DARPA Order Number DAR6D997/01
Program Code Number HJ1500-6225-0885
Contract Number DABT-96-C-0059

CLIN 0002
CDRL A001

Northrop Grumman Reference G. O. 50086

Micro-Accelerators for Sensor Data Processing
Final Technical Report

October 1998

Principal Investigators:

Michael R. Lucas
410-765-5037
michael_r_lucas@mail.northgrum.com

Richard F. Webb
410-765-9487
richard_f_webb@mail.northgrum.com

Prepared by:
Northrop Grumman Electronic Sensors and Systems Sector
PO Box 746/MS 499
Baltimore, MD 21203

Prepared for:
Defense Advanced Research Projects Agency
Information Technology Office
3701 N. Fairfax Drive
Arlington, VA 22203-1714

DARPA Order Number DAR6D997/01
Program Code Number HJ1500-6225-0885
Contract Number DABT-96-C-0059

CLIN 0002
CDRL A001

Northrop Grumman Reference G. O. 50086

Micro-Accelerators for Sensor Data Processing
Final Technical Report

October 1998

Principal Investigators:
Michael R. Lucas
410-765-5037
michael_r_lucas@mail.northgrum.com

Richard F. Webb
410-765-9487
richard_f_webb@mail.northgrum.com

Prepared by:
Northrop Grumman Electronic Sensors and Systems Sector
PO Box 746/MS 499
Baltimore, MD 21203

Prepared for:
Defense Advanced Research Projects Agency
Information Technology Office
3701 N. Fairfax Drive
Arlington, VA 22203-1714

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Oct 1998	Final Technical Report 23 Sept 96-23 Sept 98

4. TITLE AND SUBTITLE
Micro-Accelerators for Sensor Data Processing

5. FUNDING NUMBERS
DABT-96-C-0059

6. AUTHOR(S)
Michael R. Lucas, Richard F. Webb, Steven S. Gercken

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northrop Grumman Electronic Sensors and Systems Sector
Box 746/ MS 499
Baltimore, MD 21203

8. PERFORMING ORGANIZATION REPORT NUMBER
NA

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency, ITO
3701 North Fairfax Drive, Arlington VA 22203-1714 (Sponsor)
Directorate of Contracting, ATZS-DKO-I
PO Box 12748, Fort Huachuca, AZ 85670-2748 (Monitor)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NA

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILTY STATEMENT
UL

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A method for accelerating COTS processor performance using processing nodes implemented with advanced reconfigurable integrated circuit devices has been developed. This technique uses emerging Field Programmable Gate Array technologies configured as special purpose processing nodes, integrated into a COTS architecture, to perform the high throughput functions in military sensor processing. Data formats are tailored to provide maximum performance at minimum size, weight, power and cost. Simulations of the approach have been demonstrated using the high computation throughput functions within radar Space Time Adaptive Processing (STAP) algorithms.

14. SUBJECT TERMS
Accelerator, Radar, Signal Processing, Reconfigurable Integrated Circuit, Field, Programmable Gate Array, Space-Time Adaptive Processing

15. NUMBER OF PAGES
99

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Table of Contents

List of Figures

List of Tables

1. SUMMARY

A method for accelerating COTS processor performance using processing nodes implemented with advanced reconfigurable integrated circuit devices has been developed. This technique uses emerging FPGA technologies configured as special purpose processing nodes, integrated into a COTS architecture, to perform the high throughput functions in military sensor processing. Data formats are tailored to provide maximum performance at minimum size, weight, power and cost. Simulations of the approach have been demonstrated using the high computation throughput functions within radar Space Time Adaptive Processing (STAP) algorithms.

1.1 Approach

Requirements for embedded high performance processors in sensor systems performing air to air surveillance, synthetic aperture radar (SAR), ground moving target indicator (GMTI), and automatic target recognition (ATR) are increasing to hundreds or thousands of giga-floating point operations per second (GFLOPS). This increase is due to operational requirements for increased area coverage, multiple onboard sensors, more detailed image understanding, and digital beam forming with Space Time Adaptive Processing. Heterogeneous processing architectures containing a mix of programmable DSPs, microprocessors, and custom hardware preprocessors or accelerators have been proven on a number of sensor systems to offer benefits in overall processor performance, cost, size, weight, and power.

The COTS community has made great strides in developing heterogeneous processor products with DSP and microprocessor nodes, and a number of systems now flying contain COTS processors. However, the addition of preprocessors and accelerators to these COTS systems has not been well supported. An approach is required where low cost preprocessor and accelerator nodes that are configured to perform the very high throughput portions of the algorithm can be efficiently integrated into COTS processing systems from both a hardware and software perspective. Portability of the sensor algorithm over multiple generations of hardware evolution is also a requirement due to the long life of sensor systems relative to hardware generations.

Northrop Grumman has explored under this contract the use of the emerging generation of fast, large reconfigurable devices, such as Field Programmable Gate Arrays (FPGAs), to implement such a reconfigurable preprocessor or accelerator node. This approach has the potential for very high computational throughput for certain algorithms due to elimination of the large control overheads required in a more general purpose device, as well as efficiencies due to the tailoring of the data precisions used to permit the maximum packing of arithmetic functions in the device. The reconfigurable node is integrated within the communication structure of a COTS processor and functions as an alternate processing resource on which to host the targeted portions of the sensor algorithm. The hardware descriptions of the configurations of the devices were developed in VHDL to allow transportability across multiple, increasingly dense generations of reconfigurable devices.

1.2 System Requirements

Military systems containing sensors such as radar and Forward Looking Infra-Red (FLIR) have the conflicting requirements of high performance in small volume/weight/power at low cost. The most demanding functions performed by these sensors include SAR mapping, GMTI, STAP for airborne early warning, and ATR. Processing throughputs of 1 to 10 GFLOPS are required in systems currently being fielded, with throughputs of 10 to 100 GFLOPS required for emerging applications. These functions are typically performed in physically-constrained platforms which are airborne (UAVs, fighter aircraft, helicopters) or ground-mobile (wheeled vehicles, tanks), where volume, power and weight is at a premium. Additionally, since many of these platforms are produced in quantity (from hundreds to thousands), low recurring unit cost is a primary driver.

A key characteristic of many of these sensor processing requirements is that a relatively small family of mathematical computations typically drives the total system processing throughput. For example, the Q/R decomposition function, consisting primarily of matrix operations (complex multiplies) drives the STAP algorithm. Either FFT or FIR filtering drives SAR, depending on the implementation used. Similarly, GMTI is driven by the pulse compression FFTs, and ATR throughput is dominated by mean and energy filtering to perform detection. These core driving functions are usually well structured and relatively simple to code and can generally be configured so that a high level of parallelism can be achieved. Approaches which reduce the size, weight, power and cost of performing the driving computations can significantly impact the overall processing system.

1.3 Precision Analysis Studies

In typical military sensor data processing systems containing commercial microprocessors and DSPs, all computations are normally done in IEEE 754 standard single precision 32 bit floating point format, as this is the data type for which these commercial devices have been optimized. Some devices have alternate data format capability, such as 16 bit fixed point, which consequently have higher processing throughput and have been used for data preprocessing functions in radar and E/O systems. However, a rigorous analysis is not usually performed as to the specific precision required for various functions of the algorithm, as there typically is no way to exploit the use of alternate data types.

The selection of the precision used for any application depends on the system performance requirements for all environments expected to be encountered. Preliminary studies performed at Northrop Grumman indicate the potential for using reduced precision formats in certain parts of sensor processing algorithms with minimal effect on system end performance. In this project, the impact of varying the precision used in the Q/R decomposition computations used to compute adaptive beam forming weights as part of a Space Time Adaptive Processing (STAP) algorithm was studied. For example, using real life collected radar data, double precision floating point arithmetic was used to generate weights which were then applied to form adaptive radar beams to eliminate main beam clutter. The use of floating point arithmetic using reduced precision (e.g.,

eight bit mantissa) to perform the same weight computations provided beam pattern results nearly identical to the double precision floating point computation. Use of even lower precision arithmetic (e.g., 4 bit mantissa) was shown to still provide very good results.

1.4 Reconfigurable Device Characteristics

A very rapid growth rate in reconfigurable device capabilities is currently underway. Currently available devices are in the 100,000 to 200,000 equivalent gate ranges. Multiple vendors are developing next generation devices which will double these equivalent gate densities using leading edge integrated circuit technologies and feature sizes, and devices with over a million equivalent gates are projected for the year 2000 timeframe. Northrop Grumman has investigated the capabilities of the latest generations of devices to provide highly parallel, very high throughput, reduced precision computational elements for use in the reconfigurable accelerator node. Multi-GFLOPS throughputs have been demonstrated to be obtainable in a single device, when using the precision ranges in the STAP application example discussed in the previous section.

1.5 Micro-Accelerator Architecture

A reconfigurable accelerator node was developed on the project, consisting of a processing engine configured from multiple reconfigurable devices, high speed local memory, and a network interface to the COTS processing system's communication network. This approach allows the reconfigurable node to follow the same architecture and programming paradigm as DSP or microprocessor nodes (e.g., SHARC and PowerPC) currently used in a number of COTS processors.

The interface for the accelerator which was developed uses the Raceway interface, as it is an open standard with reasonably high bandwidth, and it can be rapidly inserted into a number of DoD systems currently using the Raceway standard. This interface can be easily extended to target PCI, Myrinet, or other open-standard communication systems. The reconfigurable node conforms to a standard daughter card format size and I/O developed by Mercury computer and is intended to be inserted into COTS processing systems using the Raceway interface on 6U or 9U VME processing boards. A detailed daughter card design was performed, and a printed circuit daughter card board was fabricated on the project. Northrop Grumman is pursuing alternate funds to demonstrate the capability of this micro-accelerator within a Raceway-based processing system.

1.6 System Benefits

The benefits of the micro-accelerator approach have been evaluated versus competing conventional approaches. The hardware configuration required for a COTS processor using DSP

and microprocessor processing nodes to perform a representative STAP algorithm within a typical latency requirement was developed from data derived from the Mountain Top program. This algorithm uses a high percentage of the total throughput for performing the Q/R decomposition function. An alternate configuration that replaces some of the DSP nodes with accelerator nodes to perform the Q/R decomposition was sized. Order of magnitude improvements in size, weight, and power are projected. Northrop Grumman is currently evaluating military sensor systems for insertion of the approach.

1.7 Report Organization

This report is organized into six sections, including this summary. Section 2 provides the system requirements rationale for the need for the micro-accelerator technology in sensor systems, and describes the computation precision studies performed which provide the basis for the computing engine design. Section 3 describes the study of the candidate reconfigurable devices and the simulation results of the required micro-accelerator functions on these devices. Section 4 describes the micro-accelerator architecture and system level simulations conducted to demonstrate the performance projections. Section 5 describes the detailed design of the micro-accelerator. Section 6 provides conclusions and thoughts on future research areas.

The Appendix provides VHDL code listings of the high performance 16 bit floating point multiplier and adder/subtractor developed for use in QR factorization.

2. SYSTEM ANALYSIS

A system analysis of the processing needs of emerging sensor systems was performed to define the requirements for the micro-accelerator. This systems analysis consisted of two parts: a projection of the throughput requirements and platform constraints of next generation platforms with radar and EO sensors, and a detailed study of the precision requirements of the driving functions of the selected application regime, Space-Time Adaptive Processing.

2.1 Emerging Sensor System Processing Requirements

The technical rational for development of the micro-accelerator is derived from the demanding processing requirements of advanced and emerging military sensor systems. These requirements are in three main areas: physical constraints, performance requirements, and cost.

Processor physical constraints are a function of the sensor platform and remain relatively constant over the life of the system. The primary constraints of interest are volume, weight, and power. The sensor platforms of interest range all the way from very small and light with very low power availability, such as is the case with submunitions with active seekers, all the way to ground based vehicles, where the processing is allocated a portion of a large van or truck. Northrop Grumman provides numerous sensors for a wide range of platforms, and representative constraint data from these systems is shown in Table 2-1. This table shows various platform classes with representative systems by name, and their sensor functions. Typical ranges of size/weight/power allocations for these embedded sensor processors are provided.

Table 2-1. Representative Sensor Platforms and Processor Physical Constraints.

Platform	Representative Systems	Functions	Volume (cu ft)	Weight (lbs)	Power (watts)
Brilliant Submunitions	IBAT	MMW Radar, IR Seeker	0.03- 0.3	10-40	<20
Combat Ground Vehicles	M-1 Target Finder	Search Radar	0.5	40	150
UAVs	Tier 2, Tier 2+, Tier 3 Minus	SAR, E/O Surveillance	1-1.5	50-100	500-1000
Attack Helicopter	Apache Longbow, Comanche	E/O & TV ATR, MMW Radar	1-1.5	50-100	1000-1500
Multirole Fighter	F-16, F-22, JSF	Search/Track Radar, SAR, EO Search, ESM	1-3	50-200	1500-2500
Penetration Bomber	B-1, B-2	TF/TA, SAR, ESM	1-3	50-200	1500-2500
Airborne Wide Area Surveillance	E-3 AWACS, E-8C JSTARS	Search/Track Radar, SAR, GMTI	10-40	500-1000	10,000
Ground Stations	ETRAC, SAIP	SAR, E/O ATR	10-50	50-100	10,000-20,000

Although the physical constraints for a sensor processor may remain constant for the life of the platform, the performance requirements increase over the system lifetime due to the need for additional operational capability as new threats and requirements emerge. Two major classes of sensor missions were identified as having desired future processing requirements which are beyond the limits of current embedded processing capabilities; these are air-to-ground surveillance, with typical functions such as SAR mapping, ATR, and GMTI, and air-to-air surveillance. Table 2-2 illustrates the requirements for a number of types of SAR mode functions, in which image maps are created from radar data. Some of these modes also include performing automatic target recognition on the imagery and/or performing GMTI either interleaved with SAR or concurrently from the same radar I/Q data. The processing requirements have been summarized in terms of operations required for each pixel generated in the SAR map.

Table 2-2. SAR and ATR Mode Processing Requirements.

Mode	Description	Operations per Pixel
SAR Search	"Standard" SAR strip mode	2100
Bi-Static SAR Search	SAR strip with bi-static illuminator (required for covert platforms)	2600
Multi-Phase Center SAR Search	Required when high resolution required at long ranges	6300
Interferometric SAR	Measures Height/Topography of Terrain	7200
UHF/Foilage Penetration SAR Search	SAR at UHF to penetrate Foilage. Huge coherent integration times drive memory requirement	2300
Interleaved SAR/GMTI	SAR search with interleaved dwells of GMTI for simultaneous display of target types	2100
Simultaneous SAR/GMTI	SAR and GMTI display formats generated from the same I/Q data	12,000
On Board ATR	ATR running on board, cued/recognized targets provided to operator	3,000-15,000

To determine the processing requirements rate for the above functions, the pixel generation rate must be factored into the above computation figures. The pixel generation rate depends on the map image resolution (a finer resolution increases the number of pixels by the inverse square of the resolution ratios) and the coverage rate, in terms of area (e.g., square miles) per time. The coverage rate is determined by the swath width (minimum to maximum surveillance distance out from the platform) and the platform speed. Table 2-3 illustrates representative pixel rates and corresponding processing throughput rates for a selected subset of the above radar modes and various coverage rates and resolutions. As the coverage rates and resolutions increase, the processing rate go up to the TeraFLOPS region. There is also an operational desire to increase resolution to 0.1 meter to improve ATR capability for some future applications, which will further drive the processing rates.

Table 2-3. Processing Requirements for Air to Ground Surveillance.

Mode/Capability	Mpixels/sec	GFLOPS
10km Swath x 120 m/sec @ 1m		
- SAR only	1.9	4.0
- SAR + SimGMTI + ATR	1.9	23
- SAR + SimGMTI + ATR	1.9	49
10km Swath x 120 m/sec @ .6m		
- SAR only	5	10.5
- SAR + SimGMTI	5	60
- SAR + SimGMTI + ATR	5	130
10km Swath x 120 m/sec @ .3m		
- MPC SAR only	21	44
- SAR + SimGMTI	21	252
- SAR + SimGMTI + ATR	21	546
10km Swath x 240 m/sec @ .3m		
- MPC SAR only	42	88
- MPC SAR + SimGMTI	42	504
- MPC SAR + SimGMTI + ATR	42	1092

Air to air surveillance also has continuously increasing requirements. These functions are performed by large surveillance platforms such as the E-3 AWACS or the E-2 Hawkeye, on UAVs in future applications, and also as part of the radar mode suite in tactical fighters such as F-22. Table 2-4 illustrates the processing requirements drivers and requirements for air to air surveillance. Various radar system parameters are driving a multiplicative increase in processing requirements. A key driving requirement for air to air surveillance systems is the need to see smaller and smaller radar cross section targets in heavy clutter and in jamming environments. A technique for providing this performance is Space-Time Adaptive Processing (STAP), in which radar beams are formed adaptively to cancel out clutter and jamming. Several DoD programs are evaluating and demonstrating this technique for potential application to the airborne surveillance platforms.

The table illustrates that the total increase in processing requirements is growing at a faster rate than which can be provided by Moore's law (Moore's law says that evolutionary IC device improvements will provide a doubling of computing capability about every 18 months.) The processing requirement for these systems in nine years will have grown by about one thousand, and Moore's law can be expected to provide an improvement of only about 64. Techniques providing an additional ten to twenty-fold improvement in processing are required.

Table 2-4. Processing Requirements Projections for Air to Air Surveillance.

Radar System Parameter	Multiplicative Increase in Radar System Requirements*			Processing Impact	Processing Scaling	Multiplicative Increase in Processing Requirements*		
	Year 3	Year 6	Year 9			Year 3	Year 6	Year 9
Range gate rate	2	3	4	Search ("all-range-gate") processing New CFAR and New ECM algorithms	Linear Nonlinear	2 2	3 4	4 8
Doppler resolution	2	4	8	Search processing New CFAR	(log Mn)/log n Nonlinear	1.2 2	1.3 3	1.5 4
Antenna/receiver channels	1	1.5	2	Adaptive cancellation	Linear	1	1.5	2
Target density	1	2	3	Affects Post-CFAR Processing	Linear	1	2	3
Total Required Processing Throughput Increase						9.6	140	1152

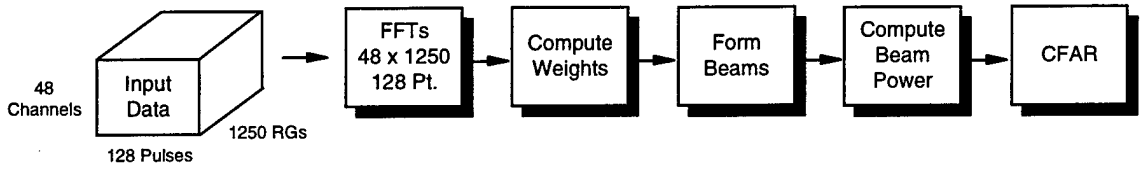
* Requirements Relative to Year 0 (Today)

A more detailed analysis of these processing requirements reveals that many of the applications are dominated by just a few types of computations, as shown in Table 2-5. These computations are algorithmically simple, often repetitive and require relatively little software code to implement. Often, input data word sizes are small (8-16 bits), and computation precision requirements are less than that provided by IEEE single precision floating point. The micro-accelerator concept is based on identifying these functions which have high payoff potential and implementing them very efficiently in micro-accelerator nodes which are embedded in the system processor. The more algorithmically complex portions of the processing are performed in the software programmable nodes of the processor, such as PowerPC microprocessors or DSPs.

Table 2-5. Processing Throughput Drivers are Concentrated in a Few Functions.

Sensor Function/Mode	Processing Total Throughput	Dominant Processor Computations	Dominant Computation % of Throughput
Radar Air to Air Surveillance / STAP	10-100 GFLOPS	Q/R Factorization FFT	50-90 20-50
Synthetic Aperture Radar/Image Formation	10-100 GFLOPS	Range/Azimuth FFT Range/Azimuth Prefilter	30-50 30-50
Radar Air to Ground Surveillance/ GMTI	50-300 GFLOPS	Pulse Compression (FFT/IFFT)	70%
Forward Looking Infrared/ ATR	10-100 GOPS	Detection Filtering	90%

Based on the emerging system processing requirements analysis and the micro-accelerator concept of targeting high throughput portions of the problem, a stressing function was selected for detailed analysis using the micro-accelerator approach. STAP was selected, as STAP systems have a need for very high throughput; size, weight, and power reduction techniques are critical; and STAP has driving functions which fit the micro-accelerator criteria. Figure 2-1 shows a STAP algorithm which was used on the Mountain Top program, for which benchmark data had been developed using a large DSP-based multiprocessor system. The most stressing portion of the algorithm is the computation of the beam pattern weights. The bulk of the weight computation is the QR Factorization function. This function was picked for further analysis.



Operations (x 10^6)	269	12,288	384	1.3	16
Throughput (MOPS)	390	17,821	557	2	23

256 QR Factorizations
Each 288 x 144

256(8 MN^2) Operations
= (256)(8)(288)$(144)^2$
= 12,230 x 10^6

Figure 2-1. Space-Time Adaptive Processing Algorithm.

2.2 Computing Precision Requirements

Using the STAP problem as a driving test case to guide the micro-accelerator development, a task was performed to assess the impact of reducing the data word size (and subsequently, the computational precision) in order to increase the number of computing resources that could be put into the configurable devices and to increase the computation rate. For example, a 12 x 12 multiplier requires about one fourth as many logic "gates" in a configurable device as a 24 x 24 multiplier, and can be computed at a higher clock rate. Effort focused on the QR decomposition function, since it is usually one of the highest throughput drivers (up to 95%) in some STAP algorithms.

The approach used simulations in MATLAB which simulated the reduced precision computations for the QR decomposition. These values could then be compared against results computed with double precision floating point and/or entered into larger MATLAB simulations

of an entire STAP algorithm. The QR decomposition algorithm is sensitive to the "condition", or dynamic range characteristics, of the input data sets, so multiple types of data sets were used to explore the problem domain.

To analyze the precision effects, a variable precision QR function was developed, consisting of a group of MATLAB functions that compute the QR factorization using the Fast Givens approach with a specified number of bits in the mantissa to simulate potential hardware implementations. This approach still uses floating point, due to the dynamic range expected in the calculations. In the variable precision QR function, there is one statement for each arithmetic operation. After each arithmetic operation, data values are transformed to a floating point representation having the user specified number of bits in the mantissa, to be used in the next calculation. A sample of this code is shown in Figure 2-2.

```
function y=vprec(x,z)
%
% This Function Converts A Number x To A Reduced Precision
% Number y Having z Bits In It's Mantissa
%
% Find n defined as the largest power of 2 in the number
if x==0
  y=0;
else
 n=real(fix(log2(x)));
if n<=0
 n=n-1;
end
%
% Find w by dividing x by 2^(n-z) and truncating the fractional part
% of this number. The multiplication scales the number so that only
% the correct bits are lopped off.
%
% Scale the number back to the appropriate level by multiplying w by
% 2^(n-z);
y=(round(x/(2^(n-z))))*(2^(n-z));
end
```

Figure 2-2. Sample Code from Variable Precision QR.

This variable precision QR function was then used to calculate weights using input data developed by Mitre as part of their recently developed STAP RT compact benchmark. The Mitre data sets vary in matrix size and condition, as shown in Table 2-6. Mitre had also developed an error criteria against which to measure the accuracy of the results, shown in Figure 2-3.

$$\max_i \; 10*\log_{10} \frac{\left| w(i) - \hat{w}(i) \right|_2^2}{\left| w(i) \right|_2^2} < -10$$

$$\max_i \; \left| w(i) - \hat{w}(i) \right|^2 < .1 \left| w(i) \right|^2$$

$$\max_i \; \left| w(i) - \hat{w}(i) \right| < .316 \left| w(i) \right|$$

Figure 2-3. Mitre QR Benchmark Error Criteria.

This variable precision QR function was then used to calculate weights using input data developed by Mitre as part of their recently developed STAP RT compact benchmark. The Mitre data sets vary in matrix size and condition, as shown in Table 2-6. Mitre had also developed an error criteria against which to measure the accuracy of the results, shown in Figure 2-3.

Table 2-6. QR Benchmark Data Sets

Size of Data Matrix Y	Data Matrix File Name	Condition(dB) of Matrix Y^HY	Weight Vector File Name
32 x 16	Y32x16.dat	75	W32x16.dat
64 x 32	Y64x32.dat	80	W64x32.dat
96 x 48	Y96x48.dat	85	W96x48.dat
128 x 64	Y128x64.dat	90	W128x64.dat
160 x 80	Y160x80.dat	95	W160x80.dat
192 x 96	Y192x96.dat	100	W192x96.dat
64 x 16	Y64x16.dat	75	W64x16.dat
128 x 32	Y128x32.dat	80	W128x32.dat
192 x 48	Y192x48.dat	85	W192x48.dat
256 x 64	Y256x64.dat	90	W256x64.dat
320 x 80	Y320x80.dat	95	W320x80.dat
384 x 96	Y384x96.dat	100	W384x96.dat

The results of using the variable precision Q/R with the Mitre error criteria is shown in Table 2-7. As can be seen, a 22 bit mantissa is required to meet the error criteria for the largest matrix sizes with the highest condition values. When queried on the origin of the error criteria, Mitre responded that the matrix conditions and error criteria were developed to determine if the machine being benchmarked was computing with at least single precision floating point (32 bit)

accuracy. The 22 bit mantissa requirement seen for the worst case matrices essentially corresponds to the 23 bit mantissa of single precision floating point.

Table 2-7. Variable Precision QR Results with Mitre Error Criteria.

Size of Data Matrix Y	Condition(dB) of Matrix Y^HY	Mantissa Bits
32 x 16	75	14
64 x 32	80	17
96 x 48	85	19
128 x 64	90	19
160 x 80	95	20
192 x 96	100	22

Since the Mitre data set conditions were developed to test the accuracy of the computations but not the requirements needed in a radar algorithm, alternate sets and error criteria were then processed using the variable precision QR. A signal generator tool developed by Northrop Grumman was used to develop data representative of different radar scenarios. This signal generator creates 2 dimensional matrices (channels x range samples) that correspond with the signal that would be seen by a 2 dimensional phased array antenna. The user can specify various parameters to be used in this generator, including number of input signals, amplitude of input signals, output matrix size, and other antenna parameters that were not varied for this application. Data sets were created using this signal generator having the same matrix sizes and conditions as the Mitre data sets, and then the precision required to satisfy the Mitre criteria was determined, as shown in Table 2-8. Fewer mantissa bits were required to satisfy the error criteria using the synthetic radar data, leading to the conclusion that the Mitre benchmark data set appears to be more stressing than what typical radar environments would be.

Table 2-8. Variable Precision QR Results with Radar Signal Generator Data.

Size of Data Matrix Y	Mitre Data		Signal Generator	
	Condition(dB) of Matrix Y^HY	Mantissa Bits	Condition(dB) of Matrix Y^HY	Mantissa Bits
32 x 16	75	14	75	14
64 x 32	80	17	80	15
96 x 48	85	19	85	16
128 x 64	90	19	90	17
160 x 80	95	20	95	17
192 x 96	100	22	100	20

This issue was explored further using additional generated data containing multiple jamming signals representative of airborne surveillance scenarios, instead of contraining the generator to output data sets of a specific condition level. Realistic data sets generated using this approach had lower condition values than the benchmark sets. Also, the error criteria developed by Mitre for benchmarking use did not appear appropriate for determining the effect of the errors on the application of weights when developing beam patterns. Certain weight values end up being very small and their effect on radar performance is small. However, these values are the ones having the largest errors since the precision effects most impact their computation. Table 2-9 illustrates the results of using variable precision on the data sets with 5 jamming signals of 40 dB strength. In addition to computing the mantissa bits required to meet the original Mitre benchmarking criteria, and also to meet an alternate criteria based on norms (Norm (w-w^ / Norm (w) <= 0.31), in which the effect of small errors is averaged out, is shown. Fewer precision bits are required to meet the norm-based criteria.

Table 2-9. Variable Precision QR Results with Representative Jammer Signals.

Size of Data Matrix Y	Mitre Data		Signal Generator		Signal Generator 5 Signals, 40 dB		Signal Generator* 5 Signals, 40 dB	
	Cond Y^HY(dB)	Mantissa Bits	Cond Y^HY(dB)	Mantissa Bits	Cond Y^HY(dB)	Mantissa Bits	Cond Y^HY(dB)	Mantissa Bits
32 x 16	75	14	75	14	64	12	64	9
64 x 32	80	17	80	15	68	13	68	10
96 x 48	85	19	85	16	70	13	70	10
128 x 64	90	19	90	17	71	14	71	11
160 x 80	95	20	95	17	72	15	72	11
192 x 96	100	22	100	20	72	15	72	11

* Error Criteria Is Norm $(w-\hat{w})$ / Norm $(w) \leq .31$

The above analyses led to the conclusion that there was potential in using greatly reduced precision in the computations when applied to data expected in realistic radar environments. The error criteria evaluated so far was based on rather arbitrary criteria and the desired goal was to determine the precision requirements for real radar applications. To provide this "real-life" validation, radar data collected from flight tests was processed using the variable precision QR, and radar beam patterns were formed from these results. The reduced precision radar beam patterns were compared to a reference beam pattern computed with double precision floating point. This process is shown in Figure 2-3. (To speed up the computation process, a variation of the variable QR routine, called the truncated MATLAB QR, was developed and tested.)

The data collection scenario consisted of the Northrop Grumman-owned flying radar testbed with a 24 channel radar. Under funding from Rome Labs on the Multi-Channel Airborne Radar Measurements (MCARM) program, flight test radar data was collected in a number of scenarios. The data used for the variable precision analysis consisted of the MCARM aircraft flying south over the Chesapeake Bay. A moving target simulator provided by Rome Labs, which generated 10 targets equally spaced in Doppler shift, was placed on the Eastern shore of the Bay. This data was collected on high speed magnetic tape recorders for use in a variety of radar evaluations.

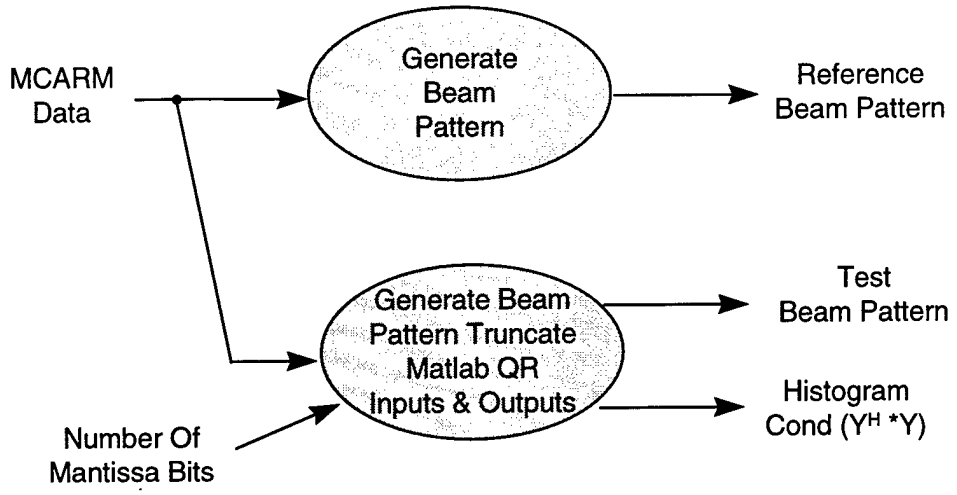
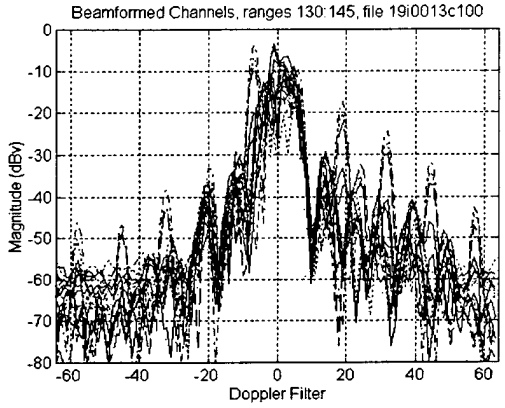


Figure 2-3. Reduced Precision Evaluation Approach Using Collected Radar Data.

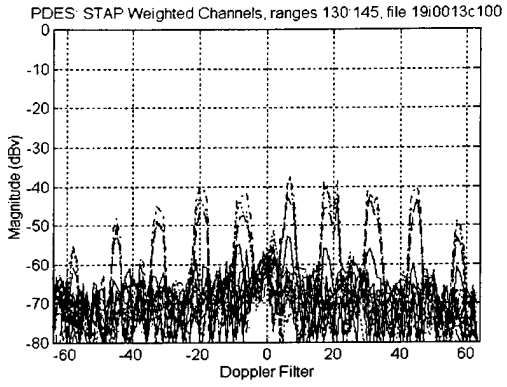
The results of using the variable precision approach on this data are shown in Figure 2-4. The condition of the matrices of this data were from 34 to 73 dB. The main criteria is visibility of targets above the main beam clutter. It was found that very good beam patterns can be obtained with relatively small precision in the QR computation. Further analysis of the beam pattern weights is shown in Table 2-10. Even with the small number of mantissa bits, the number of weights with errors greater than 0.31 is a small portion of the total, and their impact on formation of the beam patterns appears to be very small.

Table 2-10. Analysis of Weight Errors Computed with Variable Precision QR.

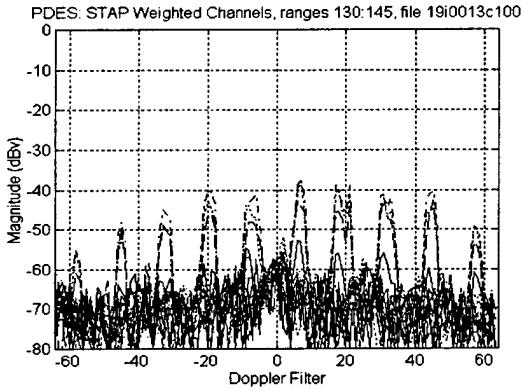
Mantissa Bits	Number Of Weights Error ≤ .31	Number Of Weights Error > .31
14	32,767	1
12	32,766	2
11	32,755	13
10	32,679	89
8	31,818	956



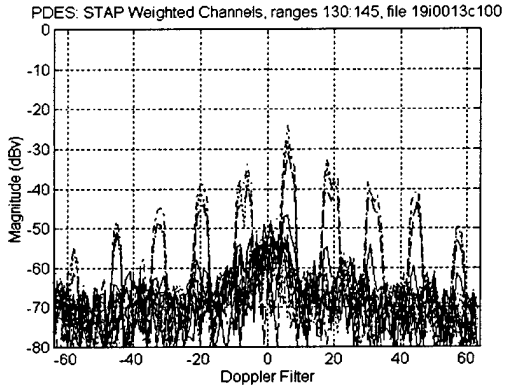
a) Beamforming with no STAP processing. Ten targets are equally spaced in Doppler dimension. Two center targets are obscured in main beam clutter.



b) Beamforming with STAP using 32 range gates to generate weights. ***Double precision floating point arithmetic*** used in weight computation. Main beam clutter has been suppressed, center targets are clearly visible.



c) Beamforming with STAP using ***8-bit mantissa floating point arithmetic*** in weight computation. Result is nearly identical to double precision weight computation.



d) Beamforming with STAP using ***4-bit mantissa floating point arithmetic*** in weight computation. Main beam clutter starting to rise, but targets are still clearly visible.

Figure 2-4. Radar Beam Patterns with Adaptive Weights Generated Using Variable Precision QR.

Conclusions from this precision analysis relative to the development of the micro-accelerator are:

- The accuracy requirements for QR factorization for STAP are application dependent and depend on the number of signals to cancel, amplitude of signals, probability of detection required, probability of false alarms, and other factors.
- Establishing a criteria for representative data sets and measuring errors is only an approximate means of determining accuracy requirements, and detailed system analysis with representative data from the actual application environment must be performed.
- A 16 bit floating point word format (10 bit mantissa, 1 bit sign, 5 bit exponent) was selected for micro-accelerator development and exploration as it appears to be useful for real STAP applications and the Modulo 2 word size eases initial application. The number of bits can be expanded as FPGA technology improves to satisfy more demanding applications.

Also, an number of follow study areas were identified to further provide precision requirement guidance:

- Speed up the execution time of the variable precision QR by linking C programs into MATLAB and using faster platforms, which allows exploration of larger matrix sizes.
- Perform analyses that include the actual detection algorithm to determine end radar performance instead of beam pattern comparisons.
- Analyze reduced rank QR algorithms for suitability to acceleration and accuracy requirements.

3. RECONFIGURABLE DEVICE EVALUATION AND SIMULATION

A study was performed to assess the capability of emerging devices to perform the core computations required for accelerating the driving requirements of the future sensor systems discussed in the previous section. The study consisted of high level evaluation of the device options available and downselection to the most promising for more detailed analysis. Timing and sizing simulations of the required arithmetic functions were performed.

3.1 FPGA Technology Trades

The initial reconfigurable device study consisted of information collection on devices available or soon to be available commercially, as well as the developmental devices being developed on the DARPA Adaptive Computing Systems (ACS) initiative. An initial review yielded six promising candidates. The various vendors/developers were contacted, and additional data was gathered. A top level summary of this effort is shown in Table 3-1. From this initial data, a first downselect was made to the three which had the highest micro-accelerator implementation potential: the Gatefield product using flash memory technology which was under development under a DARPA ACS contract, and the commercial Altera and Xilinx products. The other three initial candidates were eliminated for a variety of reasons. The NSC RSP part did not have the gate density and I/O required for high performance arithmetic problems such as STAP. The University of Washington RaPiD parts did not support floating point arithmetic. The Actel SPGA parts had a promising development plan, but the maturity and projected availability were not compatible with this project's schedule.

Table 3-1. Initial Configurable Device Study Results and Downselection.

Vendor & Contact	Device or Family	Description	Advertised Density	Bench	Comments
NSC Mark Landguth	RSP	GP Core With Configurable Array	6144 Cells 55K Gates	Y	Low Density Only 96 Conf. I/O
Univ Of Wash Carl Ebeling	RaPiD	Coarse Grained Architecture	-	-	No Floating Point Arithmetic Elements
Actel Jay McKibben	SPGA	SRAM Based Fine Grained Architecture	100-400K Gates	N	30K Anti-Fuse Device Densest Avail 5/97 SPGA Is 1st Prog. Part 100K Projected 1Q98
Gatefield Gary Kling	GF250	Flash Memory Based Fine Grained Architecture Sea Of Tiles	18,750 Tiles 150K Gates	Y	Investigate Further
Altera Dave Richard	FLEX 10K	SRAM Based Fine Grained Architecture LUT	10-250K Gates	N	Investigate Further
Xilinx Brian Stephens	XV	SRAM Based Fine Grained Architecture LUT	125-250K Gates	Y	Investigate Further

Each of the downselected technologies were evaluated in more detail. Key factors were gate density and speed as determined by benchmarks and a continued technology evolution plan which provided aggressive increases in these parameters over the next few years. To perform the density benchmarking, a standard of reference was required. Each supplier used a different approach to develop an "equivalent gate" count used in marketing the part. Gatefield uses tiles as the basic element with 8 gates per tile, Altera uses Logic Elements(LEs), and Xilinx uses Configurable Logic Blocks (CLBs). Each of these methods results in a "gate count" for each type and size of device. The gate counts of the largest devices from each of these suppliers projected to be available in the project timeframe, and thus the prime candidates for micro-accelerator implementation, are shown in Table 3-2.

Table 3-2. Equivalent Gate Count Methods for Device Candidates.

Device	Method	Gates
Gatefield GF250F150	18,750 Tiles x 8 Gates/Tile	150,000
Altera EFP10K130	6656 LEs x 12 Gates/LE = 79,872 .35 x 16 EABs 2Kbits/EAB x 4 Gates/bit = 45,872 .65 x 16 EABs x 150 Gates/EAB = 1,560	127,307
Xilinx XC40125XV	4624 CLBs x 2.375 Logic Cells/CLB x 12 Gates/Logic Cell	131,784

This gate count method does not provide much information regarding the actual performance of the devices when implementing specific functions. After reviewing various methods of comparison, it was decided that the most meaningful approach for the micro-accelerator application was to benchmark the arithmetic functions required for the identified sensor processing functions. Table 3-3 shows the results of running a reduced precision (16 bit) floating point multiply, based on the system requirements of the previous section. The table shows the three candidate devices implementing the multiply, using multiple pipeline stages. Simulated clock frequencies were derived to provide speed comparisons, as was device utilization, which is the percent of device resources required to implement the function, to provide a figure of merit of circuit density.

Although the three devices all had nominally the same "equivalent gate count", the percent of the total resources in the device required to implement the function varied by a factor of nearly two to one. The Gatefield and Altera devices had nearly the same device resource utilization, but the Xilinx parts implemented the function in about half the resources as the other two. Also, the Xilinx devices had over a 50% higher speed than the other two.

Table 3-3. 16-bit Floating Point Multiplier Benchmark Results.

Pipeline Stages	Gatefield GF250F150			Altera EPF10K130-3			Xilinx XC40125XV-09		
	Clock Freq. (MHz)	Tiles	Device Util. (%)	Clock Freq. (MHz)	LEs	Device Util. (%)	Clock Freq. (MHz)	CLBs	Device Util. (%)
0	12.3	1006	5.4	14.6	356	5.4	16.5	161	3.5
3	30.2	1277	6.8	39.5	457	6.9	54.6	167	3.6
5	38.1	1304	7.0	-	-	-	79.3	184	4.0

Additional factors used in selecting an FPGA family for further study for the micro-accelerator included vendor support and product roadmaps. Xilinx provided a very high level of technical, and had a long term roadmap with continuous product improvements in terms of density and speed, and devices with one million gates were projected in the 1999-2000 timeframe. The final result of the evaluation was to use the Xilinx family for more detailed timing and sizing studies and for the micro-accelerator prototype implementation.

3.2 Floating Element Compute Definitions

Based on the system analysis and precision studies, variable precision floating point compute elements were deemed necessary for maximum benefit of the micro-accelerator approach as applied to the STAP problem. Northrop Grumman developed a complete definition of the reduced precision floating point format to be used and defined the architecture of the various floating point compute elements required for the QR function: multiplier, adder/subtractor, and divider. These were used as the basis for VHDL models from which the FPGA implementations were synthesized.

Referring back to the conclusion reached in Section 2.2 to use a 16 bit floating point format for initial work, Figure 3-1 illustrates the format defined for bit ordering, data types and ranges. This format is based on the IEEE 754 standard definitions, as shown in Table 3-4. The format is a subset of the standard, in that plus/minus infinity, not-a-number, and subnormal numbers are not presently supported.

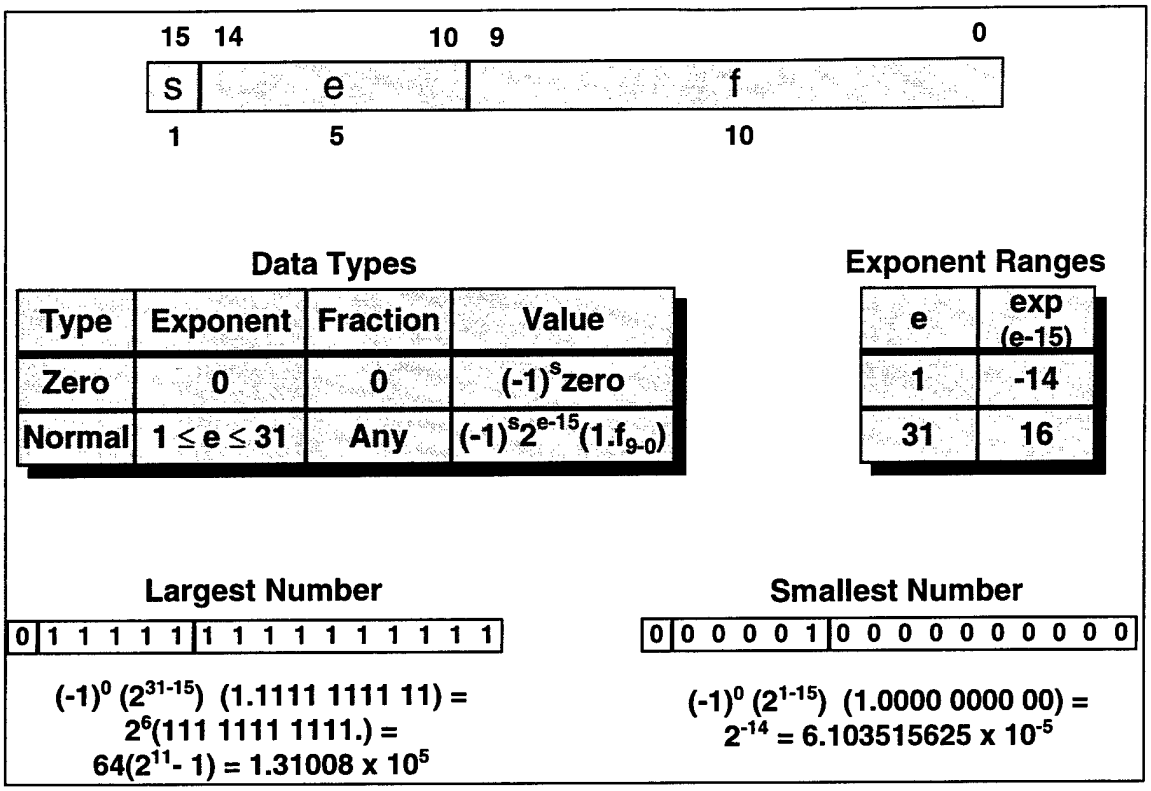


Figure 3-1. 16 bit Floating Point Format

Table 3-4. Floating Point Format Relationship to IEEE 754 Definitions.

Exponent	Fraction	Value
255	0	± ∞
255	non-zero	NaN
0	non-zero	$(-1)^s 2^{-126}(0.f)$ (subnormal)
0	0	± zero
0 < e < 255	any	$(-1)^s 2^{-126}(1.f)$ (normal)

The Micro-Accelerator Formats Are A Subset Of IEEE 754
± ∞, NaN And Subnormal Numbers Not Presently Supported

Using these floating point definitions, architectures for the compute elements were defined. Although these architectures are for the 16 bit floating point format, they can easily be applied to alternate word formats if desired. Figure 3-2 shows the floating point multiplier architecture.

Figure 3-3 shows the floating point adder/subtractor. Figure 3-4 shows the floating point divider, which can also be used as a multiplier by bypassing the reciprocal function.

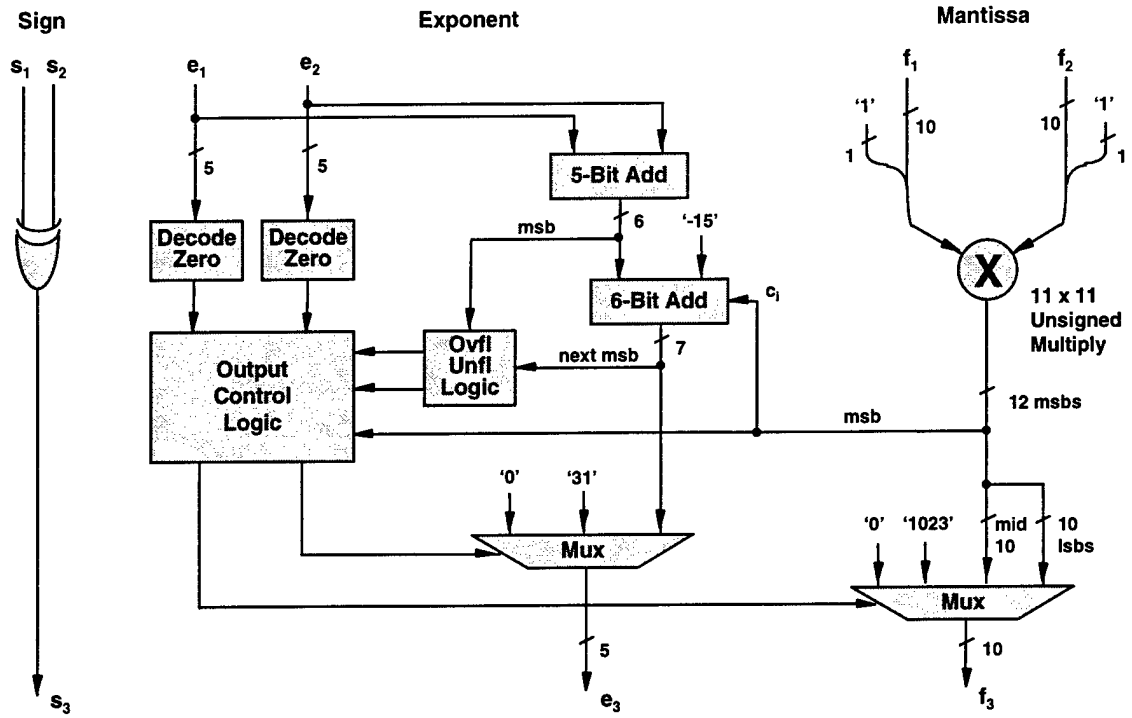


Figure 3-2. 16 Bit Floating Point Multiplier.

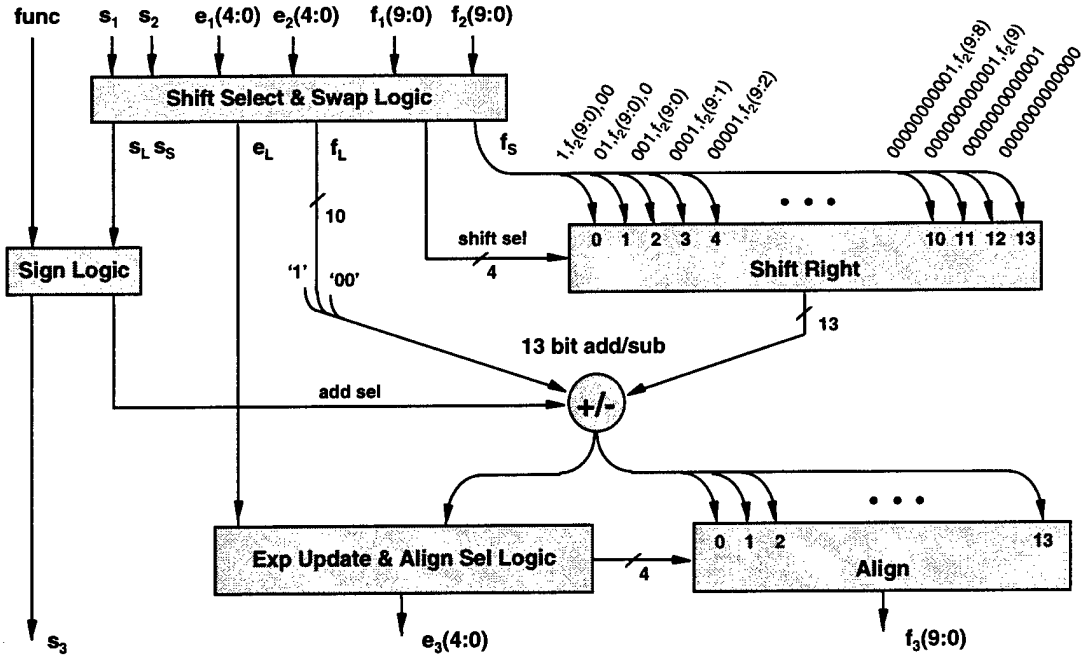


Figure 3-3. 16 Bit Floating Point Adder/Subtractor.

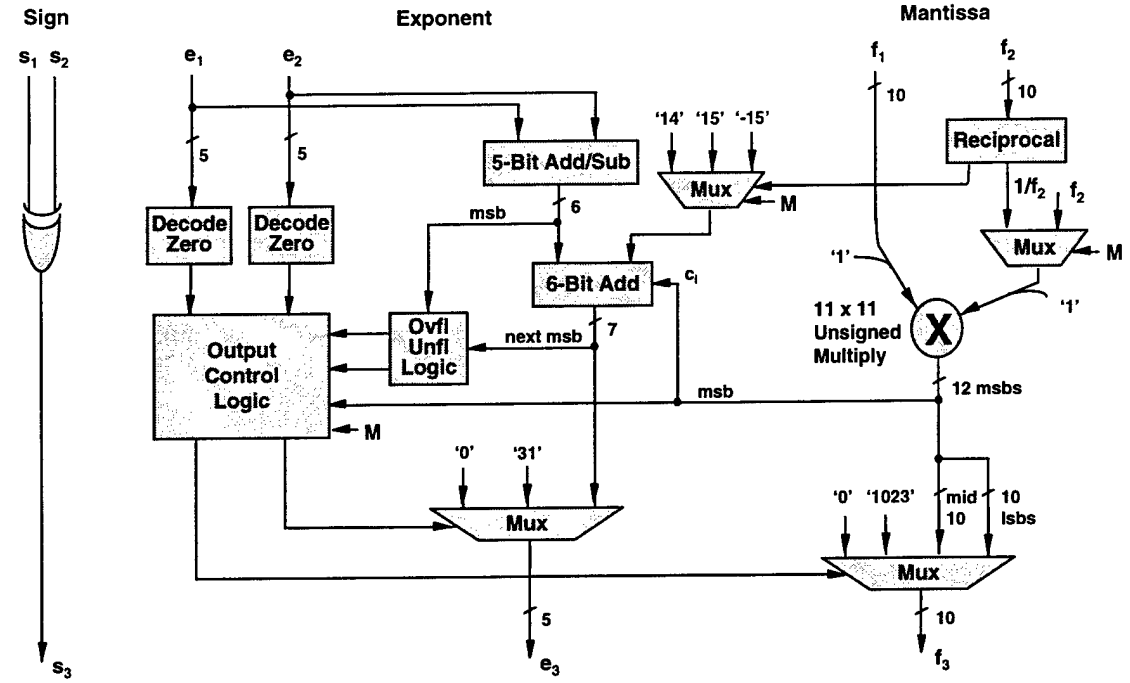


Figure 3-4. 16 Bit Floating Point Divider.

3.3 VHDL Simulation

Using the architectures developed for the compute elements, VHDL modeling and FPGA timing and sizing simulations were performed. The task objectives included developing VHDL descriptions of the floating point arithmetic functions as well as dual port memory. The targeted performance constraint was 80 to 100 MHz operation, and pipelining was allowed. Performance data on the compute elements from this task was also fed into a higher level system performance model to obtain data on the entire QR factorization and is discussed in Section 4. Another objective of the VHDL simulation task was to determine sizing requirements for partitioning the QR functions into multiple FPGAs. A final objective of the task was to evaluate the VHDL design methodology and software development tools of the Xilinx family of FPGAs chosen for the prototype.

The FPGA design flow is shown in Figure 3-5. VHDL source files were created for the various functional elements. The Synplicity synthesis tool was used to develop a netlist for the functions. The netlist is then input to vendor specific tools for placement and routing on the FPGA structure. Constraints can be entered into the FPGA tools at this point; the primary constraint used was timing. The place and route tools iterate until either the timing constraint is met or a specified maximum number of iterations is performed. The output data can be fed into a functional simulation of the design using tools by ModelTech. After the design has met all goals, programming files are generated to program the image into the FPGA.

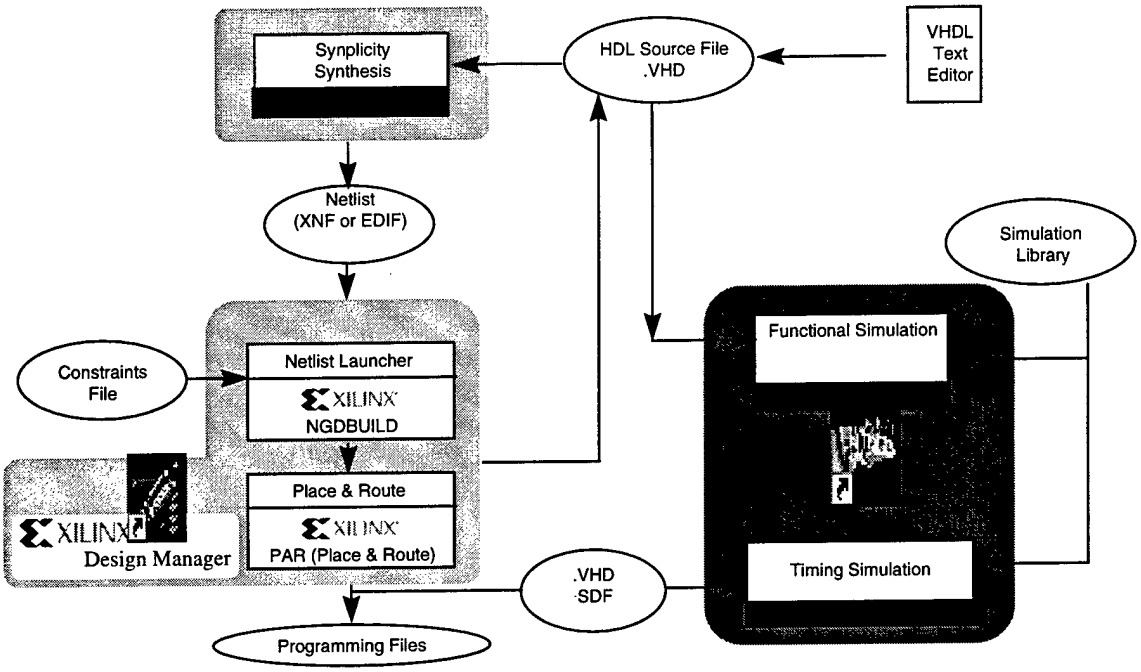


Figure 3-5. FPGA Design Flow.

Since the goals of the micro-accelerator effort were directed towards high performance arithmetic to accelerate stressing portions of the military sensor processing problem, the VHDL design effort focused on high clock speed. To achieve this performance, a pipelined approach was used for the arithmetic elements. Pipelining involves breaking the function into a number of stages which can be computed within a short clock time, with registers inserted between stages. Although this approach involves some latency (multiple clock cycles corresponding to the number of pipeline stages) in getting the first computation through the computation stages, the following computations are provided one every clock cycle. For the long vectors typical of the targeted sensor problems, this approach is very efficient and is commonly used. When using this approach, one attempts to achieve a high clock rate within a moderate number of stages, typically less than ten.

The methodology used for arithmetic element design development is shown in Figure 3-6. A VHDL description is created for an element, such as a multiplier, and the design is synthesized and a best case timing analysis is created. If the timing goal (e.g., 80 MHz) is met, placement and routing is performed to see if the design still meets timing constraints after routing. If the timing goals are not met, either a pipeline stage is inserted or the VHDL is recoded for an alternate approach. This iteration cycle is performed until the timing goals are met.

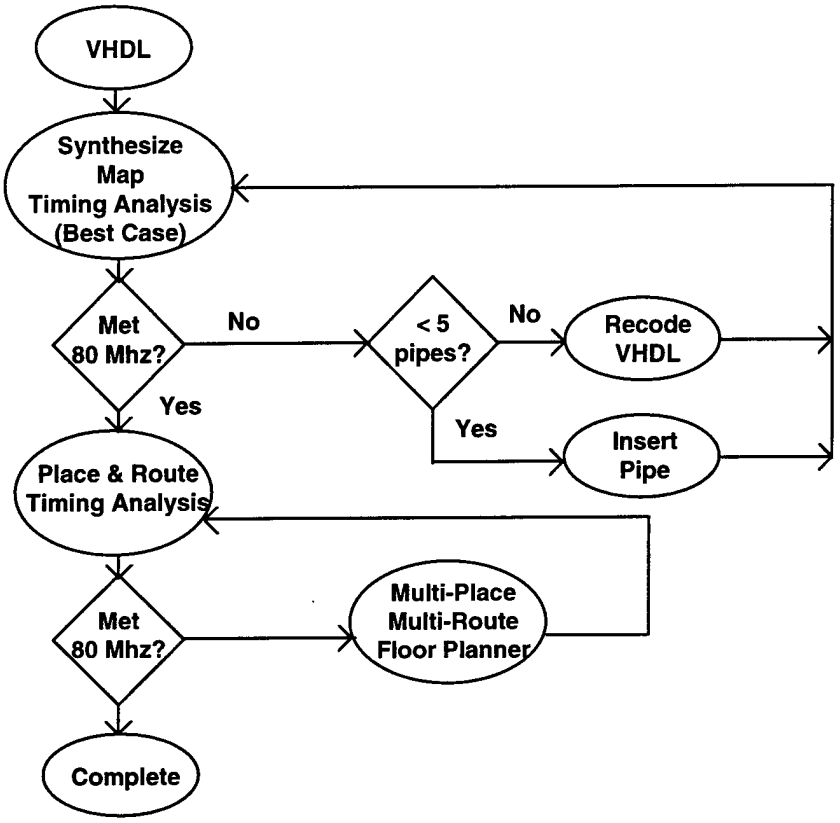


Figure 3-6. Arithmetic Element Design Flow.

Using the multiplier as an example, multiple options can be used when creating the VHDL code. The simplest approach is to use the VHDL "*" operator to multiply two standard logic vectors. In this case, the architecture is defined by the synthesizer tool one is using, and customizations such as pipeline registers can't be added. Another option is to use the vendor-supplied multiply macros. In this case, these elements did not meet performance criteria. Also, there is no capability in these vendor designs for adding pipeline registers or controlling their placement, and limited bus width options are available. The option used on this effort was to design the multiplier at the gate level, which provides total control over the number and placement of the pipeline registers and enables meaningful comparison of different multiplier architectures. This last approach is the most time consuming, and it is hoped that a standard family of these elements can be defined and reused to reduce effort in future applications. However, this approach allowed a number of multiplier architectures to be evaluated, including the carry save-adder tree, Booth's algorithm, and the summation of partial products, as shown in Figure 3-7. The summation of partial products was down selected for area, speed, and pipelining ease considerations.

VHDL code for the compute elements with varying numbers and locations of pipeline stages were created and the designs simulated on the Xilinx FPGA software. Figure 3-8 shows the results for the 16 bit floating point multiplier. Using 5 stages allowed the design to run at

approximately 80 MHz, which was the design goal for the initial micro-accelerator implementation.

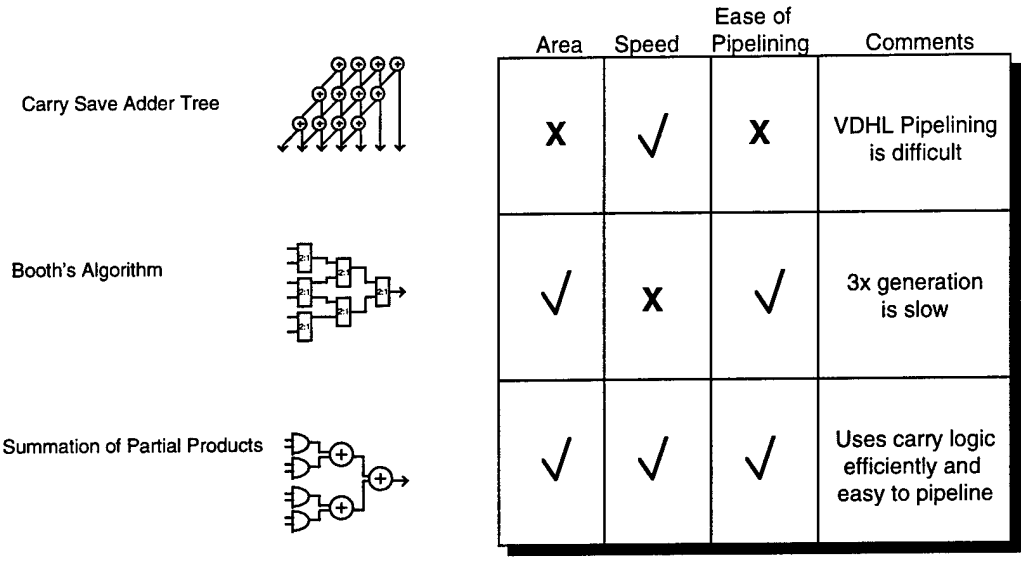


Figure 3-7. Multiplier Architectures Evaluated.

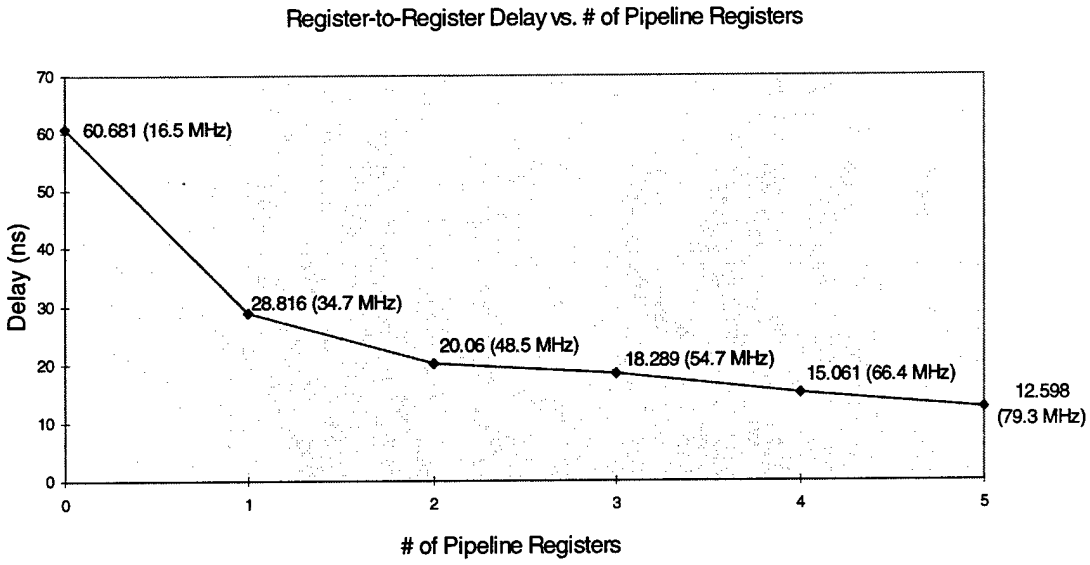


Figure 3-8. 16 bit Floating Point Multiplier Timing Simulations.

The 16 bit floating point add/subtract performance as a function of pipeline registers is shown in Figure 3-9. The adder structure was more difficult to pipeline to achieve improved performance. As a result, the adder required an additional pipeline stage (6 total) to operate at 80 MHz.

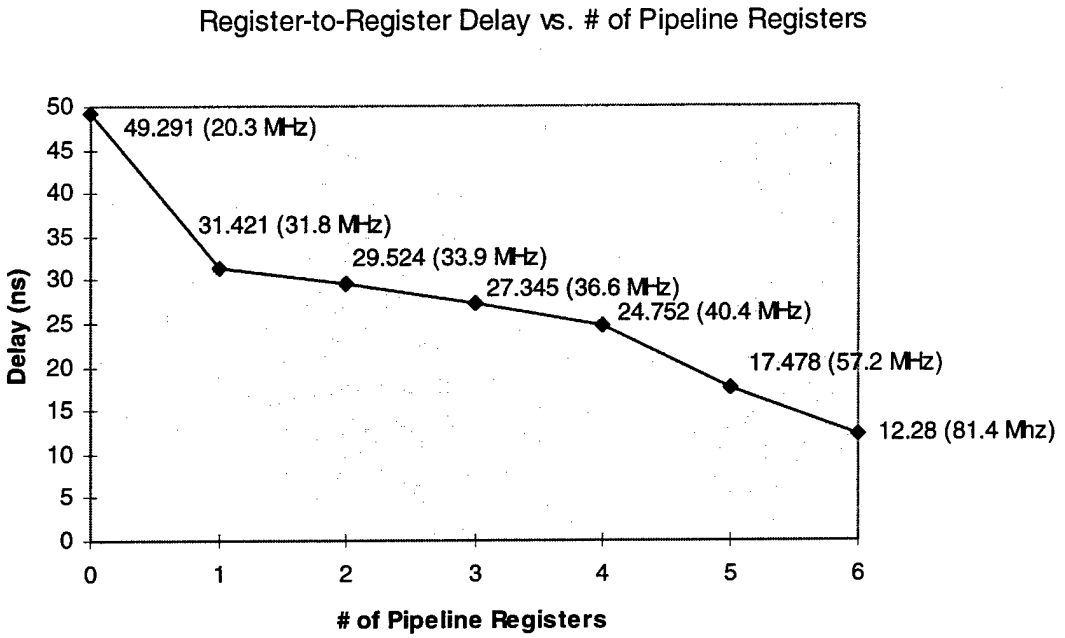


Figure 3-9. 16 bit Floating Point Adder Timing Simulations.

A number of high speed design techniques were used in developing these elements. These techniques addressed functions which were either slow or used configurable logic blocks inefficiently. Table 3-5 provides a summary of the techniques used in developing optimized arithmetic functions.

Table 3-5. Optimization Techniques for Arithmetic Functions.

A<B, A>B, A=B Comparison Functions
- Problem: Equivalence function is slow .
- Solution: Use > or < functions (because they use the carry logic) and combine carry outs to produce A=B.

Normalization/Barrel Shifter Functions
- Problem: The wide data paths require several levels of logic
- Solution: Divide the paths into 4-bit groups (to take advantage of the FPGA architecture) and multiplex the intermediate results. This also allows intermediate pipeline registers to be inserted.

Priority Encoder Function
- Problem: Multiple logic levels are required which impacts speed negatively.
- Solution: Designing 2 or 4 bit priority encoding blocks to encode inputs into small groups. Then encode intermediate results into a final result.

Smart Partitioning
- Problem: Miscellaneous logic (such as overflow/underflow detection) inputs exceed CLB port width (thus requiring multiple CLBs).
- Solution: Use parallel logic to decrease CLB inputs to allow more efficient CLB usage (this increases logic utilization to meet speed requirements).

Application of the above techniques resulted in the final VHDL descriptions for high performance elements for the floating point multiply, add/subtract, divide, and memory. Table 3-6 summarizes the final results achieved in terms of CLB usage, number of pipeline stages, and maximum worst case operating speed. The maximum operating speeds of the multiply and divide could be increased slightly to reach 80 MHz by either letting the place and route routines run longer to optimize performance, using a higher speed grade part than was simulated, or restricting operation to conditions (i.e., reduced temperature range or tighter power supply voltage tolerance) more favorable than worst case.

Table 3-6. Final Timing and Sizing Results for Arithmetic Elements.

	CLB Count	# of Pipelines	Max Speed
Floating Point Multiply	**184**	**5**	**79.3 MHz**
Floating Point Add/Subtract	**138**	**6**	**81.4 MHz**
144 x 32 Dual Port Memory	**488**	**2 write 3 read**	**80 MHz**
Floating Point Divide	**205**	**9**	**79.3 MHz**

4. MICRO-ACCELERATOR ARCHITECTURE AND SIMULATION

Based in the system analysis requirements and the timing and sizing studies for reconfigurable devices, an architecture for performing the QR factorization was developed. This architecture was simulated to determine computing efficiency performance when performing the QR factorization. Multiple options were evaluated to optimize the implementation, and sustained performance efficiencies over 50% were achieved. Performance comparisons against DSP and microprocessor based approaches was analyzed at the chip, daughter card, and VME module level, and order of magnitude improvements were projected.

4.1 Algorithm Overview

The STAP QR factorization function (which will be called QR for short) was selected as the initial target for the micro-accelerator. The QR is a key operation in developing weights for creating radar beams which can adapt to clutter and jamming. The mathematical operation performed in the QR is to convert a matrix of data to upper triangular form, as shown in Figure 4-1.

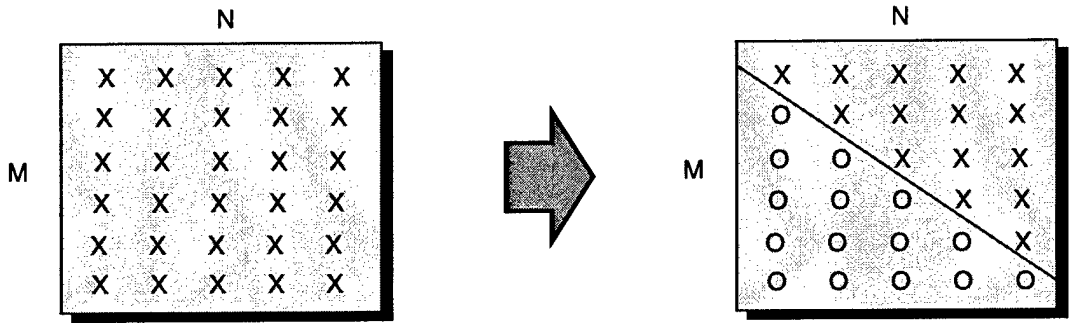


Figure 4-1. QR Factorization converts matrix to upper triangular form.

The QR approach chosen for implementation studies was the Fast Givens technique, shown in Figure 4-2. To zero the diagonal, terms called alpha, beta, and the D-vector are first computed using the first elements in the bottom two rows. Two of these terms, alpha and beta, are applied to the two rows to create a zero in the first column of the bottom row. This process is repeated until all data below the diagonal has been zeroed.

There are ordering constraints for zeroing, as shown in Figure 4-3. To zero out element "A" requires that elements to the left and below are zeroed as shown. These constraints enter into the design tradeoffs made when developing the QR implementation which can greatly impact the efficiency of the computations.

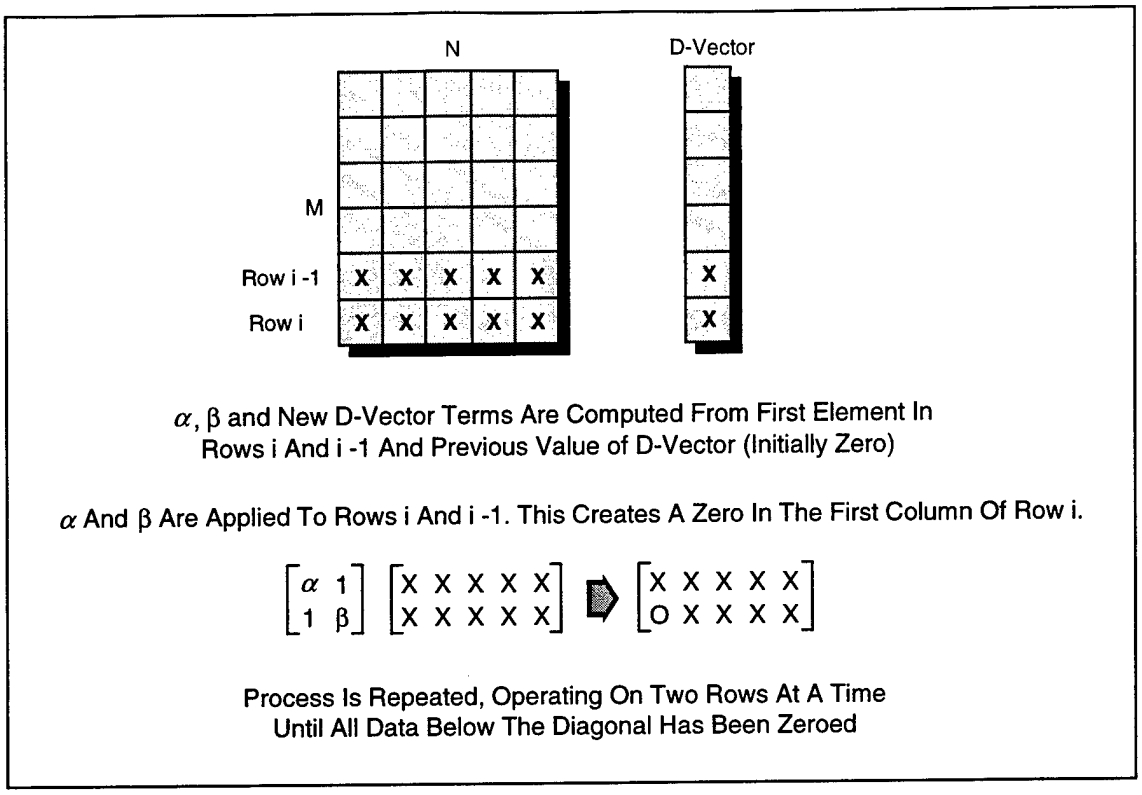


Figure 4-2. Fast Givens QR Factorization Calculations.

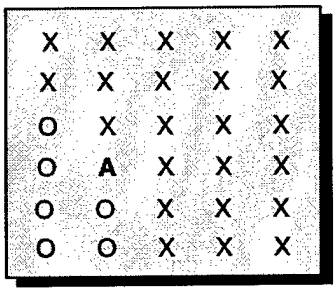


Figure 4-3. Ordering Constraints for Zeroing.

4.2 Baseline Architecture

An architecture for performing the QR which could be implemented on the micro-accelerator was then derived and tradeoffs performed. The architecture consists of three main elements: an Arithmetic Element (AE) with a small local memory for applying the alpha/beta terms to the matrix terms, an Alpha/Beta (AB) Generator for developing the alpha/beta terms, and high speed main memory to provide storage for the matrix. This architecture is shown in Figure 4-4. The initial version of the architecture had four AEs that could operate on two words from two columns at once. The AB Generator computed two alpha/beta terms in parallel to supply the AEs.

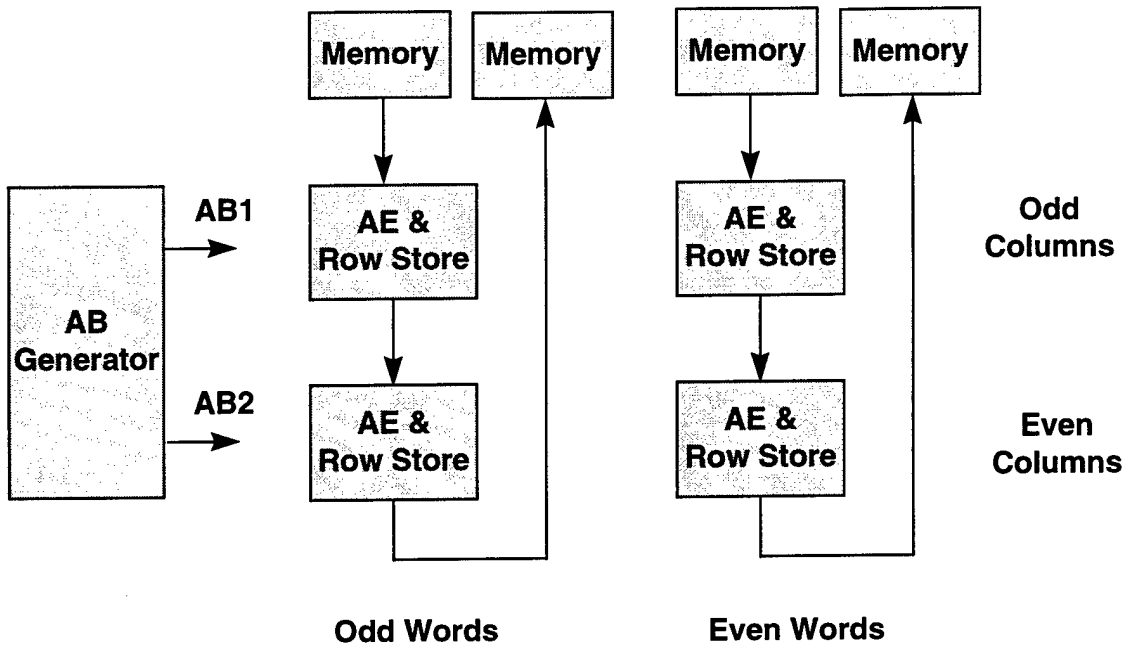


Figure 4-4. Initial QR Factorization Architecture.

The basic function performed by the AE is a complex multiplication. Data inputs to the AE consist of the two complex words from the matrix rows and the alpha/beta complex words. The outputs are two complex words. The arithmetic structure to perform this function is shown in Figure 4-5. The AE performs a total of eight real multiplies and eight real adds/subtracts.

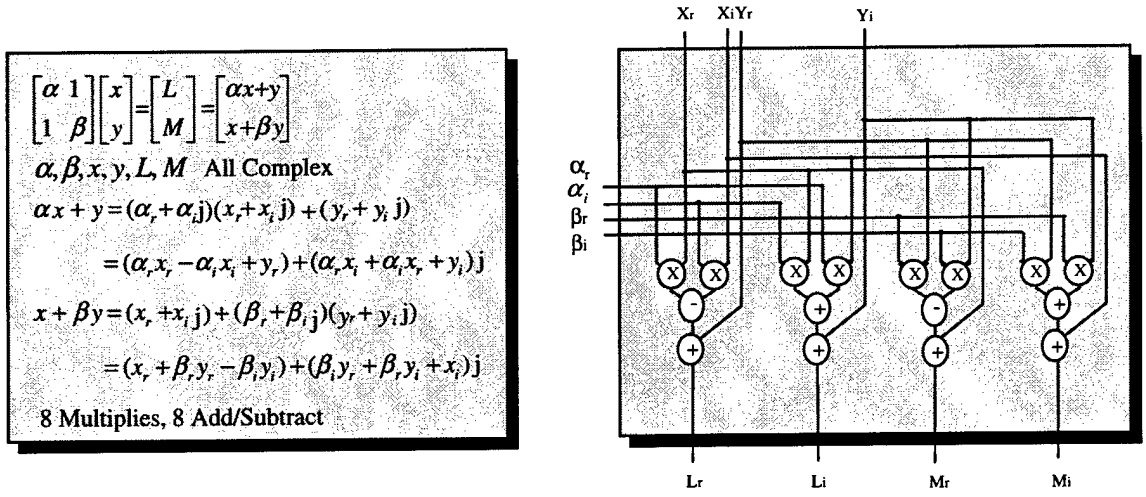


Figure 4-5. Arithmetic Element (AE) Calculations and Architecture.

Key features of the architecture include the row store to reduce memory bandwidth requirements and the cascading of the AEs to allow multiple zeroing computations at once, as well as reducing device I/O. The row store is used to store the value of a row of data which is used twice, both for zeroing elements in the row below it, as well as its own row. Saving and reusing this value as opposed to bringing it in from main memory cuts memory bandwidth requirements for this computation essentially in half. The cascading of AEs also allows elements in two rows to be computed simultaneously, without increasing the memory bandwidth requirements by feeding the results of an upper row calculation to the lower row calculation. The computation flow with these architectural features is shown in Figure 4-6.

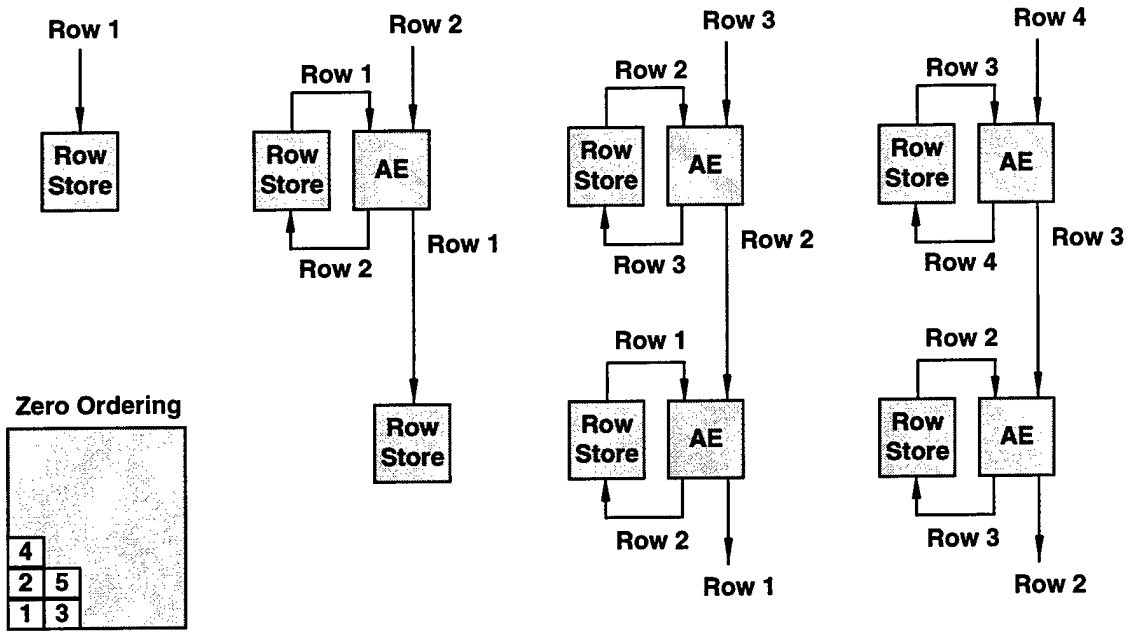


Figure 4-6. Row Store and Cascaded AEs.

The AB Generator performs a number of computations in developing the alpha/beta terms. A summary of the computations is shown in Figure 4-7. A total of 18 multiplies, 7 adds, and 3 divides are required. To enable the AB Generator to operate independently of the AEs, it computes the alpha/beta terms and applies them to the first two points in a row, as shown in Figure 4-8.

Computations	Multiplies	Adds	Divides
temp1 = conj(X(i -1)) * X(i -1)	2	1	-
temp2 = conj(X(i)) * X(i)	2	1	-
if d(i) * temp1 < d(i -1) * temp2	2	1	-
alpha = -conj(X(i -1)) * X(i) / temp2	4	2	2
temp1 = d(i) / d(i -1)	-	-	1
beta = -temp1 * conj(alpha)	2	-	-
oneplus = 1 - alpha * beta	2	2	-
d(i) = d(i -1) * oneplus	1	-	-
d(i -1) = d(i) * temp1	1	-	-
X(i -1) = oneplus * X(i)	2	-	-
Totals	18	7	3

Figure 4-7. AB Generator Computations.

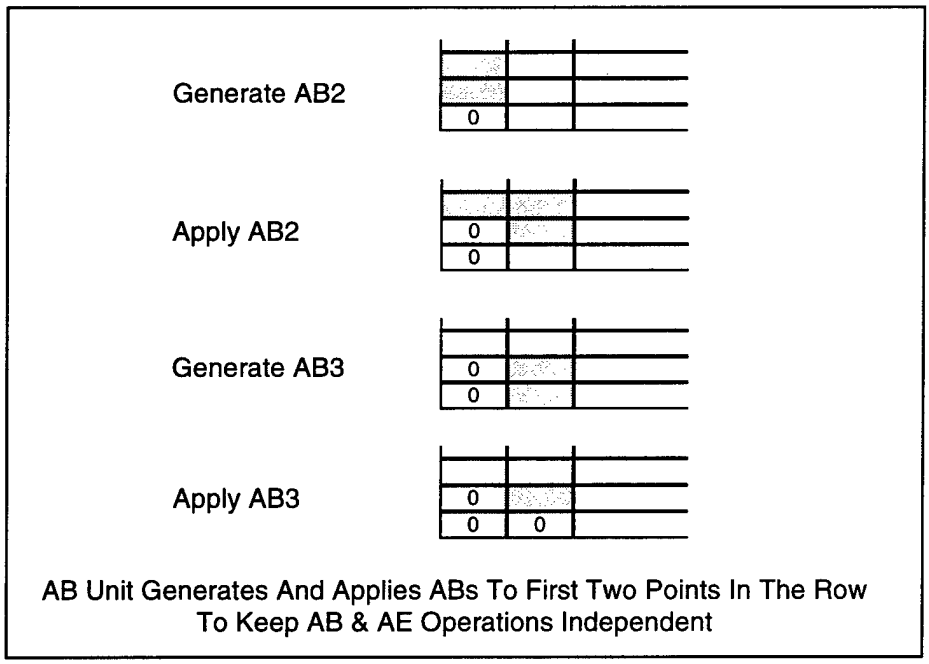


Figure 4-8. Functions Performed in AB Generator.

Based on the above computation requirements, an architecture was developed for the AB Generator function, shown in Figure 4-9. In this architecture, only 4 multipliers, 3 adders, and 2 dividers are required in the FPGA implementation. Multiple passes are made through the

functional elements to perform the required computations. This minimizes resources required for the AB Generator FPGA implementation, which can be quite significant.

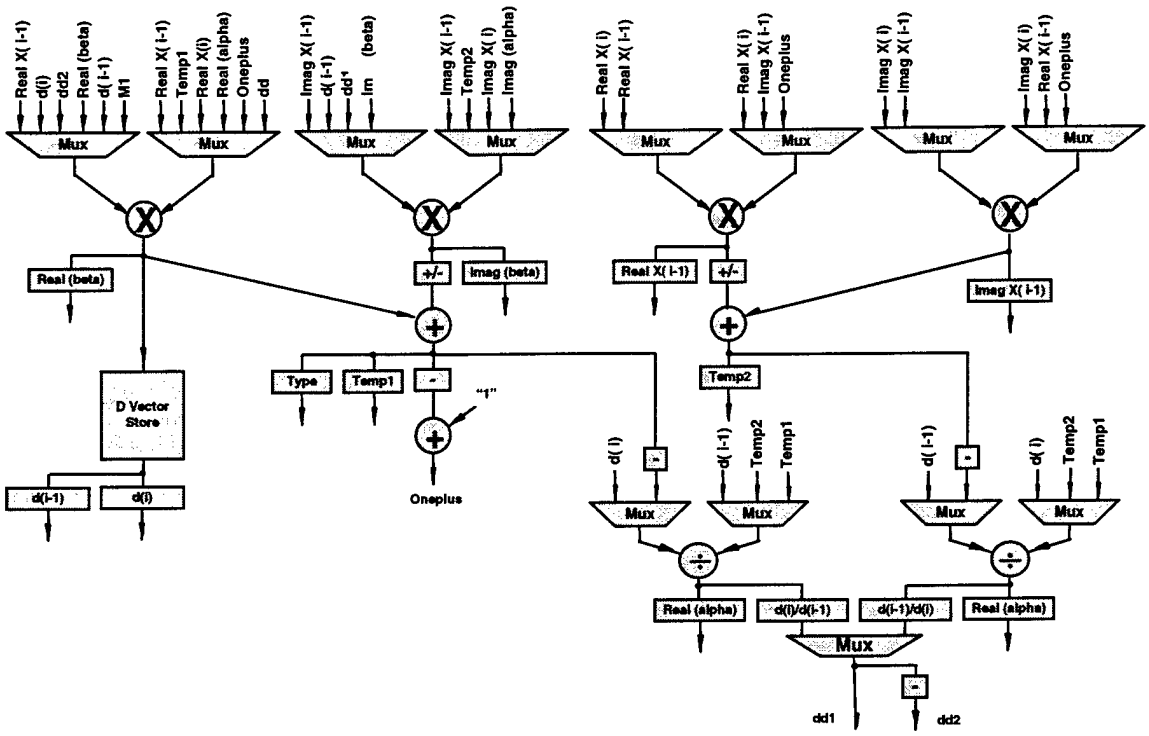


Figure 4-9. AB Generator Architecture.

4.3 Performance Modeling and Enhancements

After development of the architecture of the AE and AB Generators, performance modeling of the execution of the QR in the micro-accelerator was performed. The goal of this task was to determine efficiency of the architecture, and perform optimizations in terms of pipeline stages, computation parallelization, etc.

A high level diagram of the timeline for the baseline architecture is shown in Figure 4-10. Functions being performed are reading data in from main memory, computing alpha/beta terms, writing the upper and lower row stores, applying the alpha/beta terms to the rows of data, and writing the data back to main memory.

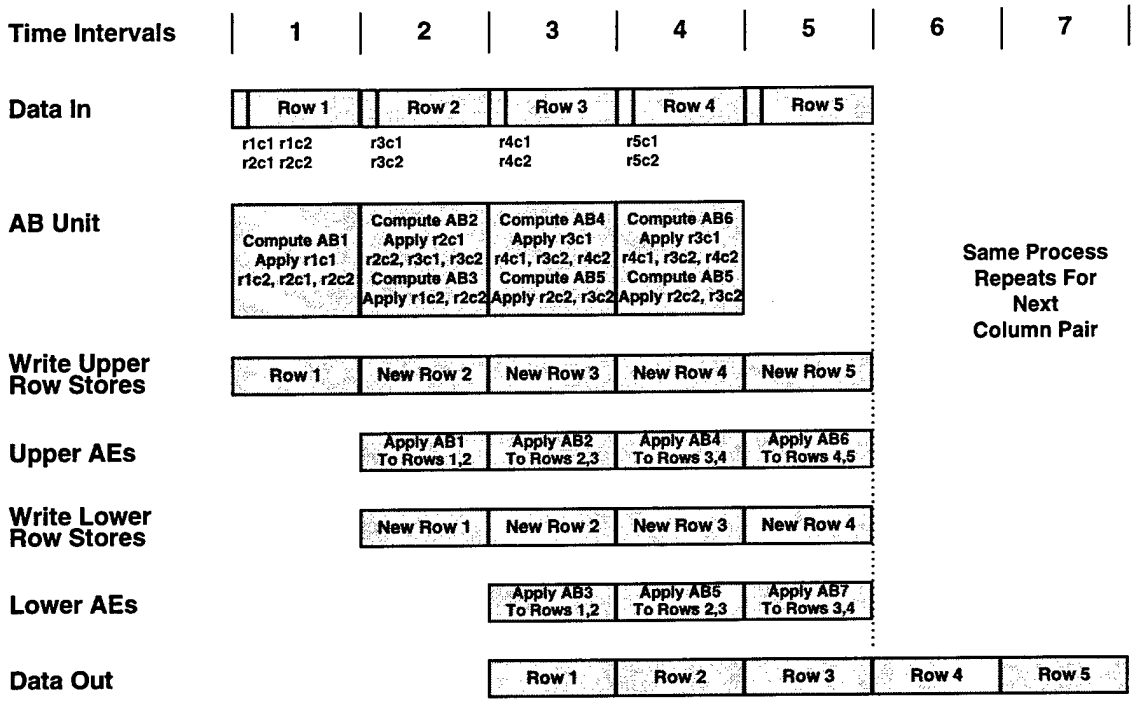


Figure 4-10. QR Factorization Timeline.

A key aspect of this timing diagram is the overlapping of the alpha/beta generation with the alpha/beta application by the AEs. This means that the relative timing of the functions is critical in terms of achieving high computing efficiency. If the alpha/beta terms are not computed for the next row by the time the AEs are finished computing the current row, the AEs will be idle for the period of time to finish alpha/beta generation. There is also a limit to how far the AB Generator can "work ahead", as data from the first elements of the current row are needed before the AB Generator can compute terms for the next row. Thus for optimum computing efficiency, a balance in the timing between the AB Generator and AE functions is desired. This balance is complicated by the computation approach in that as the matrix is diagonalized, the rows become shorter, decreasing the AE function time for the row. The AB generator computations, however, and therefore the alpha/beta terms computing time, remain the same.

An engineering analysis and tradeoff was required to optimize the processing efficiency and performance. Table 4-1 illustrates the number of clock cycles of latency required by the AB Generator to create the alpha/beta terms and by the AEs to apply them. The number of latency cycles is dependent on the number of pipeline stages in the multiplier, adder, and divider. More pipeline stages result in higher processing throughput, but with a higher latency; less pipeline stages reduce the throughput, but also reduce the latency.

Table 4-1. AB Generation Timing.

Function	Passes	Cycles
Generate AB1	1 2 3 4 5 6	mpipes + apipes mpipes + apipes mpipes + apipes + dpipes mpipes mpipes + 2apipes mpipes
Apply AB1	1 2 3	mpipes mpipes mpipes
Generate AB2	6	Same As AB1
Apply AB2	1	mpipes
Total	16	16 mpipes + 10 apipes + 2 dpipes

To analyze the performance of the entire QR, a performance model was developed and exercised. The model has the following features:

- Inputs
  - Clock Rate
  - Number of Pipeline Stages for Multiply, Divide an Add
- Outputs
  - Peak Throughput = Arithmetic Elements x Clock Rate
  - Minimum Compute Time = Operations Required / Peak Throughput
  - Actual Compute Time = Clock Cycles Required / Clock Rate
  - Efficiency = Minimum Compute Time / Actual Compute Time
  - Sustained Throughput = Efficiency x Peak Throughput
- Factors
  - Prolog and Epilog for Each Column Pair
  - For Each Row Processed, Whether Dominated by AB Time or AE Time
- Usage
  - Run Model for Different Clock Rate and Number of Pipelines form FPGA Design Activity (More Pipelining Makes AE Time Shorter But Increase AB Latency)

The results of the model using the baseline architecture developed are shown in Figure 4-11. The peak processing performance with a single AB Generation unit feeding four AEs is high (nearly six GFLOPS at 80 MHz), but the sustained throughput is very low: about one GFLOPS. This was found to be due to the very low utilization of the AEs due to lack of timely computation of the alpha/beta terms. To improve the efficiency, the use of only two AEs was modeled, but sustained performance and resulting efficiency was still very low, due to the mismatch still remaining in AB versus AE computation times.

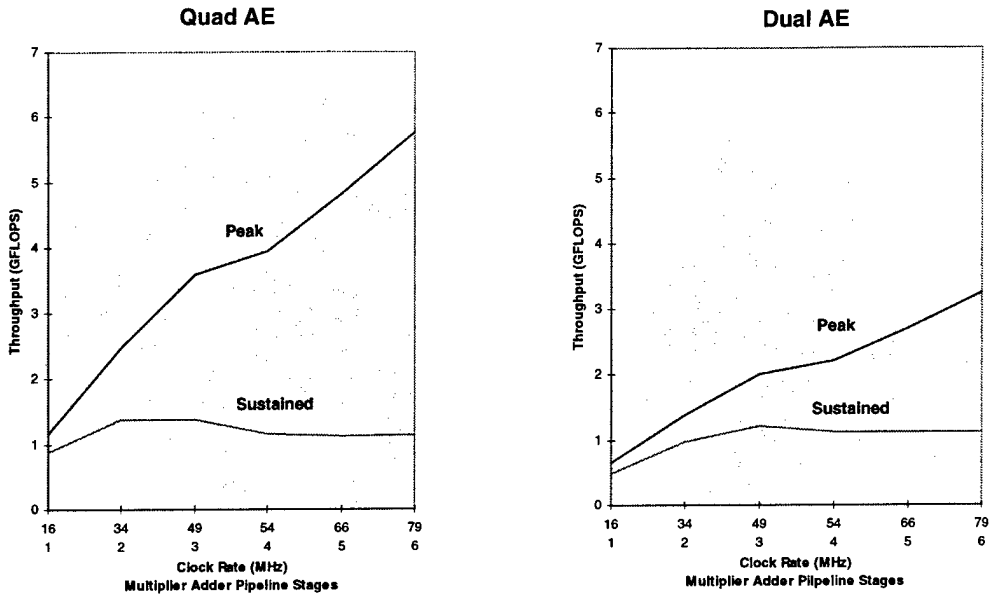


Figure 4-11. Micro-Accelerator Performance with Single AB Generation Unit.

Exploring the problem further, an approach for dual AB Generation units was developed. To compute alpha/beta terms in parallel requires following a strict ordering approach, shown in Figure 4-12.

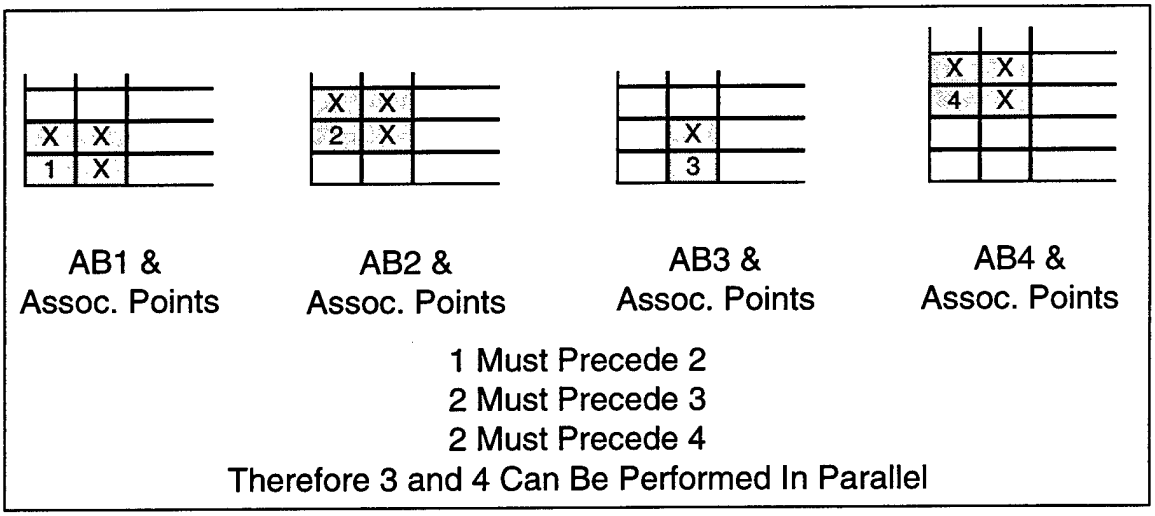


Figure 4-12. Ordering for Dual AB Generation Units.

Using this technique, the performance model was rerun, as shown in Figure 4-13. Peak performance improved slightly due to the additional computations provided by the second AB Generator. Sustained performance for the quad AE configuration improved up to 2 GFLOPS, but then peaked. Dual AE performance continued to improve with clock frequency, but was starting to flatten out also.

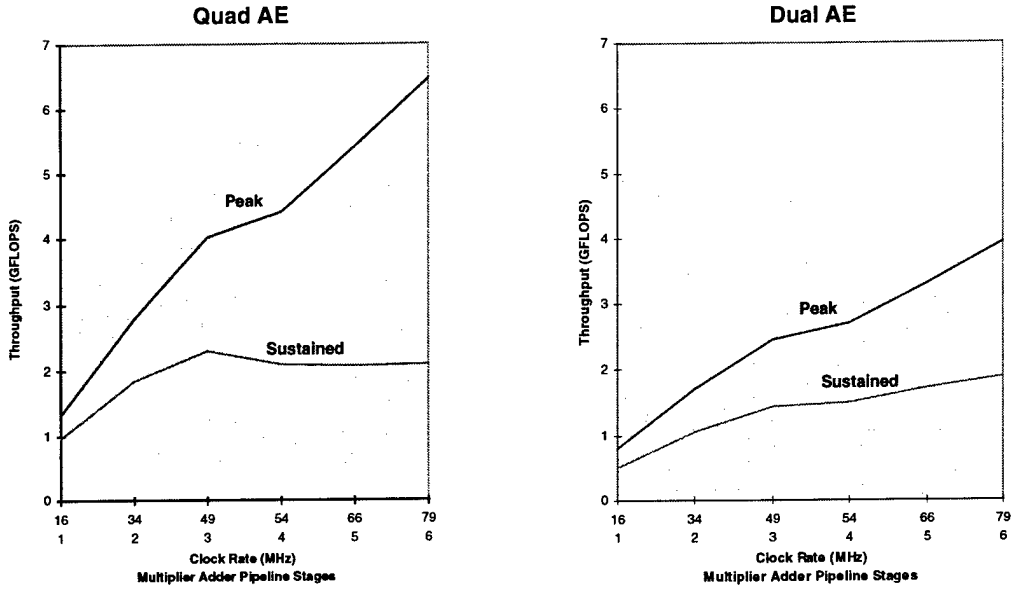
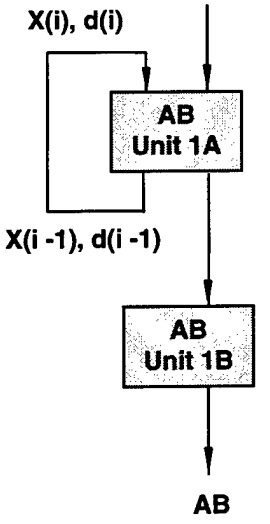


Figure 4-13. Micro-Accelerator Performance with Dual AB Generator Units.

Sustained performance was still not deemed adequate due the latency in computing the alpha/beta terms. An approach was developed to overlap computation of multiple alpha/beta terms, using intermediate results from one computation to feed the next computation. This approach, shown in Figure 4-14, greatly reduced the computation latency cycles.



	Compute Elements	Cycles To Compute & Apply AB1 & AB2
Original AB	4 multipliers 3 adders 2 dividers	16 mpipes + 10 apipes + 2 dpipes
Pipelined AB	5 multipliers 3 mult/dividers 6 adders	6 mpipes + 5 apipes + 2 dpipes

Figure 4-14. Pipelined AB Generation Unit.

The performance model was then rerun using the pipelined structure for a single AB Generator, feeding both a quad AE and a dual AE configuration. Although sustained performance was

somewhat improved over previous configurations, a flattening out of the sustained throughput was still seen, as shown in Figure 4-15.

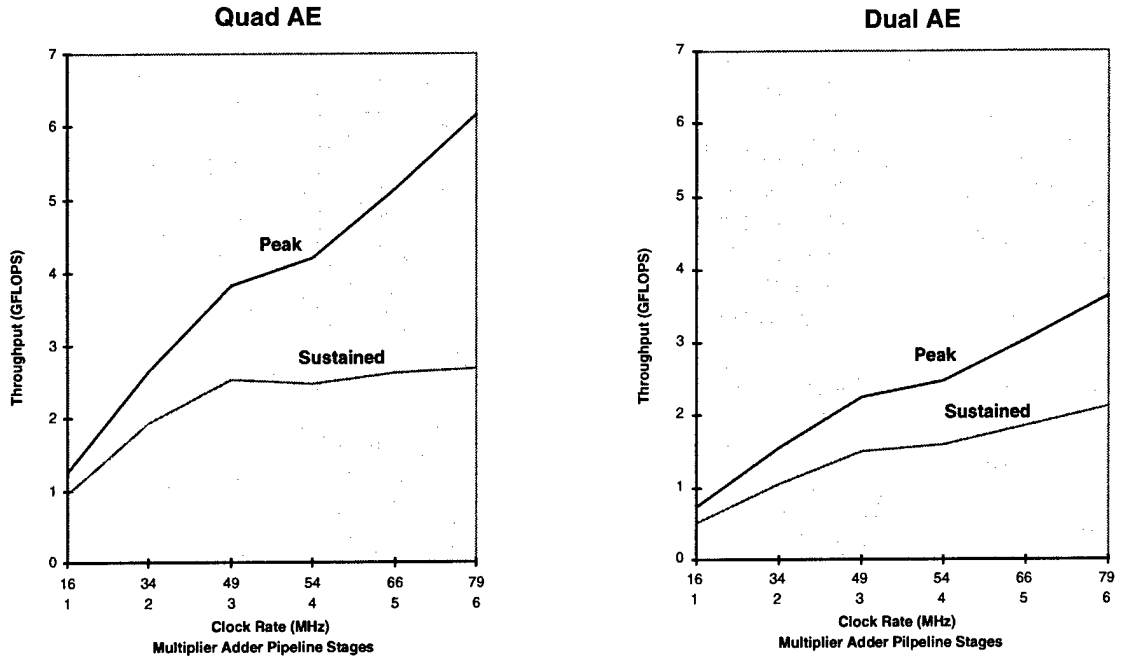


Figure 4-15. Single Pipelined AB Generator Performance.

The model was then changed to add dual pipelined AB Generators. For this final configuration, sustained performance continued to increase with clock speed and increased multiplier/adder/divider pipelines, as shown in Figure 4-16. Over 50% efficiency was achieved, and a quad AE configuration achieved nearly 8 GFLOPS peak throughput with around 4 GFLOPS sustained throughput at an 80 MHz clock rate. With faster speed grade parts due out soon, clock speeds could be increased to 100 MHz, providing nearly 10 GFLOPS peak, 5 GFLOPS sustained performance per micro-accelerator processing node.

Based on the FPGA timing and sizing simulations performed with the Xilinx devices, the various configurations run on the performance model were sized. The number of configurable logic blocks (CLBs) for the different elements, as well as the total, are shown in Table 4-3.

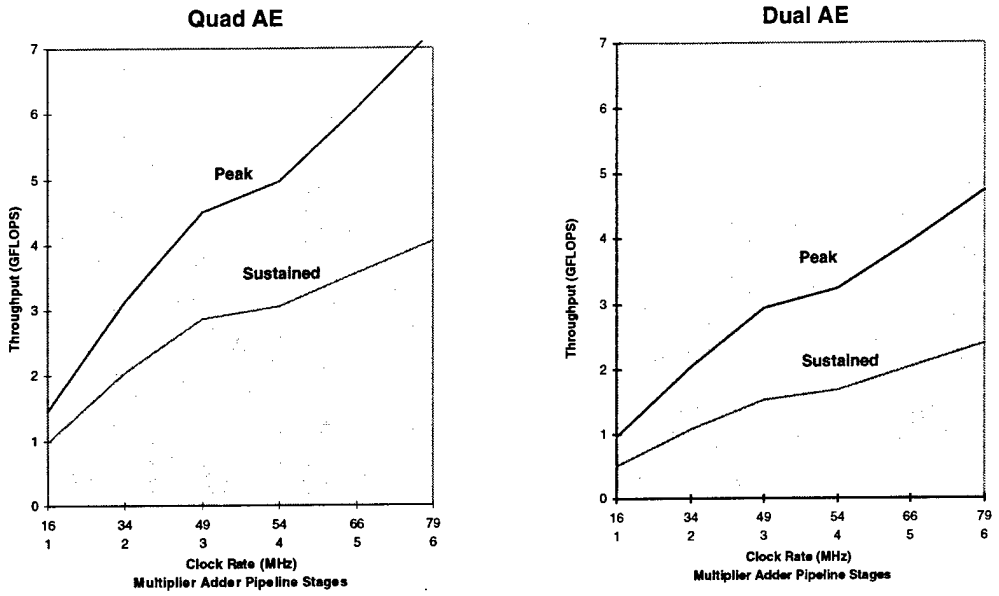


Figure 4-16. Dual Pipelined AB Generator Performance.

Table 4-3. QR Configuration Functional Element Sizing.

	Mults	CLB Per Mult	Mult/ Div	CLBs Per Mult/ Div	Adds	CLBs Per Add	Row Store	CLBs Per Row Store	CLB Sub-total	Other CLBs	Total CLBs
Pipelined AB	5	184	3	205	6	138	0	488	2363	1632	3995
Dual Pipelined AB	10	184	6	205	12	138	0	488	4726	2864	7590
Dual AE	16	184	0	205	16	138	2	488	6128	1278	7406
Quad AE	32	184	0	205	32	138	4	488	12256	1956	14212

These sizings were then partitioned into the different generations of devices either available or becoming available, as shown in Table 4-4. The design goal for the micro-accelerator hardware was to use a small number of FPGAs, to allow a daughter card form factor which could be integrated onto COTS VME processor motherboards. Since performance was the primary goal, the dual pipelined AB/Quad AE configuration was the desired implementation. The largest FPGAs available during the design phase of the project were the XV125 devices, which would have required six devices on one daughter card, along with memory and interfaces. However the Virtex devices were projected out soon, which allowed the desired configuration to be placed in two devices.

Table 4-4. FPGA Device Count for QR Micro-Accelerator Configurations.

	XV125 (4624 CLBs)	XV250 (8464 CLBs)	Virtex (13K CLBs)
Single Pipelined AB	1	1	1
Dual AE	2	1	
Single Pipelined AB	1	1	1
Quad AE	4	2	1
Dual Pipelined AB	2	1	1
Dual AE	2	1	1
Dual Pipelined AB	2	1	1
Quad AE	4	2	1

Based on the projected continuing increased density and performance of the candidate FPGA devices, the decision was made to develop the prototype micro-accelerator to accommodate two FPGA devices, and take advantage of the device improvements over time. A phased plan for initial development and improvement was developed, shown in Figure 4-17. The initial daughter card design was configured to use the available 125XV parts. However, the footprint of the FPGAs was made to be upward compatible with the Virtex family of devices, so the same board could be upgraded with Virtex parts when they became available. Also, due to limited funds on the project, the Raceway interface would be designed in, but not implemented until a future date. The prototype board has a maximum growth capability of nearly 10 GFLOPS peak performance, 5 GFLOPS sustained performance when performing the QR. It is anticipated that a future generation of Virtex, with .18 micron feature sizes and 1.8 volt cores could be implemented on a design upgrade of the daughter card and provide over 10 GFLOPS sustained performance in the year 2000 timeframe.

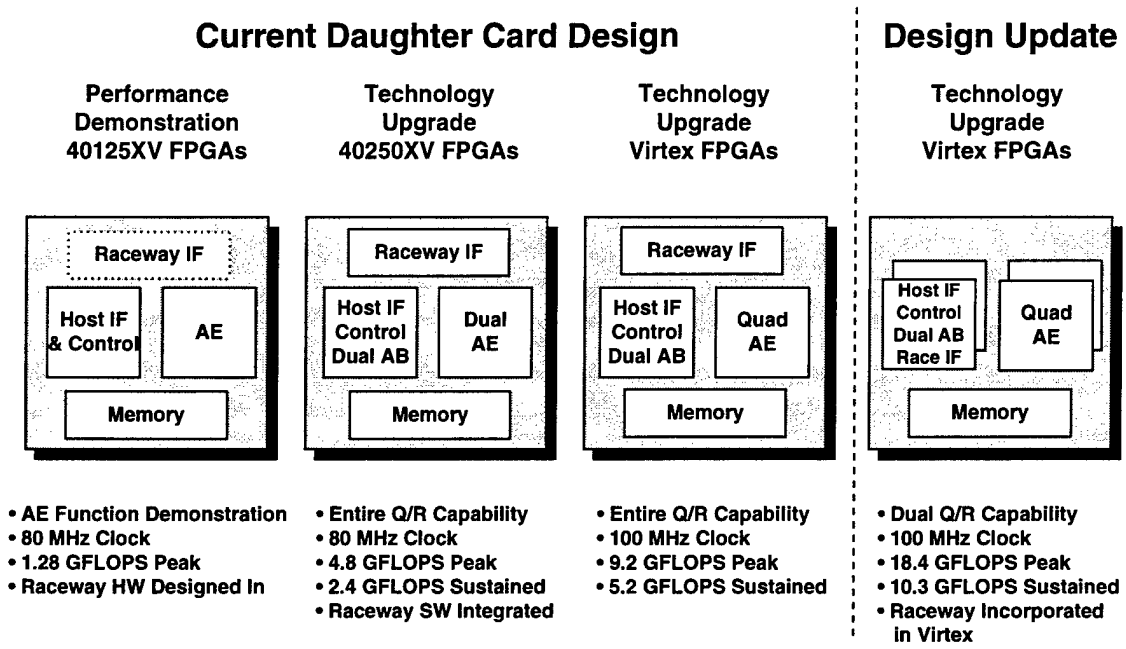


Figure 4-17. Daughter Card Phased Upgrade Plan.

System level performance comparisons of the micro-accelerator for performing the QR were developed. Table 4-5 compares sustained performance at the daughter card level for existing devices (i860, SHARC), a DSP emerging in the year 2000 timeframe, and the micro-accelerator. Performance improvements at the daughter card level are nearly 20 to 1 when comparing equivalent-year technologies.

Table 4-5. Performance Comparison with Commercial Devices.

	Peak Throughput Per Chip/Set (GFLOPS)	Chips/Sets Per Daughter Card	Efficiency	Sustained Throughput Per Daughter Card (GFLOPS)	Factor
Intel i860	.08	2	.3	.048	215
ADI SHARC	.12	6	.3	.216	48
ADI Hammerhead SHARC	.6	6	.15	.54	19
Micro-Accelerator (Virtex FPGA, Updated Daugh)	9.2	2	.56	10.3	1

The performance comparison was then extended to a specific STAP application, in which a critical timeline had to be met. In this comparison, a system was configured using current and future DSP devices (SHARCs and Hammerhead SHARCs). The impact of adding micro-accelerators was then assessed, where the timeline still had to be met, but some of the processing was performed in accelerators. Chip count reductions at the system level were over 10 to 1, as shown in Table 4-6.

Table 4-6. Impact of Micro-Accelerator Insertion.

Accelerators	Sustained Throughput (GFLOPS)	QR Compute Time (sec)	Time Left For Other Operations (sec)	SHARC Chips Required	Hammerhead SHARC Chips Required
0	0	-	-	603	242
4	41.2	.297	.300	68	27
6	61.8	.198	.399	51	21
8	82.4	.149	.448	46	18

A final performance comparison was made at the VME processing module level, again comparing DSP based configurations (SHARC and Hammerhead SHARC) without and with micro-accelerator daughter card insertion. The results, shown in Table 4-6, again indicate a large reduction in module count (6 or 7 to 1 improvements). As technology improves in both the DSP regime and in FPGAs in generations beyond those studied, these improvements are expected to remain relatively constant.

Table 4-6. Board Level Impact of Micro-Accelerator Insertion.

	SHARC Chips Only	SHARCs With Micro-Accel
SHARC Chips	603	46
Micro-Accelerators	0	8
SHARC Dght Cards	101	8
Micro-Accel Dght Cards	0	4
Total Daughter Cards	101	12
9U Modules	13	2
6U Modules	51	6

	HH SHARC Chips Only	HH SHARCs With Micro-Accel
HH SHARC Chips	242	18
Micro-Accelerators	0	8
HH SHARC Dght Cards	41	3
Micro-Accel Dght Cards	0	4
Total Daughter Cards	41	7
9U Modules	6	1
6U Modules	21	4

5. MICRO-ACCELERATOR DAUGHTER CARD DESIGN

Using the results of the system performance studies and the VHDL modeling, the design for a prototype micro-accelerator daughter card was developed. Design requirements were developed commensurate with the goals of providing a generic architecture which allowed the micro-accelerator to perform a wide variety of high throughput sensor functions, with specific analysis on performing the QR factorization function efficiently. Detailed design, layout, and routing of the daughter card were performed. The daughter card printed circuit board was fabricated.

5.1 Architecture

The key features of design derived from the system requirements are:

- Two Xilinx 40125XV FPGAs configured for AE, AB, and control
- Eight 64K x 32 Sync SRAMS for data memory
- Clock input/distribution and memory devices to support 100 MHz+ operation
- On-board 4M x 1 NVM to store FPGA configuration data
- Designed for upgrade with larger Virtex FPGAs
- Test interface to PC host allowing ease of experimental configurations
- Compatible with COTS VME mother boards (Mercury Type B Daughter Card), Raceway interface compatible

The architecture developed for the micro-accelerator daughter card consists basically of an arithmetic section, memory, and I/O, as shown in Figure 5-1. The arithmetic portion consists of two FPGAs, which will be programmed for the acceleration functions desired. In the QR implementation, one FPGA is intended to be programmed with AE units and the other with AB Generation and control functions. Since the functions to be accelerated are typically highly parallel and require high memory bandwidths for high computing efficiency, eight banks of synchronous SRAM are used to provide a very wide and thus very high bandwidth between the FPGAs and memory. The memory devices themselves are wide (32 bits), hold 2 Mbits of memory each, and can operate at up to 133 MHz. System I/O is a Raceway interface, allowing the daughter card to be used directly on Raceway based mother boards and mixed and matched with other types of heterogeneous nodes such as PowerPCs and SHARC/Hammerhead SHARC DSPs. A test header has been placed on the daughter card allowing initial micro-accelerator and demonstration without the Raceway using the JTAG interface. On-card nonvolatile memory devices provide for storing the application programming images for fielded applications. For laboratory demonstrations, FPGA images can be loaded through the test header, allowing demonstration of different functions without changing the NVM devices. Clock distribution is provided from two selectable sources. An onboard oscillator can provide a fixed clock for fielded operation. A clock signal from an off-board clock generator source can also be used to allow experimentation with different clock rates for different applications for laboratory experimentation. Power distribution includes 5V and 3.3V from the Raceway connector. Since the FPGA cores operate at reduced voltage (2.5V), on-card voltage regulators convert 3.3V to 2.5V.

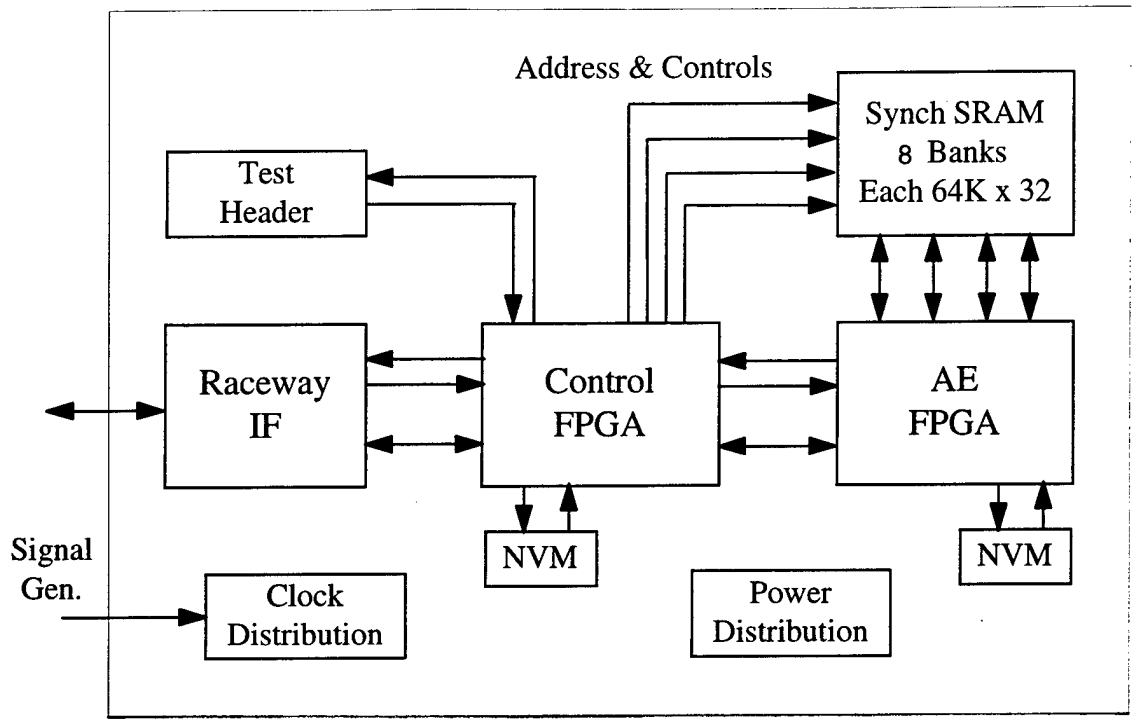


Figure 5-1. Micro-Accelerator Daughter Card Architecture.

The core of the micro-accelerator is the section consisting of the two large Xilinx FPGAs. To assist in proper development of the architecture of the daughter card, a configuration with the FPGAs partitioned into the functions needed for the QR function was used to drive architecture decisions. It was felt that this application was the most stressing in terms of memory interface, so an architecture suitable for QR would also be suitable for less complex problems such as EO automatic target recognition filtering. The block diagram of the FPGAs configured for the QR are shown below. Figure 5-2 shows the Arithmetic Element configuration for one FPGA, and Figure 5-3 shows the control portion of the AB/Control configuration for the other FPGA.

FPGA configuration can be accomplished in multiple ways. For initial test and demonstrations, the FPGAs are configured using a PC host, as shown in Figure 5-4. The FPGA programs are loaded into the FPGAs in slave serial mode using a commercial cable/interface from Xilinx. This allows great flexibility in collecting data on different FPGA functions, totally under software control.

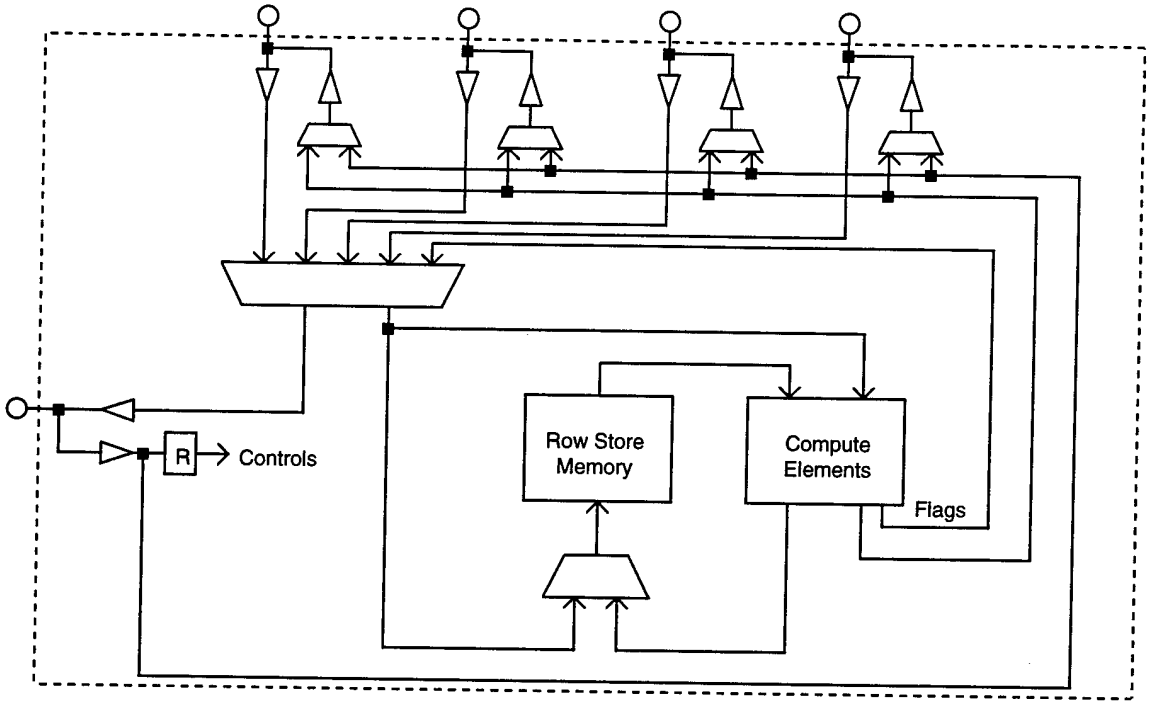


Figure 5-2. Arithmetic Element Configuration for FPGA 1.

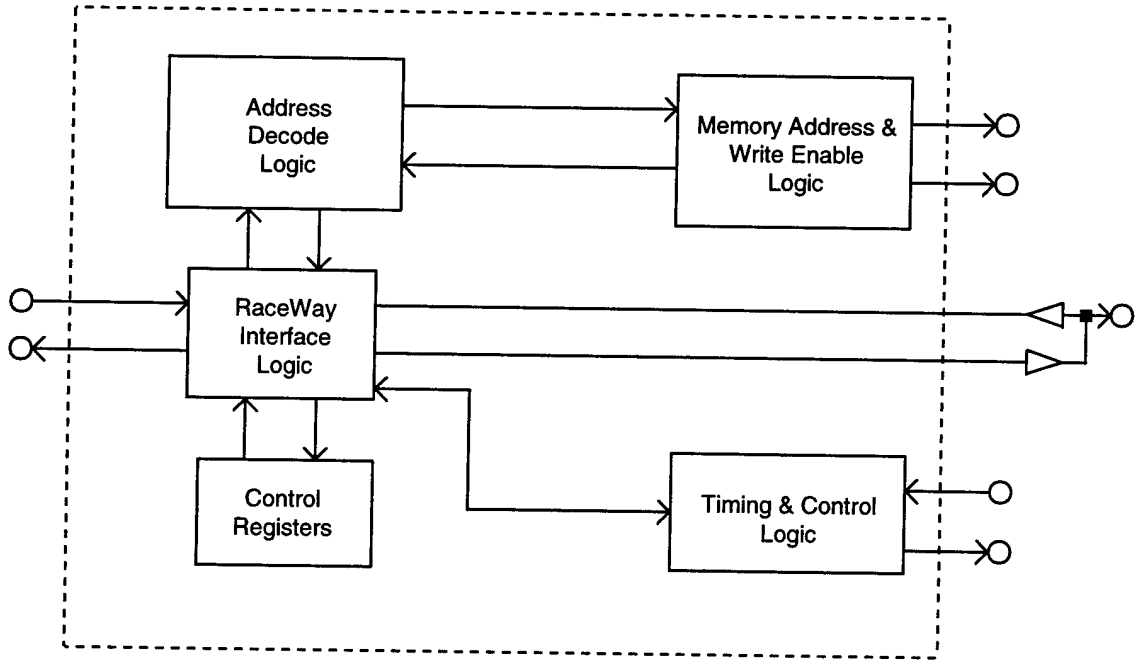


Figure 5-3. Control Configuration for FPGA 2.

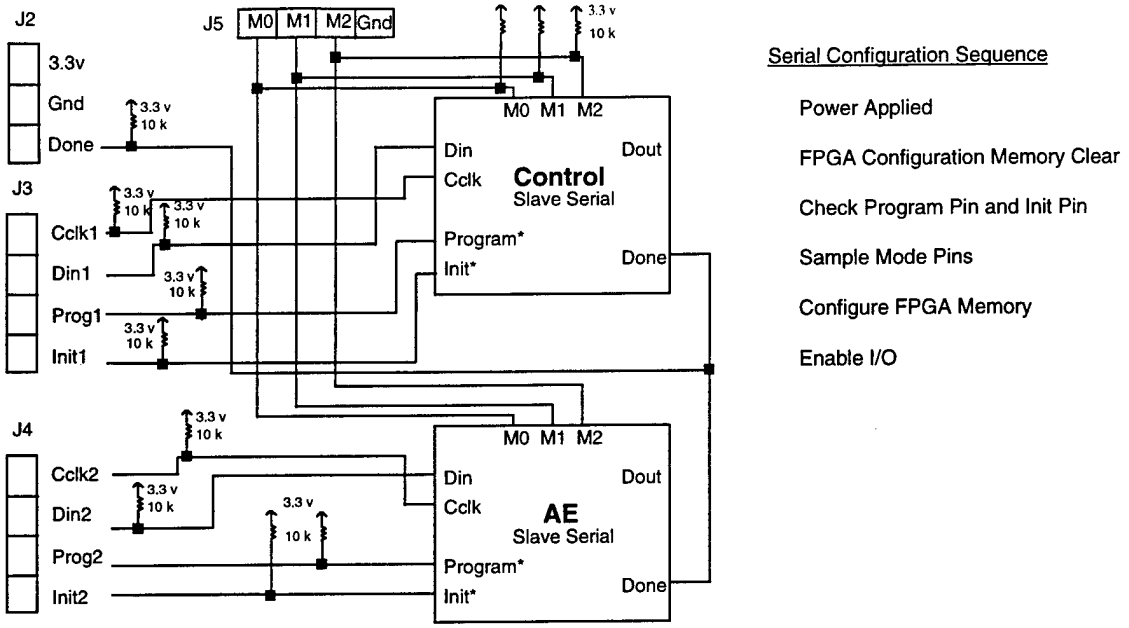


Figure 5-4. Micro-Accelerator Configuration From PC Host.

For application within a COTS system, the FPGAs can be configured from non volatile flash memory on the daughter card. This allows the daughter card to be configured for operation without any support equipment, as would be required in a fielded, flyable system. This configuration mode is shown in Figure 5-5.

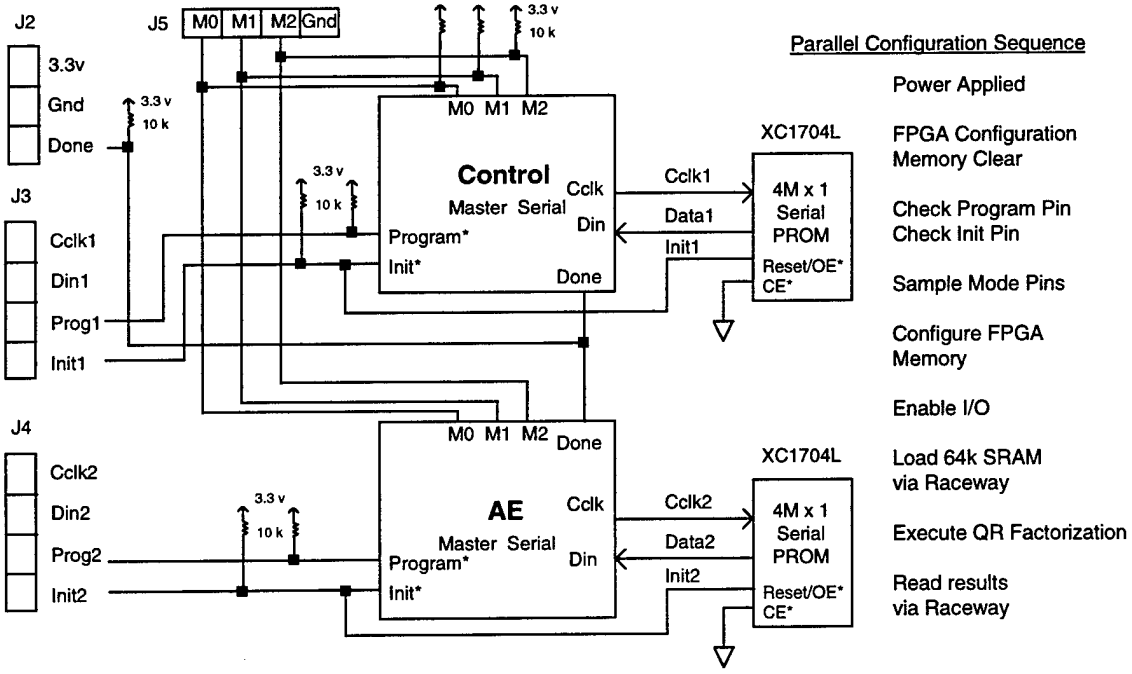


Figure 5-5. Micro-Accelerator Configuration From Non-Volatile Memory.

An additional feature of the design is the incorporation of JTAG boundary scan capability. This can be used for self test of the daughter card, and can also be used as a mechanism for loading data into the daughter card during early checkout stages. The use of JTAG for this initial data loading is shown in Figure 5-6.

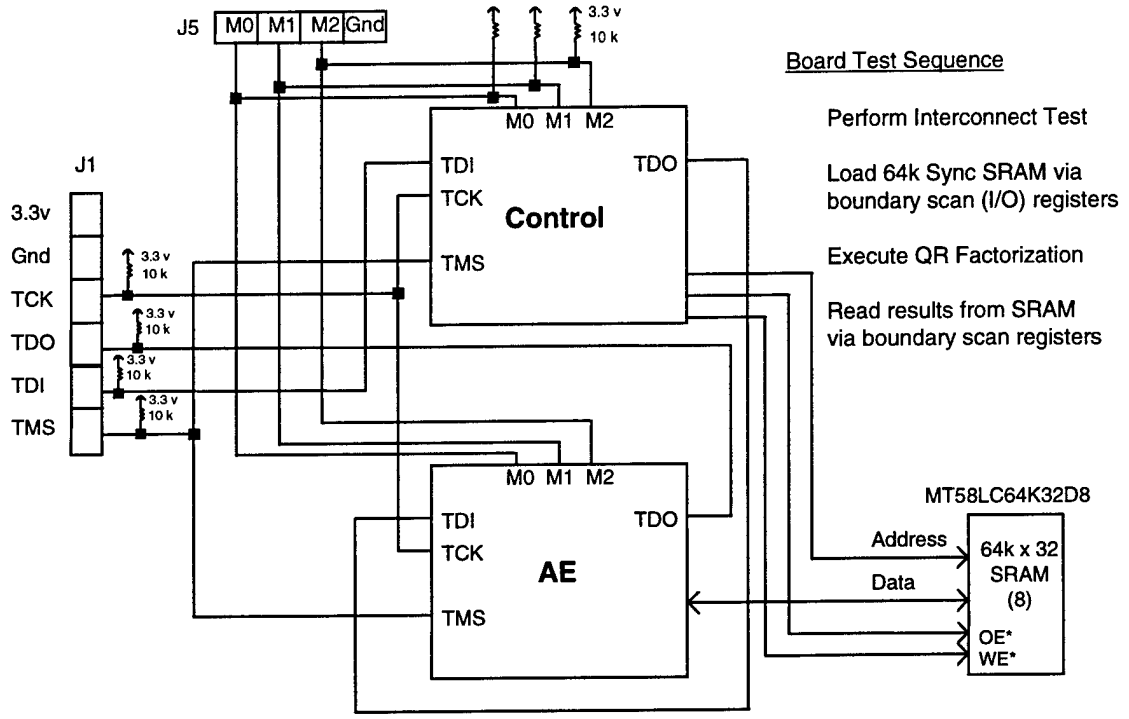


Figure 5-6. JTAG Boundary Scan Design.

The daughter card system interface to the Raceway is shown in Figure 5-7. This interface is implemented using a commercial service available from Mercury Computer Systems called RaceTrack, in which the Data Path and Sequencer chips shown are provided from Mercury, along with software and engineering support required to integrate the interface. The daughter card was designed to accommodate this interface, and assistance from Mercury was provided in checking the board to ensure correct wiring and physical design. Although the Mercury chips and software were not purchased and integrated on this contract, the board is Raceway ready and integration of the interface is anticipated on a future effort.

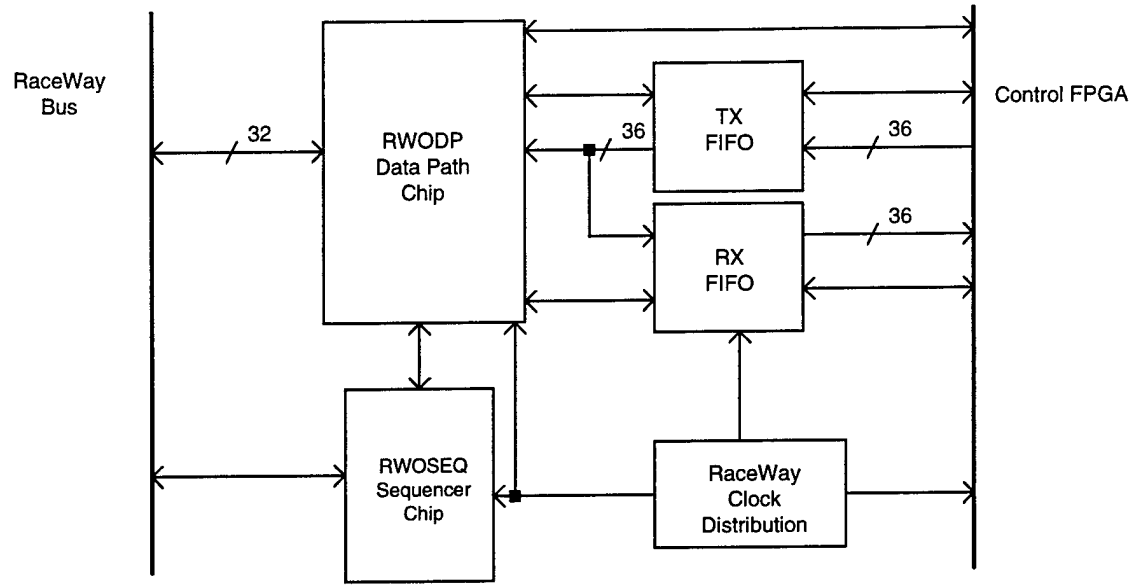


Figure 5-7. Raceway Interface to Micro-Accelerator Daughter Card.

As discussed previously, multiple clocking schemes are provided in the design to allow maximum experimentation, yet still provide an insertion ready design. The clock distribution approach is shown in Figure 5-8 The clock can be selected via the test header from two sources: an on-board oscillator and an external clock. An 80 MHz oscillator was used for the initial implementation, as that is the speed targeted for the initial QR function. This on-board clock can be changed by changing oscillators. Alternatively, an external clock source can be brought in through a mini-coaxial connector from a clock generator. This allows any clock frequency to be used, and the clock frequency to be changed without changing the oscillator part.

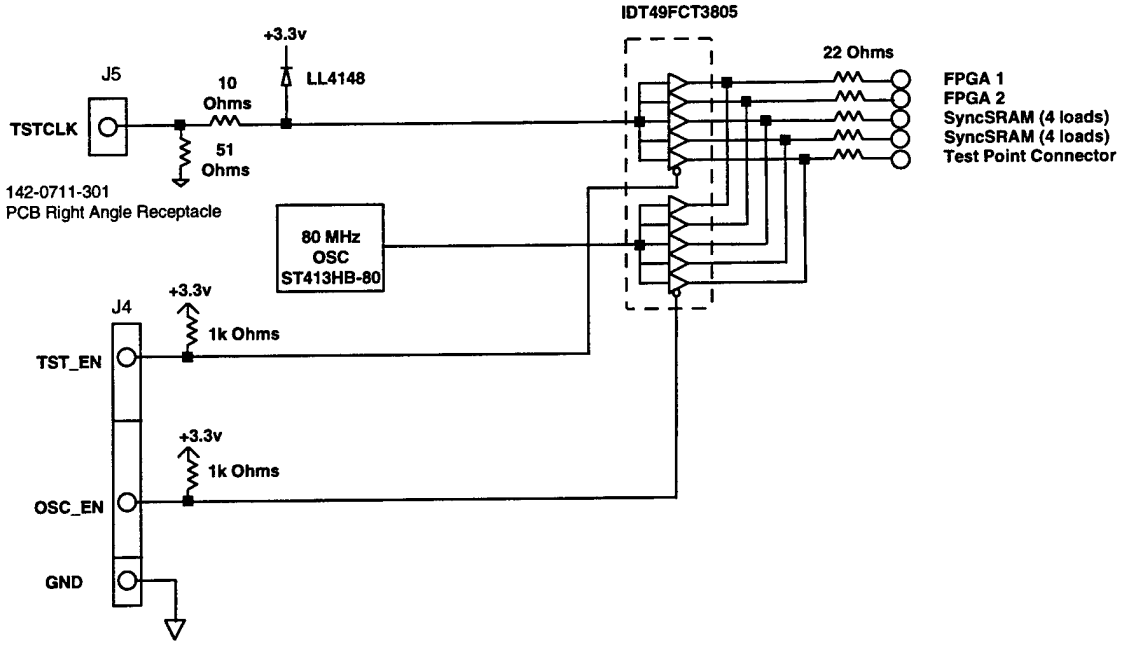


Figure 5-8. Clock Distribution Approach.

Since the FPGA technology is growing in capability very rapidly and other device technologies are also undergoing improvements according to Moore's law, the daughter card design incorporates a number of features allowing upgrades without board redesign. These growth capabilities are shown in Table 5-1.

Table 5-1. Daughter Card Design Supports Future Growth Paths.

Current Part		Future Part
Xilinx XC40125XV FPGA	=>	XC40250XV FPGA
	=>	Virtex FPGA
Micron MT58LC64K32D8 64k x 32 Sync SRAM	=>	128k x 32 Sync SRAM
SaRonix ST410H 80 Mhz Oscillator	=>	100Mhz Oscillator
XC1704L-VQ44C 4M x 1 Serial PROM	=>	8M x 1 Serial PROM

5.2 Detailed Design

The daughter card was taken through the design, layout, and board fabrication stage on this project. The parts list and power estimates are shown in Table 5-2. The FPGA devices' power dissipations depend on clock speed and utilization. Due to the 6 watt or greater dissipations projected on the daughter card, a thermal analysis was performed to determine the limits of operation. The analysis, summarized in Figure 5-9, shows that in conditions representative of a set of military environments using existing COTS processors, commercial grade FPGA parts will suffice to meet the thermal environment. The use of industrial grade parts with higher temperature ratings will enlarge the envelope of operation.

Table 5-2. Micro-Accelerator Daughter Card Parts List and Power Estimate.

Part #	Description	Mfg	Package	Pkg Height	Qty	Power	Total Power
XC40125XV-09BG560C	FPGA	Xilinx	BG560	1.7(.067)	2	6.247	12.494
MT58LC64K32D8LG-7.5	133 MHz Sync SRAM	Micron	100 TQFP	1.6(.063)	8	0.363	2.904
XC1704L-VQ44C	4M x 1 Serial PROM	Xilinx	44 VQFP	1.2(.048)	2	0.033	0.066
LT1581CT7-2.5	Voltage Regulator	Linear	TO-220	4.57(.180)	2	2	4
ST413HB-80.00	80 MHz Oscillator	SaRonix	4-pin SMD	4.7(.185)	1	0.2	0.2
IDT49FCT3805AQ	Clock Driver	IDT	SSOP20	1.6(.064)	1	0.109	0.109
EPM7128EQC100-10	FPGA	Altera	100 PQFP	3.4(.134)	1	0.8	0.8
A1460A-STD-PQ-208C	FPGA	Actel	208 PQFP	4.1(.161)	1	0.9	0.9
CY7B991-5JC	Clock Buffer	Cypress	32 PLCC	3.56(.140)	1	0.5	0.5
IDT2245LB15PF	4k x 18 Sync FIFO	IDT	64 TQFP	1.6(.063)	4	0.225	0.9
0508YC274MAT	Capacitor, 0,27 uF	AVX	2-pin	1.3(.051)	127	0	0
594D226X0020C2T	Capacitor, 22 uF	Sprague	2-pin	2.5(.098)	2	0	0
594D337X9010R2T	Capacitor, 330 uF	Sprague	2-pin	3.5(.136)	4	0	0
CRCW0402100J	Resistor, 10 ohm	Dale	SO16	0.35(0.14)	6	0	0
CRCW0402102J	Resistor, 1.0 k ohm	Dale	2-pin	0.35(0.14)	0	0	0
CRCW0402103J	Resistor, 10k ohm	Dale	2-pin	0.35(0.14)	0	0	0
CRCW0402111J	Resistor, 110 ohm	Dale	2-pin	0.35(0.14)	0	0	0
CRCW0402220J	Resistor, 22 ohm	Dale	2-pin	0.35(0.14)	0	0	0
CRCW0402221J	Resistor, 220 ohm	Dale	2-pin	0.35(0.14)	0	0	0
CRCW0402510J	Resistor, 51 ohm	Dale	2-pin	0.35(0.14)	0	0	0
LL4148	Diode, Surface Mount	AVX	2-pin	0.5(0.2)	1	0	0
142-0711-301	PCB Right Angle Recep	Amphenol	5-pin	9.53(.375)	1	0	0
TS-120-G-A-1	32 pin Terminal Strip	SamTec	32 pin	4.22(.166)	1	0	0
8903-080-177-MS-D-A	Mercury I/F Connector	KEL	80 pin	1.7(.067)	1	0	0
3416D0F00000	Thin-Fin Heat Sink	AAvid	Heat Sink	12.7(0.5)	2	0	0
Package Height in mm(inches)				Total Power (W)			22.873

Xilinx XC40125XV-560BGA

T_j max (plastic) = 125° C

T_j operation (commercial) = 85° C

T_j operation (industrial) = 100° C

P_{diss}= 6w

T_{air} = 40° C (Mercury Spec)

Condition	$Theta_{ja}$	DeltaT, °C	Tj, °C
Still Air,Max	10	60	100
Still Air,Typ	9	54	94
Still Air,Min	8	48	88
250 LFM	7	42	82
500 LFM	6	36	76
750 LFM	5	30	70

Mercury Chassis Conditions ⇨

Figure 5-10. Thermal Analysis Summary.

The daughter card detailed design was completed and layout and routing were performed. The daughter card layout is shown in Figure 5-11. A double sided construction was used to provide high circuit density. The final board design is implemented with 14 metal layers.

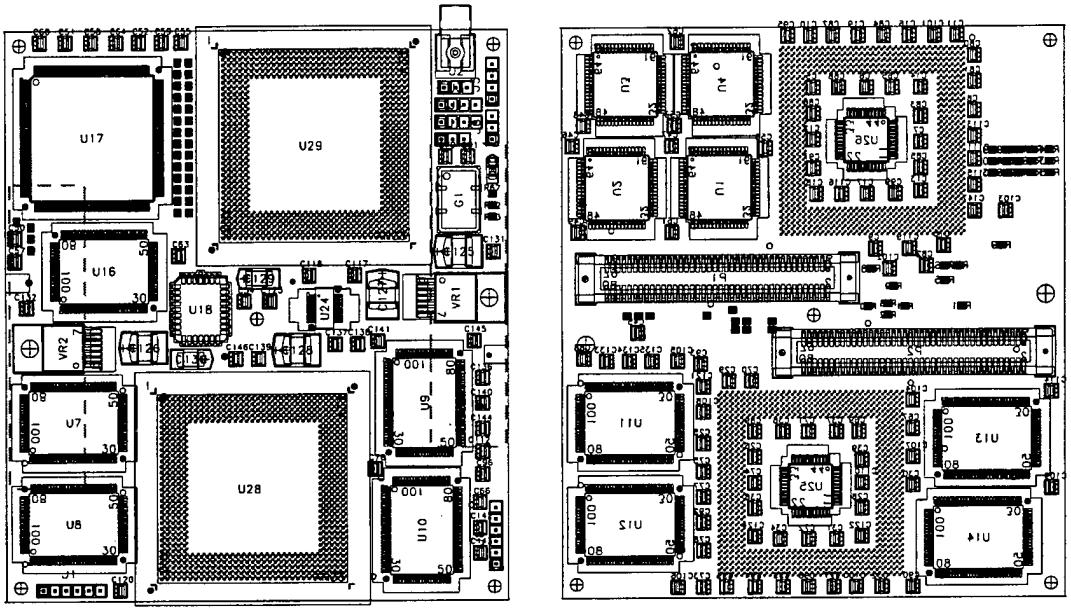


Figure 5-11. Micro-Accelerator Daughter Card Layout.

6. CONCLUSIONS

Based on the efforts on this project, the following major conclusions can be drawn:

- System requirements for signal processing on future military sensors will continue to grow. Future systems will be processing-limited if processing growth increases only at the rate predicted by Moore's law.
- Typically the algorithms of advanced sensor systems are driven by only a few high throughput functions, often comprising 50 to 90% of the total throughput.
- Often, the driving functions can be performed with less precision than the standard IEEE 754 single precision (32 bit) floating point format provides. This must be determined on an application by application basis.
- The rapid rate of growth in reconfigurable device (especially FPGA) capabilities will soon allow one million gates on a device with clock speeds upwards of 100 MHz.
- Implementing the driving functions in sensor processing very efficiently by targeting the hardware configuration specifically to the problem to minimize control overheads and by using the smallest word size and minimum precision required for the application can provide orders of magnitude higher throughput in a high density FPGA versus a similar generation DSP or microprocessor for selected algorithmic functions.
- A heterogeneous processing system with the proper mix of reconfigurable nodes and software programmable nodes can provide nearly an order of magnitude reduction at the chassis level in processing size, weight, and power for the applicable sensor algorithms.

However, to implement the above heterogeneous approach on a large scale, additional tools and techniques are still needed. The primary development areas still required to incorporate this capability into the mainstream are:

- Methods of automating the determination of minimum precision needed for the various stages of the algorithm computations. The simulation methods using MATLAB in this project were time consuming, and a full precision analysis for even one application area (STAP QR) was beyond the scope of the project using available methods.
- Automation of the optimum generation of variable precision arithmetic elements in FPGA devices. Current vendor libraries are insufficiently complete to fully exploit reduced precision algorithms in an efficient manner. The creation of custom FPGA configurations and optimization of clock speed versus pipelines is a labor intensive task, but the current state of the art still requires an experienced human designer to conduct the optimization trades and develop the optimum configurations.
- Availability of flyable COTS micro-accelerator cards which are compatible with vendors' processing systems. Discussion with COTS vendors about this approach have met with lukewarm responses, as successful incorporation will reduce the amount of products sold by the vendor. There is no incentive to support such an approach until forced to do so by competitive pressures.

- A system level approach for programming the entire heterogeneous system so that the mapping of various algorithm functions to the optimum type of processing node is accomplished either automatically or semi-automatically. Currently, an "expert' is required to partition the problem and iterate to find the optimum point.

Despite the requirement for additional development, the micro-accelerator approach has promise to provide significant benefits for military sensor processing. Active pursuit of the remaining technology hurdles should be pursued.

APPENDIX. VHDL Listings of Arithmetic Elements

A.1 VHDL Listings for 5 Pipeline Stage 16 bit Floating Point Multiplier

```
--- ------------------------------------------------
-- Floating Point Multiply
-- Configurable Micro-Accelerator for QR Factorization
-- FPMULTP5.VHD
-- ------------------------------------------------
-- Steven Gercken
-- Created: 2/6/98
-- Revised:
-- ------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity FPMULTp5 is
   port(Clock          : in std_logic;
       Resetn         : in std_logic;
       SignA          : in std_logic;
       SignB          : in std_logic;
       MantissaA      : in std_logic_vector(9 downto 0);
       MantissaB      : in std_logic_vector(9 downto 0);
       ExponentA      : in std_logic_vector(4 downto 0);
       ExponentB      : in std_logic_vector(4 downto 0);
       Sign_Out       : out std_logic;
       Mant_Out       : out std_logic_vector(9 downto 0);
       Exp_Out        : out std_logic_vector(4 downto 0)
       );
end FPMULTp5;

architecture BEHAVIOR of FPMULTp5 is

-- Declare the black_box as a boolean attribute
attribute black_box: boolean;

------------------------------------------------------------
-- Components
------------------------------------------------------------
```

```
component M11x11u4
    port (Resetn     : in  std_logic;
          Clock      : in  std_logic;
          a          : in  std_logic_vector(10 downto 0);
          b          : in  std_logic_vector(10 downto 0);
          Product    : out std_logic_vector(21 downto 0));
end component;

--attribute black_box of Mul11x11u:  component is true;

component MUX4_5
    port(InA      : in std_logic_vector(4 downto 0);
         InB      : in std_logic_vector(4 downto 0);
         InC      : in std_logic_vector(4 downto 0);
         InD      : in std_logic_vector(4 downto 0);
         Sel0     : in std_logic;
         Sel1     : in std_logic;
         DataOut  : out std_logic_vector(4 downto 0));
end component;

component MUX4_10
    port(InA      : in std_logic_vector(9 downto 0);
         InB      : in std_logic_vector(9 downto 0);
         InC      : in std_logic_vector(9 downto 0);
         InD      : in std_logic_vector(9 downto 0);
         Sel0     : in std_logic;
         Sel1     : in std_logic;
         DataOut  : out std_logic_vector(9 downto 0));
end component;



signal SignA_Q        : std_logic;
signal SignB_Q        : std_logic;
signal ExponentA_Q    : std_logic_vector(4 downto 0);
signal ExponentB_Q    : std_logic_vector(4 downto 0);
signal MantissaA_Q    : std_logic_vector(9 downto 0);
signal MantissaB_Q    : std_logic_vector(9 downto 0);

signal Bias_Correction : std_logic_vector(5 downto 0);

signal ExponentA_DD   : std_logic_vector(4 downto 0);
signal ExponentB_DD   : std_logic_vector(4 downto 0);
signal ExponentA_QQ   : std_logic_vector(4 downto 0);
signal ExponentB_QQ   : std_logic_vector(4 downto 0);
```

```
signal Exp_Zero       : std_logic_vector(4 downto 0);
signal Exp_Max        : std_logic_vector(4 downto 0);
signal Exp5_Sum       : std_logic_vector(5 downto 0);
signal Exp5_Sum_Q     : std_logic_vector(5 downto 0);
signal Exp5_Sum_DD    : std_logic_vector(5 downto 0);
signal Exp5_Sum_QQ    : std_logic_vector(5 downto 0);
signal Exp5_Sum_DDD   : std_logic_vector(5 downto 0);
signal Exp5_Sum_QQQ   : std_logic_vector(5 downto 0);
signal Exp6_Sum       : std_logic_vector(6 downto 0);
signal Exp6_Sum_Q     : std_logic_vector(4 downto 0);
signal ExpA_Zero      : std_logic;
signal ExpB_Zero      : std_logic;
signal ExpA_Zero_Q    : std_logic;
signal ExpB_Zero_Q    : std_logic;
signal ExpA_Zero_DD   : std_logic;
signal ExpB_Zero_DD   : std_logic;
signal ExpA_Zero_QQ   : std_logic;
signal ExpB_Zero_QQ   : std_logic;
signal ExpA_Zero_DDD  : std_logic;
signal ExpB_Zero_DDD  : std_logic;
signal ExpA_Zero_QQQ  : std_logic;
signal ExpB_Zero_QQQ  : std_logic;
signal Overflow       : std_logic;
signal Underflow      : std_logic;
signal Exp_Mux_Sel0   : std_logic;
signal Exp_Mux_Sel1   : std_logic;
signal Exp_Mux_Sel0_Q : std_logic;
signal Exp_Mux_Sel1_Q : std_logic;

signal MantA          : std_logic_vector(10 downto 0);
signal MantB          : std_logic_vector(10 downto 0);
signal ImpliedOne     : std_logic;
signal Product        : std_logic_vector(21 downto 0);
signal Mant_Mux_Sel0  : std_logic;
signal Mant_Mux_Sel1  : std_logic;
signal Mant_Mux_Sel0_Q : std_logic;
signal Mant_Mux_Sel1_Q : std_logic;
signal Mant_Zero      : std_logic_vector(9 downto 0);
signal Mant_Max       : std_logic_vector(9 downto 0);

signal SignA_DD       : std_logic;
signal SignB_DD       : std_logic;
signal SignA_QQ       : std_logic;
signal SignB_QQ       : std_logic;
signal SignOut        : std_logic;
```

```
signal SignOut1_Q     : std_logic;
signal SignOut2_D     : std_logic;
signal SignOut2_Q     : std_logic;
signal SignOut3_D     : std_logic;
signal SignOut3_Q     : std_logic;
signal SignOut4_D     : std_logic;
signal SignOut4_Q     : std_logic;
signal ExpOut         : std_logic_vector(4 downto 0);
signal MantOut        : std_logic_vector(9 downto 0);

signal Sign_Q         : std_logic;
signal Exp_Q          : std_logic_vector(4 downto 0);
signal Mant_Q         : std_logic_vector(9 downto 0);

signal Product_Q      : std_logic_vector(20 downto 10);

begin

-- Register fpmult inputs
 process (Resetn,Clock)
  begin
  if (Resetn = '0') then
      SignA_Q <= '0';
      SignB_Q <= '0';
      MantissaA_Q <= (others => '0');
      MantissaB_Q <= (others => '0');
      ExponentA_Q <= (others => '0');
      ExponentB_Q <= (others => '0');
   else
    if (Clock'event and Clock = '1') then
      SignA_Q <= SignA;
      SignB_Q <= SignB;
      MantissaA_Q <= MantissaA;
      MantissaB_Q <= MantissaB;
      ExponentA_Q <= ExponentA;
      ExponentB_Q <= ExponentB;
    end if;
   end if;
 end process;

-- Sign Logic

 SignOut <= SignA_QQ xor SignB_QQ;

-- Exponent Logic
```

```
 Exp5_Sum <= ( '0' & ExponentA_QQ)  + ( '0' & ExponentB_QQ);

 ExpA_Zero  <= '1' when (ExponentA_QQ = "00000")
           else '0';

 ExpB_Zero  <= '1' when (ExponentB_QQ = "00000")
           else '0';


-- Pipeline Register

 SignA_DD    <= SignA_Q;
 SignB_DD    <= SignB_Q;
 ExponentA_DD <= ExponentA_Q;
 ExponentB_DD <= ExponentB_Q;

 process (Resetn,Clock)
  begin
  if (Resetn = '0') then
      SignA_QQ <= '0';
      SignB_QQ <= '0';
      SignOut1_Q <= '0';
      SignOut2_Q <= '0';
      SignOut3_Q <= '0';
      SignOut4_Q <= '0';
      ExponentA_QQ <= (others => '0');
      ExponentB_QQ <= (others => '0');
      Exp5_Sum_Q <= (others => '0');
      Exp5_Sum_QQ <= (others => '0');
      Exp5_Sum_QQQ <= (others => '0');
      ExpA_Zero_Q <= '0';
      ExpB_Zero_Q <= '0';
      ExpA_Zero_QQ <= '0';
      ExpB_Zero_QQ <= '0';
      ExpA_Zero_QQQ <= '0';
      ExpB_Zero_QQQ <= '0';
      Exp_Mux_Sel0_Q <= '0';
      Exp_Mux_Sel1_Q <= '0';
      Mant_Mux_Sel0_Q <= '0';
      Mant_Mux_Sel1_Q <= '0';
   else
   if (Clock'event and Clock = '1') then
      SignA_QQ <= SignA_DD;
      SignB_QQ <= SignB_DD;
```

```
      SignOut1_Q <= SignOut;
      SignOut2_Q <= SignOut2_D;
      SignOut3_Q <= SignOut3_D;
      SignOut4_Q <= SignOut4_D;
      ExponentA_QQ <= ExponentA_DD;
      ExponentB_QQ <= ExponentB_DD;
      Exp5_Sum_Q <= Exp5_Sum;
      Exp5_Sum_QQ <= Exp5_Sum_DD;
      Exp5_Sum_QQQ <= Exp5_Sum_DDD;
      ExpA_Zero_Q <= ExpA_Zero;
      ExpB_Zero_Q <= ExpB_Zero;
      ExpA_Zero_QQ <= ExpA_Zero_DD;
      ExpB_Zero_QQ <= ExpB_Zero_DD;
      ExpA_Zero_QQQ <= ExpA_Zero_DDD;
      ExpB_Zero_QQQ <= ExpB_Zero_DDD;
      Exp_Mux_Sel0_Q <= Exp_Mux_Sel0;
      Exp_Mux_Sel1_Q <= Exp_Mux_Sel1;
      Mant_Mux_Sel0_Q <= Mant_Mux_Sel0;
      Mant_Mux_Sel1_Q <= Mant_Mux_Sel1;
    end if;
  end if;
end process;

SignOut2_D <= SignOut1_Q;
SignOut3_D <= SignOut2_Q;
SignOut4_D <= SignOut3_Q;
Exp5_Sum_DD <= Exp5_Sum_Q;
Exp5_Sum_DDD <= Exp5_Sum_QQ;
ExpA_Zero_DD <= ExpA_Zero_Q;
ExpB_Zero_DD <= ExpB_Zero_Q;
ExpA_Zero_DDD <= ExpA_Zero_QQ;
ExpB_Zero_DDD <= ExpB_Zero_QQ;

Bias_Correction <= "110001";
Exp_Max         <= "11111";
Exp_Zero        <= "00000";


Exp6_Sum    <= ('0' & Exp5_Sum_QQQ) + ('0' & Bias_Correction) + Product(21);


Overflow    <= Exp5_Sum_QQQ(5) and Exp6_Sum(5);

Underflow   <= not(Exp5_Sum_QQQ(5)) and Exp6_Sum(5);
```

```
Exp_Mux_Sel0 <= Overflow;

Exp_Mux_Sel1 <= ExpA_Zero_QQQ or ExpB_Zero_QQQ or Underflow;


Exp_Mux: MUX4_5 port map
    (InA       => Exp6_Sum_Q(4 downto 0),
     InB       => Exp_Max,
     InC       => Exp_Zero,
     InD       => Exp_Zero,
     Sel0      => Exp_Mux_Sel0_Q,
     Sel1      => Exp_Mux_Sel1_Q,
     DataOut   => ExpOut);


-- Mantissa Logic

ImpliedOne   <= '1';
Mant_Max     <= "1111111111";
Mant_Zero    <= "0000000000";


MantA  <= ImpliedOne & MantissaA_Q;

MantB  <= ImpliedOne & MantissaB_Q;


FIXEDMULT: M11x11u4 port map
  (Resetn       => Resetn,
   Clock        => Clock,
   a            => MantA,
   b            => MantB,
   Product      => Product);


Mant_Mux_Sel0 <= (not(Overflow) and Product(21)) or Exp_Mux_Sel1;

Mant_Mux_Sel1 <= Overflow or Exp_Mux_Sel1;

Mant_Mux: MUX4_10 port map
    (InA       => Product_Q(19 downto 10),
     InB       => Product_Q(20 downto 11),
     InC       => Mant_Max,
     InD       => Mant_Zero,
     Sel0      => Mant_Mux_Sel0_Q,
```

```
      Sel1      => Mant_Mux_Sel1_Q,
      DataOut   => MantOut);


-- Register
 process (Resetn,Clock)
  begin
  if (Resetn = '0') then
      Sign_Q    <= '0';
      Exp_Q     <= (others => '0');
      Mant_Q    <= (others => '0');
      Product_Q <= (others => '0');
      Exp6_Sum_Q <= (others => '0');
   else
    if (Clock'event and Clock = '1') then
      Sign_Q    <= SignOut4_Q;
      Exp_Q     <= ExpOut;
      Mant_Q    <= MantOut;
      Product_Q <= Product(20 downto 10);
      Exp6_Sum_Q <= Exp6_Sum(4 downto 0);
    end if;
   end if;
 end process;

 Sign_Out   <= Sign_Q;
 Exp_Out    <= Exp_Q;
 Mant_Out   <= Mant_Q;


end BEHAVIOR;
```

```
--- ------------------------------------------------
-- 4:1 Mux (5-bit data bus)
-- Configurable Micro-Accelerator for QR Factorization
-- Mux4_5.VHD
-- ------------------------------------------------
-- Steven Gercken
-- Created: 12/05/97
-- Revised:
-- ------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity MUX4_5 is
   port(InA          : in std_logic_vector(4 downto 0);
      InB          : in std_logic_vector(4 downto 0);
      InC          : in std_logic_vector(4 downto 0);
      InD          : in std_logic_vector(4 downto 0);
      Sel0         : in std_logic;
      Sel1         : in std_logic;
      DataOut      : out std_logic_vector(4 downto 0)
      );
end MUX4_5;

architecture BEHAVIOR of MUX4_5 is


signal Mux_Sel        : std_logic_vector(1 downto 0);

begin

 Mux_Sel <= Sel1 & Sel0;

 process (InA, InB, InC, InD, Mux_Sel)
  begin
   case Mux_Sel is

       when "00" =>
       DataOut <= InA;
       when "01" =>
       DataOut <= InB;
       when "10" =>
```

```
        DataOut <= InC;
        when "11" =>
        DataOut <= InD;
      when others =>
        DataOut <= "00000";

    end case;
  end process;

end Behavior;
```

```
--- ------------------------------------------------
-- 4:1 Mux (10-bit data bus)
-- Configurable Micro-Accelerator for QR Factorization
-- Mux4_10.VHD
-- ------------------------------------------------
-- Steven Gercken
-- Created: 12/05/97
-- Revised:
-- ------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity MUX4_10 is
  port(InA          : in std_logic_vector(9 downto 0);
       InB          : in std_logic_vector(9 downto 0);
       InC          : in std_logic_vector(9 downto 0);
       InD          : in std_logic_vector(9 downto 0);
       Sel0         : in std_logic;
       Sel1         : in std_logic;
       DataOut      : out std_logic_vector(9 downto 0)
       );
end MUX4_10;

architecture BEHAVIOR of MUX4_10 is

signal Mux_Sel         : std_logic_vector(1 downto 0);

begin

 Mux_Sel <= Sel1 & Sel0;

 process (InA, InB, InC, InD, Mux_Sel)
  begin
   case Mux_Sel is

       when "00" =>
       DataOut <= InA;
       when "01" =>
       DataOut <= InB;
       when "10" =>
       DataOut <= InC;
```

```
        when "11" =>
          DataOut <= InD;
      when others =>
          DataOut <= (others => '0');

    end case;
  end process;

end Behavior;
```

```
--- ------------------------------------------------
-- Fixed Point 11 x 11 Multiply - Four Pipes
-- Configurable Micro-Accelerator for QR Factorization
-- m11x11u4.VHD
-- ------------------------------------------------
-- Steven Gercken
-- Created: 2/6/98
-- Revised:
-- ------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
--library altera;
--use altera.maxplus2.all;
--use util.all;



entity m11x11u4 is
   port(Resetn   : in std_logic;
        Clock    : in std_logic;
        a        : in STD_LOGIC_VECTOR (10 downto 0);
        b        : in STD_LOGIC_VECTOR (10 downto 0);
        Product  : out STD_LOGIC_VECTOR (21 downto 0)
   );
end m11x11u4;

architecture BEHAVIOR of m11x11u4 is

signal P_c0      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c1      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c2      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c3      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c4      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c5      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c6      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c7      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c8      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c9      : STD_LOGIC_VECTOR (10 downto 0);
signal P_c10     : STD_LOGIC_VECTOR (10 downto 0);
signal P_c10_Q   : STD_LOGIC_VECTOR (10 downto 0);
```

```
signal P_c0_1    : STD_LOGIC_VECTOR (12 downto 0);
signal P_c2_3    : STD_LOGIC_VECTOR (12 downto 0);
signal P_c4_5    : STD_LOGIC_VECTOR (12 downto 0);
signal P_c6_7    : STD_LOGIC_VECTOR (12 downto 0);
signal P_c8_9    : STD_LOGIC_VECTOR (12 downto 0);

signal P_c0_3    : STD_LOGIC_VECTOR (14 downto 0);
signal P_c4_7    : STD_LOGIC_VECTOR (14 downto 0);
signal P_c8_10   : STD_LOGIC_VECTOR (13 downto 0);

signal P_c4_10   : STD_LOGIC_VECTOR (17 downto 0);
signal P_c4_10_Q : STD_LOGIC_VECTOR (17 downto 0);


signal P_c0_1_Q  : STD_LOGIC_VECTOR (12 downto 0);
signal P_c2_3_Q  : STD_LOGIC_VECTOR (12 downto 0);
signal P_c4_5_Q  : STD_LOGIC_VECTOR (12 downto 0);
signal P_c6_7_Q  : STD_LOGIC_VECTOR (12 downto 0);
signal P_c8_9_Q  : STD_LOGIC_VECTOR (12 downto 0);

signal P_c0_3_Q  : STD_LOGIC_VECTOR (14 downto 0);
signal P_c0_3_QQ : STD_LOGIC_VECTOR (14 downto 0);
signal P_c4_7_Q  : STD_LOGIC_VECTOR (14 downto 0);
signal P_c8_10_Q : STD_LOGIC_VECTOR (13 downto 0);

signal Product_D : STD_LOGIC_VECTOR (21 downto 0);
signal Product_Q : STD_LOGIC_VECTOR (21 downto 0);


begin

-- Partial Products
P_c0(10 downto 0) <= a(10 downto 0) and b(0) & b(0) & b(0) & b(0) & b(0) & b(0) & b(0) &
b(0) & b(0) & b(0) & b(0);

P_c1(10 downto 0) <= a(10 downto 0) and b(1) & b(1) & b(1) & b(1) & b(1) & b(1) & b(1) &
b(1) & b(1) & b(1) & b(1);

P_c2(10 downto 0) <= a(10 downto 0) and b(2) & b(2) & b(2) & b(2) & b(2) & b(2) & b(2) &
b(2) & b(2) & b(2) & b(2);

P_c3(10 downto 0) <= a(10 downto 0) and b(3) & b(3) & b(3) & b(3) & b(3) & b(3) & b(3) &
b(3) & b(3) & b(3) & b(3);
```

```
P_c4(10 downto 0)  <= a(10 downto 0) and b(4) & b(4) & b(4) & b(4) & b(4) & b(4) & b(4) &
b(4) & b(4) & b(4) & b(4);

P_c5(10 downto 0)  <= a(10 downto 0) and b(5) & b(5) & b(5) & b(5) & b(5) & b(5) & b(5) &
b(5) & b(5) & b(5) & b(5);

P_c6(10 downto 0)  <= a(10 downto 0) and b(6) & b(6) & b(6) & b(6) & b(6) & b(6) & b(6) &
b(6) & b(6) & b(6) & b(6);

P_c7(10 downto 0)  <= a(10 downto 0) and b(7) & b(7) & b(7) & b(7) & b(7) & b(7) & b(7) &
b(7) & b(7) & b(7) & b(7);

P_c8(10 downto 0)  <= a(10 downto 0) and b(8) & b(8) & b(8) & b(8) & b(8) & b(8) & b(8) &
b(8) & b(8) & b(8) & b(8);

P_c9(10 downto 0)  <= a(10 downto 0) and b(9) & b(9) & b(9) & b(9) & b(9) & b(9) & b(9) &
b(9) & b(9) & b(9) & b(9);

P_c10(10 downto 0) <= a(10 downto 0) and b(10) & b(10) & b(10) & b(10) & b(10) & b(10) &
b(10) & b(10) & b(10) & b(10) & b(10);


-- Level 1

P_c0_1(12 downto 0) <= ("00" & P_c0) + ('0' & P_c1 & '0');
P_c2_3(12 downto 0) <= ("00" & P_c2) + ('0' & P_c3 & '0');
P_c4_5(12 downto 0) <= ("00" & P_c4) + ('0' & P_c5 & '0');
P_c6_7(12 downto 0) <= ("00" & P_c6) + ('0' & P_c7 & '0');
P_c8_9(12 downto 0) <= ("00" & P_c8) + ('0' & P_c9 & '0');

-- Register Level 1 outputs
 process (Resetn,Clock)
  begin
  if (Resetn = '0') then
      P_c0_1_Q  <= (others => '0');
      P_c2_3_Q  <= (others => '0');
      P_c4_5_Q  <= (others => '0');
      P_c6_7_Q  <= (others => '0');
      P_c8_9_Q  <= (others => '0');
      P_c10_Q   <= (others => '0');
   else
    if (Clock'event and Clock = '1') then
      P_c0_1_Q  <= P_c0_1;
      P_c2_3_Q  <= P_c2_3;
      P_c4_5_Q  <= P_c4_5;
```

```
        P_c6_7_Q  <= P_c6_7;
        P_c8_9_Q  <= P_c8_9;
        P_c10_Q   <= P_c10;
      end if;
     end if;
  end process;

-- Level 2a

P_c0_3(14 downto 0) <= ("00" & P_c0_1_Q) + (P_c2_3_Q & "00");
P_c4_7(14 downto 0) <= ("00" & P_c4_5_Q) + (P_c6_7_Q & "00");

-- Level 2b

P_c8_10(13 downto 0)        <= ('0'  & P_c8_9_Q) + (P_c10_Q  & "000");


-- Register Level 2a and Level 2b outputs
  process (Resetn,Clock)
    begin
    if (Resetn = '0') then
        P_c0_3_Q  <= (others => '0');
        P_c4_7_Q  <= (others => '0');
        P_c8_10_Q <= (others => '0');
     else
      if (Clock'event and Clock = '1') then
        P_c0_3_Q  <= P_c0_3;
        P_c4_7_Q  <= P_c4_7;
        P_c8_10_Q <= P_c8_10;
      end if;
     end if;
  end process;

-- Level 3

P_c4_10(17 downto 0)        <= ("000" & P_c4_7_Q) + (P_c8_10_Q & "0000");

-- Register Level 3 outputs
  process (Resetn,Clock)
    begin
    if (Resetn = '0') then
        P_c0_3_QQ  <= (others => '0');
        P_c4_10_Q <= (others => '0');
     else
      if (Clock'event and Clock = '1') then
```

```
      P_c0_3_QQ <= P_c0_3_Q;
      P_c4_10_Q <= P_c4_10;
    end if;
   end if;
 end process;

-- Level 4

Product_D          <= ("0000000" & P_c0_3_QQ) + (P_c4_10_Q & "0000");

-- Register Product outputs
 process (Resetn,Clock)
  begin
  if (Resetn = '0') then
      Product_Q <= (others => '0');
   else
    if (Clock'event and Clock = '1') then
      Product_Q <= Product_D;
    end if;
   end if;
 end process;


Product <= Product_Q;

end BEHAVIOR;
```

A.2 VHDL Listings for 6 Pipeline Stage 16 bit Floating Point Adder

```
--- ------------------------------------------------
-- Floating Point Adder Subtractor
-- Configurable Micro-Accelerator for QR Factorization
-- fpadd6.VHD
-- ------------------------------------------------
-- Steven Gercken
-- Created: 1/28/98
-- Revised:
-- ------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity fpadd6 is
   port(Clock          : in std_logic;
      Resetn          : in std_logic;
      Func              : in std_logic;
      SignA           : in std_logic;
      SignB           : in std_logic;
      MantissaA       : in std_logic_vector(9 downto 0);
      MantissaB       : in std_logic_vector(9 downto 0);
      ExponentA       : in std_logic_vector(4 downto 0);
      ExponentB       : in std_logic_vector(4 downto 0);
      Sign_Out        : out std_logic;
      Mant_Out        : out std_logic_vector(9 downto 0);
      Exp_Out         : out std_logic_vector(4 downto 0)
      );
end fpadd6;

architecture BEHAVIOR of fpadd6 is


component MUX4_5
   port(InA          : in std_logic_vector(4 downto 0);
      InB          : in std_logic_vector(4 downto 0);
      InC          : in std_logic_vector(4 downto 0);
      InD          : in std_logic_vector(4 downto 0);
      Sel0         : in std_logic;
      Sel1         : in std_logic;
      DataOut      : out std_logic_vector(4 downto 0)
```

```
    );
end component;

component MUX4_10
  port(InA          : in std_logic_vector(9 downto 0);
    InB          : in std_logic_vector(9 downto 0);
    InC          : in std_logic_vector(9 downto 0);
    InD          : in std_logic_vector(9 downto 0);
    Sel0         : in std_logic;
    Sel1         : in std_logic;
    DataOut      : out std_logic_vector(9 downto 0)
    );
end component;

component swap
    port(Clock      : in std_logic;
      Resetn     : in std_logic;
      ExponentA   : in std_logic_vector(4 downto 0);
      ExponentB   : in std_logic_vector(4 downto 0);
      MantissaA   : in std_logic_vector(9 downto 0);
      MantissaB   : in std_logic_vector(9 downto 0);
      SignA       : in std_logic;
      SignB       : in std_logic;
      Exp_Temp1   : in std_logic_vector(4 downto 0);
      Exp_Temp2   : in std_logic_vector(4 downto 0);
      AeqB        : in std_logic_vector(0 downto 0);
      Func        : in std_logic;
      ExponentL   : out std_logic_vector(4 downto 0);
      ExponentS   : out std_logic_vector(4 downto 0);
      MantissaL   : out std_logic_vector(9 downto 0);
      MantissaS   : out std_logic_vector(9 downto 0);
      SignL       : out std_logic;
      SignS       : out std_logic;
      Shift_Sel   : out std_logic_vector(4 downto 0);
      Func_Q      : out std_logic
      );
end component;

component smshift
    port(Clock      : in std_logic;
      Resetn     : in std_logic;
      Shift_Sel   : in std_logic_vector(4 downto 0);
      MantissaS   : in std_logic_vector(9 downto 0);
      MantS       : out std_logic_vector(12 downto 0)
      );
```

```
end component;

component exposel3
    port(MantSum      : in std_logic_vector(13 downto 0);
       Exp_update   : out std_logic_vector(4 downto 0);
       MuxSel      : out std_logic_vector(3 downto 0);
       Exp_out_zero  : out std_logic
       );
end component;

component mantmux
    port(MantSum      : in std_logic_vector(13 downto 0);
       MuxSel      : in std_logic_vector(3 downto 0);
       MantOut     : out std_logic_vector(9 downto 0)
       );
end component;


signal Func_Q          : std_logic;
signal Func_Q1         : std_logic;
signal SignA_Q         : std_logic;
signal SignB_Q         : std_logic;
signal ExponentA_Q     : std_logic_vector(4 downto 0);
signal ExponentB_Q     : std_logic_vector(4 downto 0);
signal MantissaA_Q     : std_logic_vector(9 downto 0);
signal MantissaB_Q     : std_logic_vector(9 downto 0);

signal SignL           : std_logic;
signal SignS           : std_logic;
signal SignOut         : std_logic;
signal SignOut_Q2      : std_logic;
signal SignOut_Q3      : std_logic;
signal SignOut_Q4      : std_logic;
signal SignOut_Q5      : std_logic;
signal SignOut_Q6      : std_logic;
signal AddSel          : std_logic;
signal AddSel_Q2       : std_logic;
signal AddSel_Q3       : std_logic;

signal ExponentL       : std_logic_vector(4 downto 0);
signal ExponentL_Q2    : std_logic_vector(4 downto 0);
signal ExponentL_Q3    : std_logic_vector(4 downto 0);
signal ExponentL_Q4    : std_logic_vector(4 downto 0);
signal ExponentL_Q5    : std_logic_vector(4 downto 0);
signal ExponentL_Q6    : std_logic_vector(4 downto 0);
```

```
signal ExponentS       : std_logic_vector(4 downto 0);

signal Exp_Temp1       : std_logic_vector(5 downto 0);
signal Exp_Temp2       : std_logic_vector(5 downto 0);
signal Shift_Sel       : std_logic_vector(4 downto 0);
signal AeqB            : std_logic_vector(0 downto 0);

signal MantissaL       : std_logic_vector(9 downto 0);
signal MantissaL_Q2    : std_logic_vector(9 downto 0);
signal MantissaS       : std_logic_vector(9 downto 0);
signal MantL           : std_logic_vector(12 downto 0);
signal MantL_Q         : std_logic_vector(12 downto 0);
signal MantS           : std_logic_vector(12 downto 0);
signal MantS_Q         : std_logic_vector(12 downto 0);
signal MantSum         : std_logic_vector(13 downto 0);
signal MantSum_Q       : std_logic_vector(13 downto 0);
signal MantSum_Q5      : std_logic_vector(13 downto 0);
signal MantMuxOut      : std_logic_vector(9 downto 0);
signal MantMuxOut_Q    : std_logic_vector(9 downto 0);
signal MuxSel          : std_logic_vector(3 downto 0);
signal MuxSel_Q        : std_logic_vector(3 downto 0);

signal Guard_Bits      : std_logic_vector(1 downto 0);

signal Exp_update      : std_logic_vector(4 downto 0);
signal Exp_update_Q    : std_logic_vector(4 downto 0);
signal Exp_Sum         : std_logic_vector(5 downto 0);
signal Exp_Sum_Q       : std_logic_vector(5 downto 0);
signal Exp_out_zero    : std_logic;
signal Exp_out_zero_Q  : std_logic;
signal Exp_out_zero_Q6 : std_logic;

signal Exp_Max         : std_logic_vector(4 downto 0);
signal Exp_Zero        : std_logic_vector(4 downto 0);
signal Mant_Max        : std_logic_vector(9 downto 0);
signal Mant_Zero       : std_logic_vector(9 downto 0);
signal Overflow        : std_logic;
signal Underflow       : std_logic;


signal MantOut         : std_logic_vector(9 downto 0);
signal ExpOut          : std_logic_vector(4 downto 0);

begin
```

```
-- Register fpadd inputs
 process (Resetn,Clock)
  begin
  if (Resetn = '0') then
      SignA_Q    <= '0';
      SignB_Q    <= '0';
      Func_Q     <= '0';
      MantissaA_Q <= (others => '0');
      MantissaB_Q <= (others => '0');
      ExponentA_Q <= (others => '0');
      ExponentB_Q <= (others => '0');
   else
    if (Clock'event and Clock = '1') then
      SignA_Q    <= SignA;
      SignB_Q    <= SignB;
      Func_Q     <= Func;
      MantissaA_Q <= MantissaA;
      MantissaB_Q <= MantissaB;
      ExponentA_Q <= ExponentA;
      ExponentB_Q <= ExponentB;
    end if;
   end if;
 end process;

-- Exponent Difference & Swap Logic

Exp_Temp1   <= (ExponentA_Q(4 downto 4) & ExponentA_Q) - (ExponentB_Q(4 downto 4)
& ExponentB_Q);
Exp_Temp2   <= (ExponentB_Q(4 downto 4) & ExponentB_Q) - (ExponentA_Q(4 downto 4)
& ExponentA_Q);

AeqB        <= Exp_Temp1(5 downto 5) nor Exp_Temp2(5 downto 5);

-- Pipeline Register 1 is inside swap component

Compare: swap port map
   (Clock       => Clock,
    Resetn      => Resetn,
    ExponentA   => ExponentA_Q,
    ExponentB   => ExponentB_Q,
    MantissaA   => MantissaA_Q,
    MantissaB   => MantissaB_Q,
    SignA       => SignA_Q,
    SignB       => SignB_Q,
    Exp_Temp1   => Exp_Temp1(4 downto 0),
```

```
   Exp_Temp2    => Exp_Temp2(4 downto 0),
   AeqB         => AeqB,
   Func         => Func_Q,
   ExponentL    => ExponentL,
   ExponentS    => ExponentS,
   MantissaL    => MantissaL,
   MantissaS    => MantissaS,
   SignL        => SignL,
   SignS        => SignS,
   Shift_Sel    => Shift_Sel,
   Func_Q       => Func_Q1
   );


-- Sign Logic

AddSel  <= (SignL xor SignS) xor Func_Q1;

SignOut <= SignL;

-- MantissaS Shifter with Register 2

ManS_Shift: smshift port map
   (Clock       => Clock,
    Resetn      => Resetn,
    Shift_Sel   => Shift_Sel,
    MantissaS   => MantissaS,
    MantS       => MantS);

-- Register 2 for remaining signals
  process (Resetn,Clock)
   begin
    if (Resetn = '0') then
       AddSel_Q2   <= '0';
       SignOut_Q2  <= '0';
       ExponentL_Q2 <= (others => '0');
       MantissaL_Q2 <= (others => '0');
    else
     if (Clock'event and Clock = '1') then
       AddSel_Q2   <= AddSel;
       SignOut_Q2  <= SignOut;
       ExponentL_Q2 <= ExponentL;
       MantissaL_Q2 <= MantissaL;
     end if;
    end if;
```

```
 end process;


-- MantissaL Vector

Guard_Bits <= '0' & '0';

MantL  <= '1' & MantissaL_Q2 & Guard_Bits;



-- Register 3
 process (Resetn,Clock)
  begin
   if (Resetn = '0') then
      AddSel_Q3     <= '0';
      SignOut_Q3    <= '0';
      ExponentL_Q3  <= (others => '0');
      MantL_Q       <= (others => '0');
      MantS_Q       <= (others => '0');
   else
    if (Clock'event and Clock = '1') then
      AddSel_Q3     <= AddSel_Q2;
      SignOut_Q3    <= SignOut_Q2;
      ExponentL_Q3  <= ExponentL_Q2;
      MantL_Q       <= MantL;
      MantS_Q       <= MantS;
    end if;
   end if;
 end process;


-- 13 bit Add/Subtract
process(AddSel_Q3, MantS_Q, MantL_Q)
begin
 if AddSel_Q3 = '0' then
    MantSum  <= ('0' & MantL_Q) + ('0' & MantS_Q);
 else
    MantSum  <= ('0' & MantL_Q) - ('0' & MantS_Q);
 end if;
end process;


-- Register 4
 process (Resetn,Clock)
```

```
  begin
   if (Resetn = '0') then
      MantSum_Q     <= (others => '0');
      SignOut_Q4    <= '0';
      ExponentL_Q4  <= (others => '0');
   else
    if (Clock'event and Clock = '1') then
      MantSum_Q     <= MantSum;
      SignOut_Q4    <= SignOut_Q3;
      ExponentL_Q4  <= ExponentL_Q3;
    end if;
   end if;
 end process;


-- Exponent Update Logic

Expo_Update: Exposel3 port map
   (MantSum      => MantSum_Q,
    Exp_update   => Exp_update,
    MuxSel       => MuxSel,
    Exp_out_zero => Exp_out_zero);

-- Register 5
 process (Resetn,Clock)
   begin
    if (Resetn = '0') then
       MantSum_Q5    <= (others => '0');
       SignOut_Q5    <= '0';
       ExponentL_Q5  <= (others => '0');
       Exp_update_Q  <= (others => '0');
       MuxSel_Q      <= (others => '0');
       Exp_out_zero_Q <= '0';
    else
     if (Clock'event and Clock = '1') then
       MantSum_Q5    <= MantSum_Q;
       SignOut_Q5    <= SignOut_Q4;
       ExponentL_Q5  <= ExponentL_Q4;
       Exp_update_Q  <= Exp_update;
       MuxSel_Q      <= MuxSel;
       Exp_out_zero_Q <= Exp_out_zero;
     end if;
    end if;
 end process;
```

```
Exp_Sum <= ('0' & ExponentL_Q5) + ('0' & Exp_update_Q);


-- Mant Align Mux

Mant_Mux: Mantmux port map
   (MantSum      => MantSum_Q5,
    MuxSel       => MuxSel_Q,
    MantOut      => MantMuxOut);


-- Register 6
 process (Resetn,Clock)
  begin
   if (Resetn = '0') then
      MantMuxOut_Q  <= (others => '0');
      SignOut_Q6    <= '0';
      Exp_Sum_Q     <= (others => '0');
      ExponentL_Q6  <= (others => '0');
      Exp_out_zero_Q6 <= '0';
   else
    if (Clock'event and Clock = '1') then
      MantMuxOut_Q  <= MantMuxOut;
      SignOut_Q6    <= SignOut_Q5;
      Exp_Sum_Q     <= Exp_Sum;
      ExponentL_Q6  <= ExponentL_Q5;
      Exp_out_zero_Q6 <= Exp_out_zero_Q;
    end if;
   end if;
 end process;

-- Overflow/Underflow logic

 Exp_Max       <= "11111";
 Exp_Zero      <= "00000";

 Overflow   <= ExponentL_Q6(4) and Exp_Sum_Q(4);

 Underflow  <= (not(ExponentL_Q6(4)) and Exp_Sum_Q(4)) or Exp_out_zero_Q6;


-- Exponent Output Mux

 Exp_Out_Mux: MUX4_5 port map
     (InA       => Exp_Sum_Q(4 downto 0),
```

```
      InB      => Exp_Max,
      InC      => Exp_Zero,
      InD      => Exp_Zero,
      Sel0     => Overflow,
      Sel1     => Underflow,
      DataOut   => ExpOut);

-- Mantissa Output Mux

 Mant_Max     <= "1111111111";
 Mant_Zero    <= "0000000000";


 Mant_Out_Mux: MUX4_10 port map
     (InA      => MantMuxOut_Q,
      InB      => Mant_Max,
      InC      => Mant_Zero,
      InD      => Mant_Zero,
      Sel0     => Overflow,
      Sel1     => Underflow,
      DataOut   => MantOut);


-- Register fpadd outputs
 process (Resetn,Clock)
  begin
   if (Resetn = '0') then
       Sign_Out <= '0';
       Exp_Out  <= (others => '0');
       Mant_Out <= (others => '0');
   else
    if (Clock'event and Clock = '1') then
       Sign_Out <= SignOut_Q6;
       Exp_Out  <= ExpOut;
       Mant_Out <= MantOut;
    end if;
   end if;
 end process;


end BEHAVIOR;
```

```
--- ----------------------------------------------
-- 4:1 Mux (5-bit data bus)
-- Configurable Micro-Accelerator for QR Factorization
-- Mux4_5.VHD
-- ----------------------------------------------
-- Steven Gercken
-- Created: 12/05/97
-- Revised:
-- ----------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity MUX4_5 is
   port(InA          : in std_logic_vector(4 downto 0);
      InB          : in std_logic_vector(4 downto 0);
      InC          : in std_logic_vector(4 downto 0);
      InD          : in std_logic_vector(4 downto 0);
      Sel0         : in std_logic;
      Sel1         : in std_logic;
      DataOut      : out std_logic_vector(4 downto 0)
      );
end MUX4_5;

architecture BEHAVIOR of MUX4_5 is


signal Mux_Sel        : std_logic_vector(1 downto 0);

begin

 Mux_Sel <= Sel1 & Sel0;

 process (InA, InB, InC, InD, Mux_Sel)
  begin
   case Mux_Sel is

       when "00" =>
       DataOut <= InA;
       when "01" =>
       DataOut <= InB;
       when "10" =>
```

```
        DataOut <= InC;
        when "11" =>
        DataOut <= InD;
      when others =>
        DataOut <= "00000";

    end case;
  end process;

end Behavior;
```

```
--- ----------------------------------------------
-- 4:1 Mux (10-bit data bus)
-- Configurable Micro-Accelerator for QR Factorization
-- Mux4_10.VHD
-- ----------------------------------------------
-- Steven Gercken
-- Created: 12/05/97
-- Revised:
-- ----------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity MUX4_10 is
   port(InA          : in std_logic_vector(9 downto 0);
        InB          : in std_logic_vector(9 downto 0);
        InC          : in std_logic_vector(9 downto 0);
        InD          : in std_logic_vector(9 downto 0);
        Sel0         : in std_logic;
        Sel1         : in std_logic;
        DataOut      : out std_logic_vector(9 downto 0)
        );
end MUX4_10;

architecture BEHAVIOR of MUX4_10 is

signal Mux_Sel          : std_logic_vector(1 downto 0);

begin

 Mux_Sel <= Sel1 & Sel0;

 process (InA, InB, InC, InD, Mux_Sel)
  begin
   case Mux_Sel is

        when "00" =>
        DataOut <= InA;
        when "01" =>
        DataOut <= InB;
        when "10" =>
        DataOut <= InC;
```

```
        when "11" =>
          DataOut <= InD;
      when others =>
          DataOut <= (others => '0');

    end case;
  end process;

end Behavior;
```

```
--- ------------------------------------------------
-- Floating Point Adder Subtractor
-- Configurable Micro-Accelerator for QR Factorization
-- mantmux.VHD
-- ------------------------------------------------
-- Steven Gercken
-- Created: 1/29/98
-- Revised:
-- ------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mantmux is
      port(MantSum      : in std_logic_vector(13 downto 0);
         MuxSel       : in std_logic_vector(3 downto 0);
         MantOut      : out std_logic_vector(9 downto 0)
         );
end mantmux;

architecture BEHAVIOR of mantmux is


signal LoMuxSel     : std_logic_vector(1 downto 0);
signal HiMuxSel     : std_logic_vector(1 downto 0);
signal LoMuxOut     : std_logic_vector(13 downto 0);
signal HiMuxOut     : std_logic_vector(13 downto 0);
signal MantIn       : std_logic_vector(14 downto 1);


begin

LoMuxSel  <= MuxSel(1 downto 0);
HiMuxSel  <= MuxSel(3 downto 2);

MantIn    <= "00" & MantSum(12 downto 1);

process(HiMuxSel,MantIn)
begin
 case HiMuxSel is

        when "00" =>
           HiMuxOut   <= MantIn(2 downto 1)  & "000000000000";
```

```
        when "01" =>
          HiMuxOut   <= MantIn(6 downto 1)  & "00000000";
        when "10" =>
          HiMuxOut   <= MantIn(10 downto 1) & "0000";
        when "11" =>
          HiMuxOut   <= MantIn(14 downto 1);
        when others =>
          HiMuxOut   <= (others => '0');

        end case;
end process;


process(LoMuxSel,HiMuxOut)
begin
 case LoMuxSel is

        when "00" =>
          LoMuxOut   <= HiMuxOut(10 downto 0) & "000";
        when "01" =>
          LoMuxOut   <= HiMuxOut(11 downto 0) & "00";
        when "10" =>
          LoMuxOut   <= HiMuxOut(12 downto 0) & '0';
        when "11" =>
          LoMuxOut   <= HiMuxOut(13 downto 0);
        when others =>
          LoMuxOut   <= (others => '0');

        end case;
end process;

MantOut  <= LoMuxOut(13 downto 4);


end BEHAVIOR;
```

```
--- ------------------------------------------------
-- Floating Point Adder Subtractor
-- Configurable Micro-Accelerator for QR Factorization
-- SWAP.VHD
-- ------------------------------------------------
-- Steven Gercken
-- Created: 1/28/98
-- Revised:
-- ------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity SWAP is
    port(Clock       : in std_logic;
       Resetn      : in std_logic;
       ExponentA    : in std_logic_vector(4 downto 0);
       ExponentB    : in std_logic_vector(4 downto 0);
       MantissaA    : in std_logic_vector(9 downto 0);
       MantissaB    : in std_logic_vector(9 downto 0);
       SignA        : in std_logic;
       SignB        : in std_logic;
       Exp_Temp1    : in std_logic_vector(4 downto 0);
       Exp_Temp2    : in std_logic_vector(4 downto 0);
       AeqB         : in std_logic_vector(0 downto 0);
       Func         : in std_logic;
       ExponentL    : out std_logic_vector(4 downto 0);
       ExponentS    : out std_logic_vector(4 downto 0);
       MantissaL    : out std_logic_vector(9 downto 0);
       MantissaS    : out std_logic_vector(9 downto 0);
       SignL        : out std_logic;
       SignS        : out std_logic;
       Shift_Sel    : out std_logic_vector(4 downto 0);
       Func_Q       : out std_logic
       );
end SWAP;

architecture BEHAVIOR of SWAP is

signal AgtB             : std_logic;
signal AgtB_Q           : std_logic;
signal ExponentA_Q      : std_logic_vector(4 downto 0);
signal ExponentB_Q      : std_logic_vector(4 downto 0);
```

```
signal MantissaA_Q      : std_logic_vector(9 downto 0);
signal MantissaB_Q      : std_logic_vector(9 downto 0);
signal SignA_Q          : std_logic;
signal SignB_Q          : std_logic;
signal Exp_Diff         : std_logic_vector(4 downto 0);
signal Exp_Temp1_Q      : std_logic_vector(4 downto 0);
signal Exp_Temp2_Q      : std_logic_vector(4 downto 0);
signal Exp_Result       : std_logic_vector(4 downto 0);

begin


process(AeqB,ExponentA,ExponentB,MantissaA,MantissaB)
begin
 if (AeqB = "1") then
   if MantissaA >= MantissaB then
      AgtB     <= '1';
   else
      AgtB     <= '0';
   end if;
 else
   if ExponentA > ExponentB then
      AgtB     <= '1';
   else
      AgtB     <= '0';
   end if;
 end if;
end process;


-- Register
 process (Resetn,Clock)
  begin
  if (Resetn = '0') then
      Exp_Temp1_Q <= (others => '0');
      Exp_Temp2_Q <= (others => '0');
      AgtB_Q      <= '0';
      ExponentA_Q <= (others => '0');
      ExponentB_Q <= (others => '0');
      MantissaA_Q <= (others => '0');
      MantissaB_Q <= (others => '0');
      SignA_Q     <= '0';
      SignB_Q     <= '0';
      Func_Q      <= '0';
   else
```

```
    if (Clock'event and Clock = '1') then
      Exp_Temp1_Q <= Exp_Temp1;
      Exp_Temp2_Q <= Exp_Temp2;
      AgtB_Q      <= AgtB;
      ExponentA_Q <= ExponentA;
      ExponentB_Q <= ExponentB;
      MantissaA_Q <= MantissaA;
      MantissaB_Q <= MantissaB;
      SignA_Q     <= SignA;
      SignB_Q     <= SignB;
      Func_Q      <= Func;
    end if;
   end if;
 end process;

process(AgtB_Q,Exp_Temp1_Q,Exp_Temp2_Q)
begin
 case AgtB_Q is

    when '1' =>
      Exp_Diff <= Exp_Temp1_Q;

    when '0' =>
      Exp_Diff <= Exp_Temp2_Q;

    when others =>
      Exp_Diff <= Exp_Temp1_Q;
 end case;
end process;

process(Exp_Diff)
begin
 if Exp_Diff  >= "01101" then
   Shift_Sel <= "01101";
 else
   Shift_Sel <= Exp_Diff;
 end if;
end process;

ExponentL  <= ExponentA_Q when AgtB_Q = '1'
          else ExponentB_Q;

ExponentS  <= ExponentA_Q when AgtB_Q = '0'
          else ExponentB_Q;
```

```
MantissaL  <= MantissaA_Q when AgtB_Q = '1'
        else MantissaB_Q;

MantissaS  <= MantissaA_Q when AgtB_Q = '0'
        else MantissaB_Q;

SignL    <= SignA_Q when AgtB_Q = '1'
        else SignB_Q;

SignS    <= SignA_Q when AgtB_Q = '0'
        else SignB_Q;



end BEHAVIOR;
```

```
--- ------------------------------------------------
-- Floating Point Adder Subtractor
-- Configurable Micro-Accelerator for QR Factorization
-- SMALLSHIFT.VHD
-- ------------------------------------------------
-- Steven Gercken
-- Created: 1/21/98
-- Revised:
-- ------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity SMALLSHIFT is
   port(Clock       : in std_logic;
      Resetn      : in std_logic;
      Shift_Sel   : in std_logic_vector(4 downto 0);
      MantissaS   : in std_logic_vector(9 downto 0);
      MantS       : out std_logic_vector(12 downto 0)
      );
end SMALLSHIFT;

architecture BEHAVIOR of SMALLSHIFT is

signal MantSmall     : std_logic_vector(12 downto 0);
signal MantSmall_Q   : std_logic_vector(12 downto 0);
signal MantHi        : std_logic_vector(12 downto 0);
signal MantLo        : std_logic_vector(12 downto 0);
signal HiShift       : std_logic_vector(1 downto 0);
signal HiShift_Q     : std_logic_vector(1 downto 0);
signal LoShift       : std_logic_vector(1 downto 0);
signal LoShift_Q     : std_logic_vector(1 downto 0);

begin

-- Shift_Sel(4) is always 0, so why use it in barrel shifter?
HiShift   <= Shift_Sel(3 downto 2);
LoShift   <= Shift_Sel(1 downto 0);

MantSmall <= '1' & MantissaS & '0' & '0';

-- Register 2
 process (Resetn,Clock)
```

```
  begin
  if (Resetn = '0') then
      LoShift_Q  <= (others => '0');
      HiShift_Q  <= (others => '0');
      MantSmall_Q <= (others => '0');
   else
    if (Clock'event and Clock = '1') then
      LoShift_Q  <= LoShift;
      HiShift_Q  <= HiShift;
      MantSmall_Q <= MantSmall;
    end if;
   end if;
 end process;



process(LoShift_Q,MantSmall_Q)
begin
 case LoShift_Q is

     when "00"  =>
          MantLo <= MantSmall_Q;

     when "01"  =>
          MantLo <= '0'  & MantSmall_Q(12 downto 1);

     when "10"  =>
          MantLo <= "00"  & MantSmall_Q(12 downto 2);

     when "11"  =>
          MantLo <= "000" & MantSmall_Q(12 downto 3);

       when others =>
          MantLo <= (others => '0');

       end case;
end process;

process(HiShift_Q,MantLo)
begin
 case HiShift_Q is

     when "00"  =>
          MantHi <= MantLo;
```

```
      when "01"  =>
          MantHi <= "0000"         & MantLo(12 downto 4);

      when "10"  =>
          MantHi <= "00000000"     & MantLo(12 downto 8);

      when "11"  =>
          MantHi <= "000000000000" & MantLo(12 downto 12);

        when others =>
          MantHi <= (others => '0');

        end case;
end process;

MantS  <= MantHi;

end BEHAVIOR;
```

```
--- ------------------------------------------------
-- Floating Point Adder Subtractor
-- Configurable Micro-Accelerator for QR Factorization
-- exposel3.VHD
-- ------------------------------------------------
-- Steven Gercken
-- Created: 2/9/98
-- Revised:
-- ------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity exposel3 is
      port(MantSum      : in std_logic_vector(13 downto 0);
         Exp_update    : out std_logic_vector(4 downto 0);
         MuxSel        : out std_logic_vector(3 downto 0);
         Exp_out_zero  : out std_logic
         );
end exposel3;

architecture BEHAVIOR of exposel3 is


signal Mantin        : std_logic_vector(15 downto 0);
signal A             : std_logic;
signal B             : std_logic;
signal C             : std_logic;
signal D             : std_logic;
signal U             : std_logic;
signal B1U           : std_logic;
signal B1L           : std_logic;
signal B0A           : std_logic;
signal B0B           : std_logic;
signal B0C           : std_logic;
signal B0D           : std_logic;
signal Carry         : std_logic;
signal MuxSel_Int    : std_logic_vector(3 downto 0);

begin

Mantin <= "00" & MantSum(13 downto 1) & "0";
```

```
-- Level 1

A  <= Mantin(15) or Mantin(14) or Mantin(13) or Mantin(12);

B  <= Mantin(11) or Mantin(10) or Mantin(9)  or Mantin(8);

C  <= Mantin(7)  or Mantin(6)  or Mantin(5)  or Mantin(4);

D  <= Mantin(3)  or Mantin(2)  or Mantin(1)  or Mantin(0);

U  <= Mantin(15) or Mantin(14) or Mantin(13) or Mantin(12) or Mantin(11) or Mantin(10) or
Mantin(9) or Mantin(8);

B1U <= Mantin(15) or Mantin(14) or (not(Mantin(13)) and not(Mantin(12)) and (Mantin(11) or
Mantin(10)));

B1L <= Mantin(7) or Mantin(6) or (not(Mantin(5)) and not(Mantin(4)) and (Mantin(3) or
Mantin(2)));


B0A <= Mantin(15) or (not(Mantin(14)) and Mantin(13));

B0B <= Mantin(11) or (not(Mantin(10)) and Mantin(9));

B0C <= Mantin(7) or (not(Mantin(6)) and Mantin(5));

B0D <= Mantin(3) or (not(Mantin(2)) and Mantin(1));

-- Level 2

Carry       <= A or B or C or D;

MuxSel_Int(3) <= U;

MuxSel_Int(2) <= A or (not(B) and C);

MuxSel_Int(1) <= B1U or (not(U) and B1L);

MuxSel_Int(0) <= B0A or (not(A) and B0B) or (not(U) and (B0C or (not(C) and B0D)));


Exp_out_zero  <=  not(Carry) and not(MuxSel_Int(0)) and not(MuxSel_Int(1))
                  and not(MuxSel_Int(2)) and not(MuxSel_Int(3));
```

```
MuxSel      <= MuxSel_Int;

process(MantSum)
begin
 if MantSum(13) = '1' then
      Exp_update <= "00001";
--      MuxSel    <= "0000";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 12) = "01" then
      Exp_update <= "00000";
--      MuxSel    <= "0001";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 11) = "001" then
      Exp_update <= "11111";
--      MuxSel    <= "0010";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 10) = "0001" then
      Exp_update <= "11110";
--      MuxSel    <= "0011";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 9)  = "00001" then
      Exp_update <= "11101";
--      MuxSel    <= "0100";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 8)  = "000001" then
      Exp_update <= "11100";
--      MuxSel    <= "0101";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 7)  = "0000001" then
      Exp_update <= "11011";
--      MuxSel    <= "0110";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 6)  = "00000001" then
      Exp_update <= "11010";
--      MuxSel    <= "0111";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 5)  = "000000001" then
      Exp_update <= "11001";
--      MuxSel    <= "1000";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 4)  = "0000000001" then
      Exp_update <= "11000";
--      MuxSel    <= "1001";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 3)  = "00000000001" then
```

```
      Exp_update <= "10111";
--      MuxSel    <= "1010";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 2) = "000000000001" then
      Exp_update <= "10110";
--      MuxSel    <= "1011";
--      Exp_out_zero <= '0';
 elsif MantSum(13 downto 1) = "0000000000001" then
      Exp_update <= "10101";
--      MuxSel    <= "1100";
--      Exp_out_zero <= '0';
-- elsif MantSum(13 downto 1) = "0000000000000" then
--      Exp_update <= "00000";
--      MuxSel    <= "1101";
--      Exp_out_zero <= '1';
 else
      Exp_update <= "00000";
--      MuxSel    <= "0000";
--      Exp_out_zero <= '0';
 end if;
end process;

end BEHAVIOR;
```