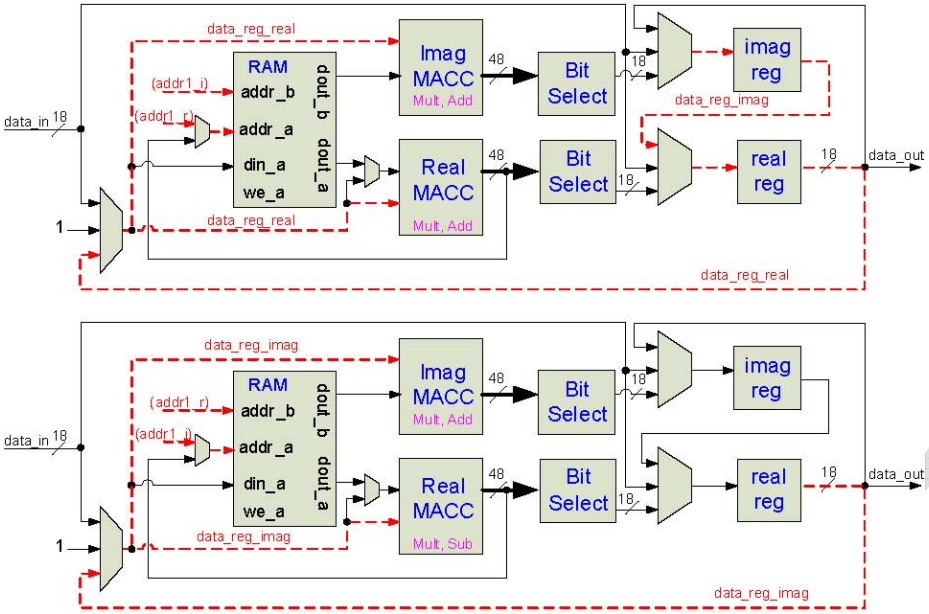

Figure 5-11 The *macc_loopback* instruction

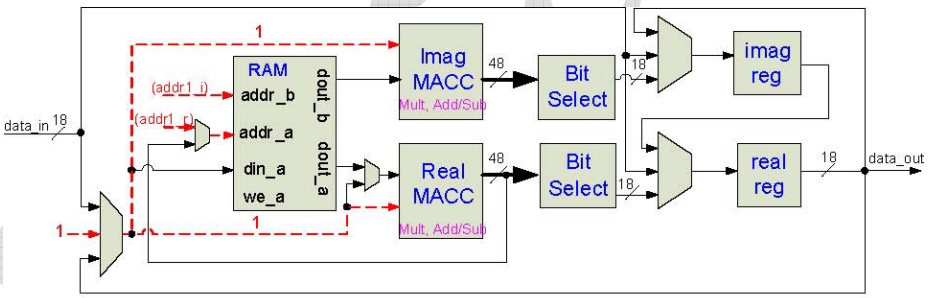

Figure 5-12 The *add* or *sub* instruction

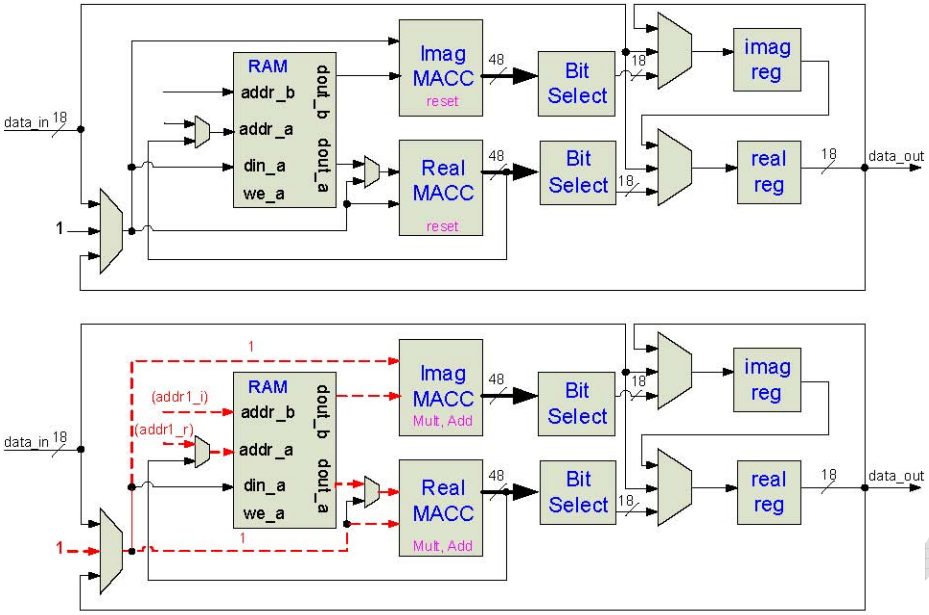

Figure 5-13 The *rd_ram* instruction

5.5.2.5 wr ram

The *wr ram* instruction writes the data registers into the BRAM in two phases, first the real part and then the imaginary part, as shown in Figure 3.15. In order to preserve the contents of the real and imaginary registers, the data are looped back into their inputs.

5.5.2.6 wr ram indirect

The *wr ram indirect* instruction is used to write the content of the real data register to a single local BRAM location, indexed by the lower order bits of the real MACC, instead of the usual global counter, as shown in Figure 3.16.

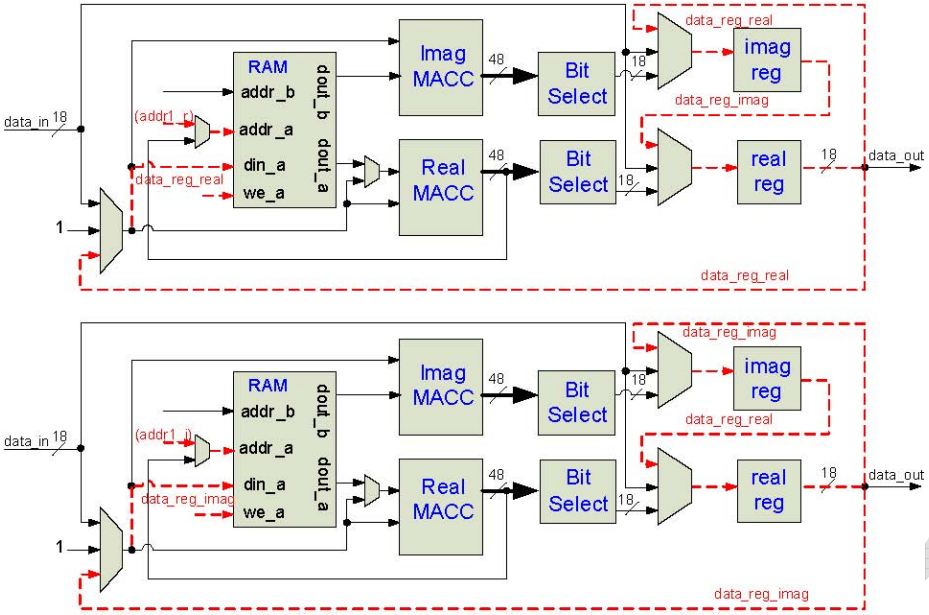

Figure 5-14 The *wr_ram* instruction

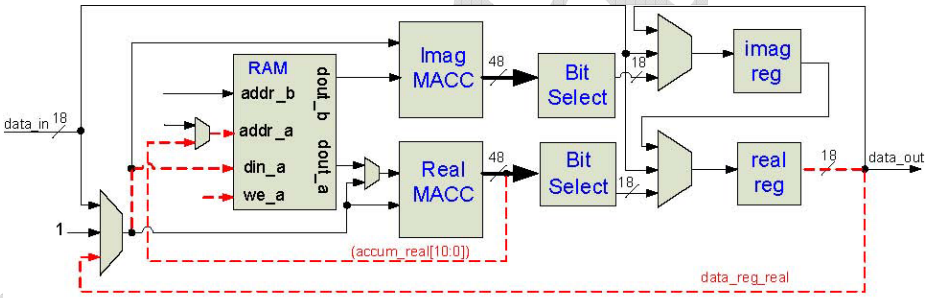

Figure 5-15 The *wr_ram_indirect* instruction

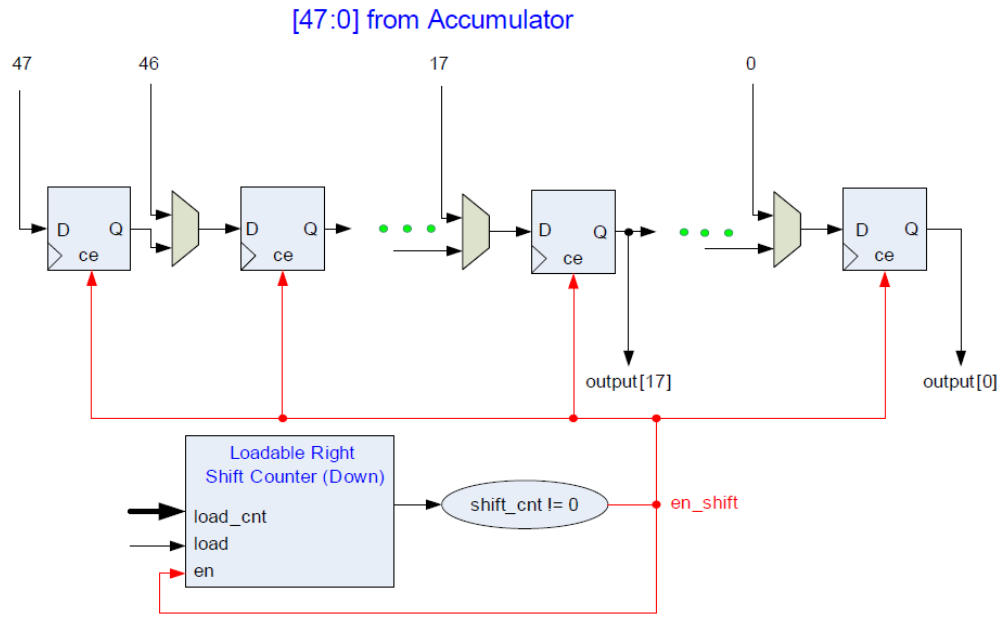

Figure 5-16 The bit selection logic

5.5.2.7 rtshift store and noshift store

The "BitSelect" block shown in all PE diagrams is used only by the *rtshift store* and *noshift store* instructions. By using the *rtshift store* instruction, the programmer can control how much the 48-bit data that are in the MACCs can be right-shifted before the data are stored in one of the 18-bit data registers. This is useful anywhere where the system needs to scale down data by a factor of 2X, such as normalization after a DFT. The *noshift store* instruction simply transfers the lowest 18 bits of the accumulator to the data registers, and therefore uses fewer cycles. The bit selection logic is shown in Figure 3.17.

5.5.2.8 refresh regs

The *refresh regs* instruction opens the I/O ports on the mesh boundary and shifts new data in while at the same time shifting results out, as shown in

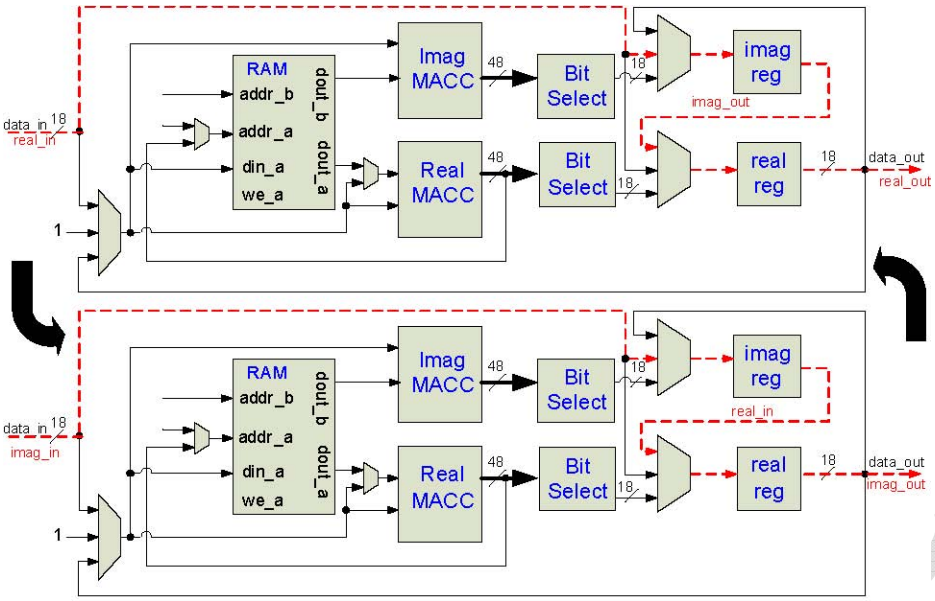

Figure 5-17 The *refresh_regs* instruction

Figure 3.18. This instruction is used when the AXEY has reached convergence and it is ready to accept new measured data from the wavefront sensors.

5.5.2.9 advance regs

The *advance regs* instruction simply enables the real and imaginary registers for one clock cycle, as shown in Figure 3.19. Data are looped back into the imaginary register in case they are needed again. For example, two sequential *advance regs* instruction would return the data registers to their original contents. This is the same concept that preserves the data registers in the *wr ram* instruction.

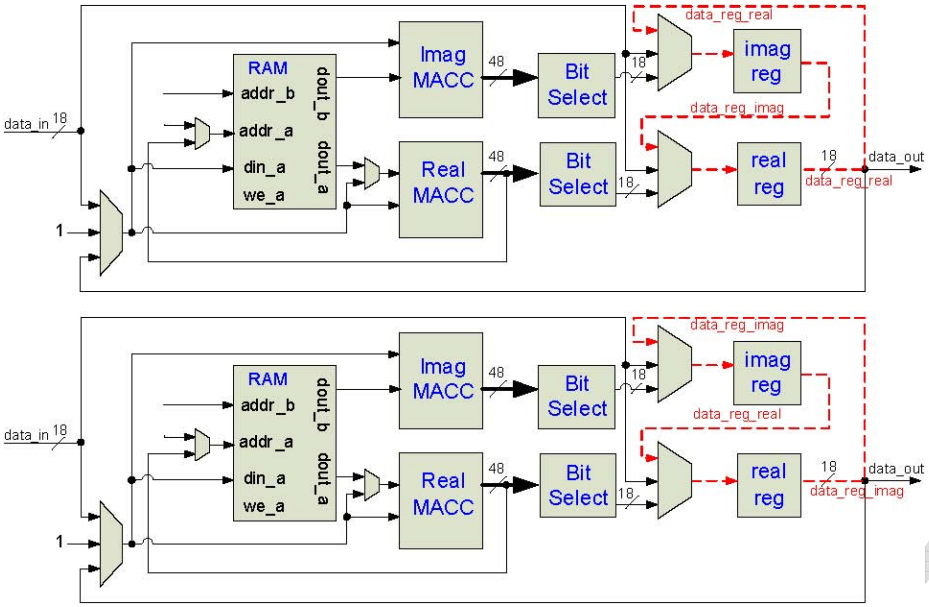

Figure 5-18 The *advance_regs* instruction

5.5.2.10 branch if neg

The *branch if neg* instruction uses a high order bit from the real MACC in the PE located at row zero, column zero, layer zero to conditionally load the CACS program counter with a new index value, as shown in Figure 3.20. If the high order bit is a one, which would indicate that the accumulator contains a negative value, then the CACS counter is loaded. If the bit is a zero, then the accumulator value must be positive so no load is performed.

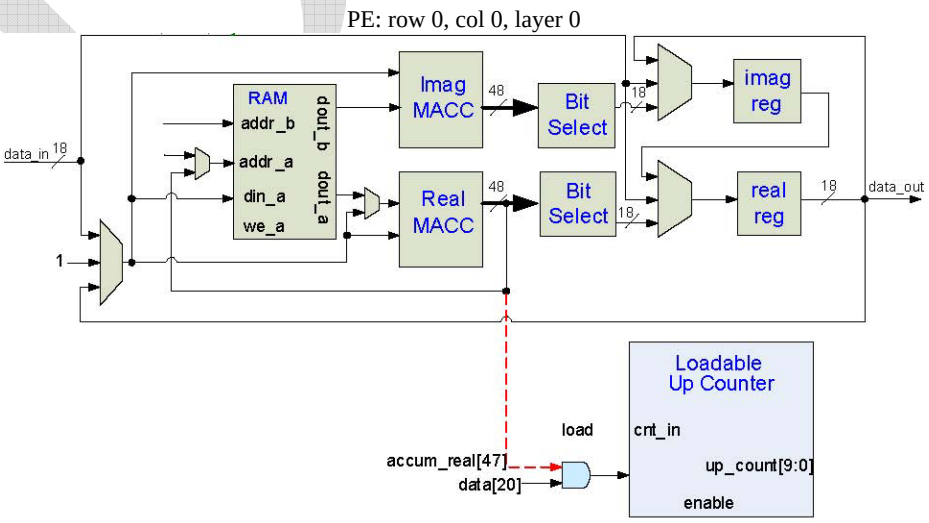

Figure 5-19 The *branch_if_nea* instruction

5.5.2.11 ld ramcnt indirect

Like branch if neg, the ld ramcnt indirect instruction uses information from only the first PE, which is at location row zero, column zero, layer zero. The low order bit data from both the real and imaginary MACCs are used to load the global real and imaginary coefficient up counters, as shown in Figure 3.21.

5.6 Algorithm mapping

The basic loop program is shown in Figure 3.22. In this section, we will explain how the main parts of the program are mapped into instruction sequences. The complete program can be viewed in Appendix A.

1. Forward propagation is performed by a macc layer instruction on shift values for the current guide star.
2. Adjusted forward propagation is done by a *dft* instruction using the inverse dft coefficient set that is stored in RAM.
3. An aperture is taken on the register data by a *macc loopback* instruction that multiplies the data by 1 or 0, depending on if the PE is inside or outside of the aperture.
4. The error is calculated by first writing the adjusted forward-propagated value to a temporary RAM location. The measured value is subtracted from the adjusted forward value.
5. A *dft* instruction using the forward coefficients is taken to bring the data back into the Fourier domain.
6. The error is written to RAM to be used later for back propagation. The error address location is loaded and then offset by the current guide star index. Once the correct location is in the real accumulator, register data are written using *wr ram indirect*.
7. An error magnitude is taken by first squaring and accumulating along the east/west direction with *square rows*. The *add reals ns* instruction is then used to add the real result of each row in the north/south direction. The final error magnitude for the guide star is written to a specific location using *wr ram indirect*.
8. The global error is calculated using *add gstars reals*. This instruction takes the address of guide star error magnitudes and sums them.
9. Back propagation is done by the macc *gstar* instruction, which takes the errors that were written by step 6 and multiply-accumulates them by the C_n^2 of the layer.
10. Adjusted coefficient error is simply a complex multiplication of the Kolmogorov filter value by the coefficient error.
11. A new estimated value is calculated by adding the adjusted coefficient error to the current estimated value and storing the result in RAM.

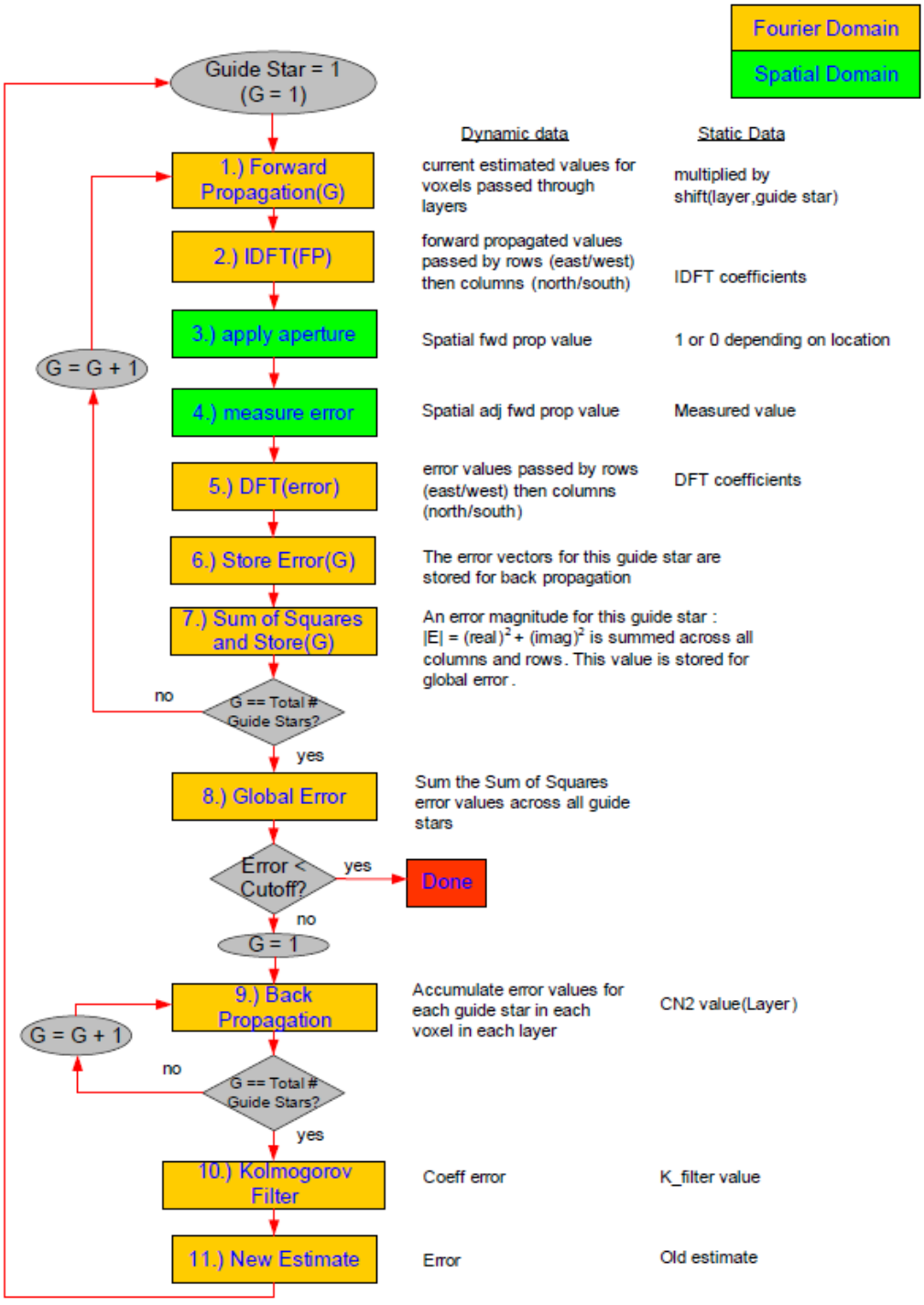

Figure 5-20 The Basic Loop program

5.7 Scalability

The system is intended for use on a variety of AO systems. Certain characteristics of the system such as aperture of the telescope will determine the size and shape of the systolic array. The AXEY architecture therefore needs to have the capability of being generated according to the traits of the target AO system.

5.8 Centralized definitions

The dimensions of the meshed box are specified in definitions .h. All the Ruby scripts and Verilog [15] files reference this central file for dynamically creating module definitions and memory contents. An example of a definitions file is shown below where the 3-D array is specified as 64 columns wide, 64 rows deep, and 8 layers high:

```
‘define COLUMNS 64
‘define ROWS 64
‘define LAYERS 8
‘define CONN_WIDTH 18
‘define RAM_ADDR_BITS 11
‘define RAM_WIDTH 18
‘define SCALE_BITS 10
```

5.9 Verilog architecture generation

For simplicity and usability, Ruby [16] was selected for Verilog generation. The Verilog *generate* statement was not used because it cannot dynamically reference the external memory content files. Each module is explicitly defined with reference to the memory content file that it uses, as shown below for a PE at location column 0, row 3, layer 0:

```
pe_block #( .FILENAME("c0_r3_l0.data") ) pb_c0_r3_l0 (
    .clk(clk),
    .rst(rst),
    .load_acc(load_acc),
    .ce_dsp_real(ce_dsp_real),
    …
);
```

The switch-lattice pattern shown in Figure3.3 is also generated according to dimensions defined in definitions .h. Any system with Ruby installed can run the scripts that generate the top level Verilog files *top.v* and *box.v*. The hierarchy of architecture files is shown below with their brief descriptions.

1. top.v : top level Verilog interface file
    a. box.v : routing and MUXs for switch lattice
        i. muxer4.v : 4 input MUX with 3 inputs for switch lattice
        ii. muxer3.v : 4 input MUX with 4 inputs for switch lattice
        iii. pe block.v : The basic Processing Element
            1. dsp e.v : A Verilog wrapper file for Xilinx DSP48E primitive
            2. bram infer.v : Infers the Xilinx BlockRAM primitive for a PE
            3. bit select.v : A loadable right shifter
    b. cacs.v : The SIMD controller and associated logic
        i. brom infer.v : Infers the Xilinx BlockRAM primitive for program memory

5.10 RAM data structure

The static data that each PE needs are calculated before run time and loaded into each PE's BlockRAM on configuration. A BlockRAM generation script references the definitions file and writes a memory content file for each BlockRAM in the system. Each particular BlockRAM contains DFT and IDFT coefficients, aperture values, filter values, constants, as well as some dynamic data, such as estimated and intermediate error values.

In addition to defining the memories, an address map file shows where data have been placed in all of the memories. The program compiler references this file for address keywords, such as *kolm* for the Kolmogorov filter value or *cn2* for the C_n^2 value, as seen below. The smaller data sets are located at low addresses (0 to 99 in the example below) and the large DFT and IDFT coefficients are written last (100 onward below). It is straightforward to add additional data sets by modifying the BlockRAM generation script.

```
shift 0 4
cn2 812
aperture 16 18
kolm 20 22
afp 24 25
ev 26 27
const1 28 30
inv_dft 100 164
fwd_dft 228 292
```

This functionality hides the memory mapping details from the programmer. For example, a *rd_ram cn2* instruction would be compiled to first load the real coefficient up counter (as shown in Figure 3.5) with an 8, and then load the imaginary coefficient up counter with a 12. The instruction then proceeds, as shown in Figure 3.14 and the accumulators now contain the real and imaginary C_n^2 values.

5.11 Verification

The Verilog simulation tool of the AXEY system is ModelSim SE [17]. ModelSim-SE is an industry-proven RTL simulator that is available in the UCSC Microarchitecture Lab. Xilinx primitives are directly instantiated and simulated in a ModelSim cycle-accurate simulation. Ruby-VPI [18] scripts drive the AXEY test bench together with ModelSim.

5.11.1 Simulation size

The AXEY system that will work on real data sets will require an array of PEs numbering in the thousands. Even the most capable desktop systems with behavioral RTL simulators would take hours if not days just to get through a single iteration of the algorithmic loop. However, because the scale of the array does not affect operational validity, a relatively small array can be quickly simulated and the same results would apply to any larger version.

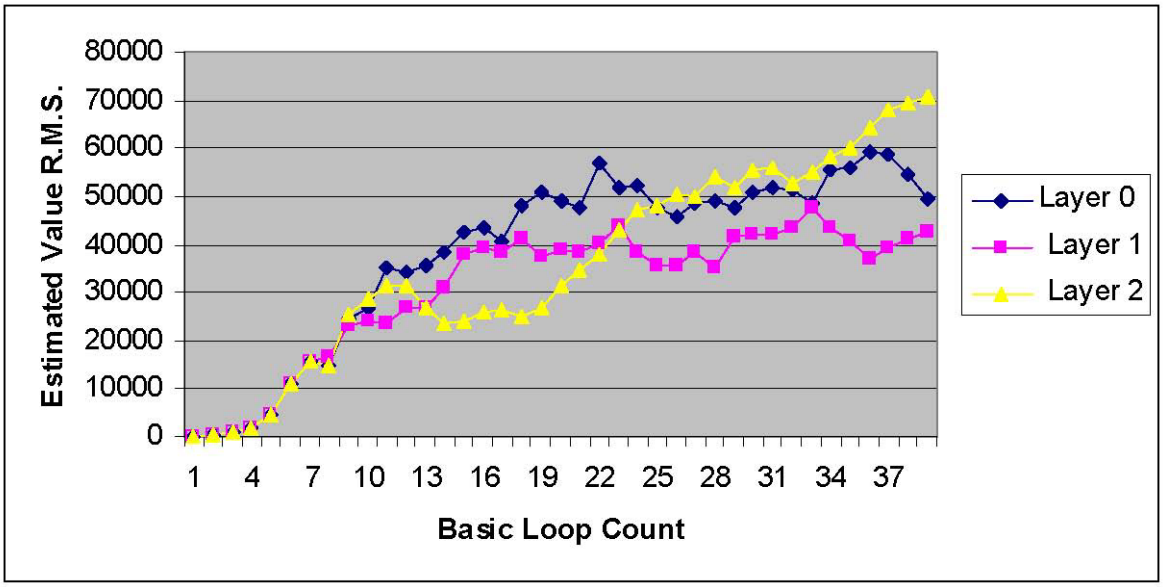

Figure 5-21 Convergence for identical C_n^2 values

5.11.2 Performance

The current version of the AXEY algorithm uses no preconditioning in the loop so the estimated values "converge" slowly. It also begins from a cold start so the estimated values start at zero and ramp up. A small 8x8x3 AXEY array is adequate for verification without consuming unreasonable amounts of memory and CPU time on the simulator desktop computer. Each iteration through the loop takes 1,900 AXEY clock cycles.

Verification of the basic iterative algorithm is performed using a fake data set composed of constant measured values for three guide stars. The set of constant values are the equivalent of there being a perfect atmosphere, and the layer estimate root mean squares should converge to the ratios set by the C_n^2 values of the layer. In Figure 5.1, the three layer RMS values roughly converge to the same level. In Figure 5.2, the Ruby scripts values are set to ratios of 0.6 for layer 0,

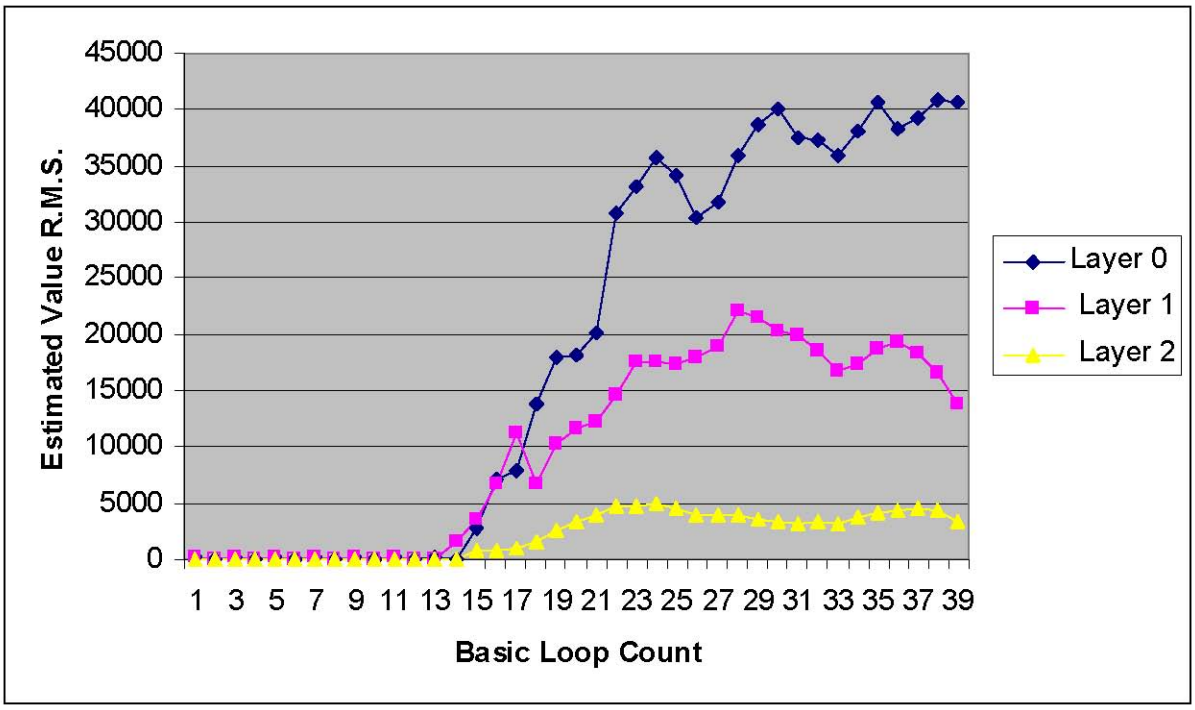

Figure 5-22 Convergence for different C_n^2 values

0.3 for layer 1, and 0.1 for layer 2 of the 65,536 scaling factor. Forty iterations through the basic loop are shown for both graphs.

5.12 Implementation

The Xilinx ISE 9.2 tools were used to synthesize, place, and route the AXEY. The design is fully functional and synthesizable for both theVirtex4 and Virtex 5SX families. To target theVirtex4, the *pe block.v* file can be changed to instantiate *dsp.v* in order to use the DSP48 primitive in the Virtex4, as opposed to *dsp e.v*, which uses the newer DSP48E primitive of the Virtex5.

5.12.1 Synthesis

The building block of the systolic array is the single processing element (PE). As the PE was first defined and later redefined (as instructions were added), the concept of "design for synthesis" was always employed. Every piece of RTL code is logically mapped by the designer to intended FPGA resources. Therefore, the synthesis tool does not have to fill in any blanks when it is inferring what resources to use. Table 6.1 shows utilization numbers in a Virtex5 device for a single PE:

Virtex5 Resource	Single PE Utilization
LUTs	223
Flip Flops	144
BlockRAM (2048x18)	1
DSP48E	2

Table 5-4 Resource Utilization

5.12.2 Place and route

AVirtex5SX95TFPGA was targeted with a4x4x3 array. Timing closure was achieved at 150MHz. The Post-PAR utilization summary is shown below. The first resource to run out is I/O. Future work on the AXEY will have to involve serialization strategies using device fabric and dedicated Serializer-Deserializer (SERDES) primitives.

Device Utilization Summary:
Number of BUFGs 1 out of 32 3%
Number of DSP48Es 96 out of 640 15%
Number of External IOBs 436 out of 640 68%
Number of LOCed IOBs 0 out of 436 0%
Number of RAMB18X2s 49 out of 244 20%
Number of Slice Registers 6981 out of 58880 11%
Number used as Flip Flops 6981
Number used as Latches 0
Number used as LatchThrus 0
Number of Slice LUTS 11663 out of 58880 19%

Table 5-5 Summary of Device Utilization

5.13 Other Issues

5.13.1 Designing for multi-FPGA, multi-board

A complete AXEY system ready to be integrated into an adaptive optics system would contain 64 rows, 64 columns, and 5 layers. As shown in Figure 3.4, if the SX95T FPGA (147PEs) is selected (64 x64 x5 = 20480 PEs), then at least 140 FPGAs are needed. Each FPGA needs to communicate with its neighbors to its immediate north, south, east, and west. The optimal number of PEs on-chip will be a balance of system-wide communication bandwidth, power, and I/O limitations for PCB and cable. On-chip high-speed SERDES components will be used to ensure that total bandwidth rates through the entire array are optimal. The elimination of long wires at row or column boundaries will be done by folding at board edges and/or torus optimal placement techniques.

5.13.2 Unused FPGA Fabric

Because the system employs many powerful FPGAs, the capability of the system can always be improved, even after complete boards are assembled. Currently, most of the logic utilization is in the fixed resources of the Virtex5 (DSP48Es and BlockRAM). Much FPGA fabric is left unused. Potential uses include:

1. The CACS control logic could be duplicated to improve timing toward a higher system clock frequency. Multiple copies of the control logic on-chip would reduce both the critical path length of control lines as well as fanout for those nets. If each chip has control logic, then the control nets do not have to be passed to other chips, which saves I/O pins.
2. For the best visibility, modern telescopes are placed in high altitude locations. These locations are more vulnerable to a single event upset (SEU). An SEU is a change of state caused by a high-energy particle strike to a sensitive node in a micro-electronic device. The ability to detect a problem and rapidly fix it is critical because the operation of the whole telescope is expensive so repeating an observation that lasts many hours is extremely costly. CRC logic could be built to verify validity of static BRAM contents while the system is in operation. The static contents could be reprogrammed with partial reconfiguration techniques while the system is still operating.

5.14 Reliability

The chips will be socketed. This has the potential to reduce reliability, so careful attention must be paid to selection of sockets and regular run time diagnostics.

5.14.1 SEUs

5.15 Maintainability

Rapid diagnostic ability to identify failure location.

The chips will be socketed to allow easy repair in the event of a failure.

- Failure location: row/column

- Failure repair: commodity chip instead of custom or low volume board

Rapid repair, replace a socketed chip.

- Spares requirements: fewer non identical components

Minimize the number of unique components and the average cost of these components to minimize the total cost of spares.

Hardware Description Language (HDL)

Register Transfer Language (RTL)

Graphical Input

Personal preference, natural mode of thought, ease of comprehension …

"How do I know what was really synthesized was what I intended?"

5.16 I/O

I/O and General Operation ???

5.17 Processing Speed and Size

Table 5-6 provides an analysis of the processing speed for a given configuration of the tomography engine.

This shows a single tomography iteration time of <70µsec.

Tomography Engine				
Current Tomography Performance				
29-Jan-08	**Device Used**			**Ver 1.3**
Assumed Parameters	**LX35**	**e Virtex 5**	**Voxels (*)**	**Units**
Layers	8	8	8	
Sodium Layer Height	90	90	90	Km
Atmosphere Height (height of highest layer)	15	15	15	Km
Guide Star Contselation Angle (full width, edge to edge)	2	2	2	Arc Minutes
Max Angle from Zenith	46	46	46	Degrees
Guide Stars (including tip/tilt)	10	10	10	
Sub Apertures in Primary Aperture	64	64	64	
Aperture Size	10	10	10	meters
Extended Sub Apertures in Atmosphere	144	144	64	
Sub Aperture Size	15.6	15.6	15.6	cm
Clock Speed	100	100	100	MHz
Transfer In/Out time	10%	10%	10%	%
Frame Time	1,000	1,000	1,000	µSec
Number of iteration cycles to converge				
Without pre conditioning	100	100	100	Cycles
With pre conditioning	8	8	8	Cycles
Use Pre Conditioning	1	1	1	
Use DCT instead of DFT	0	0	0	
Inter-Chip Bus Size	4	4	4	Bits/Transfer
Inter-Chip Bus Speed (Using DDR and 2x Transfer Clk)	4	4	4	Xfers/Clock
DSP-48's per Chip	192	512	512	
Power Dissipation per Chip	12	16	16	Watts
Chips per board	9	9	9	
Sub Apertures per chip	9	25	25	
MAC's per SUB ap	2	2	2	
Time for Each Elements of the Algorithm				
Basic Loop without aperturing, pre cond. or filtering (Incl. Fwd and	244	244	244	Cycles
Aperturing (2-D FFT-1/FFT)	1,021	1,021	458	Cycles
Pre conditioning (matrix multiply + 2-D FFT-1/FFT)	1,021	1,021	458	Cycles
Filtering (Kolmogorov applied to back propagated error)	10	10	10	Cycles
Scaling for laser guide stars	164	164	84	Cycles
Total Time per Iteration	2,714	2,714	1,508	Cycles
	27.1	**27.1**	**15.1**	µSec
Summary				
Speed				
Cycles to Converge	21,712	21,712	12,064	Cycles
Time-to-Converge	217	217	121	µSec
Max Frame Rate	**4.15**	**4.15**	**7.46**	KHz
Fastest				
Number of chips required (at the above time-to-converge)	**2,401**	**841**	**169**	Chips
Number of Boards	~ 268	~ 95	~ 20	Boards
Total Power Dissipation	< 29	< 14	< 3	KWatts
Total Voxels	208,514	208,514	40,960	Voxels
Total Compute Power	~ 92	~ 86	~ 17	Tera Ops
Power Efficiency	~ 3,190	~ 6,379	~ 6,181	Giga Ops/kW
Least Chips				
Number of chips required (relaxing time-to-converge to <1 mSec)	**~ 580**	**~ 203**	**~ 23**	Chips
Number of Boards	~ 66	~ 24	~ 4	Boards
Total FPGA Power Dissipation	< 7	< 4	< 1	KWatts
Processing Power per Chip	~ 38	~ 102	~ 102	Giga Ops
Total Processing Power	~ 22	~ 21	~ 2	Tera Ops

Table 5-6 Summary of the Tomography Engine's Performance

Item		Equation	Comment

Table 5-7 Formulas Used for Calculating Tomography Performance

Referring to section 3, if we look at the raw computational requirements of each step in our algorithm, we can see that the Fourier and inverse Fourier transforms are the largest single element.

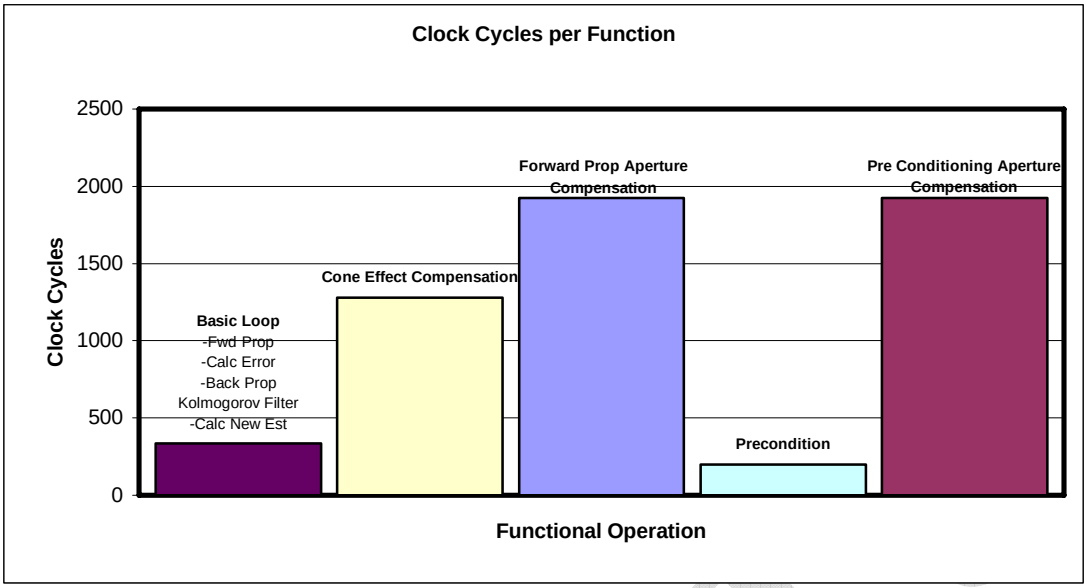

Figure 5-23 Tomography Compute Rates

(See the spread sheet)

Fastest (if it is too slow, we can't go faster)

Smallest (if it is still too large, we can't go smaller)

5.17.1 I/O

5.17.1.1 Primary Data Rates

Wave front sensor data-rates into the system and layer data-rates out of the system are substantial. We assume the following:

1. Real (not complex) data in and out (for simplicity), 2 Bytes per element
2. 64 sub-aperture across the primary
3. Number of segments on the DM's match the number of sub apertures in the primary
4. 1 Msec frame time
5. 10% of a frame time is used for overlapped transfer in/out of data
6. We don't consider the field angle or the angle from the azimuth in the calculations of data rates, since the number of DM commands out will be the same as the number of sub apertures in the primary even if the field is wide (see 3. above). These do effect the solution time and the size of the Tomography Engine, however.

Aggregate raw input data rates are ~8 MB/Sec per WFS or DM. However, we only have part of a frame to read in new WFS data and write out the new layer data.

For the above assumptions, the input rates that the system needs to handle during transfer are 82 MB per second per layer: 410 MB per second total. While this is substantial, the system can handle this transfer rate.

Sub apertures across primary	64	64	100	100
Number of WFSs				
Number of Layers	5	8	5	8
Number of MCAO DM's	5	8	5	8
Number of observing DMs (only relevant for MOAO)	10	20	20	20
Word size (Bytes)	2			
Portion of a frame used for transfer time (% frame)				
(Transfers of WFS data-in and DM data-out are overlapped)	10%			
Frame time (µSec)	1,000			
MCAO				
Input and output data rates (MB/Sec)				
Per WFS	82	82	200	200
Total	410	655	1,000	1,600
MOAO (MB/Sec) (Same input data rates as MCAO)				
Data rate per sensor	82	82	200	200
Total data rate	819	1,638	4,000	4,000
Diagnostic data (MB/Sec)				
Per WFS or Layer	8	8	20	20
Total stored data rate	82	131	200	320
Mbits/Sec	655	1049	1600	2560

Example I/O statistics for various configurations

MCAO

The data rates for layer information are similar. The number of sub apertures in the higher layers, in our estimated atmosphere, increases with field of view and angle of view from the zenith. However, it is reasonable to assume that the DM's used will have a number of segments approximately equal to the number of sub apertures in the WFSs. Thus, the commands to the DM's will be fewer than the number of sub apertures in the higher layers and the data-out rate to the DMs will be approximately the same as the data-in rate from the WFSs.

MOAO

…

Interconnect

Communications with elements of the WFSs or DMs are through standard LVDS interfaces.

For diagnostic storage of the primary data, a medium performance RAID cluster is used which supports Fiber Channel and uses SATA or other high-speed disks.

5.17.1.2 Diagnostic Data Rates

These data rates are very low compared to the primary data rates since we have an entire frame to move the data rather than a small portion of a frame. In addition, it uses a

smaller separate bus and does not compete with bandwidth used by the primary data and/or the computational data interconnect.

5.17.2 Diagnostic Computations

A variety of diagnostic computations and monitoring are supported by the system.

5.17.2.1 C_n^2 profile extraction

The C_n^2 profile can be estimated during operation using the information about the estimated layers.

5.17.2.2 Wind extraction (by layer)

Wind in each layer can be estimated. 19 20

5.17.2.3 Convergence rate Reporting

5.17.2.4 Layer height extraction

5.17.3 Capturing Diagnostic Data Streams

In addition to capturing diagnostic data, it is important to be able to capture WFS and layer data streams for analysis. This means streaming the input WFS and output layer data to a disk sub system while the tomography engine is operating, *without affecting its operation.*

With a bus-based system, the basic I/O, the diagnostic data streams, and the data access requirements of the tomographic calculations all compete for a common bus bandwidth.

A worst-case scenario might be to try to capture all data in and out of the system on a frame basis i.e., each millisecond for 8 hours. The data rate for this would be less than 320 MB per second (2.6 Gbits per second) and an 8-hour session, capturing all data for every frame, would require <10 Tera Bytes of storage. While this level of capture is unlikely, it could be supported with off-the-shelf, fiber channel, storage sub systems such as Dell's CX series, which can handle 4 Gbit data rates and ____ of total storage.

For the LAO tomography engine, the ability to capture and store data is limited by our willingness to supply an existing, off-the-shelf, storage sub-system. It is not limited by having to use exotic technology or by balancing data storage activities against system performance.

INSERT REPLACEMENT FOR THIS

5.17.4 System Control

Control for a frame is updated in parallel with the loading and unloading of the data. This is a very low bandwidth operation and has no impact on I/O or operation. Control is performed by a traditional PC using traditional methods and communications.

For the most part, inter-chip data is handled by LVDS signaling. Each signal has a possible BW of 600 Mbits per second (~75 MB/Sec)

I/O is divided into primary, system configuration and diagnostic buses

Figure 5-24 I/O bus flow

5.17.5 Primary Data Flow

5.17.6 System Configuration

5.17.7 Diagnostic Data Flow

We are now primarily limited by our willingness to save data, not by a limitation of the system to support the saving of the data or the impact that the action might have on the system function and performance.

PSF, C_n^2, RMS error, wind, convergence rate,

Illustrate extracting data while running, using the load of a SR.

Flags are set at the beginning of each frame to tell the main control to load the shift registers at the appropriate time. This changes the iteration or convergence time by only a few cycles (out of thousands), depending on when data is loaded.

Once loaded and started, control for dumping diagnostic data is handled by a completely separate state machine.

Inter chip bottleneck

5.18 Global Synchronization and Clocks

LVDS?

The ability to keep things synchronized may set the maximum clock frequency between cells, chips and boards.

Clock skew

Data delay

Data wrap delay

We may use a fast path or add a fast pipeline buffer and take an extra clock in a long series of clocks rather than slow all clocks down.

5.18.1 Integer vs. Floating Point

The algorithm we use for converging to a new estimated value is stable in the presence of noise. The effect of round off errors due to using integers instead of floating point is the equivalent of adding noise. This lengthens the convergence time, but the algorithm will still converge.

Range

Scaled to 2^n

Minimizing loss of precision

Division, shifting by ‘n’

5.18.2 Data Word Size

10 bits in from WFS, 10 bits out to DMs, 16 bits internal to Axey on 18-bit MACs with 36-bit results

Add Don’s analysis

How do we analytically determine the effect of the truncation to determine word size?

5.19 System Control and User Interface

Figure 5-25 Overall System Diagram

Show Tomography engine and control PC, diagnostic storage, DMs, Cameras …

While this is an incredibly powerful and flexible computing machine, its power is at the cost of versatility and it is not designed to handle human interface or general-purpose computational tasks. Instead, these very low bandwidth and low compute requirement tasks will be handled by a traditional PC class computer, which will be programmed by traditional methods.

This control computer stores and controls:

- Overall system architecture configuration files
- Primary topographic algorithm code
- Diagnostic code
- Frame-to-frame control to determine what diagnostic processes to perform
- Frame-to-frame control to determine what diagnostic and basic data to store

Communications between the control computer and Axey are through standard LVDS signaling.

5.20 Ease of Use

- Programming: simple state machine descriptions

Show code

Show state diagram

- Hardware upgrades by code

- System control and user interface

Diagnostic error detection and location (row, column) using spare cycles

5.20.1 Performance

Limiting factors

I/O during transform Pins

Power

Multiplier

Inter board communication and synchronization in general

The LX55 is slower because of the reduced size of the inter chip busing, but it takes less power and fewer boards because it uses fewer chips.

We limit-out on power/speed and bus bandwidth using a larger chip. We limit-out on resources on the smaller chip.

6 Summary

The initial analysis, simulation and implementation show that the architecture described here can implement the functions required for the real time computation of atmospheric tomography and control of the deformable mirrors for a large astronomical AO system can be accomplished within the specifications required.

Further work will refine the system and present trade-offs in terms of speed, over sampling, accuracy, power, etc.

Additional work will examine the suitability of the system for use with alternate algorithms.

Appendices

Appendix A Program: The Basic Loop

```
# forward propagate
beg: rd_ram ev
noshift_store

# point the real/imag cnt indexes to shift for
# current guide star
rd_ram shift_addr
ld_ramcnt_indirect
macc_layer shift
rtshift_store scale_bits

# IDFT
dft_ew inv_dft_ew
rtshift_store scale_bits
dft_ns inv_dft_ns

# in addition to scaling down, also normalize down here
rtshift_store normalize_bits

# take aperture in spatial
macc_loopback aperture
noshift_store

# take error in spatial
wr_ram temp
rd_ram meas_addr
ld_ramcnt_indirect
rd_ram_direct
sub temp
noshift_store

# move spatial error to Fourier with DFT
dft_ew fwd_dft_ew
rtshift_store scale_bits
dft_ns fwd_dft_ns
rtshift_store scale_bits
wr_ram error_temp

# write error for this guidestar to 4 memory locations
# back propagation needs it in this format
#
# procedure: write Real to locations 1 then 2

# write Imag to locations 3 then 4
# 1.) error_addr : R
# 4.) error_addr+1 : I
#.
#.
# 3.) error_addr + 2x#gstars : I
# 2.) error_addr + 2x#gstars + 1 : R
rd_ram error_addr

# writes to 1.
wr_ram_indirect
add gstar_num
add gstar_num
add unscaled_const1

# writes to 2.
wr_ram_indirect
advance_regs
sub unscaled_const1

# writes to 3.
wr_ram_indirect
sub gstar_num
sub gstar_num
add unscaled_const1
```

```
# writes to 4.
53
wr_ram_indirect

# put the error_addr at location for next gstar (in case we
# come back here)
add unscaled_const1
noshift_store
wr_ram error_addr

# get an error magnitude for this guide star
rd_ram error_temp
noshift_store

# square_rows accumulates all R^2 + I^2 values in each row in
# the real accum
square_rows

# scale down the sum of square rows, this incurs some roundoff errors*
rtshift_store scale_bits

# add_reals_ns accumulates the real values north/south
# in a real path only, bypassing the imag paths
add_reals_ns unscaled_const1
noshift_store

# writes to proper sum of squares location: (sos_addr +
# gstar_cnt)
rd_ram sos_addr
add gstar_cnt
wr_ram_indirect

# increase meas address index
rd_ram meas_addr
add dual_twos
noshift_store
wr_ram meas_addr

# increase layer shift address index for next guide star
rd_ram shift_addr

add layer2_num
noshift_store
wr_ram shift_addr

# done with a gstar loop, now increase gstar_cnt index
rd_ram gstar_cnt
add unscaled_const1
noshift_store
wr_ram gstar_cnt

sub gstar_num

# if we haven't gotten an error for each guide star,
# branch to the beginning
branch_if_neg beg

# determine if global error is small enough to exit program
# sum the sum of squares values across all guide stars
add_gstar_reals sos
sub cutoff

# here is the bailout condition
branch_if_neg stop

# Back Propagation
rd_ram cn2
noshift_store
macc_gstar error
rtshift_store scale_bits
macc_loopback unscaled_const1
```

```
# Kolm filter
macc_loopback kolm

# new estimated value
add ev
noshift_store
wr_ram ev

# we are at end of loop so reset address indexes for:
# gstar_cnt,
rd_ram const_zero
noshift_store
wr_ram gstar_cnt

# meas_addr,
rd_ram meas_addr_start
noshift_store
wr_ram meas_addr

# shift_addr,
rd_ram shift_addr_start
noshift_store
wr_ram shift_addr

# error_addr,
rd_ram error_addr_start
noshift_store
wr_ram error_addr

# now force a branch to the very beginning of loop
rd_ram neg_const
branch_if_neg beg

stop: done
```

Appendix B Framework Requirements

B.1 GenerateVerilogfiles,BlockRAMcontents, and program compilation

Ruby

NArray library for Ruby

B.2 B.2 Behavioral RTL simulation

Linux OS

Ruby

NArray library for Ruby

Ruby-VPI 16.0.0 or higher that works with ModelSim

Modelsim SE Simulator

Xilinx primitive libraries for Modelsim SE

Appendix C Ruby Scripts

C.1 Ruby scripts for Verilog Generation

1. gen top.rb : generates the top level Verilog file: “top.v”
2. gen box.rb : generates the 3-D systolic box Verilog file: “box.v”
3. gen all.rb : callsall architecturegenerationscripts,oneat atime(“gen top.rb”, “gen box.rb”, “gen all.rb”)

C.2 Ruby scripts for systolic array BlockRAM content generation

gen mesh brams.rb :

uses functions located in “gen bram functions.rb”

generates the BlockRAM content files: ”c# r# l#.data”

generates the address file: “addr.data”

C.3 Ruby scripts for CACS program compilation

compile.rb : compiles any sequence list text file into “whole sequence.seq”

gen cacs rom.rb : compiles whole sequence.seq into the final CACS rom file : “control rom.data”.

References

[1] D. Gavel. Tomography for multiconjugate adaptive optics systems using laser guide stars. SPIE Astronomical Telescopes and Instrumentation, 2004.

[11] M. Reinig. The LAO Systolic Array Tomography Engine. 2007.

---

[1] TMT document reference

[2] Luc's paper

[3] The LAO Systolic Array Tomography Engine, Matthew D. Fischler, UCSC School of Engineering, November 26, 2007.

[4] W. M. Keck Observatory. The Next Generation Adaptive Optics System: Design and Development Proposal. 2006.

[5] KAON 463: Lessons learned for Keck LGS Operations Weather Impact, Efficiency & Science Operations Model", D. Le Mignant, 12 Feb. 2007

[6] Radon transform See [**Error! Bookmark not defined.**]

[7] "General Purpose Systolic Arrays", Johnson, Hursom, Shirazi, 1993, Computer, IEEE, (p24)

[8] H. T. Kung and C. E. Leiserson. Systolic Arrays (forVLSI). Proceedings of Sparse Matrix Symposiums, pages 256–282, 1978.

[9] R. Hughey and D.P. Lopresti. Architecture of a programmable systolic array. Systolic Arrays, Proceedings of the International Conference on, pages 41–49, 1988.

[10] M. Gokhale. Splash: A Reconfigurable Linear Logic Array. Parallel Processing, International Conference on, pages 1526–1531, 1990.

[11] K.T. Johnson, A.R. Hurson, and Shirazi. General-purpose systolic arrays. Computer, 26(11):20–31, 1993.

[12] A. Krikelis and R. M. Lea. Architectural constructs for cost-effective parallel computers. pages 287–300, 1989.

[13] H.T. Kung. Why Systolic Architectures? IEEE Computer, 15(1):37–46, 1982.

[14] Xilinx, Inc. Virtex-5 Xtreme DSP Design Considerations User Guide,7 January 2007.

[15] IEEE Computer Society. IEEE Standard for Verilog Hardware Description Language. Std. 1364-2005, IEEE, 7 April 2006.

[16] D. Thomas, C. Fowler, and A. Hunt. Programming Ruby: The Pragmatic Programmer's Guide. Pragmatic Bookshelf, Raleigh, North Carolina, 2005.

[17] Mentor Graphics Corporation. ModelSim SE User's Manual, software version 6.3c edition, 2007.

[18] Kurapati. Specification-driven functional verification with Verilog PLI & VPI and System Verilog DPI. Master's thesis, UCSC, 23 April 2007.

[19] LNL

[20] Luke's work