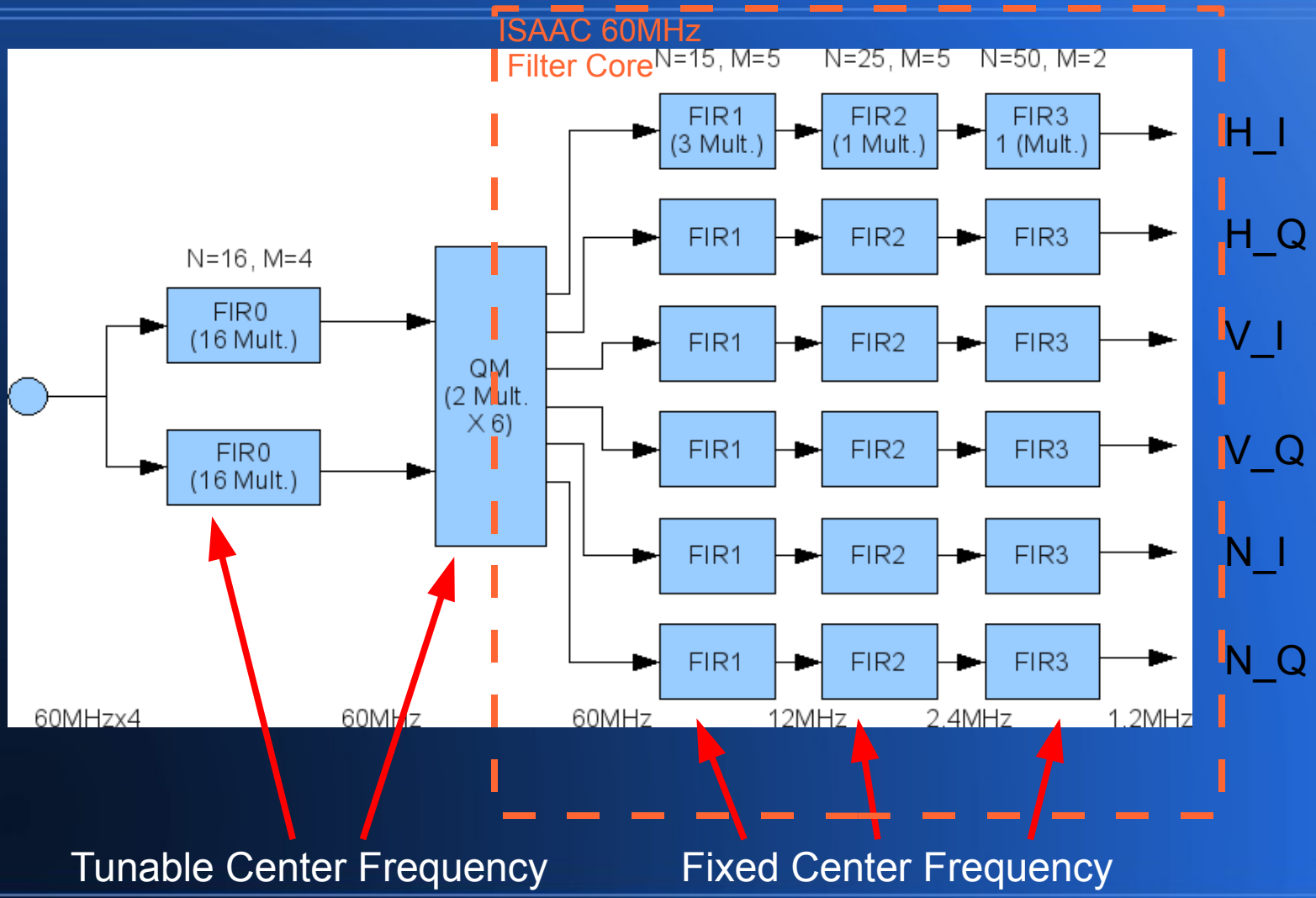
240MHz Digital Filter V2-3000 Implementation

Jason Zheng, Kayla Nguyen,
Charles Le, Yutao He

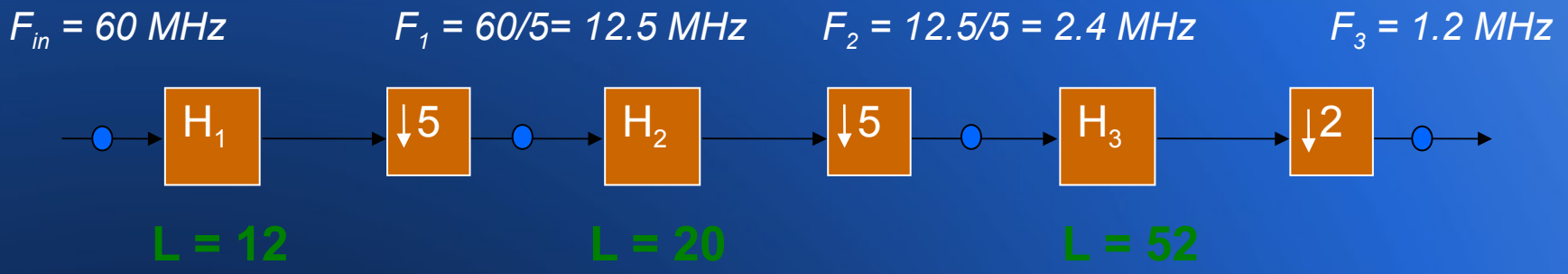
Outline

- Overview
- FIRDECIM implementation
- Coefficient Bank Implementation
- QM implementation
- FIR and QM Tuning
- Verification Results
- Synthesis Results

Overview


ISAAC 60MHz
Filter Core
N=15, M=5
N=25, M=5
N=50, M=2
N=16, M=4
FIR0
(16 Mult.)
FIR0
(16 Mult.)
QM
(2 Mult.
X 6)
FIR1
(3 Mult.)
FIR2
(1 Mult.)
FIR3
1 (Mult.)
FIR1
FIR2
FIR3
H_I
H_Q
V_I
V_Q
N_I
N_Q
60MHzx4
60MHz
60MHz
12MHz
2.4MHz
1.2MHz
Tunable Center Frequency
Fixed Center Frequency

FIRDECIM



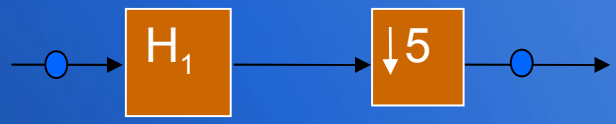
- FIR followed by a decimation stage
- Commonly referred to as polyphase FIR or firdecim
- N = number of taps, M = decimation factor

Implementation Choices

- Plain FIR + Decimation
- Simple Polyphase Filter
- Serial FIR (distributed arithmetic) + Decimation
- ISAAC Thread-Oriented FIR
- Considerations:
  - Speed and Size
  - Ease of Implementation, Modification, and Verification
  - Center Frequency Tuning
  - SEU mitigation (coefficient correctness)

Plain FIR + Decimation

- Assume no input buffer

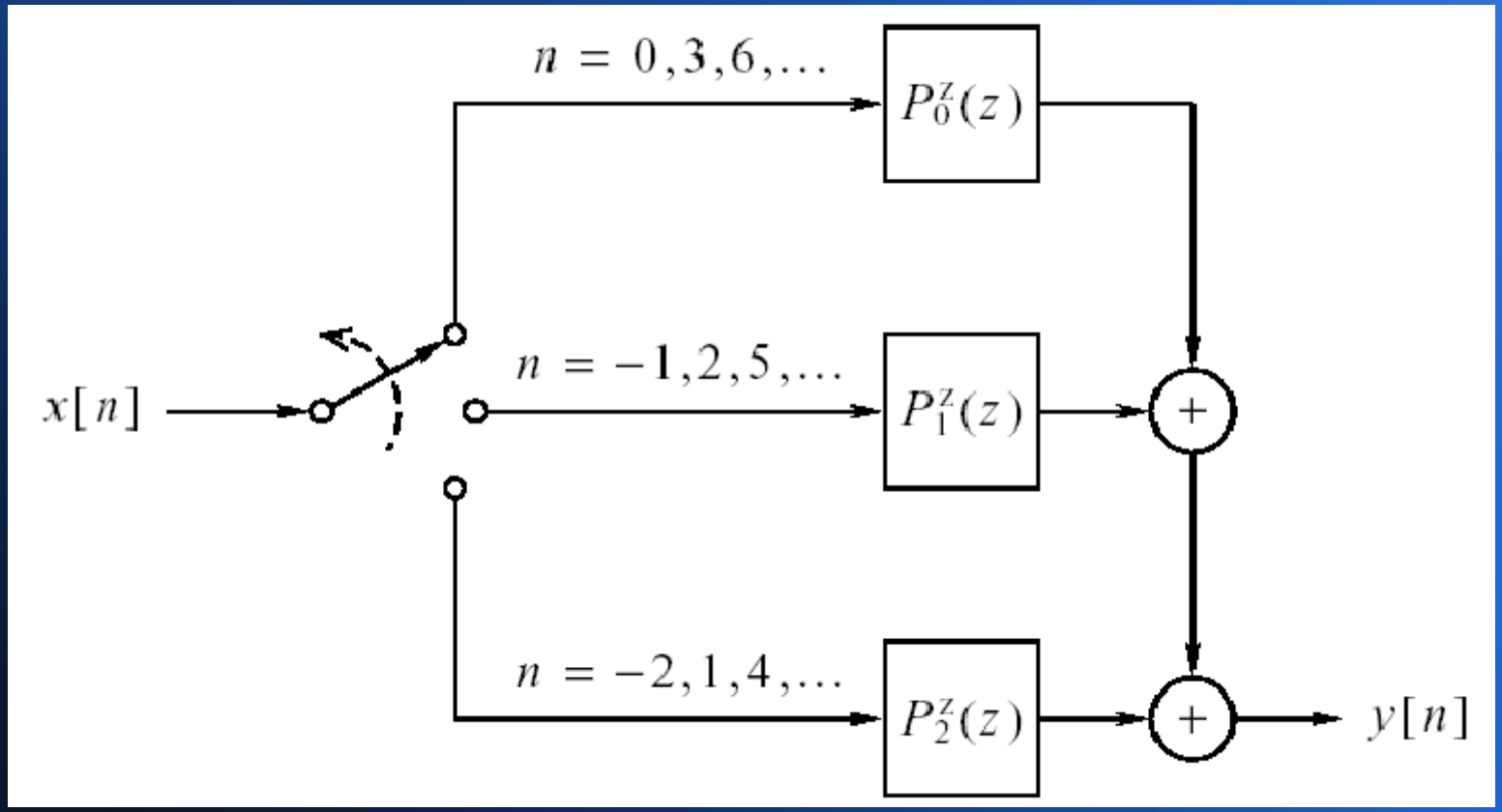


- multiplier clock to input rate ratio of R
- N taps
- Need ceil(N/R) multipliers
- Most calculations are thrown away due to decimation

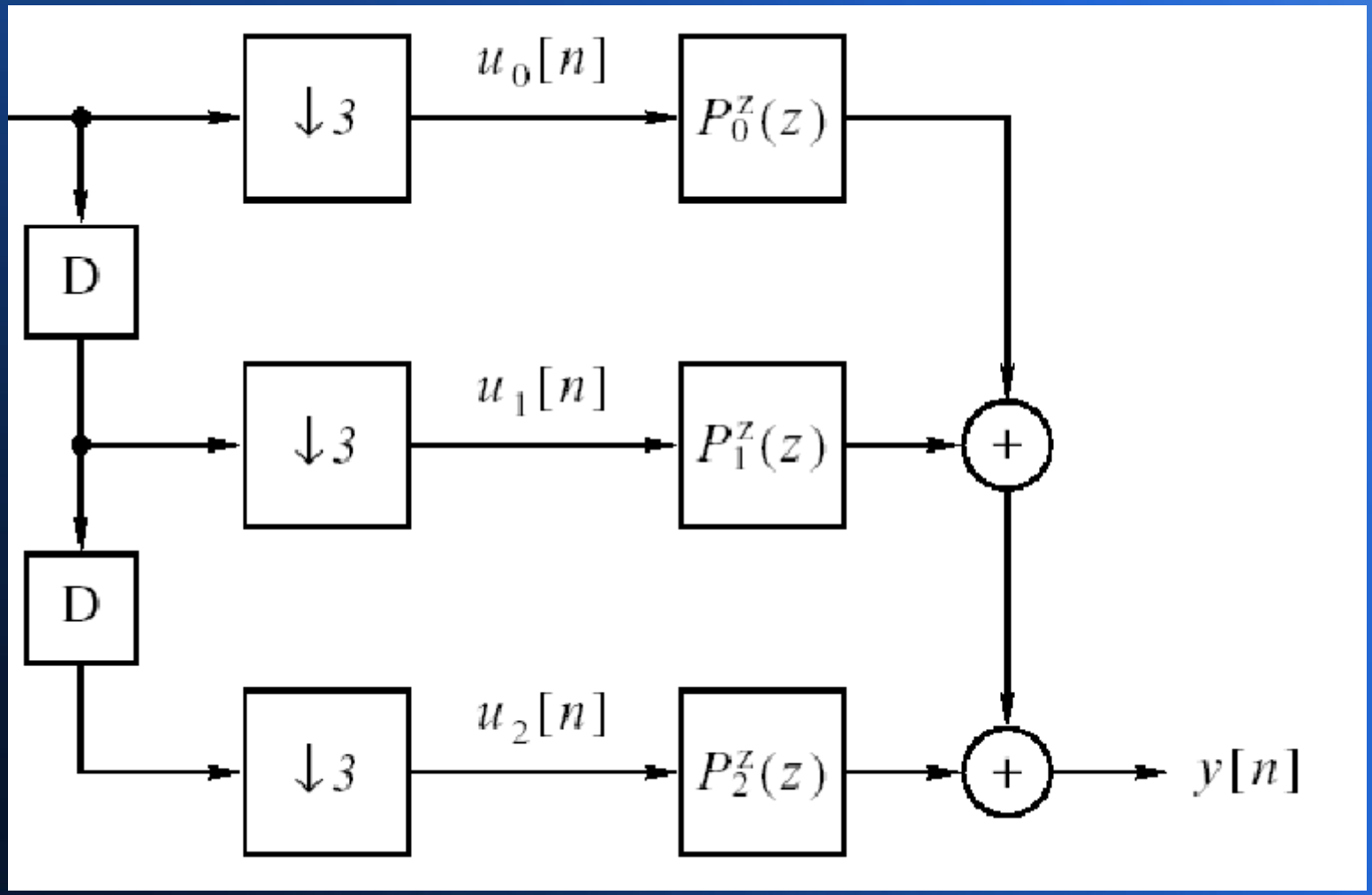
Simple Polyphase Filter

- Decimate first, then filter
- Break N-tap filter into M sub-filters
- No wasted calculations
- Require max(N/M, M) multipliers, at least 1 multiplier per sub-filter

Polyphase Filter – Switched Input View



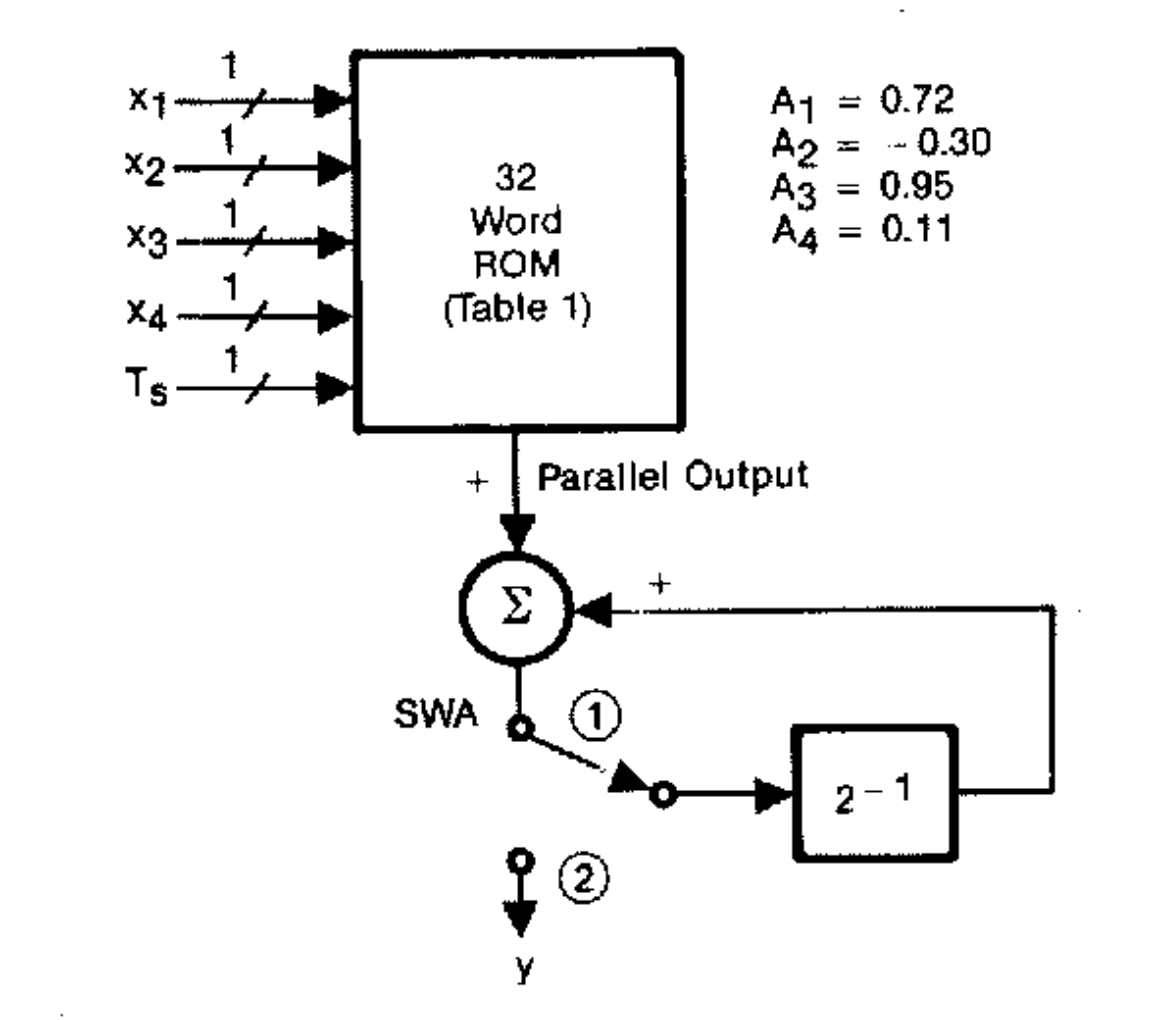
Polyphase Filter - Delayed Inputs View



Serial FIR

- Calculate N multiplications all at once, but bit-wise.
- Latency depends on fixed point width K
- Multipliers are replaced by LUTs (2^N depth)
- Internal Clock rate = K * Input rate
- Need input buffer for higher input rates
- Tuning = rewrite all LUTs

Serial FIR (From White's Paper)



Our Approach – Thread Oriented

- Based on textbook polyphase filter
- Generalize multiplier sharing with decimation factor M and clock ratio R
- Multiplier needed is ceil(N / (M*R))
- R is typically very high in a multi-stage firdecim design, esp in later stages.
- Suits well for “bottom-heavy” filters with single clock domain

Our Approach – Thread View

Each output is computed as an individual thread with a MAC unit

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16	x17	x18	x19	x20	x21	x22	x23	x24	x25	x26
output1	*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0											
		*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0										
			*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0									
				*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0								
output2					*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0							
						*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0						
							*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0					
								*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0				
output3									*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0			
										*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0			
											*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0	
												*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1	*e0
output4													*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2	*e1
														*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3	*e2
															*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4	*e3
output4																*e15	*e14	*e13	*e12	*e11	*e10	*e9	*e8	*e7	*e6	*e5	*e4

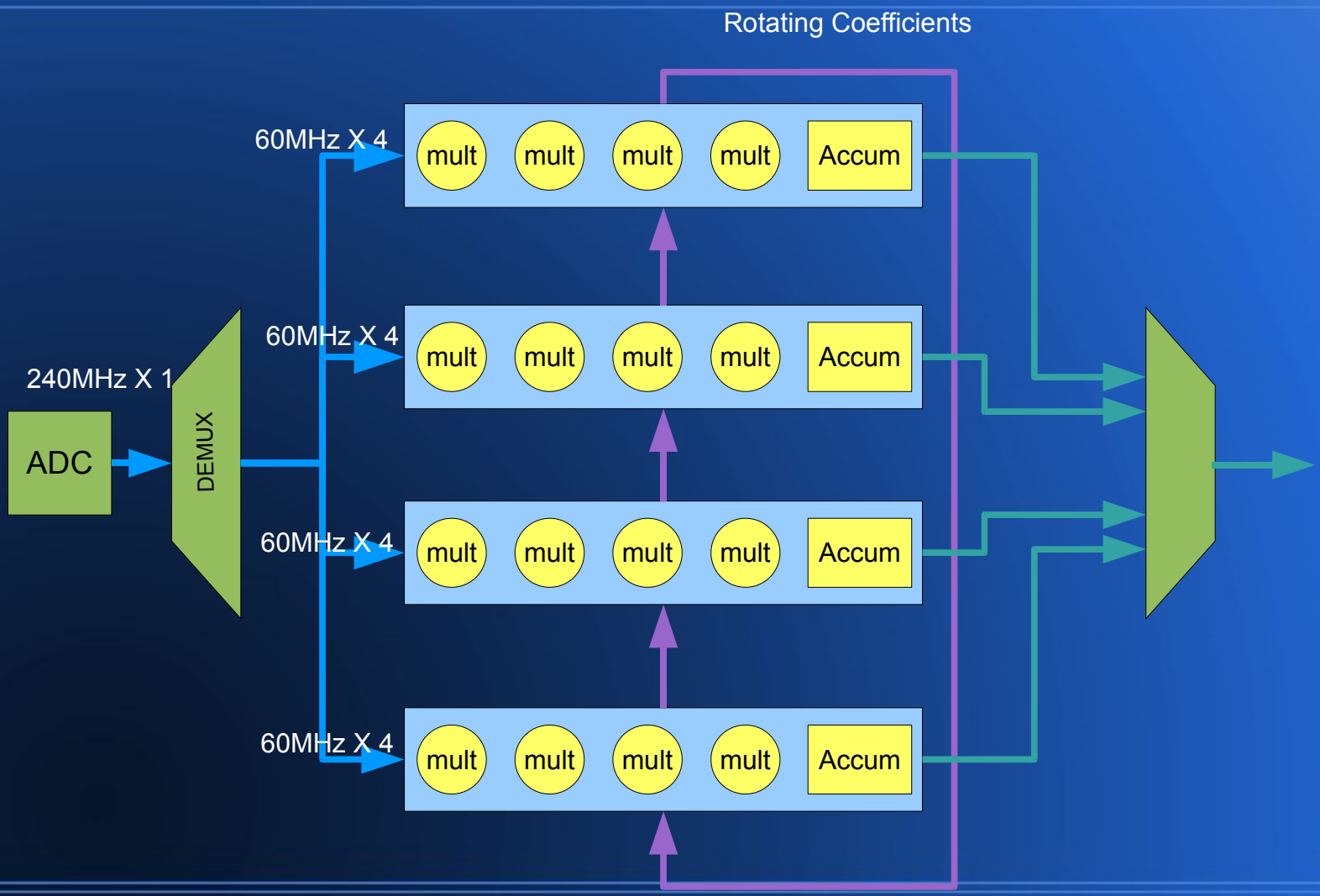
Computed by MAC 1

Computed by MAC 2

Computed by MAC 3

Computed by MAC 4

Our Approach – Thread Oriented



FIR Comparisons

	Resource Required	Pros	Cons
Plain FIR + Decimation	(N/R) Mult.	Straightforward Implementation	Wasted calculations due to decimation
Simple Polyphase FIR	(N/M) Mult.	No wasted calculations	Cannot take advantage of high R
Serial FIR	2^K Entry LUT	Small, compact FIR	Slow, need buffer for high input rates
ISAAC FIR	(N/(M*R)) Mult.	Easy implementation and analysis, low resource requirement	Uses more multipliers than Serial FIR

Coefficient Bank Implementation

- Option 1 – Shift Registers
  - XST can map automatically V2 SRL16 macros, save flip-flops for pipeline stages
  - Tuning is done by shifting 1's and 0's in proper sequence
  - However SRL16 cannot by scrubbed
  - Mitigate SEU with EDAC or TMR only

Coefficient Bank Implementation

- Option 2 – BlockRAM
  - Coefficient banks usually small, wasteful to use entire BlockRAM
  - Tuning is done by writing to the BlockRAM
  - Scrubbing can be done by enabling BlockRAM scrubbing
  - However unclear how to selectively scrub BlockRAM, i.e. other BlockRAM used as buffers should not be scrubbed

Coefficient Bank Implementation

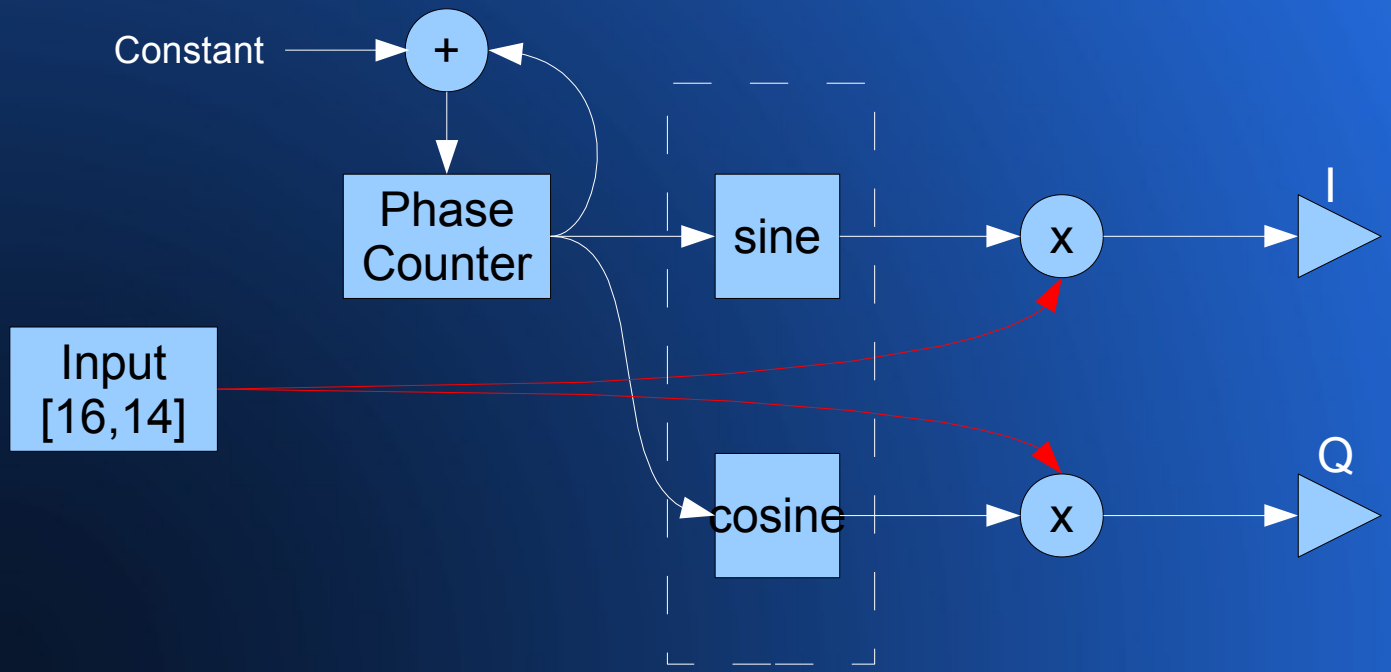
- Option 3 – Distributed RAM (LUTRAM)
  - Coefficient banks usually small enough to fit in distributed RAM
  - Tuning is done by writing to the RAM block
  - Scrubbing is free: distributed RAM is scrubbed together with config RAM.
  - Similar LUT usage as Option 1

Coefficient Bank Summary

	Resource Element	SEU Mitigation	Tuning
Shift Registers	SRL16 (4LUTs)	Cannot scrub; EDAC or TMR only	Bit-wise shift in new values
BlockRAM	BRAM Blocks	BRAM scrubbing or EDAC	Rewrite BRAM content
Distributed RAM	4LUTs	Config. RAM scrubbing or EDAC	Rewrite RAM content

QM Implementation

```
phase=wbpc*(0:N-1)=2pi*fc/f0*(0:N-1)
xbsr=xov.*cos(phase)
xbsi=xov.*sin(phase)
```



QM Implementation

- Phase counter is synchronized/reset on TX_START
- Phase counter should not accumulate quantization error
- Actual implementation uses the following equation (x,y are outputs from 2 FIR0):
  - I = x*sin(k) + y*cos(k)
  - Q = x*cos(k) + y*sin(k)
- How to implement the sine/cosine elements?

Sine/Cosine Implementation

- Option 1 – CORDIC
  - Multi-cycle iterative evaluation process
  - Need multiple clock cycles to get the result
  - Latency proportional to precision requirement
  - Two sources of error:
    - phase input quantization error (caused by input quantization)
    - output quantization error (caused by output quantization)

Sine/Cosine Implementation

- Option 2 – Full Sine/Cosine Table
  - Equivalent to CORDIC but implemented as large look-up table
  - Size is 2^(K)
  - Latency is one clock
  - Two sources of error: phase input quantization error and output quantization error
  - Most of the table is not used!

Sine/Cosine Implementation

- Option 3 – Condensed Sine/Cosine Table
  - phase=wbpc*(0:N-1)=2pi*fc/f0*(0:N-1)
  - Observe that if fc is constrained to multiples of 1MHz, phase can only take on finite set of values
  - Size is shrank from (2^K) to (f0 / 1MHz)
  - Latency is still one clock
  - Quantization error only on output; phase input quantization error is eliminated!

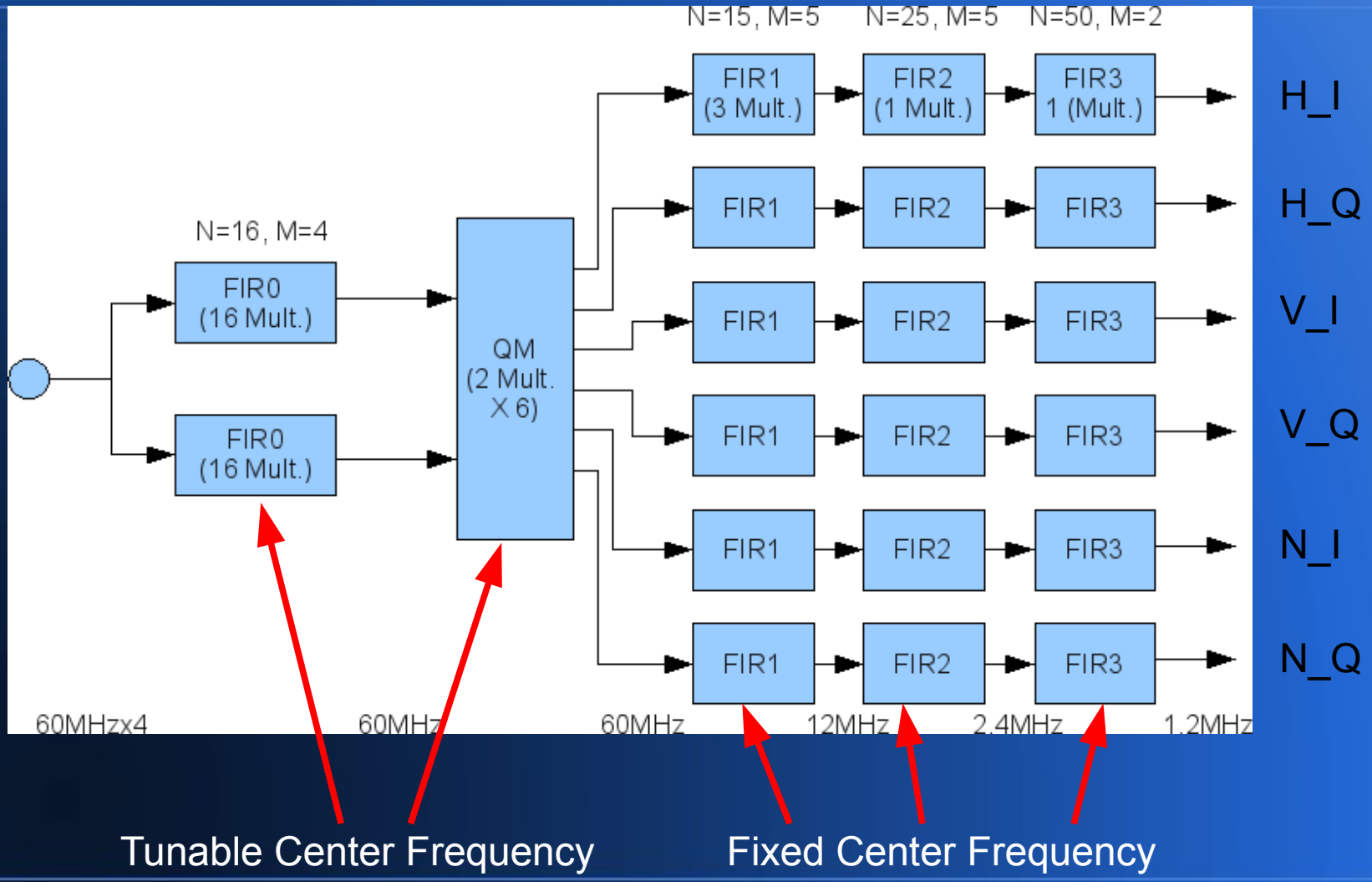
Sine/Cosine Implementation

- Consideration for SEU Mitigation
  - Tables can be implemented with BlockRAM or Distributed RAM
  - Turning on scrubbing for BlockRAM may not be desirable if other modules use BlockRAM as buffers
  - Distributed RAM is scrubbed along with other 4LUT configuration RAM
  - EDAC is also possible, but more overhead and latency

Sine/Cosine Summary

	Size	Latency	Q. Error	SEU Mitigation
CORDIC	small	20+ clock cycles	Phase Input + Function Output	Config RAM scrubbing
Full Table	2^k entries (1024)	1 clock cycle	Phase Input + Function Output	BRAM scrubbing
Condensed Table	f0 / 1MHz (240)	1 clock cycle	Function Output Only	BRAM or Distributed RAM Scrubbing

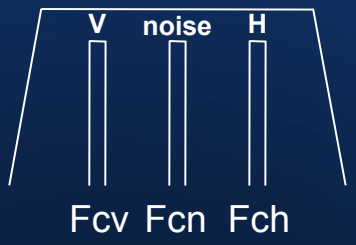
FIR and QM Tuning



Tuning FIR0

- Need to load the coefficient banks (32 loads)
- Need to re-compute the coefficients
  - wbpc= 2*pi*f_chanc/fs0;
  - hmsbpr= hmsbp.*cos(wbpc*[0:1:hlngbp-1]);
  - hmsbpi= hmsbp.*sin(wbpc*[0:1:hlngbp-1]);
- Sine and cosine functions are implemented using the same LUT used in QM, assuming f_chanc is multiples of 1MHz

Explanation of f_chanc in SMAP context



- Fcv and Fch are 5MHz apart
- Fcn is 2.5MHz away from both Fcv and Fch
- F_chanc for FIR0 is Fcn
- 1MHz contraint is only on Fcn, not on Fcv and Fch

Tuning QM

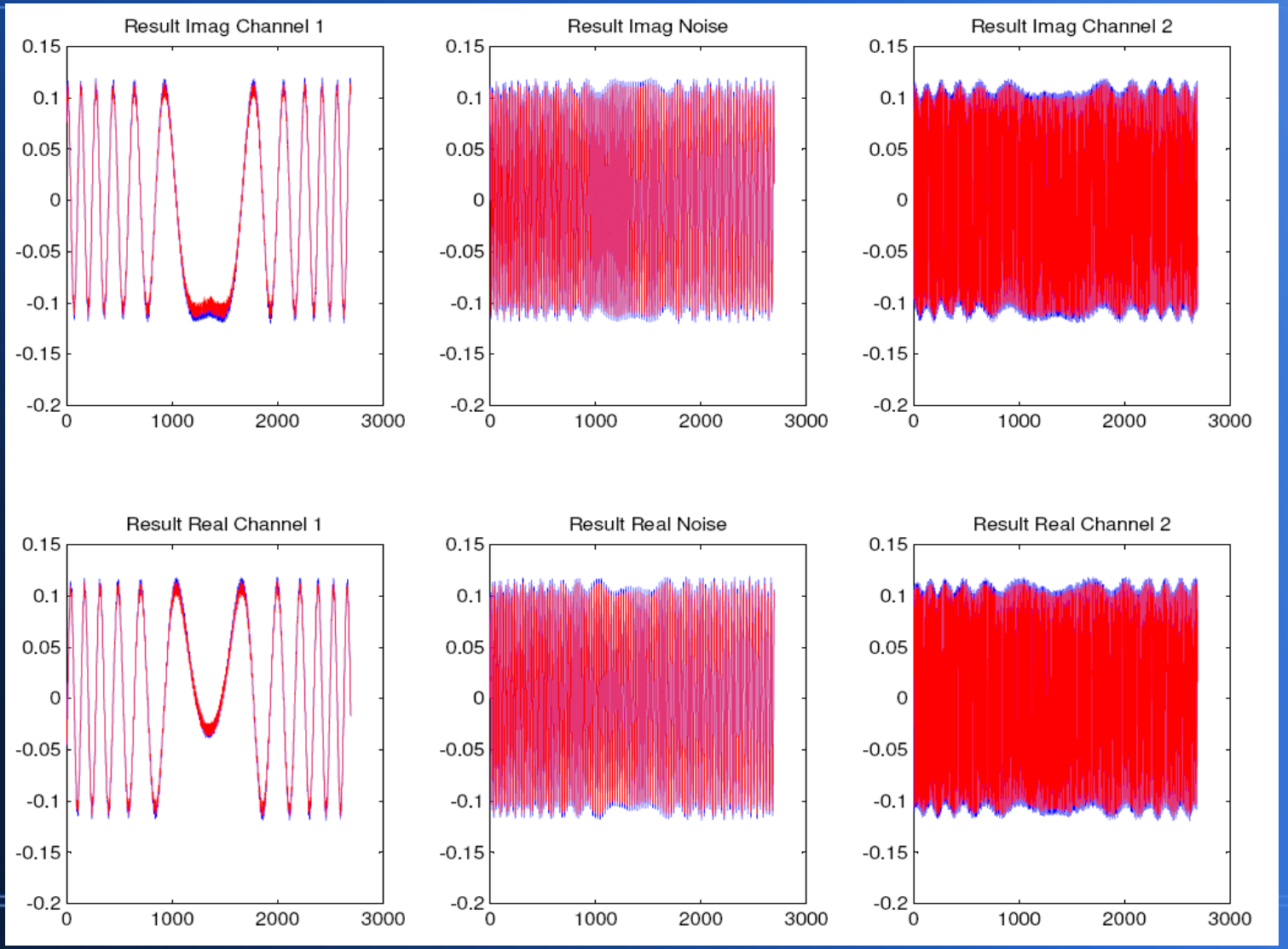
- Recall QM phase calculation:
  - phase=wbpc*(0:N-1)=2pi*fc/f0*(0:N-1)
- Tuning QM's fc only involves changing the constant increment value for the phase counter
- If fc = 1MHz, increment value is 1
- If fc = 34MHz, increment value is 34

Design Verification
Output Comparison of FIR0 and QM

Output after FIR0 and QM processing

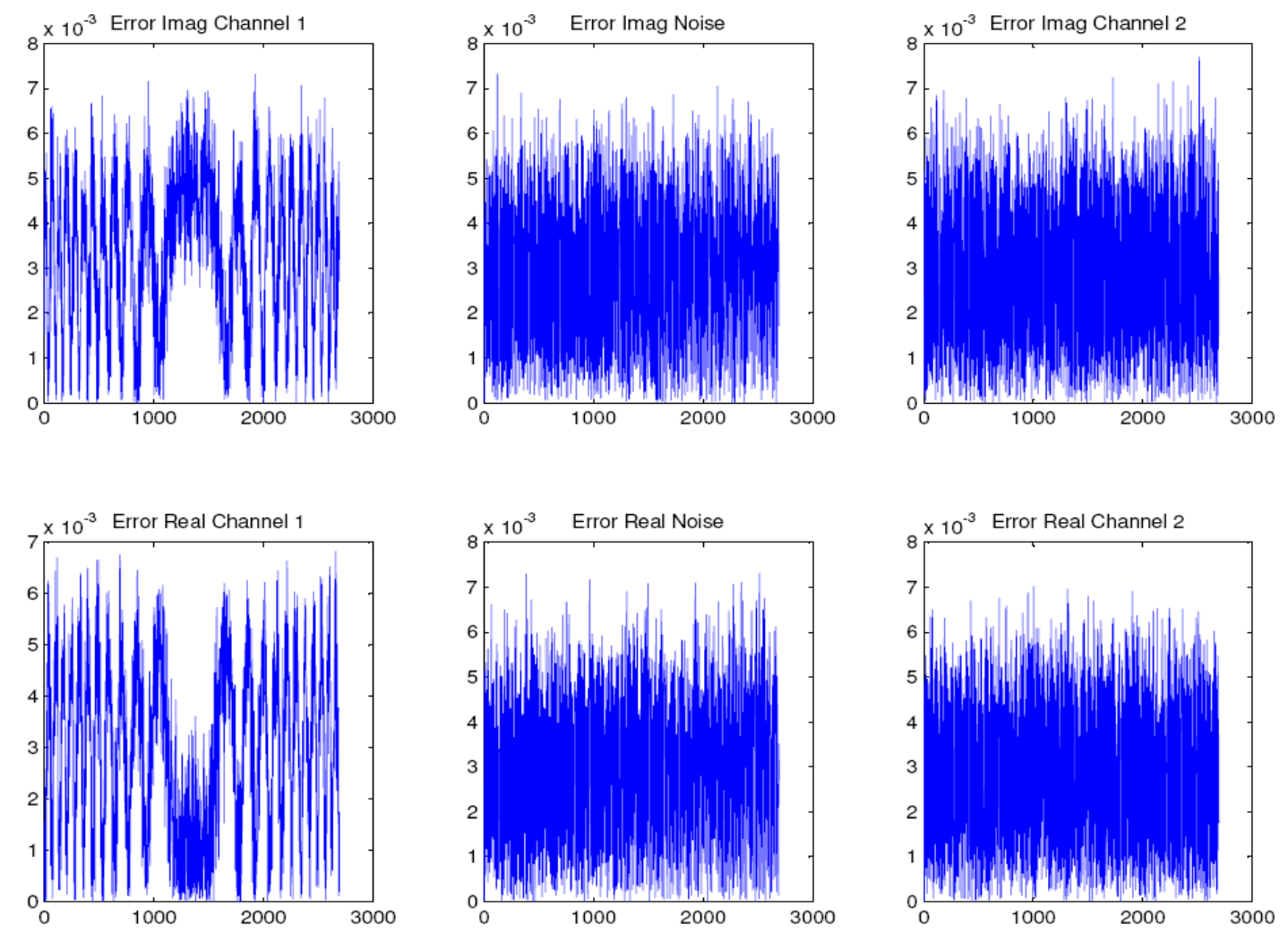
Red – RTL simulation

Blue – Matlab simulation



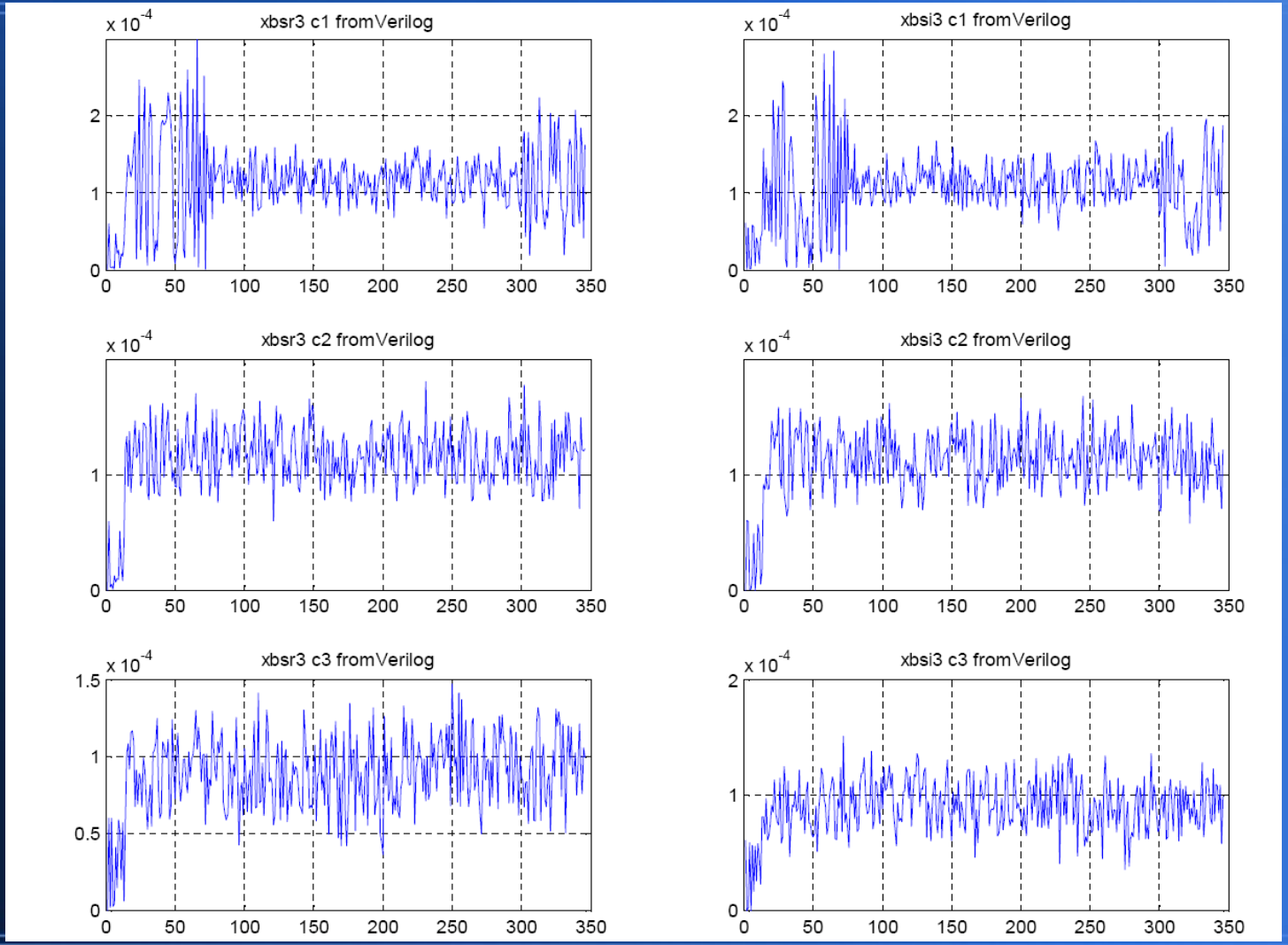
Design Verification
Error of FIR0 and QM

Error after FIR0 and QM processing



Design Verification
Error of FIR1, FIR2, FIR3

Error after
- FIR1,
- FIR2, and
- FIR3
- processing



Synthesis Result

Design	240MHz Design	
FPGA Device	Xilinx Virtex 2: speed grade 4 (space grade)	
Speed (Bandwidth)	84.9MHz	
	Used / Total	Percentage %
Number of Multipliers	76 / 96	79%
Number of Slices	8,247 / 14,336	57%
Number of Flip Flops	9,736 / 28,672	33%
Number of 4 input LUTs	13,868 / 28,672	48%

- Resource usage includes FIR0, QM, FIR1, FIR2, and FIR3 for complex data processing for H-pol, V-pol and noise

Future Work

- Further reduce multiplier count with VKCMs (constant multipliers implemented in LUTs)
- FIR0 can save 16x2 full multipliers by replacing with VKCMs
- How to detect error in computation?
- How to selectively scrub BRAMs?