# Low-density Parity-check Decoding on the Sandbridge Sandblaster SDR Platform


**By**

Murugappan Senthilvelan *
Meng Yu *
Daniel Iancu *
Mihai Sima ‡
Michael J. Schulte †

* Optimum Semiconductor Technologies Inc.
‡ University of Victoria
†University of Wisconsin – Madison

# Outline

- Research overview and contributions
- Background
  - LDPC decoding
  - CORDIC algorithm
- CORDIC algorithm design considerations
- CORDIC ISA extensions & hardware estimates
- Evaluation platform and methodology
- Results
  - Computational performance
  - Arithmetic accuracy
  - Power consumption estimates
- Summary

# Research Overview

- Low-density Parity-check (LDPC) Forward Error Correction (FEC) codes exhibit excellent error correction capability, close to Shannon capacity
- LDPC codes adopted by state-of-art wireless protocols (WiMAX, LTE) for improving spectral efficiency
- Among proposed LDPC decoding algorithms, Belief Propagation (BP) algorithm is a good approximation to optimal ideal decoder
- BP algorithm requires compute-intensive hyperbolic trigonometric functions
- 32% of computation time spent on computing hyperbolic functions

# Research Overview

- COordinate Rotation DIgital Computer (CORDIC) algorithm provides convenient ways to compute hyperbolic functions
- However, sequential CORDIC algorithm inefficient for a full-software implementation
- Instruction Set Architecture (ISA) extensions to support CORDIC algorithm proposed
- Proposed ISA extensions are evaluated on the Sandbridge Sandblaster SB3000 SDR platform

# Research Contributions

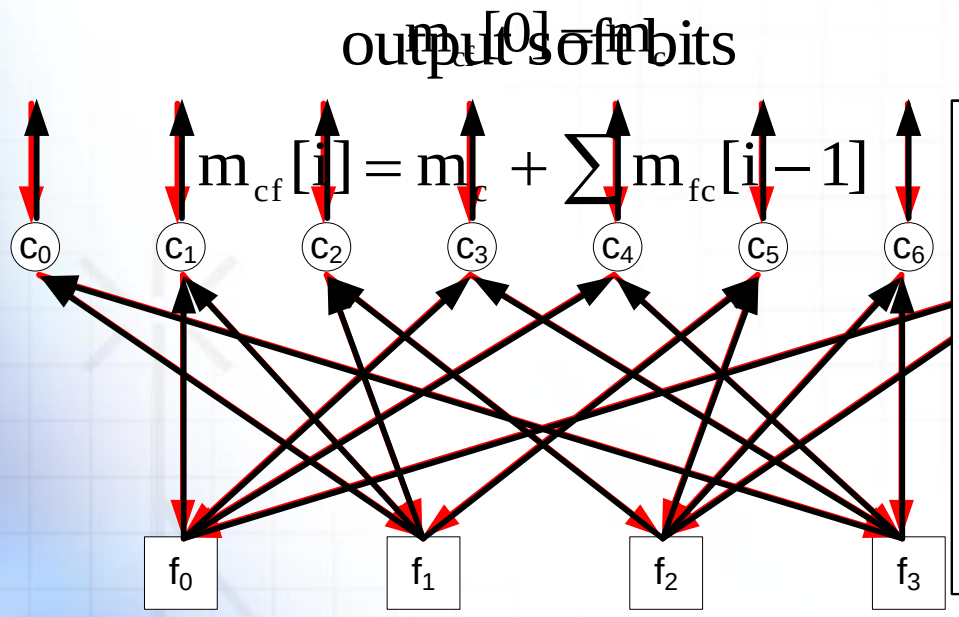
- Addresses the high computational complexity of the LDPC BP decoding algorithm on SDR platforms
- Performs analysis to determine the set of CORDIC ISA extensions to evaluate hyperbolic trigonometric functions
- Provides estimates on computation performance, arithmetic accuracy, and power consumption when using CORDIC ISA extensions
- Provides hardware estimates for CORDIC functional unit that implements the CORDIC ISA extensions

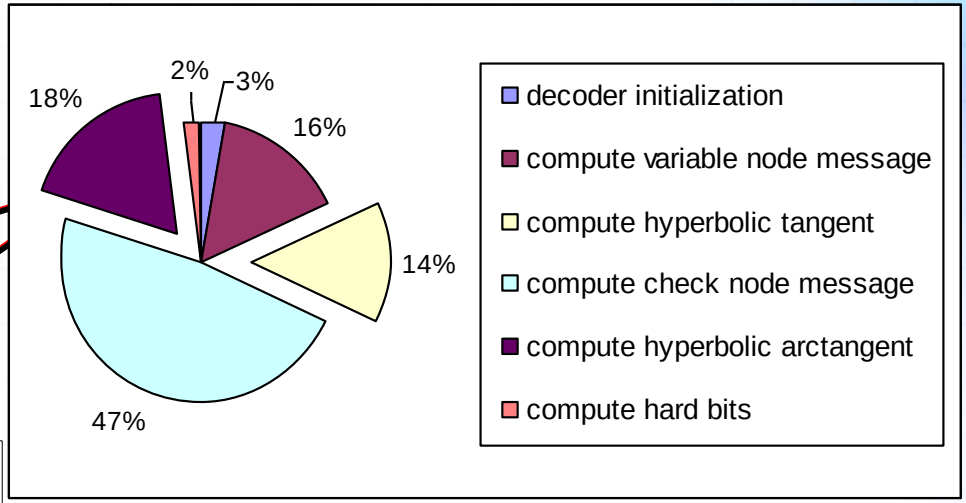
# Summary of Results

- Demonstrates computational speed-ups of 1.2*x* on the LDPC BP decoding algorithm when using CORDIC ISA extensions
- CORDIC implementations have better numerical accuracy compared to baseline implementation
- Combinatorial area of the CORDIC functional unit similar to the Vector Multiply-Accumulate unit on the SB3000 platform
- 5% to 15% power savings obtained depending on the type of power saving methodology used

# Low Density Parity Check Decoding

- Best error correcting codes, close to Shannon theoretical limits
- Decoded iteratively using message passing algorithms




$$m_{fc}[i] = 2\tanh^{-1}\left(\prod \tanh\left(m_{cf}[i]/2\right)\right)$$

**Hyperbolic Tangents**

# The CORDIC Algorithm

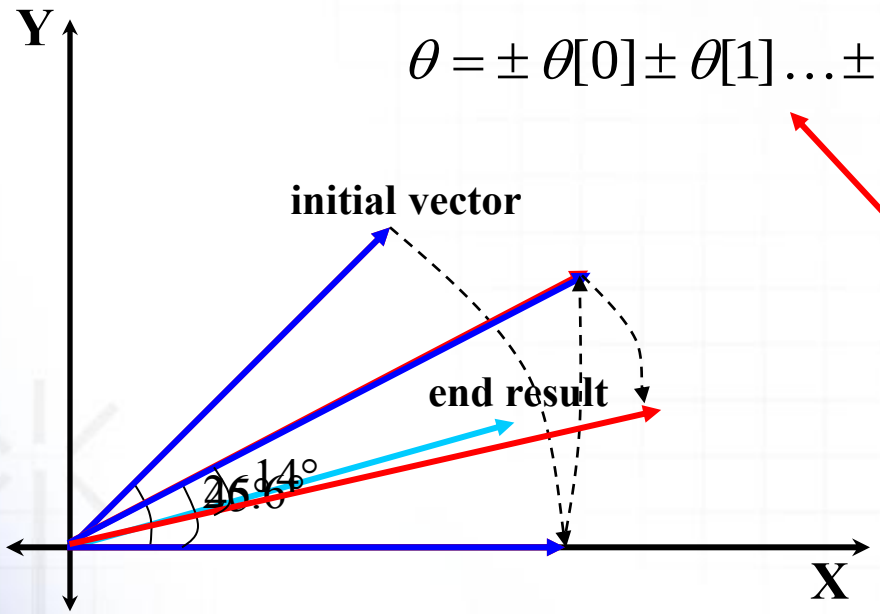
- Iterative sequential algorithm to perform vector rotations in different co-ordinate systems
- Provides convenient ways to move between Cartesian and Polar Coordinate systems
- This freedom provides natural ways to perform
  - Vector rotations
  - Multiplication and division
  - A wide variety of elementary functions
    - $\sin(\theta)$, $\cos(\theta)$, $\tan(\theta)$, $\tan^{-1}(\theta)$
    - $\sinh(\theta)$, $\cosh(\theta)$, $\tanh(\theta)$, $\tanh^{-1}(\theta)$
    - $\exp(\alpha)$, $\ln(\alpha)$, $\sqrt{\alpha}$

But…

- CORDIC algorithm slow when implemented completely in software
- Hence, we investigate processor support for the CORDIC algorithm

# Background: The CORDIC Algorithm

- Basic idea is to rotate vectors by splitting rotation angle into a sequence of micro rotations by known elementary angles




$$\theta = \pm\theta[0] \pm \theta[1] \ldots \pm \theta[n-1] = \sum_{i=0}^{n-1} \sigma_i \theta_i$$

In circular co-ordinate system elementary angles computed using formula:

$\theta[i] = \tan^{-1}(2^{-i})$

**CORDIC iteration 3**

Angle accumulator z = -19.6°

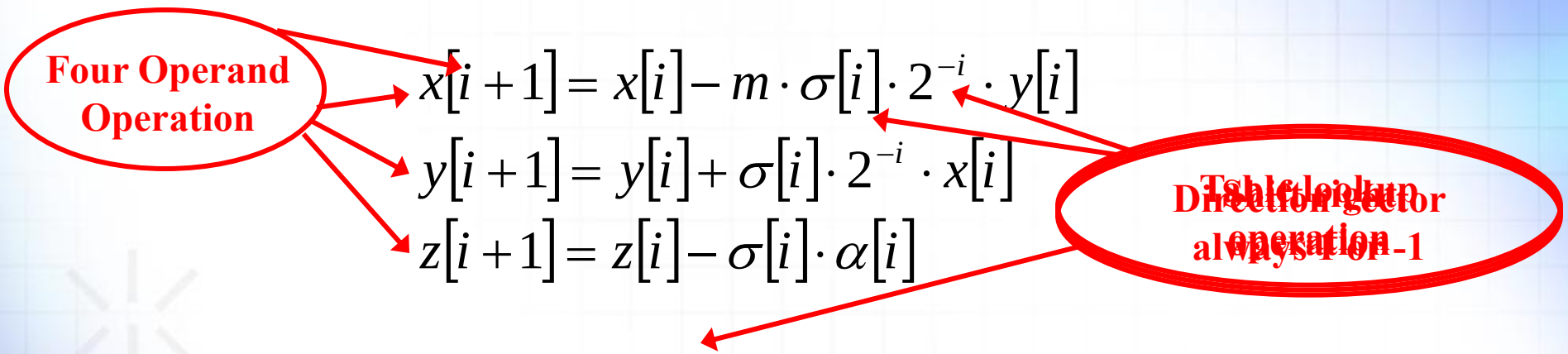
Micro-rotation angle = -45.0°

New angle accumulator = -13.5°

# Background: The CORDIC Algorithm

- Uses simple lookup, shift, and add operations

Generalized CORDIC equations for vector defined by (x,y) with an angle z




where, m = 1 and $\alpha[i] = \tan^{-1}(2^{-i})$ for circular CORDIC

m = 0 and $\alpha[i] = 2^{-i}$ for linear CORDIC

m = -1 and $\alpha[i] = \tanh^{-1}(2^{-i})$ for hyperbolic CORDIC

Scale factor compensation performed by two constant multiplications

# Background: The CORDIC Algorithm

- Two modes of operation

***Rotation mode*** ← vector rotation operation

**Angle accumulator is initialized with rotation angle.**

**Direction of rotation *σ[i]* chosen to decrease residual angle magnitude**

$$\sigma[i] = \begin{cases} -1 & if \ z[i] < 0 \\ +1 & otherwise \end{cases}$$

***Vectoring mode*** ← Cartesian to polar conversion

**Rotates input vector to align resulting vector with *x* axis**

**such that *y* approaches 0**

$$\sigma[i] = \begin{matrix} -1 & if \ y[i] \geq 0 \\ +1 & otherwise \end{matrix}$$

# CORDIC Algorithm Design Considerations

## Convergence

- Maximum angle that can be handled by CORDIC algorithm is sum of all micro-rotation angles
- Convergence angle defined by basic CORDIC algorithm for hyperbolic CORDIC approach is 64˚
- For good decoding performance, LDPC decoding req
  (± 180 ˚), there are two options
  - Use mathematical identities
  - Perform negative CORDIC iterations

Large or Small scale factors – Loss of Precision

| Coordinate System | Basic CORDIC algorithm | | | Including two extra iterations | | |
|---|---|---|---|---|---|---|
| | Iteration sequence | Convergence range | Scale factor | Iteration sequence | Convergence range | Scale factor |
| Circular | 0,1,2,..,15 | ± 99.88˚ | 1.64 | -2,-1,0,..,13 | ± 239.00˚ | 15.18 |
| Linear | 1,2,3,..,16 | ± 57.29˚ | 1.00 | -1,0,1,..,14 | ±229.18˚ | 1.00 |
| Hyperbolic | 1,2,3,4,4,..,14 | ±64.06˚ | 0.83 | -1,0,1,..,13 | ±197.38˚ | 0.26 |

# CORDIC Algorithm Design Considerations

## Operand Representation Format

- Coordinates x and y, and iteration counter
  - Represent using 2's complement numbers
- Angle Representation
  - 2's complement number in radian format
    - Commonly used representation for angles
    - No wrap-around property about Π radians
- DSP algorithms typically use radian format, so CORDIC angles represented using radian format
- Pre-computed micro-rotation angles with extra precision stored in lookup table to match the input fixed-point data format

# CORDIC Algorithm Design Considerations

## Precision and Computational Accuracy

- 'n' CORDIC iterations needed for 'n' bits of precision
  - Wireless algorithms typically use 16-bit data.
  - 16 CORDIC iterations needed
- At least $\log_2(n)$ additional fractional bits needed to reduce impact of rounding errors
  - With 16-bit data, 20-bit internal CORDIC precision

# CORDIC Algorithm Design Considerations

## CORDIC Iterations per Instruction

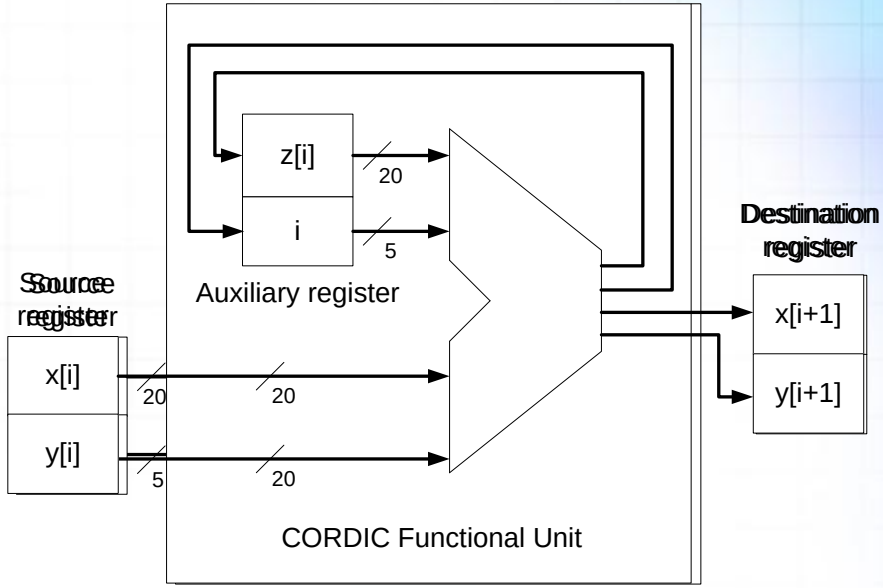
- Long latency DSP pipelines can tolerate multiple CORDIC iterations per instruction
- All 16 CORDIC iterations cannot fit in one instruction
  - Four inputs, four outputs
  - Not a typical DSP instruction (one to three inputs, one output)
  - Four inputs can be packed in two registers (feasible)
  - Four outputs require two register write backs (not feasible with one register write port)
- Two design choices
  - Use implicitly addressable temporary storage registers (Full-CORDIC approach)
  - Split the multiple data paths across multiple instructions (Semi-CORDIC approach)

# Full–CORDIC Approach

- Augment the CORDIC unit with auxiliary registers for implicit source and destination operands
- Four different instructions defined for the full-CORDIC approach
  - Configure Set CORDIC
  - Configure Read CORDIC
  - CORDIC Rotate
  - CORDIC Vector

But …

- Auxiliary register hardware overhead substantial
- Additional state needs to be saved and restored while interrupt processing

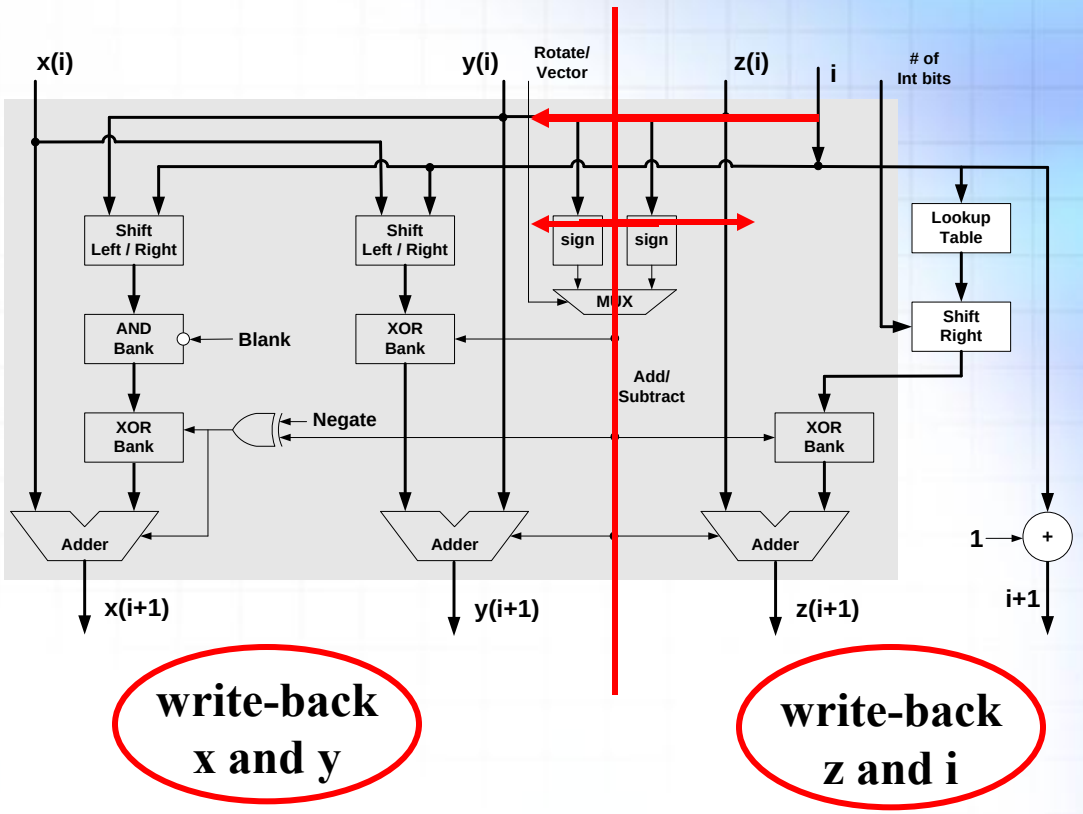


# Semi–CORDIC Approach

- Splits execution of four data paths into two instructions
- Four different instructions defined for the semi-CORDIC approach
  - XY CORDIC Rotation
  - ZI CORDIC Rotation
  - XY CORDIC Vectoring
  - ZI CORDIC Vectoring
- Fits in traditional DSP architecture
- No extra state added

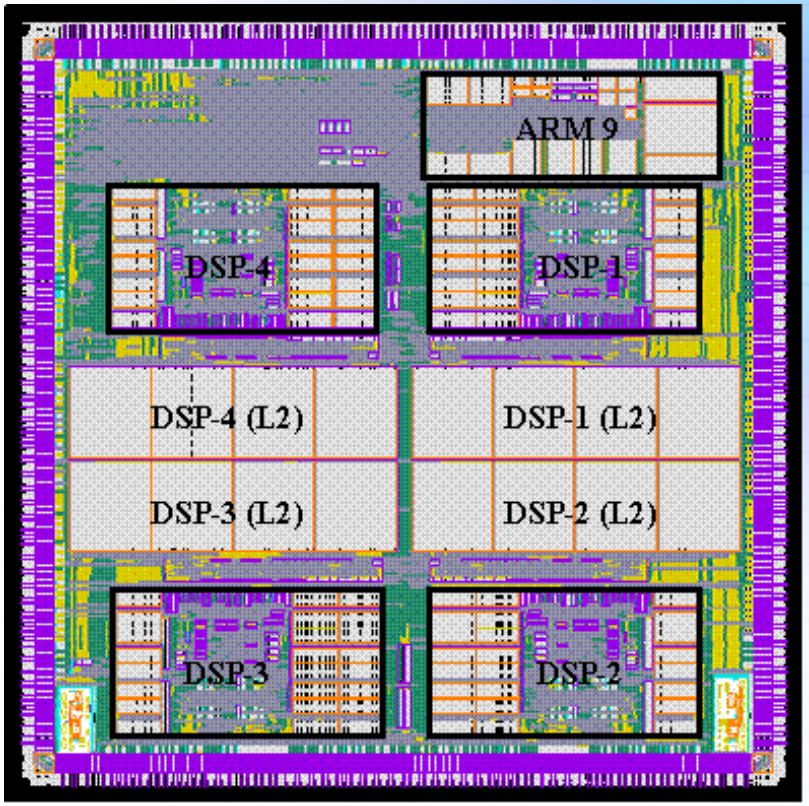
But…

- Almost doubles the number of CORDIC instructions

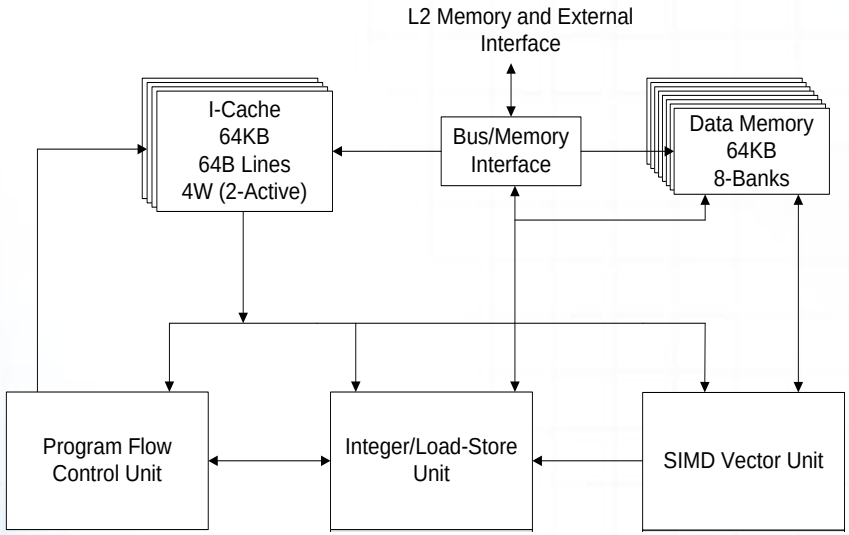


# Evaluation Platform – Sandbridge SB3000

- Designed for efficient software execution of wireless protocol physical layers, protocol stacks
- Four multithreaded SIMD VLIW DSP Cores, ARM applications processor
- High-speed RF interfaces and standard peripheral interfaces
- Multiple wireless protocols implemented; WiMAX, CDMA, WLAN, GSM/GPRS
- State-of-the-art SDR



# Evaluation Platform – Sandblaster DSP

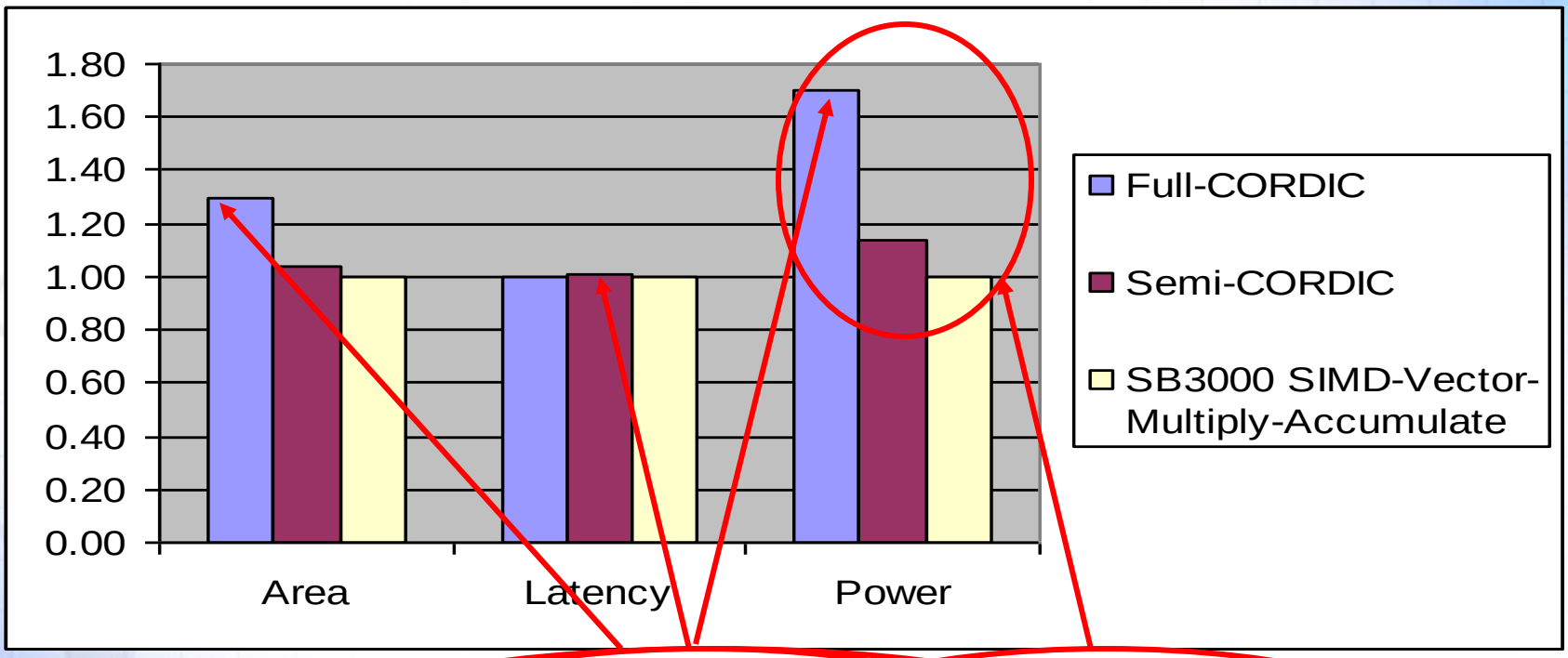



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Ld/St | Inst Dec | RF Read | Agen | XFer | Int Ext | Mem 0 | Mem 1 | Mem 2 | WB |
| ALU | Inst Dec | Wait | RF Read | Exec 1 | Exec 2 | XFer | WB | | |
| I_Mul | Inst Dec | Wait | RF Read | Exec 1 | Exec 2 | Exec 3 | XFer | WB | |
| V_Mul | Inst Dec | VRF Read | Exec 1 | Exec 2 | Exec 3 | Exec 4 | XFer | VRF WB | |

- 8 threads per processor core, 4 such cores ( 32 threads total)
- Vector processing unit performs four–way SIMD operations on 16-bit, 32-bit or 40-bit data types
- Operations performed in parallel using compound instructions
- Threads issue one instruction per cycle – 8 threads total
- Vector instruction pipeline consists of 4 execute stages

# Hardware Synthesis Estimates

- Synthesis performed using Synopsys Design Compiler and TSMC's 65nm CMOS standard cell library.
- All implementations are 4-way SIMD, fit in four execute stages of SB3000 and contain pipeline registers




Latencies of CORD
similar to SB300

Scale factor used to model power of CORDIC instructions

# CORDIC ISA Extensions in Sandblaster Tool Chain

- CORDIC instructions added to the SaDL Python files
- Sandblaster tools retargeted using new architecture definitions
- CORDIC instruction used in wireless algorithms using intrinsic calls

```
short op1[12], op2[12];

for(i=0;i<(CORDIC_PRECISION/CORDIC_ITR_PER_INST);i++){
  __sb_q_simd_vector_cordic_xy_rotation(op1, op1, op2);
  __sb_q_simd_vector_cordic_zi_rotation(op2, op2);
}
```

- Compiler inserts assembly instructions, generates supporting loads/stores, optimizes and schedules the instructions
- x86 functions that emulate the functionality of ISA extensions added to simulator to simulate application functionality

# Computational Performance & Arithmetic Error

- All memory accesses assumed to be single cycle
- Wireless algorithms are all single-threaded implementations
- Wireless algorithms simulated for functional correctness and dynamic compound instruction counts measured

$$\text{Computational Speedup} = \frac{\left(\begin{array}{l}\text{Dynamic compound instruction count of}\\ \text{non}-\text{CORDIC baseline implementation}\end{array}\right)}{\left(\begin{array}{l}\text{Dynamic compound instruction count}\\ \text{of new implementation}\end{array}\right)}$$

- Arithmetic error expressed as Normalized Root Mean Square Error when compared against ideal floating-point implementation

$$\text{Arithmetic Error} = \sqrt{\frac{\sum\left(\text{Value}_{\text{float}} - \text{Value}_{\text{fixed\_to\_float}}\right)^2}{\sum\left(\text{Value}_{\text{float}}\right)^2}} \quad \text{x} \quad 100$$

# Power Estimation

- Analytical methodology using instruction profiling and power consumption measurements from real hardware
- Sandblaster DSP instruction set divided into instruction sub-classes based on computational complexity
- Power for each instruction sub-class measured on hardware using test programs with infinite loops of instructions from same sub-class
- Power of a VLIW compound instruction modeled as

$$P_{base} + P_{inst\_pipeline1} + P_{inst\_pipeline2} + P_{inst\_pipeline3}$$

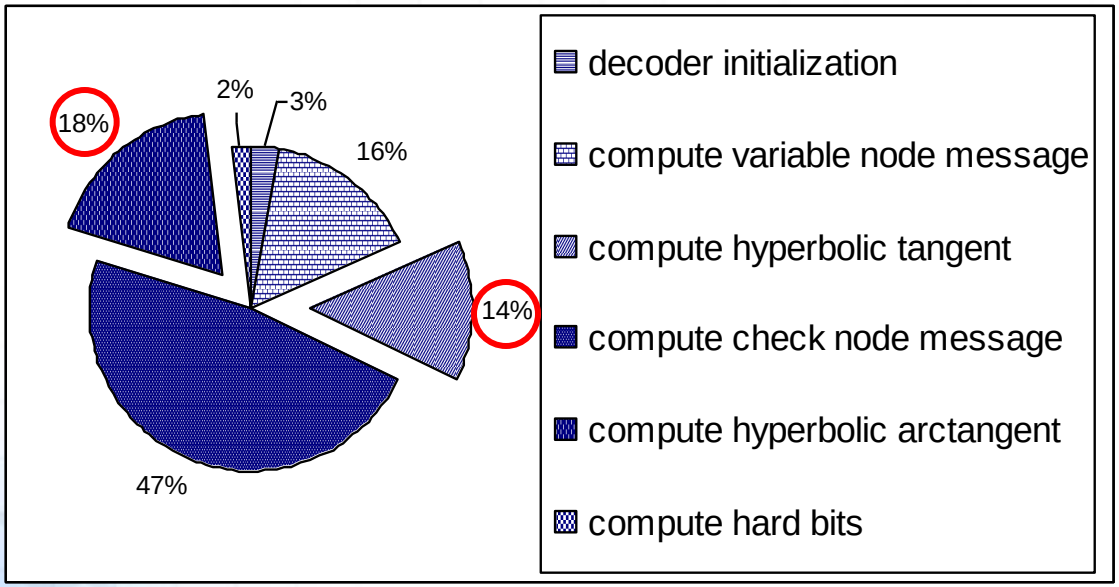
- $P_{base}$ is power of NOP instruction – does not include power of instruction pipeline
- $P_{inst\_pipeline}$ can be computed by subtracting $P_{base}$ from power of compound instruction packet with one instruction

# Power Estimation

- Wireless algorithms profiled on the simulator to measure number of instructions in each instruction sub-class
- Divide by total dynamic compound instruction count to compute fraction of computation time $T_{frac}$ utilized by instruction sub-class
- Total instruction pipeline power = $\Sigma\ T_{frac}[i] * P_{inst\_pipeline}[i]$
- To account for capacitive loading, routing, and leakage due to CORDIC unit, additional 15% of instruction pipeline power added
- Total power = Total instruction pipeline power + $P_{base}$
- When comparing two implementations, total dynamic compound instruction count picked from longer running implementation
- For shorter running implementation, three options
  - Always stay ON (base power consumed, Most power)
  - Clock gate core (leakage power consumed, Nominal power)
  - Turn OFF core (zero power consumed, Least power)
- Verification on real FIR filter yields error within 10%

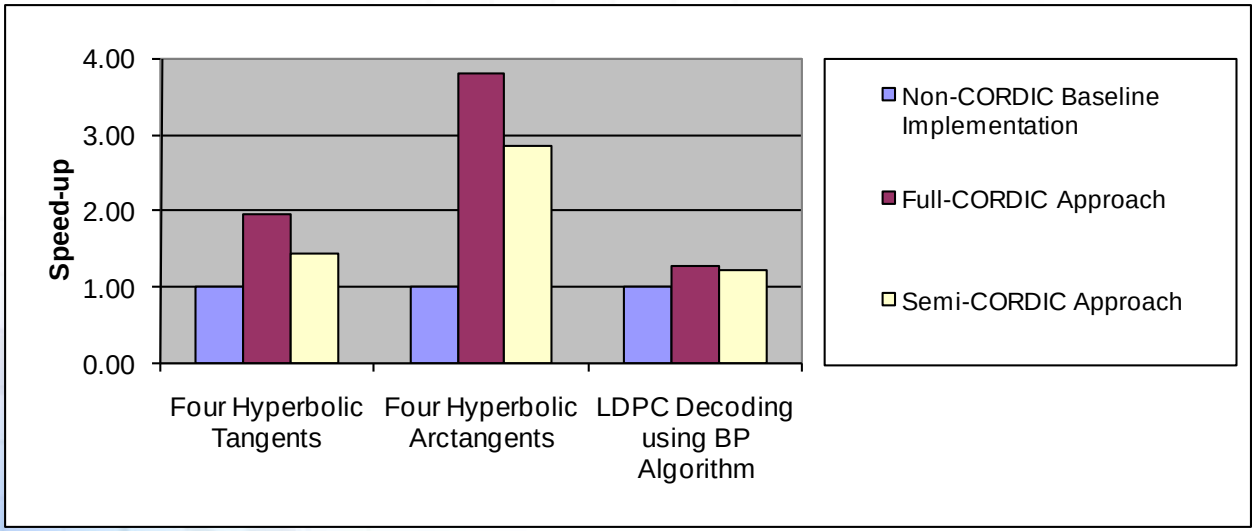
# Baseline Implementation

- LDPC decoder used in evaluation is from Mobile WiMAX IEEE 802.16e (block size =2304 bits, half-rate coding)
- Hyperbolic functions computed using polynomial approximations or table lookup operations
- To meet numerical accuracy table lookup used 8 KB and 32 KB table sizes
- 32% of computation time spent in evaluation of hyperbolic CORDIC functions



# Computational Performance

- Significant speed-ups observed on elementary operations - hyperbolic tangent and arctangent
- Speed-ups of 1.27*x* and 1.21*x* when using Full-CORDIC and Semi-CORDIC approaches
- Speed-up on elementary operations does not translate to LDPC decoder because they constitute only 32% of computation time
- If the repetitive patterns in WiMAX LDPC decoder used to improve memory performance, higher speed-ups when using CORDIC instructions



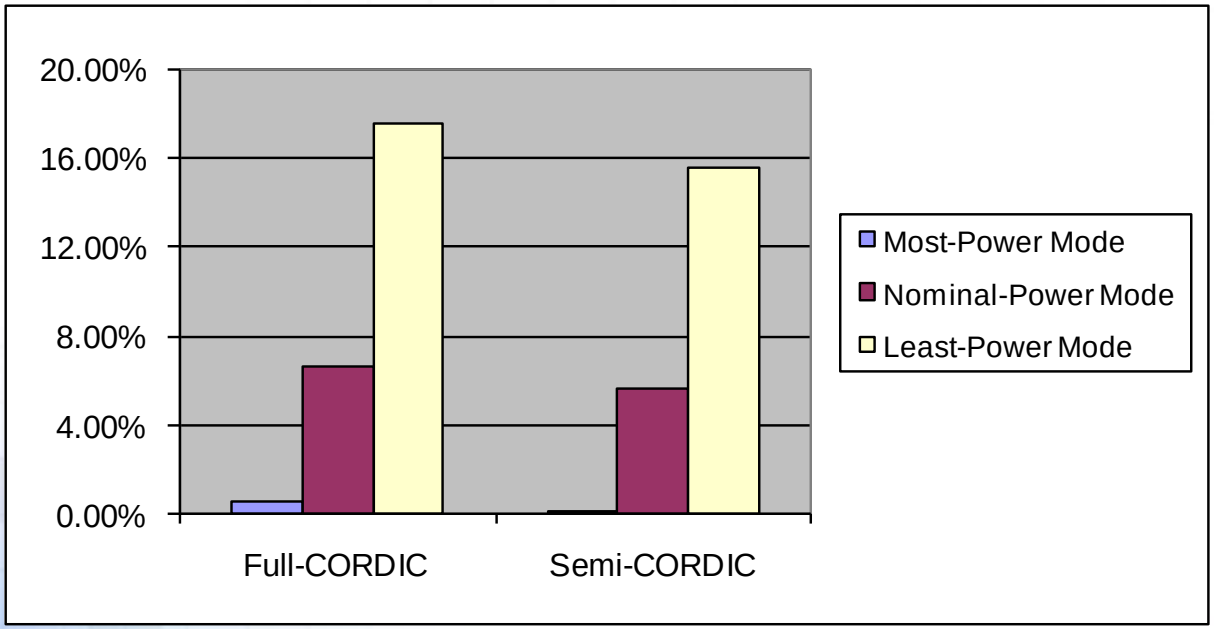
# Arithmetic Error

- Both CORDIC implementations demonstrated similar computational accuracy
- CORDIC implementations performed better than the baseline non-CORDIC baseline implementation when comparing the NRMS arithmetic error

| Non-CORDIC Baseline Implementation | CORDIC Implementations |
|---|---|
| 2.08% | 0.05% |

# Power Consumption Estimates

- Results for semi-CORDIC approach shown, similar results observed for full-CORDIC approach
- CORDIC approaches perform slightly better even without any power saving technique
- With power saving techniques an average of 5% savings in nominal power mode, 15% savings in least power mode



# Summary

- When performing LDPC decoding, CORDIC augmented Sandblaster implementations exhibits
  - 1.2x speedup over non-CORDIC baseline implementation
  - Provides better arithmetic accuracy
  - 5% to 15% reduced power consumption
  - Low memory overhead
- Further benefits can be observed if repetitive LDPC decoder patterns can be exploited
- CORDIC ISA extensions presented provide significant performance benefits with reasonable hardware overhead
- Other variations of CORDIC algorithm can provide further improvements
- Techniques presented in this research applicable to other high-performance DSP systems

# Thank You !

# Questions ?

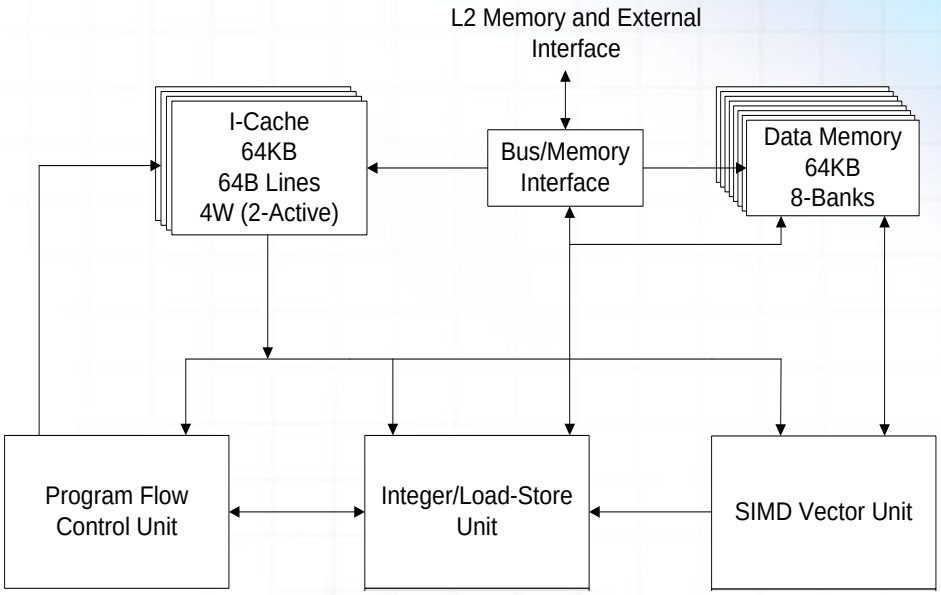
# Backup slides

# Previous Research – SDR Architectures

- SDR architectures in existence / proposed
  - Sandbridge SandBlaster SB3000
  - NXP Embedded Vector processor
  - European Research Consortium – Flexible air interface baseband processor
  - Infineon X-Gold SDR 20
  - Icera Livanto
  - University of Michigan – SODA, Ardberg, AnySP

  (common feature: SIMD architecture, Applications Processor)
- Other architectures for Signal Processing and Multimedia
  - Non-customizable Static ISA – TigerSharc, TI TMS320C6x,
  - Customizable Static ISA –ARC 700, Tensilica Xtensa
  - Customizable Dynamic ISA – Stretch S6000, REMARC, Piperench, MOLEN

# Evaluation Platform – Sandblaster DSP




- 8 threads per processor core, 4 such cores ( 32 threads total)
- Operations performed in parallel using compound instructions
- Vector processing unit performs four–way SIMD operations on 16-bit, 32-bit or 40-bit data types
- High speed 64-bit data busses allow loading four 16-bit data into the vector registers

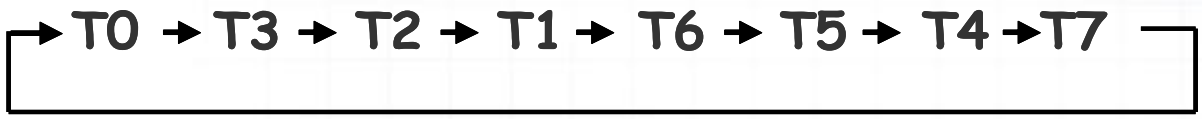
# Evaluation Platform - Sandblaster DSP

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Ld/St | Inst Dec | RF Read | Agen | XFer | Int Ext | Mem 0 | Mem 1 | Mem 2 | WB |
| ALU | Inst Dec | Wait | RF Read | Exec 1 | Exec 2 | XFer | WB | | |
| I_Mul | Inst Dec | Wait | RF Read | Exec 1 | Exec 2 | Exec 3 | XFer | WB | |
| V_Mul | Inst Dec | VRF Read | Exec 1 | Exec 2 | Exec 3 | Exec 4 | XFer | VRF WB | |

- Pipeline length – 8 stages for most instructions
- Threads issue one instruction per cycle – 8 threads total
- No data hazards in pipeline, so no dependency checking and forwarding hardware.
- Vector instruction pipeline consists of 4 execute stages.

# The Sandblaster SDR

- Multithreaded SIMD processor executing compound instructions optimized for handset radio applications
- Eight concurrent threads supported on each processor core – Token Triggered Threading




  - Each thread executes an instruction simultaneously
  - Only one thread may issue instruction on cycle boundary
  - Token passed between threads indicates next issuing thread
  - Instructions complete in 8 DSP cycles (1 thread cycle) before next instruction issued, hence all true data dependencies eliminated

- 4 such processor cores ( total of 32 threads) and 1 ARM processor on the SB3011

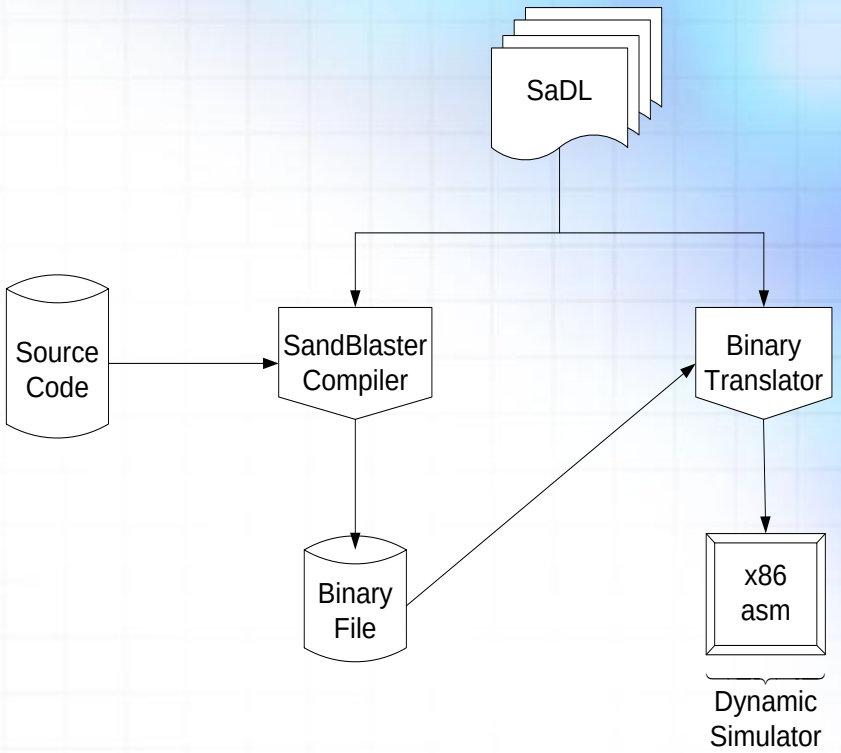
# Evaluation Platform – Sandblaster DSP

```
L0: lvu     %vr0, %r3, 8
||  vmulreds %ac0, %vr0, %vr0, %ac0
||  loop    %lc0, L0
```

- Compound instruction implements inner loop of a vector sum-of-squares computation
- 4-way SIMD vector operations make use of data level parallelism
- Compound instruction packets make use of instruction level parallelism
- Long latency vector-multiply-reduce instruction
- Latency hidden by multithreading

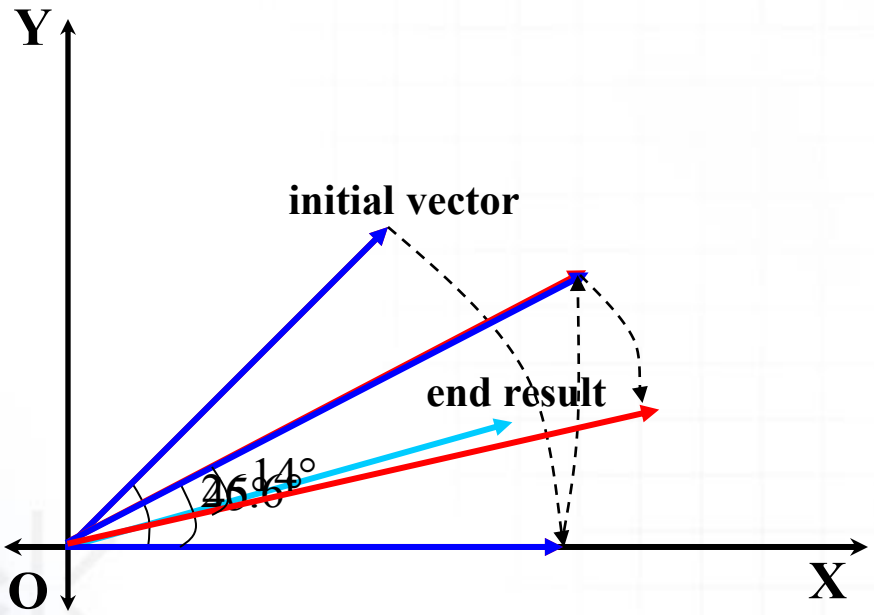
# Evaluation Platform - Sandblaster Tool chain

- Sandbridge architecture Description Language (SaDL)- collection of Python files that abstracts architecture from tools
- Compiler supports sophisticated loop optimizations, memory disambiguation, and automatic vectorization in addition to standard scalar optimizations




- Compiler packs multiple instructions in compound instructions to utilize instruction-level parallelism
- Compiler generates 4-way SIMD instructions to utilize data-level parallelism
- Simulator is cycle-accurate, system-level simulator

# CORDIC Rotation




Vector V = (1,1)

Initial angle = 45°

Rotate by = -30°

**CORDIC iteration 3**

Angle accumulator = -18.4°

Micro-rotation angle = -45.0°

New angle accumulator = -63.4°

# CORDIC Functions

| Coordinate System | Mode | Initialization | Output |
|---|---|---|---|
| Circular | Rotation | $x_0 = x$<br>$y_0 = y$<br>$z_0 = \theta$ | $x_n = A[n] \cdot (x\cos\theta - y\sin\theta)$<br>$y_n = A[n] \cdot (y\cos\theta + x\sin\theta)$<br>$z_n = 0$ |
| | | $x_0 = 1/A[n]$<br>$y_0 = 0$<br>$z_0 = \theta$ | $x_n = \cos\theta$<br>$y_n = \sin\theta$<br>$z_n = 0$ |
| Circular | Vectoring | $x_0 = x$<br>$y_0 = y$<br>$z_0 = \theta$ | $x_n = A[n] \cdot sign(x_0) \cdot \sqrt{x^2 + y^2}$<br>$y_n = 0$<br>$z_n = \theta + \tan^{-1}(y/x)$ |
| Linear | Rotation | $x_0 = x$<br>$y_0 = y$<br>$z_0 = z$ | $x_n = x$<br>$y_n = y + x \cdot z$<br>$z_n = 0$ |
| Linear | Vectoring | $x_0 = x$<br>$y_0 = y$<br>$z_0 = z$ | $x_n = x$<br>$y_n = 0$<br>$z_n = z + (y/x)$ |
| Hyperbolic | Rotation | $x_0 = x$<br>$y_0 = y$<br>$z_0 = \theta$ | $x_n = A_{-1}[n] \cdot (x\cosh\theta + y\sinh\theta)$<br>$y_n = A_{-1}[n] \cdot (y\cosh\theta + x\sinh\theta)$<br>$z_n = 0$ |
| | | $x_0 = 1/A_{-1}[n]$<br>$y_0 = 0$<br>$z_0 = \theta$ | $x_n = \cosh\theta$<br>$y_n = \sinh\theta$<br>$z_n = 0$ |
| Hyperbolic | Vectoring | $x_0 = x$<br>$y_0 = y$<br>$z_0 = \theta$ | $x_n = A_{-1}[n] \cdot sign(x_0) \cdot \sqrt{x^2 - y^2}$<br>$y_n = 0$<br>$z_n = \theta + \tan^{-1}(y/x)$ |

- A wide variety of transcendental functions can be computed using the CORDIC algorithm apart from rotating vectors

# CORDIC – Special Functions

- A wide variety of transcendental functions can be computed using the CORDIC algorithm apart from rotating vectors

$$\tan\alpha = \sin\alpha/\cos\alpha$$

$$\tanh\alpha = \sinh\alpha/\cosh\alpha$$

$$\exp\alpha = \sinh\alpha + \cosh\alpha$$

$$\ln\alpha = 2\tanh^{-1}[y/x] \text{ where } x=\alpha+1 \text{ and } y=\alpha-1$$

$$(\alpha)^{1/2} = (x^2-y^2)^{1/2} \text{ where } x=\alpha+1/4 \text{ and } y=\alpha-1/4$$

# Previous Research – CORDIC

- Variations of CORDIC algorithm proposed
  - On-line CORDIC
  - Redundant CORDIC
  - Differential CORDIC
- Hardware implementations
  - Sequential CORDIC hardware
  - Pipelined CORDIC hardware
  - Systolic array of CORDIC processing elements for SVD
- My research is first time CORDIC instructions are proposed for programmable DSPs
- Hardware designs for the CORDIC function unit will be investigated

# CORDIC Algorithm Design Considerations

## Operand Representation Format

- Coordinates x and y, and iteration counter
  - Represent using 2's complement numbers
- Angle Representation has two options
  - 2's complement number in radian format
    - Commonly used representation for angles
    - No wrap-around property
  - 2's complement number in Daggett angle format
    - Has wrap-around property
    - Overhead of converting between two formats
- All algorithms under investigation use radian format, so CORDIC angles represented using radian format

# CORDIC Algorithm Design Considerations

- **Precision and Computational Accuracy**
  - **'n' CORDIC iterations needed for 'n' bits of precision**
  - **At least $\log_2(n)$ additional fractional bits needed to reduce impact of rounding errors**

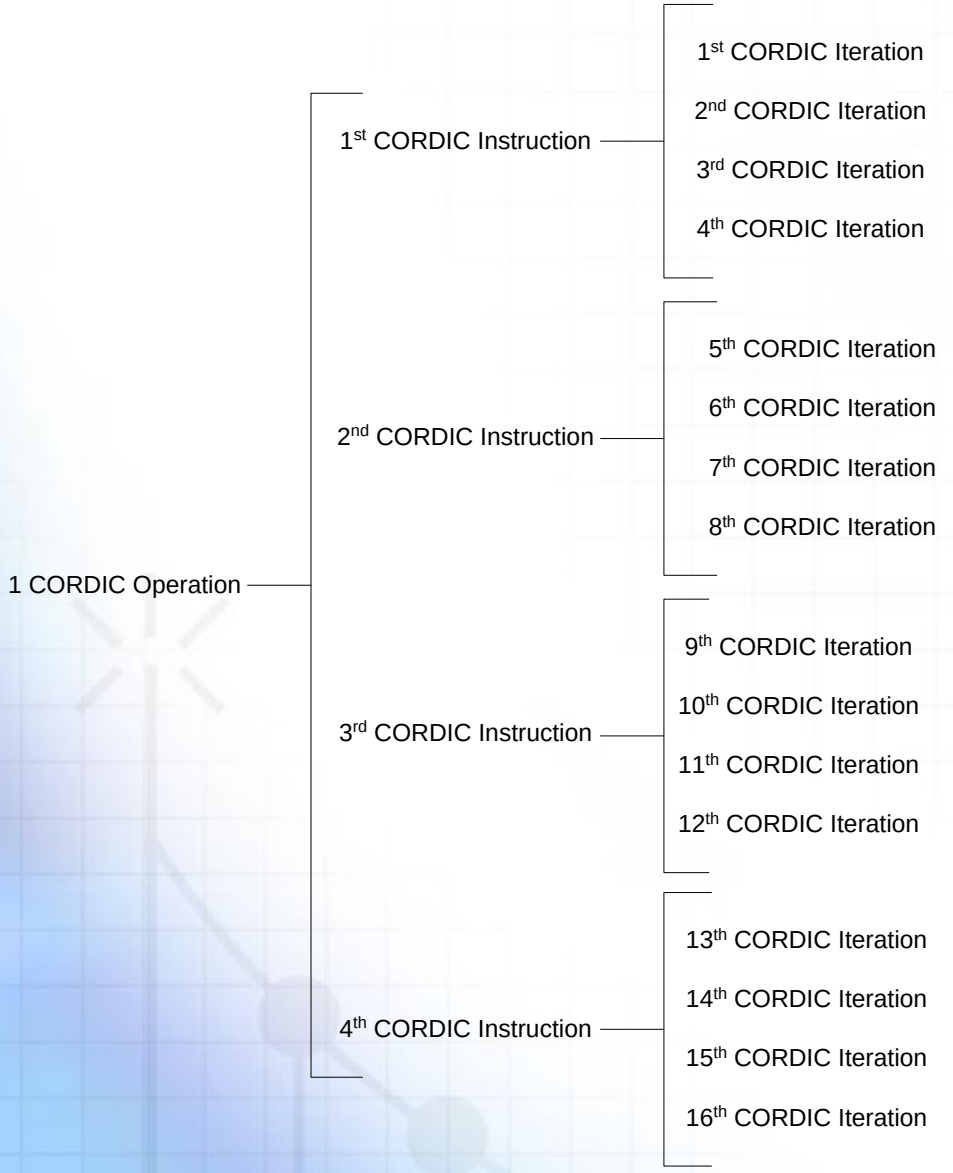
- **Angle Representation**
  - **Daggett's angle format chosen for representing the angle *z* in CORDIC iterations.**

| -180° | 90° | 45° | 22.5° | $360°/2^{n-k}$ | $360°/2^{n}$ |
|---|---|---|---|---|---|
| **1** | **1** | **1** | **0** | . . . | **0** |

**= 135°**

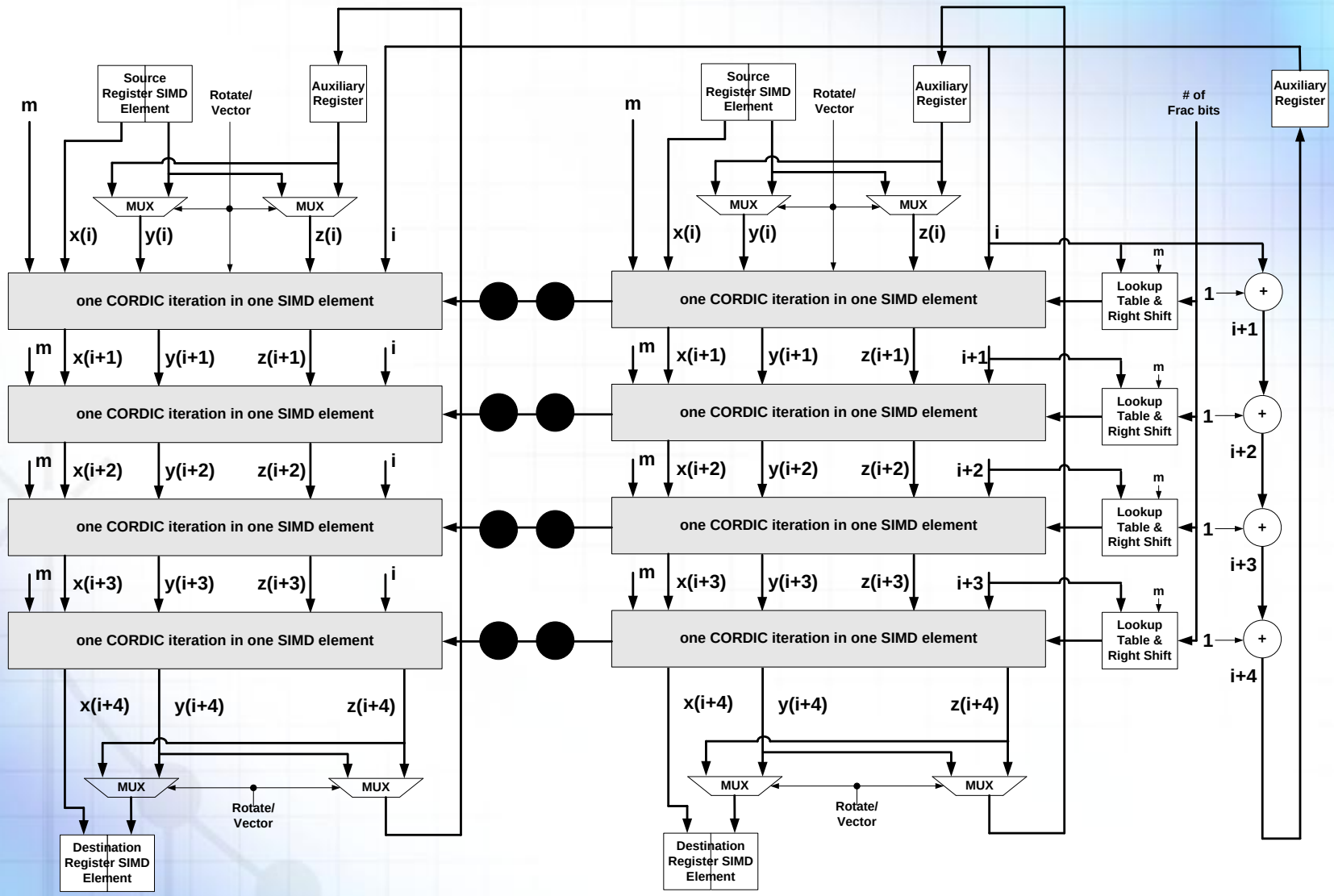
'n' is total number of bits
'k' is bit position (0 to n-1)

# Step 2: Propose CORDIC Instructions

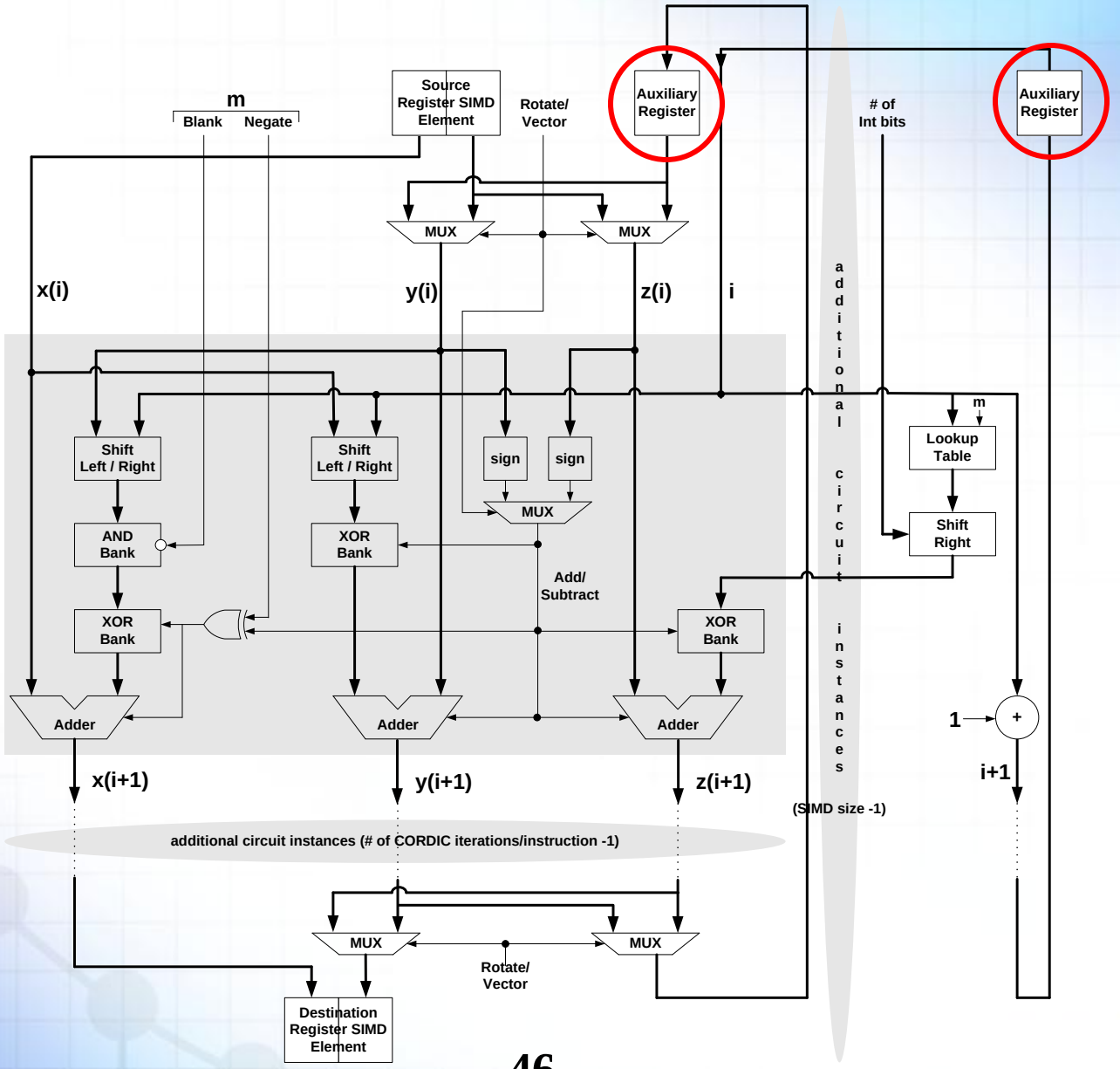



- Multiple CORDIC iterations in one CORDIC instruction
- One/Multiple CORDIC instruction(s) in one CORDIC operation
- k-way SIMD CORDIC instructions

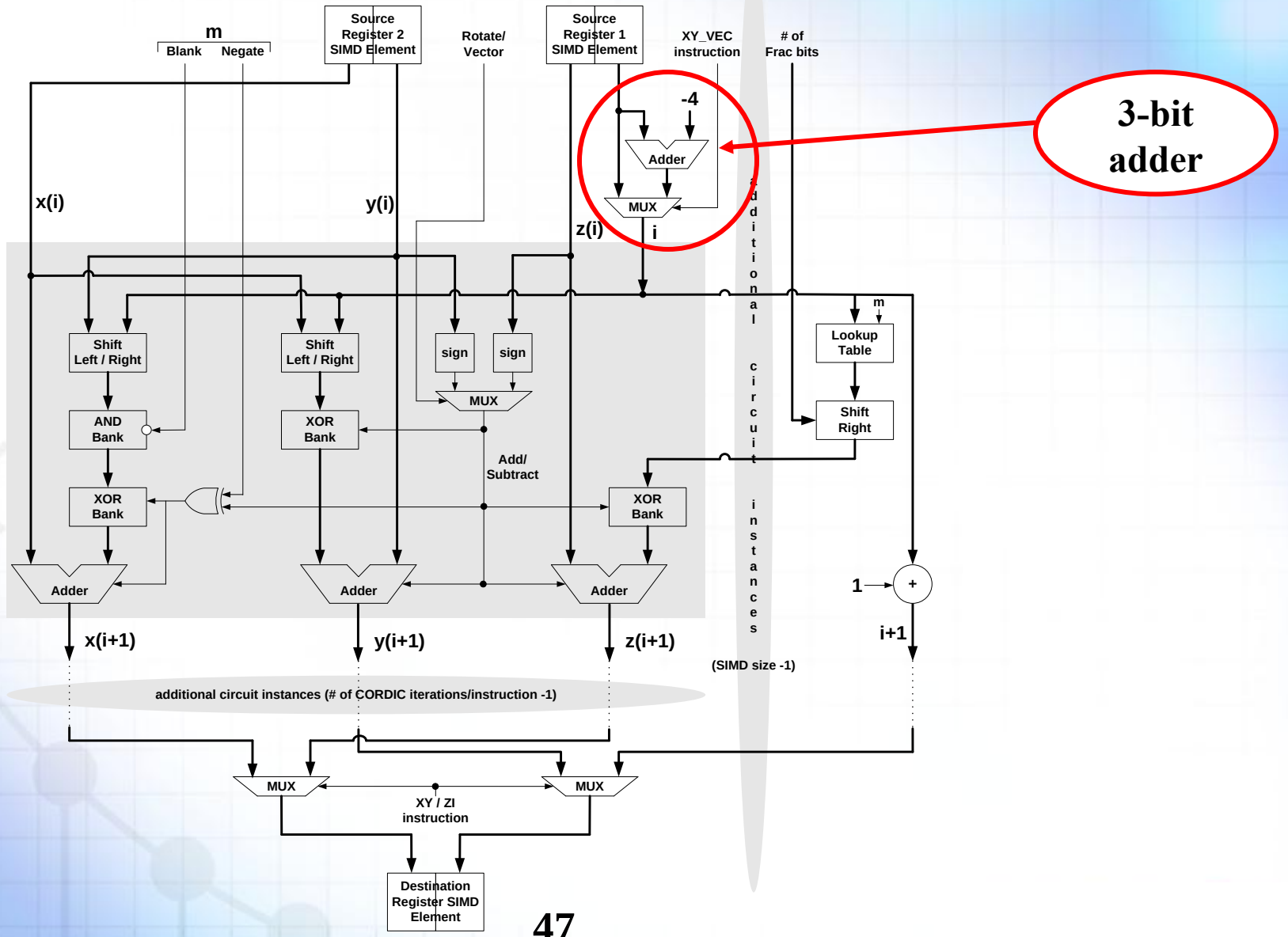
# Full–CORDIC Functional Unit



# Full–CORDIC Basic Hardware Block & Interfaces

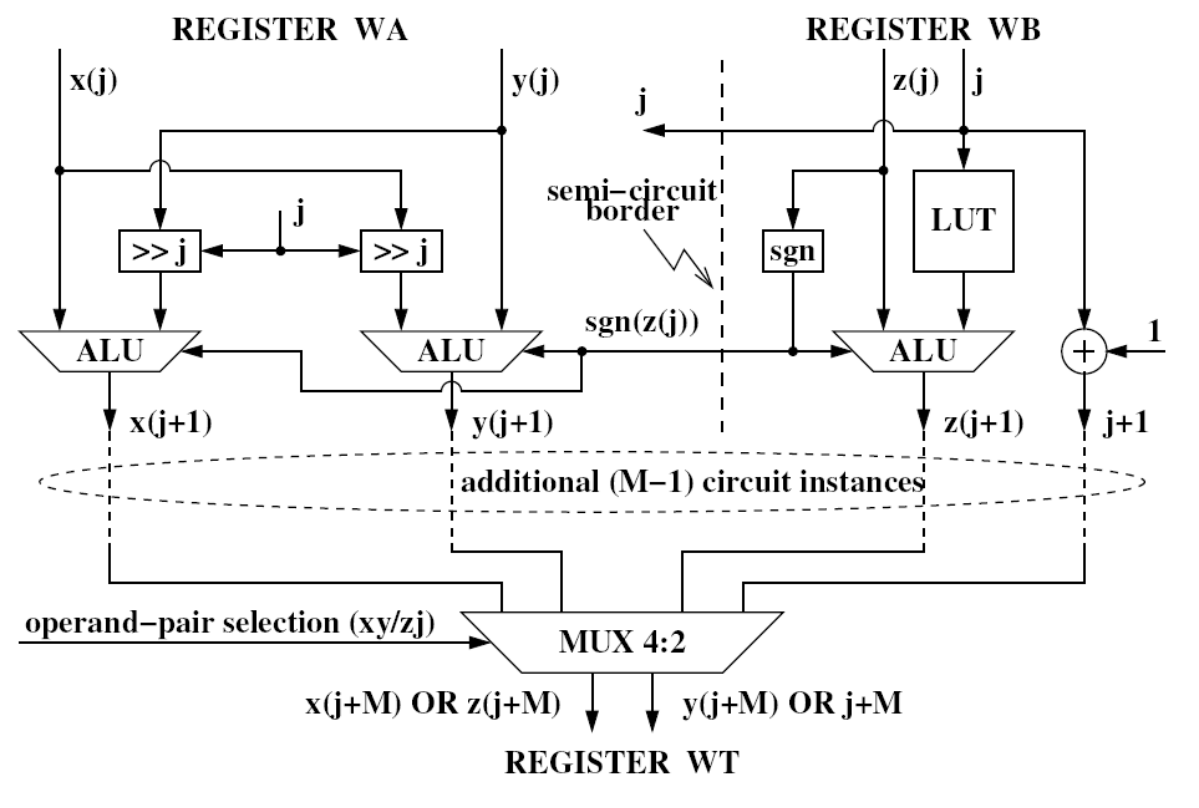


# Semi–CORDIC Basic Hardware Block & Interfaces



# Step 4: Hardware of Semi-CORDIC Rotation

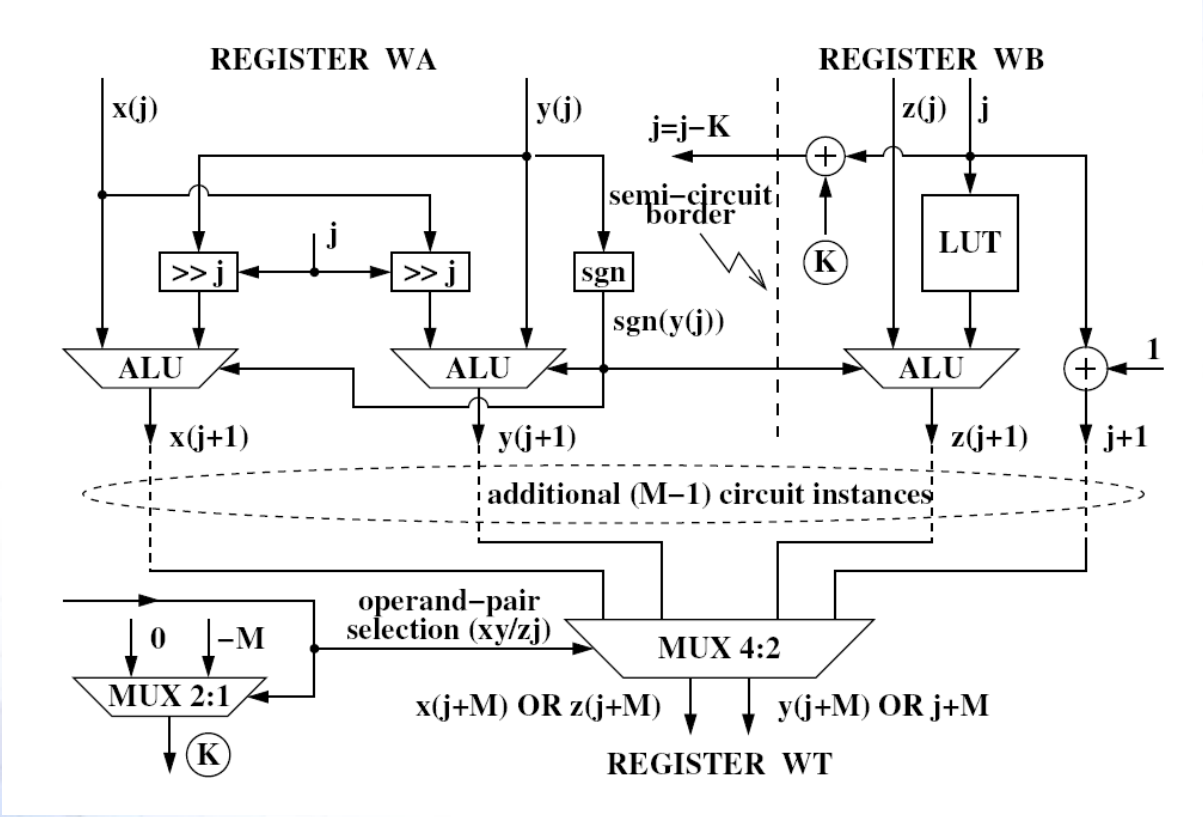
- M instances of the circuit can be deployed to perform M iterations per CORDIC
- XY_ROT_CORDIC and ZI_ROT_CORDIC must be issued in order




Semi-CORDIC rotation

# Hardware of Semi-CORDIC Vectoring

- M instances of the circuit can be deployed to perform M iterations per CORDIC
- ZI_VEC_CORDIC and XY_VEC_CORDIC must be issued in order




Semi-CORDIC vectoring

# Instruction Properties in the SaDL Python Files

- Name of the instruction
- Instruction type (scalar, SIMD vector, or memory access)
- SIMD instruction data size (16, 32, or 40-bits)
- Number and type of input registers
- Number and type of output registers
- Instruction optimization properties, such as unmovable, optimization barrier, not dead
- Instruction opcode
- Functionality of the instruction in x86 host code

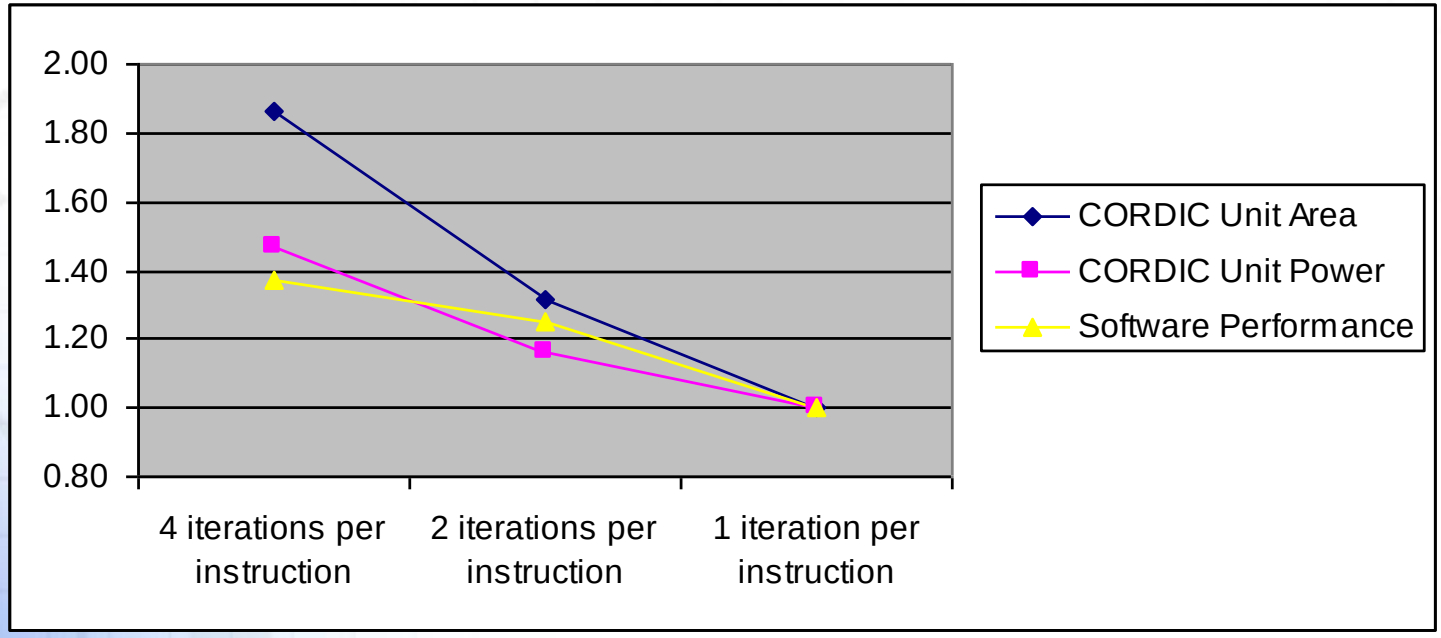
# Hardware Synthesis Estimates

- Synthesis performed using Synopsys Design Compiler and TSMC's 65nm CMOS standard cell library.

| CORDIC ISA Extensions | Area ($um^2$) | | | Latency (ns) | Power (mW) |
|---|---|---|---|---|---|
| | Combinational Area | Non-Combinational Area | Total Area | | |
| Full-CORDIC | 24540 | 11349 | 35889 | 1.64 | 10.53 |
| Semi-CORDIC | 22847 | 5964 | 28811 | 1.66 | 7.08 |
| SB3000 SIMD-Vector-Multiply-Accumulate | 24025 | 3709 | 27734 | 1.65 | 6.20 |

# Hardware - Software Performance Tradeoff

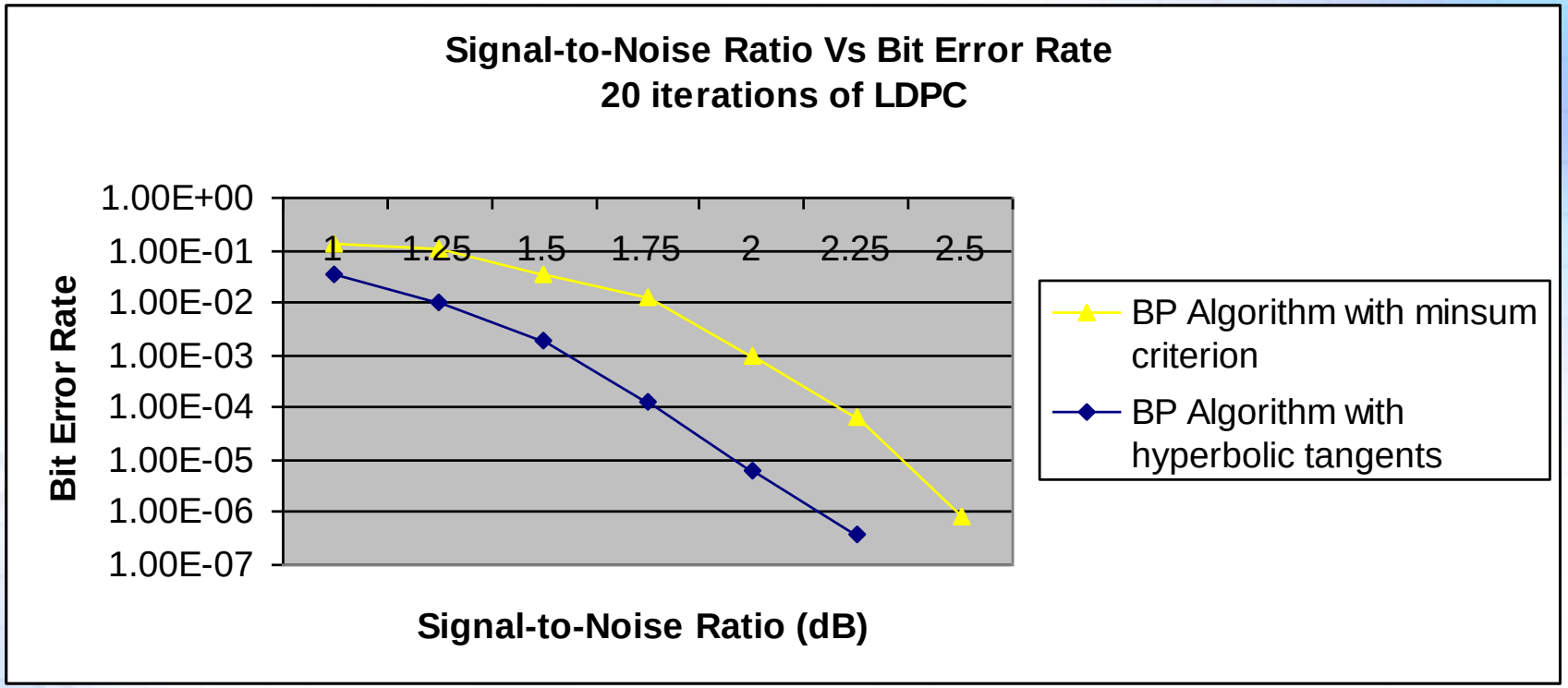
- Straight-forward way to reduce size of CORDIC unit is to reduce number of CORDIC iterations per instruction
- Software performance reduces at a slower rate than CORDIC unit area and can be traded off without too much impact
- Software performance speedup drops from 7.2*x* to 5.2*x* for an 86% reduction in area going from 4 iterations to 1 iteration per instruction



# Arithmetic Error

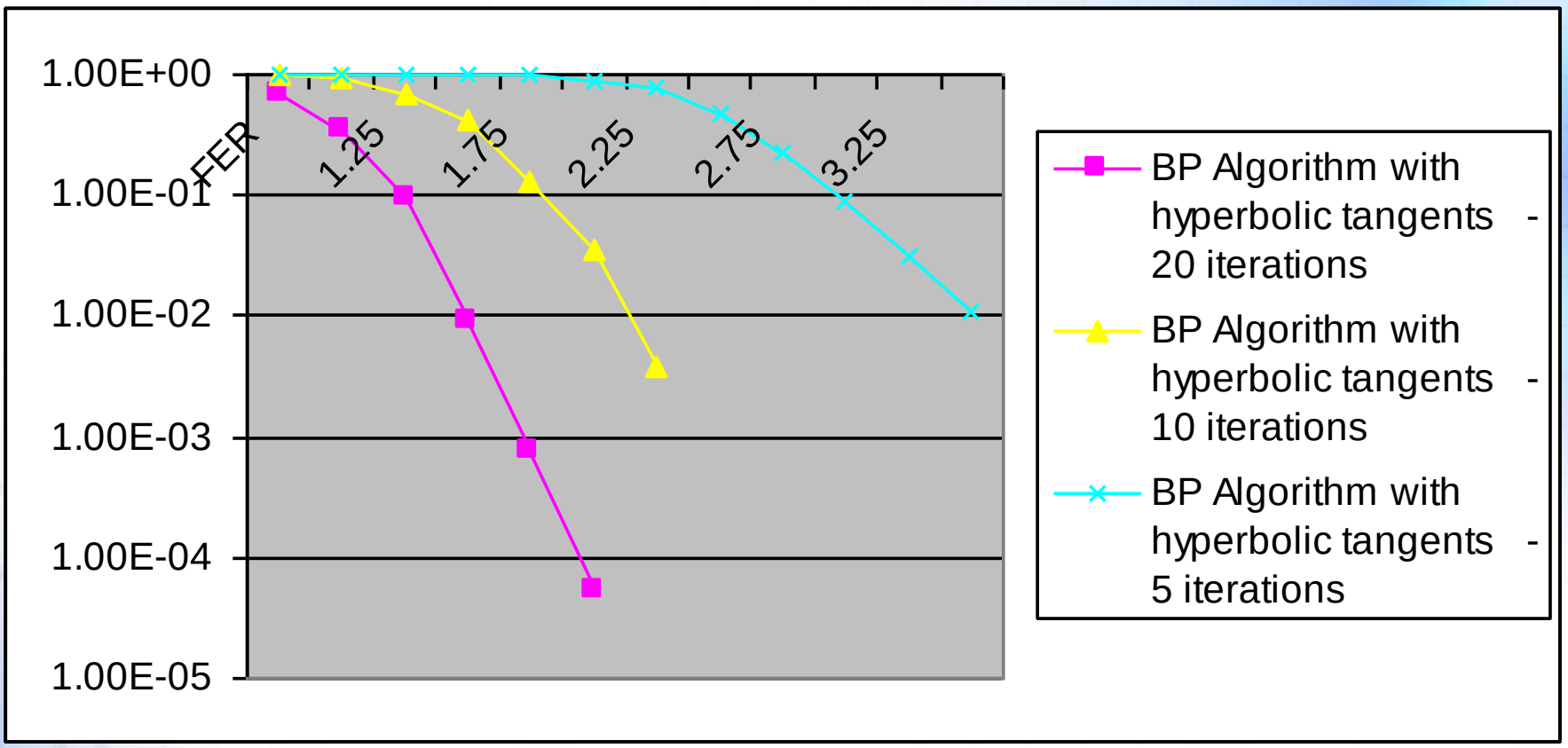
| Wireless Algorithms | Non-CORDIC Baseline Implementation | CORDIC Implementations |
|---|---|---|
| LDPC Decoder | 2.08% | 0.05% |

# LDPC- BP versus Min-Sum




- Block Size = 2304 bits input, 1152 bits output
- # of iterations = 20

# LDPC- BP Iteration Count




- Block Size = 2304 bits input, 1152 bits output