# SIPHER: Scalable Implementation of Primitives for Homomorphic EncRyption

## FPGA implementation using Simulink

HPEC 2011
Sept 21, 2011

**Dave Cousins,**
**Kurt Rohloff, Rick Schantz: BBN**
{dcousins, krohloff, schantz,}@bbn.com
**Chris Peikert: Georgia Tech**
cpeikert@cc.gatech.edu

Sponsored by Air Force Research Laboratory (AFRL) Contract No. FA8750-11-C-0098. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

Raytheon BBN Technologies

Georgia Institute of Technology

# Outline

- Motivation for Fully Homomorphic Encryption
- Our encryption scheme and primitives
- Why use an FPGA, and why Simulink?
- Examples
- Where do we go next?

# What is Fully Homomorphic Encryption?

- DoD is excited about Cloud Computing, but needs *guaranteed* security!
  - Requirement: Send high-value data to an un-trusted second party for ***processing, not just storage***.
  - *No decryption of data allowed at any point!*
- Example:
  - Need to deconflict airspace for multiple missions, but cannot share the mission tracks due to security requirements (different countries involved).
- Theoretical breakthrough by Craig Gentry (IBM) 2009*
  - Presented the first Fully Homomorphic Encryption (FHE) scheme that allows "computation" on encrypted variables without intermediate decryption.
  - Computation means multiple individual operations (and/or) on encrypted data → we are very far from full "programs"
  - Led to a rapid development of improved techniques.

*C. Gentry. Fully Homomorphic Encryption Using Ideal Lattices. STOC, 2009.

# FHE comes with great cost!

- FHE schemes allow “unlimited” computation on encrypted data.
- Based on “computationally hard” stochastic lattice theory problems.
  - Stochastics introduce noise term that grows with the number of operations applied to the encrypted data.
    - Too much noise → can no longer decrypt the data.
    - Requires a recrypt ( “decryption/encryption”) operation built out of FHE operations to “clean” the intermediate results of noise.
  - Leads to a HUGE computational burden
    - Current benchmarks on workstation PC → 30 sec / gate.

# Our goal: Build an FHE co-processor

- Reduce the time necessary for FHE computation by
  - Using more efficient algorithms
  - Hardware implementation of FHE primitives using FPGA
- Our approach:
  - Define a “Somewhat” HE scheme
  - Specify a maximum number of operations on the data before noise growth makes decryption impossible.
  - No need for recrypt → Much less burdensome
  - Uses all the same primitives as FHE
- Do this in a flexible and scalable way so we can keep up with the theory

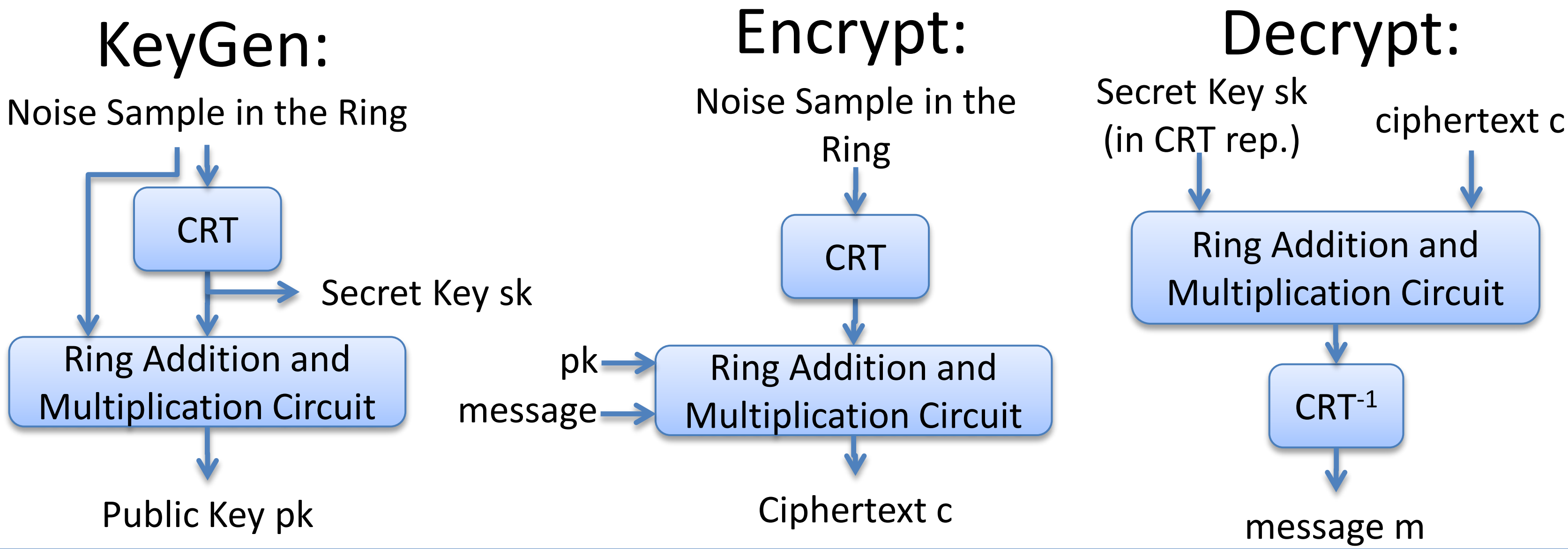
# Our chosen SHE scheme

- Based on “Learning with Errors over Rings”*
  - also stochastic lattice based
- Uses Chinese Remainder Transform (CRT) to simplify structure of “add” and “multiply” operations
  - Analogous to Fourier Transform pair: convolution ⇔ multiplication
- Formulation can encode and manipulate arrays of small integers or bits
  - Remainder of talk focuses on arrays of bits
  - Addition in the ring →XOR
  - Multiplication in the ring →AND
  - Can support unlimited # XOR, limited # AND operations

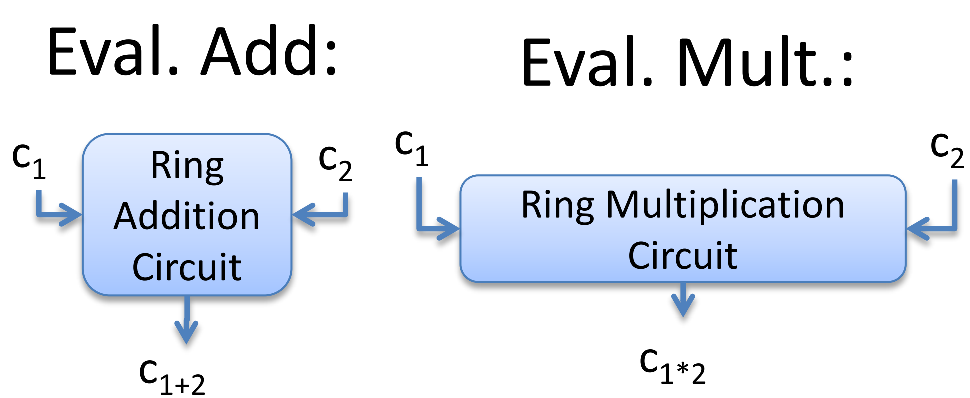
*V. Lyubashevsky, C. Peikert, and O. Regev. "On ideal lattices and learning with errors over rings". In Henri Gilbert, editor, Advances in Cryptology - EUROCRYPT 2010, V 6110 of Lecture Notes in Computer Science, Ch 1, PP 1-23. Springer Berlin / Heidelberg, Berlin.

# Baseline SHE Primitives

**Primitives that are done once**




**Primitives that are done multiple times**



# Why FPGA? Efficiency in computation!

- All primitives are computed on arrays modulo a large prime q
  - One selects the degree parameter (number of EvalMul operations supported) and a security parameter $\delta$ (encryption "hardness")
  - Generates parameters n (size of the plaintext vectors) and q.
- One bit encrypted into 2*width.q bits.
- Large bit-width modulo arithmetic is inefficient on a CPU

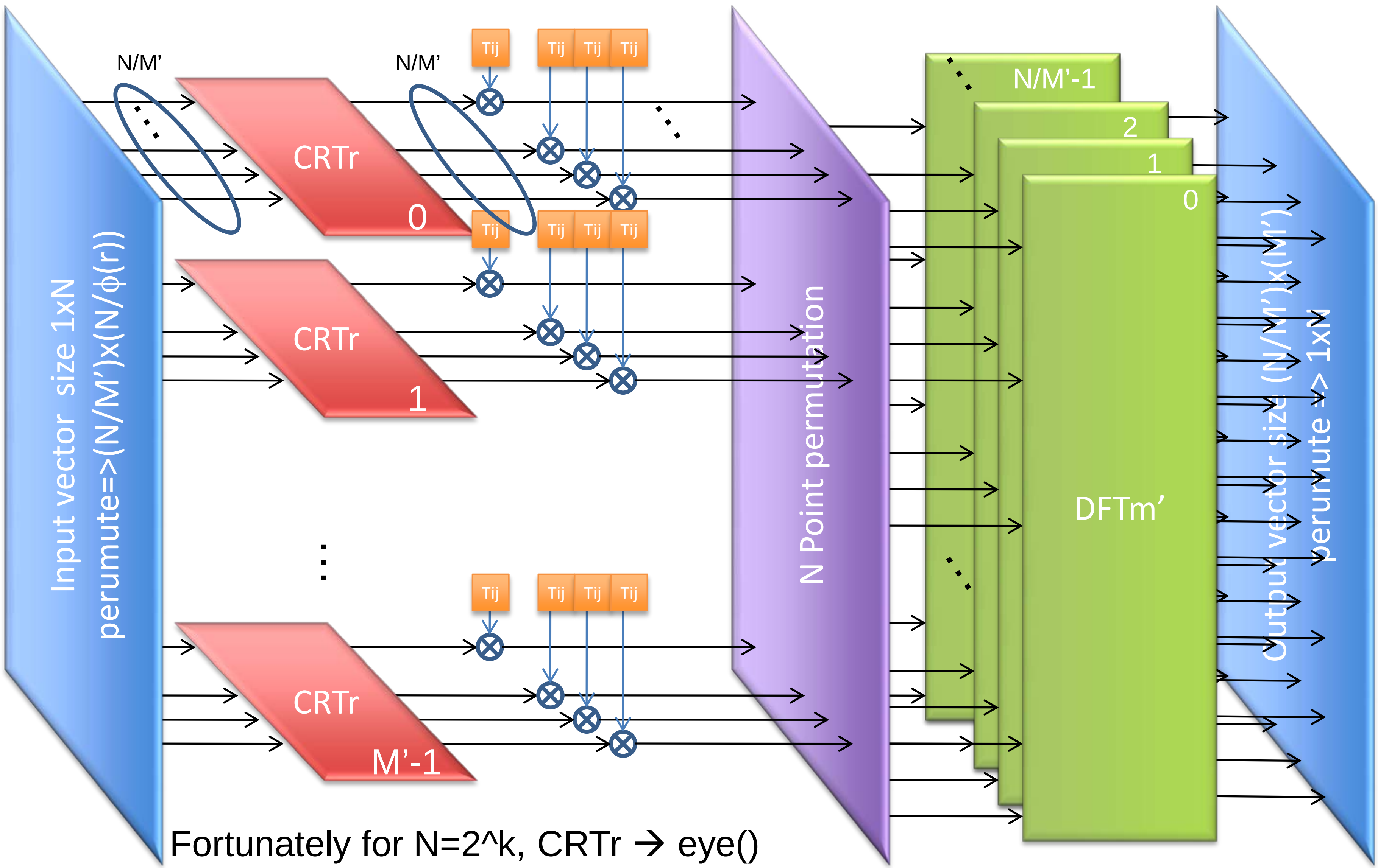
| Degree | Fair Encryption (SSN) | | Good Encryption (Credit Card) | |
|---|---|---|---|---|
| | n | q # bits | n | q # bits |
| 4 | 128 | 31 | 1024 | 37 |
| 5 | 256 | 41 | 1024 | 46 |
| 6 | 256 | 50 | 2048 | 59 |
| 7 | 256 | 58 | 2048 | 68 |
| 8 | 256 | 66 | 2048 | 78 |
| 9 | 512 | 79 | 2048 | 88 |

# Why Matlab/Simulink?

- Matlab fixed point library allows us to quickly code our SHE benchmarks in a high level language.
  - LWE uses "signal processing" constructs:
    - CRT for $n=2^k$ is based on FFT(x)%q
    - Noise selection uses discretized Gaussian random number pool
  - Theory is constantly being improved upon!
    - Our code life is about 3-4 months before new innovations require rewrite
- Simulink HDL coder lets us code our primitives directly in a data flow diagram, in parameterized form.
  - One diagram for all combinations of n and q
- Co-verification of the primitives in Matlab, Simulink and VHDL is easy!

# Example of Similarity with Signal Processing: CRT

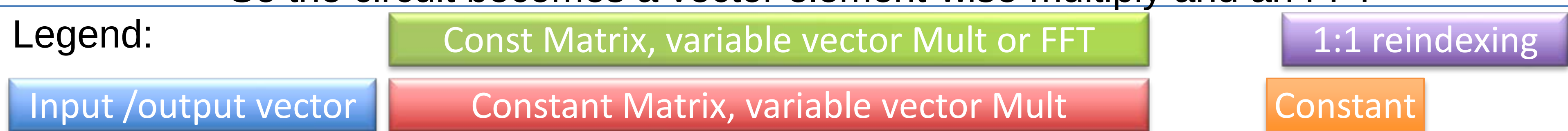




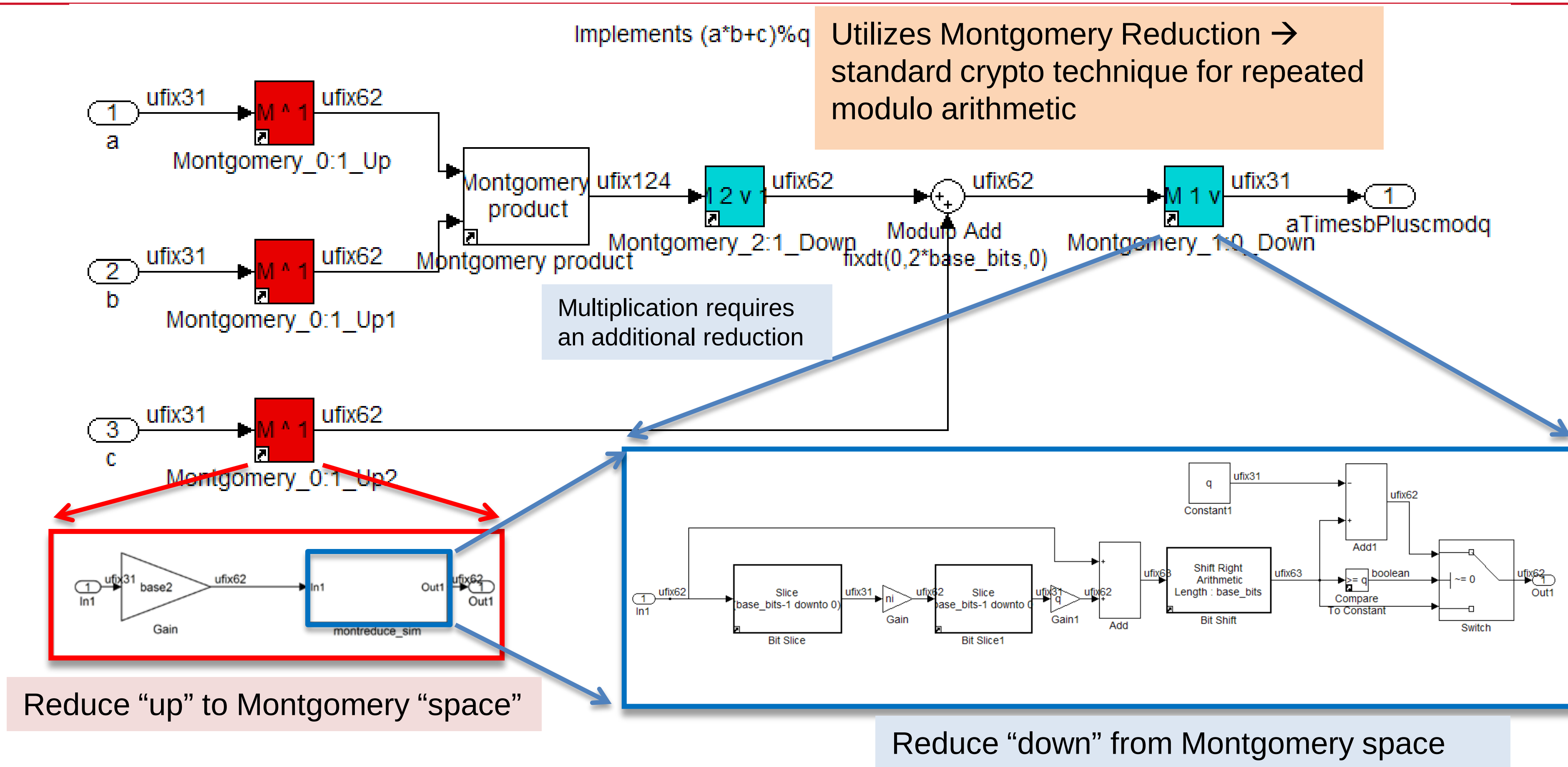

Fortunately for N=2^k, CRTr → eye()

So the circuit becomes a vector element-wise multiply and an FFT

Legend:



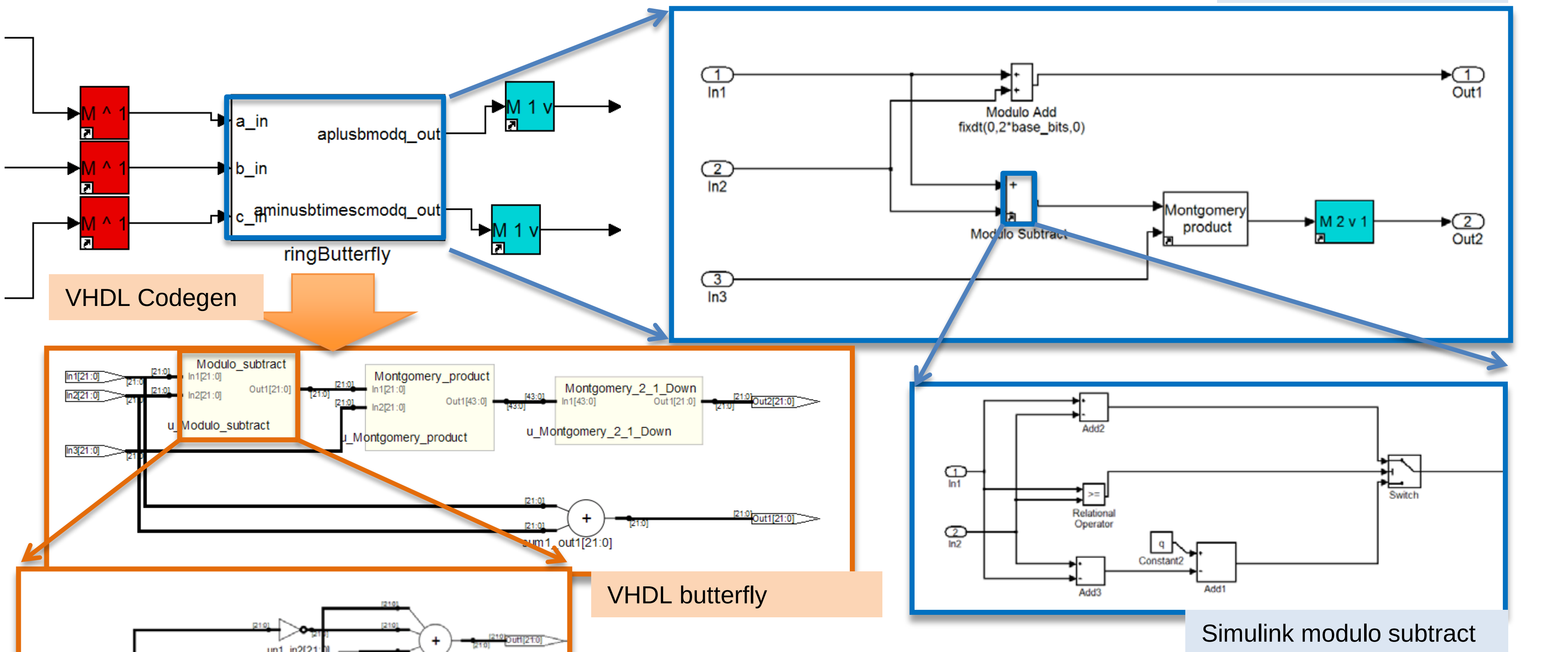
# Example: Simulink RingMultiplyAdd (a.*b+c)%q

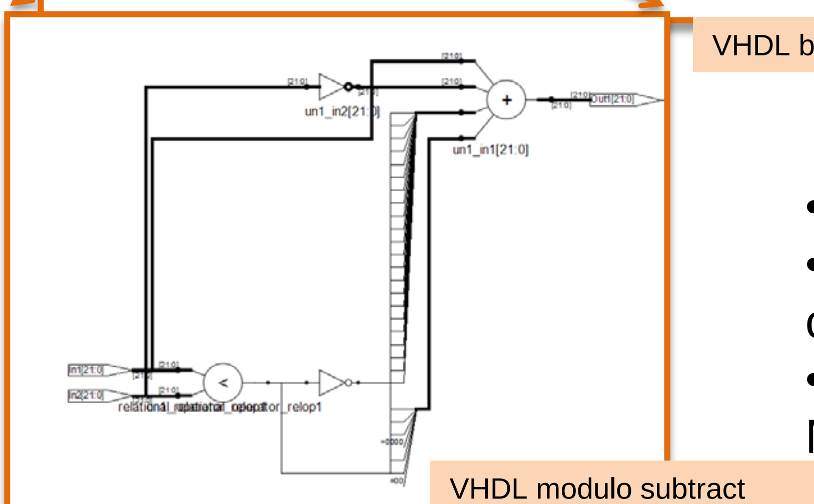



- Montgomery reduction doubles required bit-width…
  …but now Product and Add are Simulink primitives with rounding set to "wrap" → no trial division needed for modulo
- Model is serialized, iterated for n values.

Raytheon BBN Technologies · Georgia Institute of Technology

# Example: RingButterfly VHDL generation



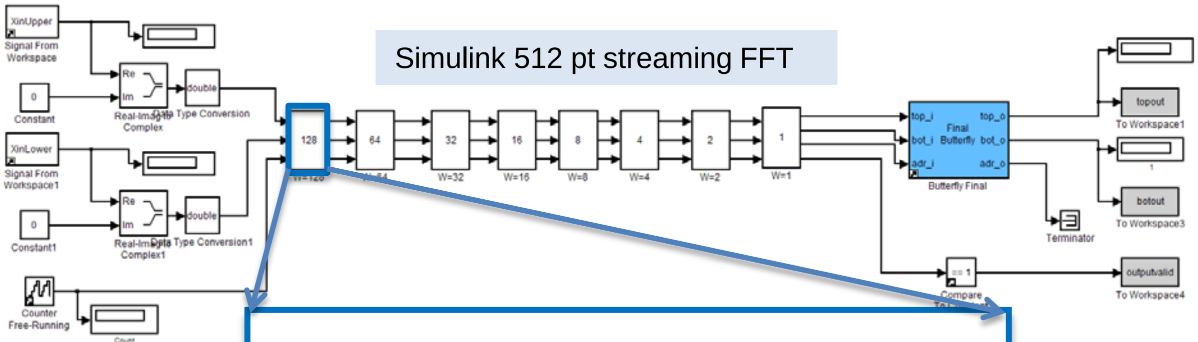




- Butterfly is *the* core component of CRT (FFT)
- Simulink tools generated VHDL code that compiled in ACTEL tool chain unedited!
- Simulink generated test benches that ran in Modelsim unedited !

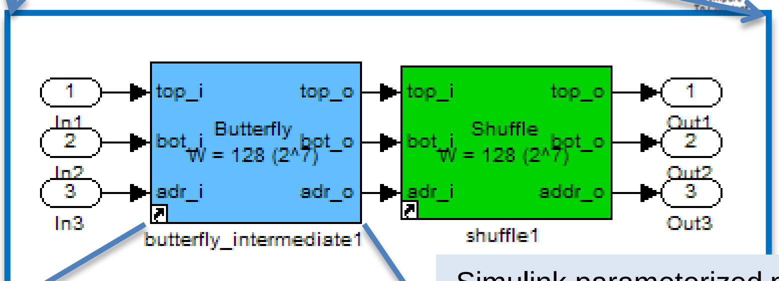
Raytheon BBN Technologies | Georgia Institute of Technology

# Example: Pipelined FFT structure

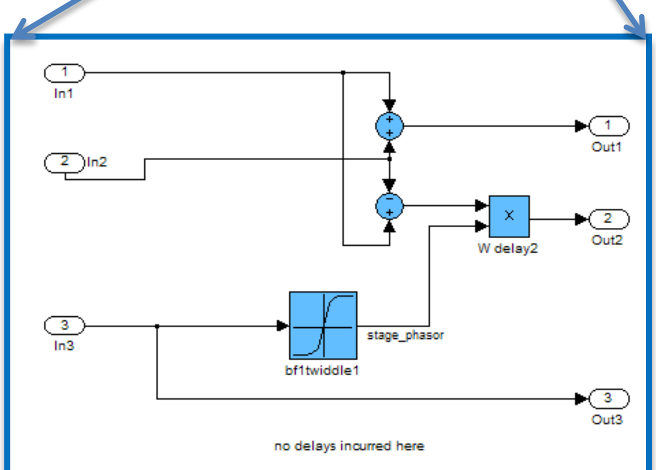
- Scalable radix 2 FFT is the basis for CRT
- Data streams in 2 points per clock cycle
- Can cascade stages for larger FFTs
- Each stage uses same library model with a different parameter



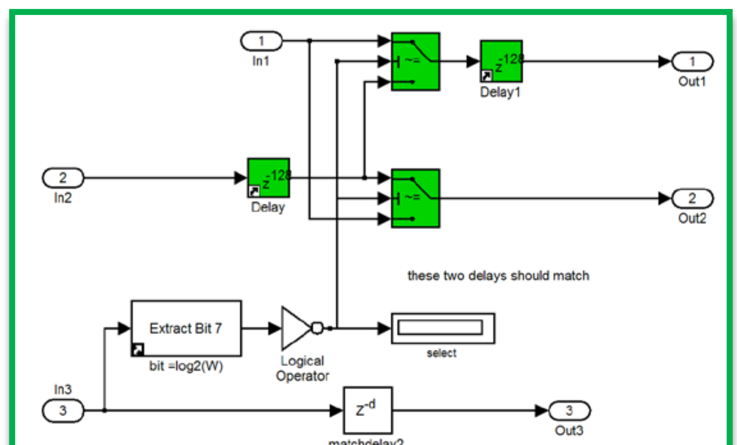

Simulink 512 pt streaming FFT




Simulink parameterized processing stage

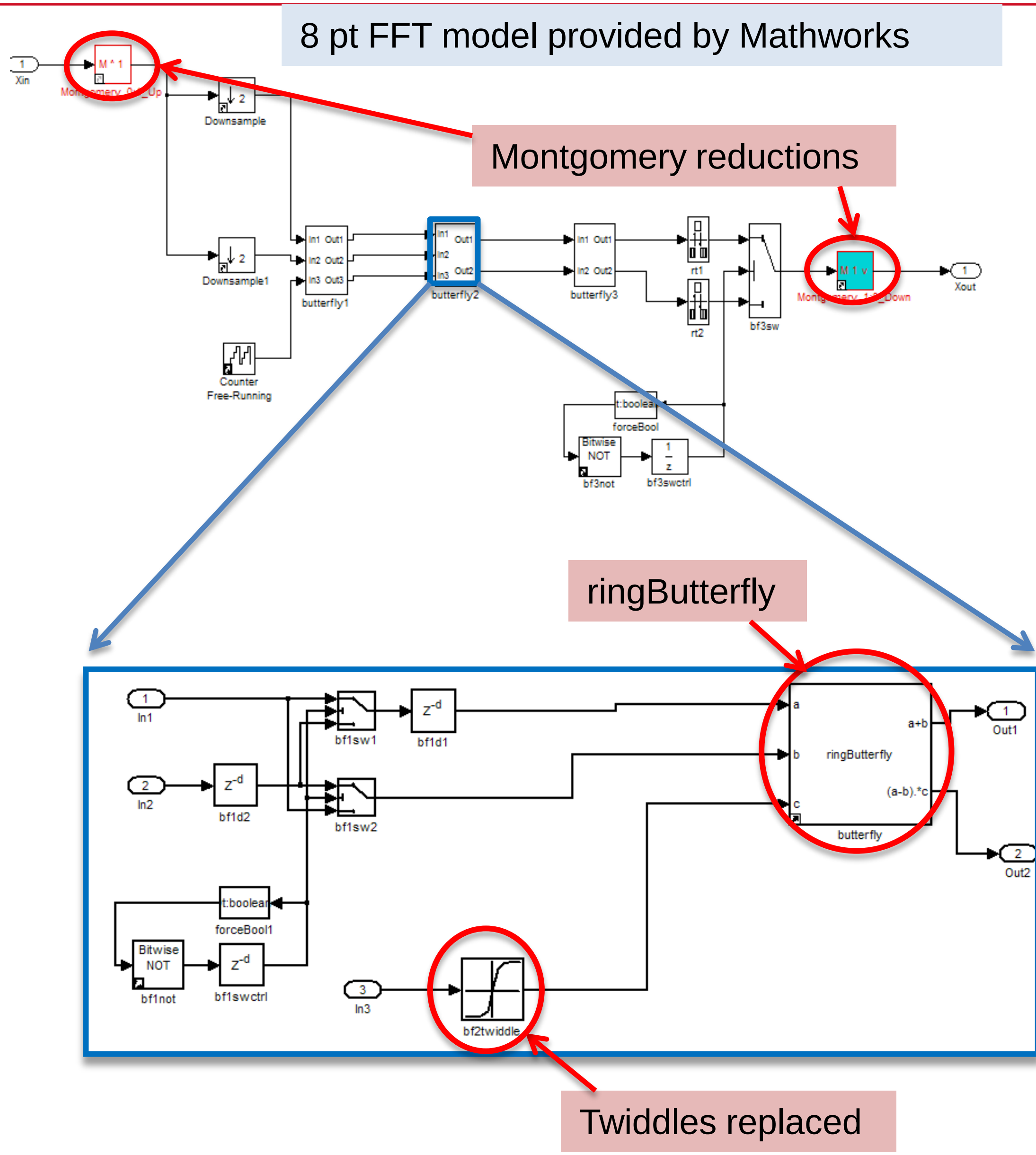



Simulink complex butterfly stage




Simulink shuffle stage

# Conversion from complex to ring arithmetic




- No major changes!!
  - Replace input variable with fixed point integers mod q
  - Twiddle factors replaced with modulo versions and stored in Montgomery form
  - Add Montgomery blocks to input and output
  - Complex butterfly block replaced with ringButterfly block
- Equivalent to mod(fft(x),q)

# Where do we go next?

- Go faster
  - Current primitives have no pipelining, so they are slow (e.g. butterfly has 88 ns cycle time on ACTEL A3PE)
    - typical of first cut code → get it to work right, then get it to work fast
  - Can use Simulink tools for adding pipelining
- Go bigger
  - Simulink is limited to 128 bit word widths.
    - Impacts multipliers in Montgomery Stages when q larger than 32 bits
    - Investigating Montgomery Multiplier (pipelined modulo multiply) and breaking multiplies into smaller chunks
  - Build full CRT and $CRT^{-1}$ models and register architecture
- Build in more functionality
  - Next year's goal is to build complete SHE processing unit on Xlinx FPGA
    - Registers, and simple ALU-like instruction set
  - Move towards FHE
    - Algorithms are constantly improving, reducing required computational load

# Special Thanks to the Mathworks Team

- Jeff Miller, Brian Ogilvie, Jared Schuler
- Mathworks second pilot program with BBN
  - First pilot program focused on Matlab to Blackfin code generation
- Taught us how to become “power users”
  - Provided sample code for Montgomery Methods using fixed point toolbox
  - Provided sample models for HDL friendly serialized Ring Ops and basic streaming FFT
  - (Bi)Weekly webconferences really bootstrapped our program