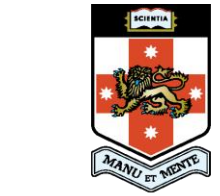
Multiplier Circuits

Conventional and Approximate Designs



UNSW
SYDNEY

Dr. Hassaan Saadat
Dr. Darshana Jayasinghe
Prof. Sri Parameswaran

Lecture Outline

- Quick Recall
  - Number-representation in computers
  - Arithmetic circuits in computers
    - With the example of Half-adder, Full-adder and Ripple-carry adder
- Conventional Multiplier Circuits
- Approximate Arithmetic Units
  - What are they?
  - Why are they needed?
- Approximate log-based multipliers and its variants

Quick Recall:
Number Representation in Computers

- Modern computer are **digital circuits** where each nodes can have ony **two states**
  - High/Low, or ON/OFF

- The mathematical binary number system is naturally suited for computers

Numbers are represented in binary format

Combination of ones and zeros – called bits

Quick Recall: Number Representation in Computers

Numbers are represented in binary format

Combination of ones and zeros – called bits

- Positional number system
  - Position of a bit determines its weight in the value of the number
  - Value of an unsigned N-bit number $d_{N-1}d_{N-2}\ldots d_1d_0$

$$value(A_{us}) = \sum_{i=0}^{N-1} 2^i d_i$$

- Other representation exists: including for signed, fractional numbers
  - Beyond the scope of this lecture
  - Further resources for interested students
    - https://en.wikipedia.org/wiki/Computer_number_format
    - https://computing-concepts.cs.uri.edu/wiki/Data_Representation_For_Computing
    - https://en.wikipedia.org/wiki/Signed_number_representations
    - https://www.stat.berkeley.edu/~nolan/stat133/Spr04/chapters/representations.pdf

Quick Recall: Arithmetic in Computers

- Digital logic circuits
  - Mimic the operations of mathematical binary number system

Quick Recall: Arithmetic in Computers

- Digital logic circuits
  - Mimic the operations of mathematical binary number system
- Consider binary addition of two bits
  - 4 possible cases

```
  0     0     1     1
+ 0   + 1   + 0   + 1
---   ---   ---   ---
```

Quick Recall: Arithmetic in Computers

- Digital logic circuits
  - Mimic the operations of mathematical binary number system
- Consider binary addition of two bits
  - 4 possible cases

```
  0     0     1     1
+ 0   + 1   + 0   + 1
-----------------------
  0 0   0 1   0 1   1 0
```

Quick Recall: Arithmetic in Computers

- Digital logic circuits
  - Mimic the operations of mathematical binary number system
- Consider binary addition of two bits
  - 4 possible cases

	0		0		1		1
+	0	+	1	+	0	+	1
	0 0		0 1		0 1		1 0
	c s		c s		c s		c s

Quick Recall: Arithmetic in Computers

- Digital logic circuits
  - Mimic the operations of mathematical binary number system
- Consider binary addition of two bits
  - 4 possible cases

```
  A: 0        A: 0        A: 1        A: 1
  B: 0        B: 1        B: 0        B: 1
+            +           +           +
-----       -----       -----       -----
  0 0         0 1         0 1         1 0
  c s         c s         c s         c s
```

Truth Table

A	B	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

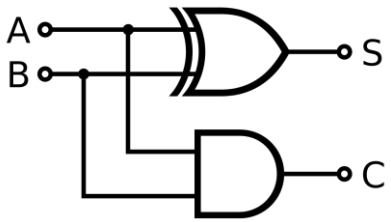
Quick Recall: Arithmetic in Computers

- Digital logic circuits
  - Mimic the operations of mathematical binary number system
- Consider binary addition of two bits
  - 4 possible cases

A: 0	A: 0	A: 1	A: 1
B: 0	B: 1	B: 0	B: 1
+	+	+	+
0 0	0 1	0 1	1 0
c s	c s	c s	c s

Truth Table

A	B	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Half adder

Quick Recall: Arithmetic in Computers

- Similarly, consider addition of 3 bits
  - 8 possible cases

Truth Table

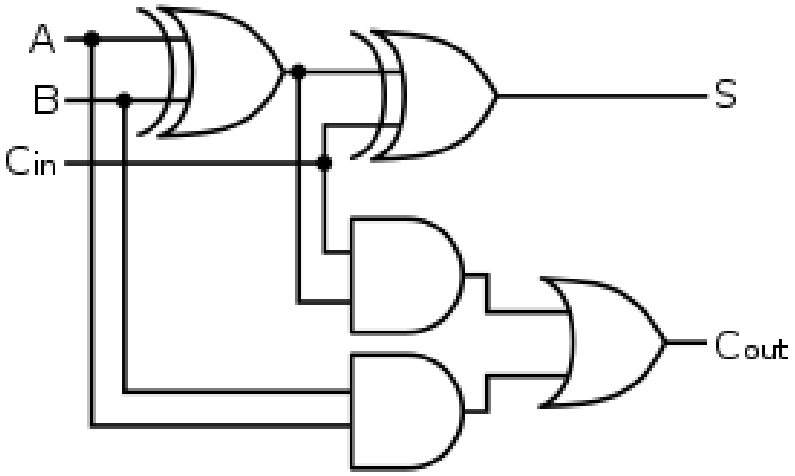
Cin	A	B	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Quick Recall: Arithmetic in Computers

- Similarly, consider addition of 3 bits
  - 8 possible cases

Truth Table

Cin	A	B	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Full-adder

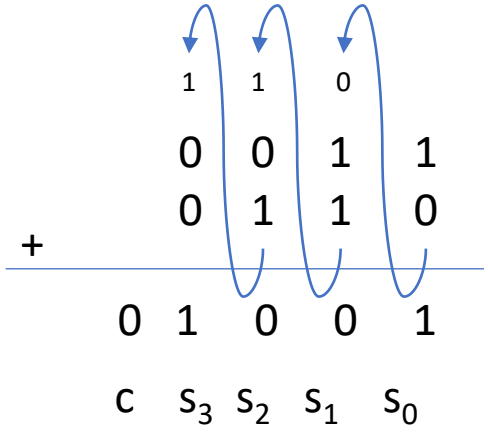
Quick Recall: Arithmetic in Computers

- What about addition of N-bit numbers?

$$
\begin{array}{cccccc}
 & 0 & 0 & 1 & 1 \\
 & 0 & 1 & 1 & 0 \\
+ & & & & \\
\hline
 & & & & \\
c & s_3 & s_2 & s_1 & s_0
\end{array}
$$

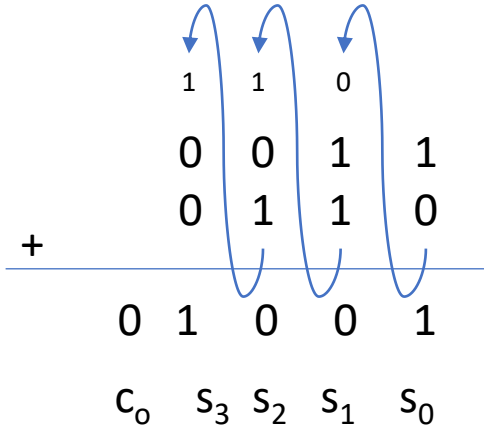
Quick Recall: Arithmetic in Computers

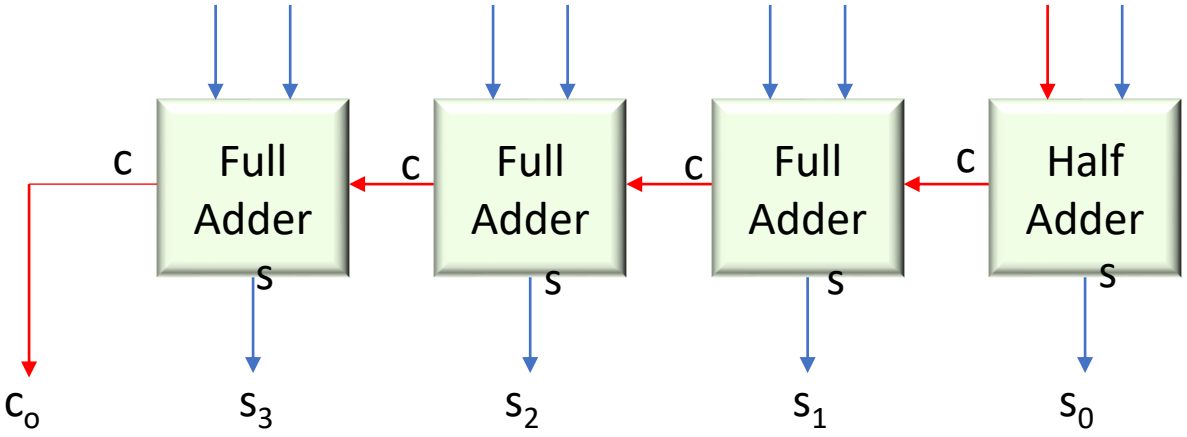
- What about addition of N-bit numbers?



Quick Recall: Arithmetic in Computers

- What about addition of N-bit numbers?





Quick Recall: Arithmetic circuits

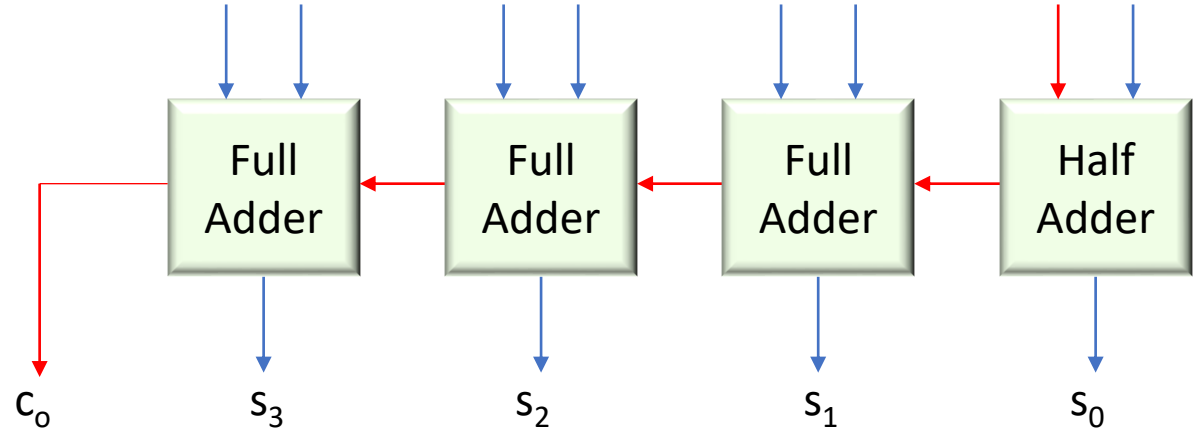
- Resource consumption and speed issues

- Logic Area:
  - Number of Gates/logic elements
  - In cases of FPGA, logic elements are CLB – LUTs
    - https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

- Speed:
  - For each logic element, its output appears after a certain delay
  - The speed of the circuit depends on the longest path of logic elements : called as the critical path

Quick Recall: Arithmetic in Computers

- Lets analyse the 4-bit RCA circuit



- Logic area: (N-1) FA + 1 HA
- Delay: delay of (N-1) FA + delay of HA

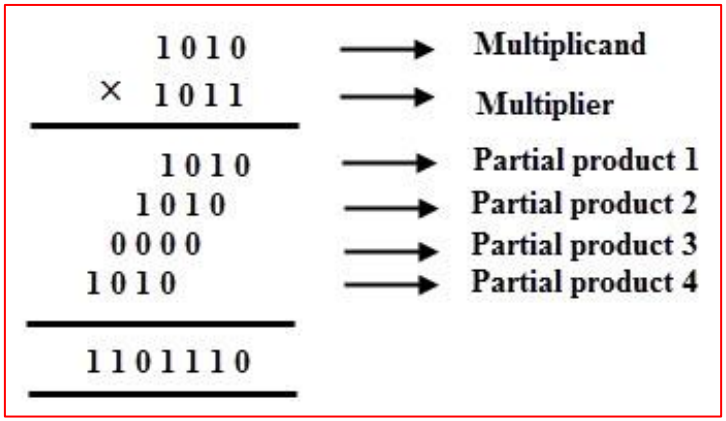
Variations in the design

- Carry look-ahead adder
  - Improved critical path, more area
  - https://en.wikipedia.org/wiki/Carry-lookahead_adder
- Bit-serial adder
  - N cycles
    - RCA and CLA are called parallel adders
  - Smaller area and critical path
  - https://en.wikipedia.org/wiki/Serial_binary_adder

Multipliers

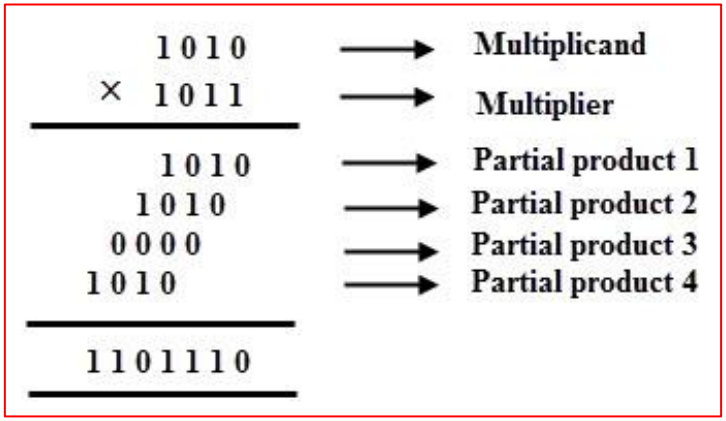
Binary Multiplication

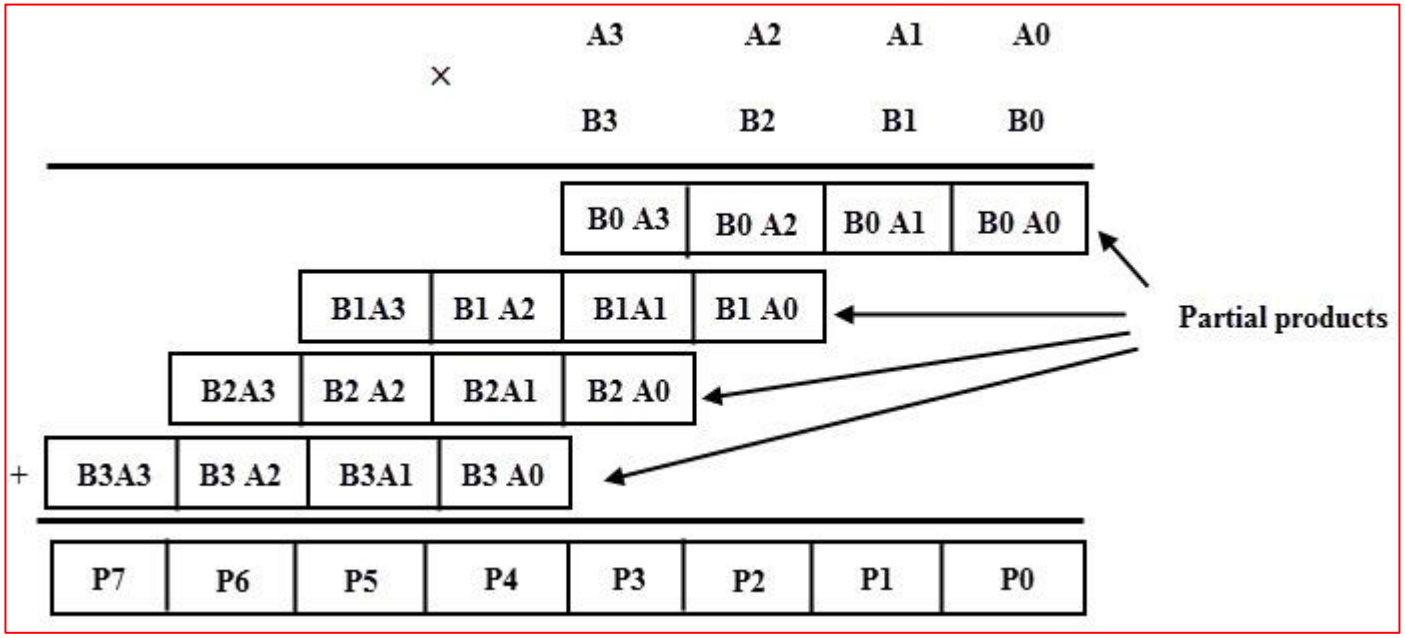
- Consider binary multiplication from high-school



Binary Multiplication

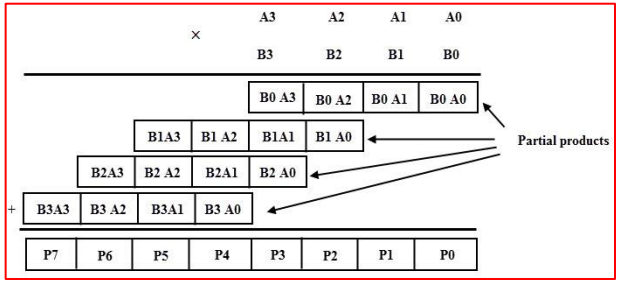
- Two steps:
  - Generate partial product
  - Add/Accumulate them

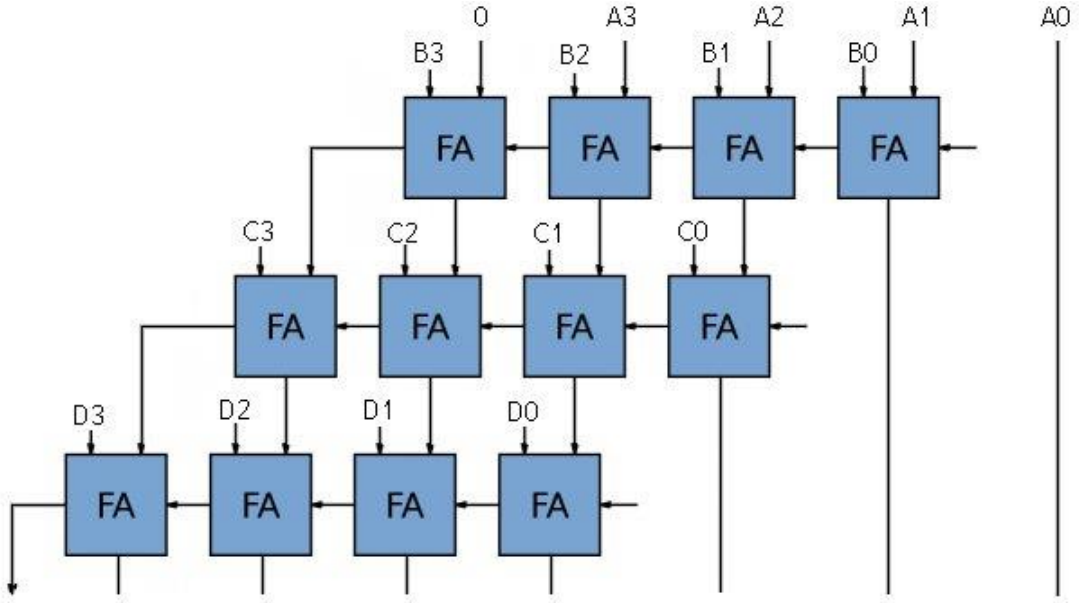




Binary multiplier Circuit

- Array multiplier
  - O(N^2) area and delay complexity





Variations in the designs

- Wallace tree reduction
  - https://en.wikipedia.org/wiki/Wallace_tree

- Dadda tree reduction
  - https://en.wikipedia.org/wiki/Dadda_multiplier

Good News!!

- Modern ASIC/FPGA synthesis tools are smart
  - recognize the + - * operators in Verilog/VHDL
  - build the circuit behind etc using available logic elements
- Division and Mod are not always supported
  - You may need IP-core
    - https://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core

Approximate Arithmetic

Use Ripple carry adder for understanding

Then move to approximate multipliers

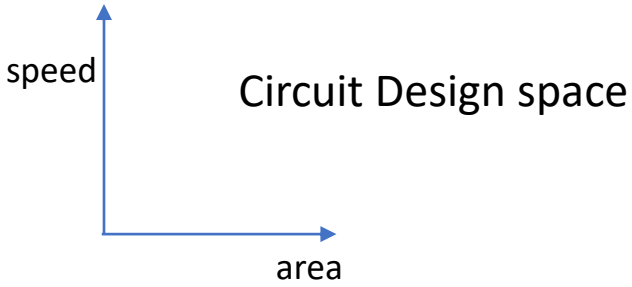
Motivation for Approximate Arithmetic

- Multiplication/addition dominant operations in many applications
  - For example, Deep neural networks
- Their resource-consumption have direct impact on overall resource-consumption of the system
- How can we improve it?

Motivation for Approximate Arithmetic

- RCA/CLA Adder?
- Array multipliers / Wallace / Dadda?

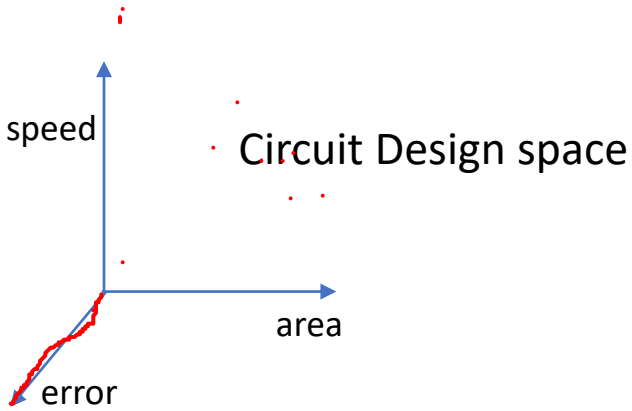
- Essentially all aim to be exact
  - correct for all possible binary input – with no error



What if we can allow certain errors?

What if we can allow certain errors?

- Another dimension in the design space



- New possibility of design points
- Gives us further margins for efficiency improvements

Error resilient applications

- You might think errors are not good?

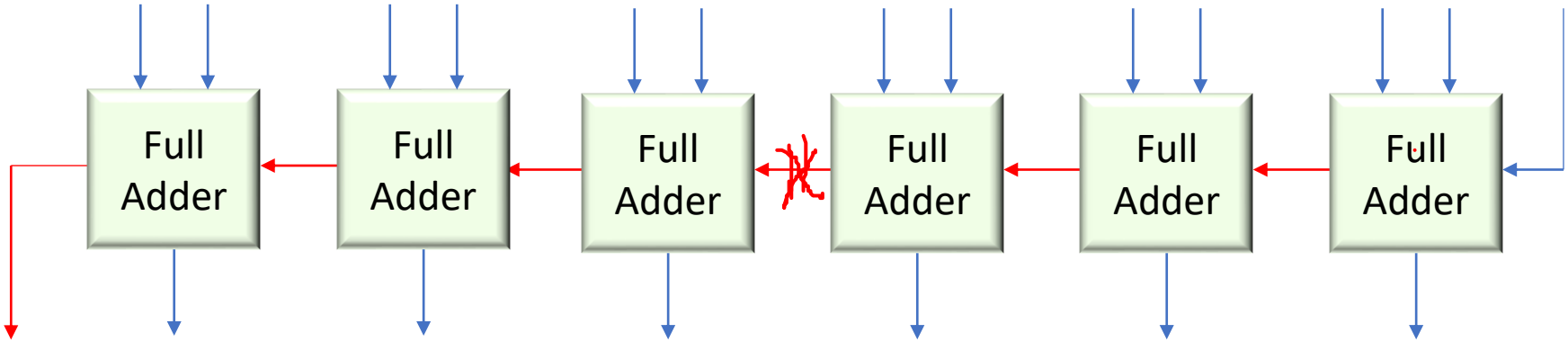
Error resilient applications

- You might think errors are not good?
- Some algorithms are inherently error resilient
- Reasons
  - Algorithm are designed to treat noisy inputs from sensors
  - Treat errors as noise and suppress them
- Minimal impact on output results

Approximate Adder

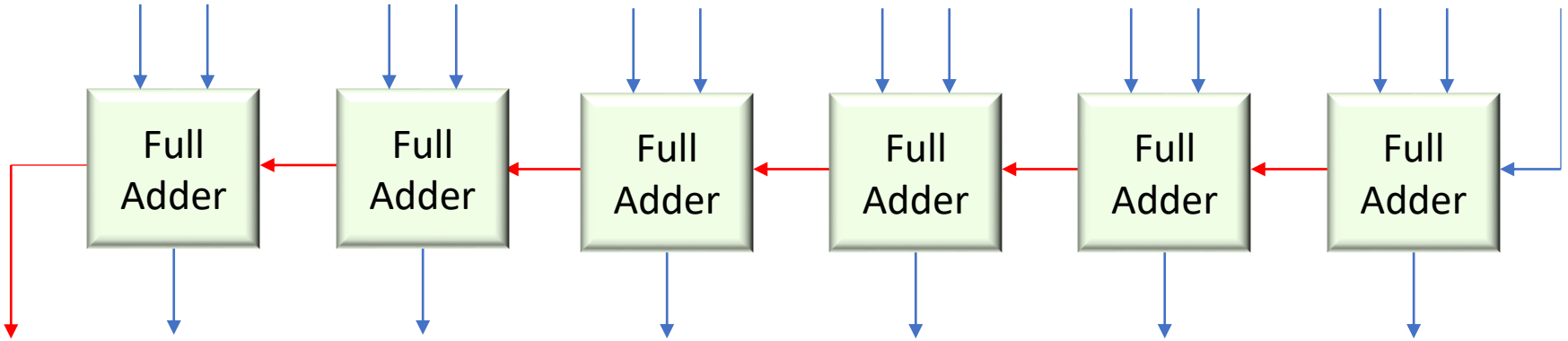
- Lets understand this concept with a basic RCA adder

Approximate Adder

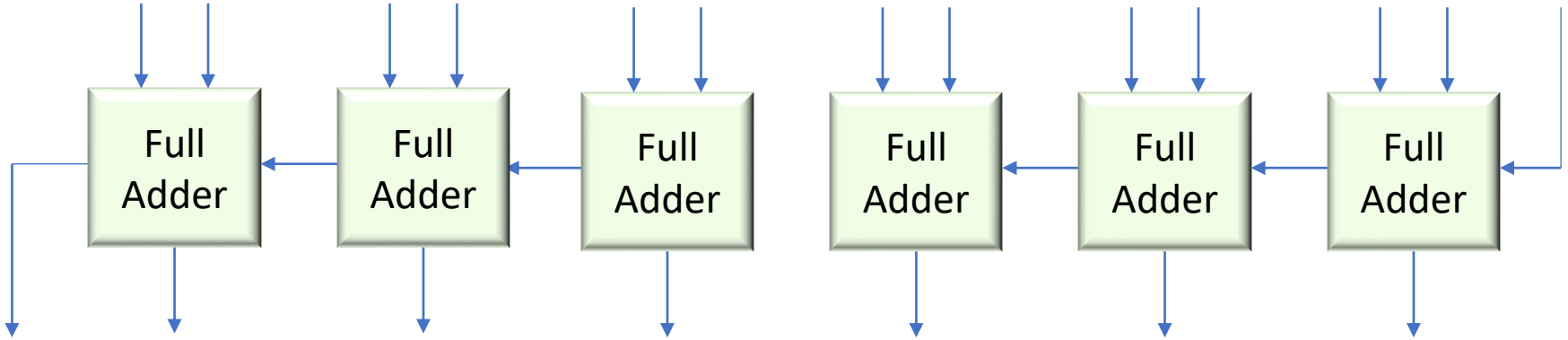


- Delay: 6 Full-adders

Approximate Adder



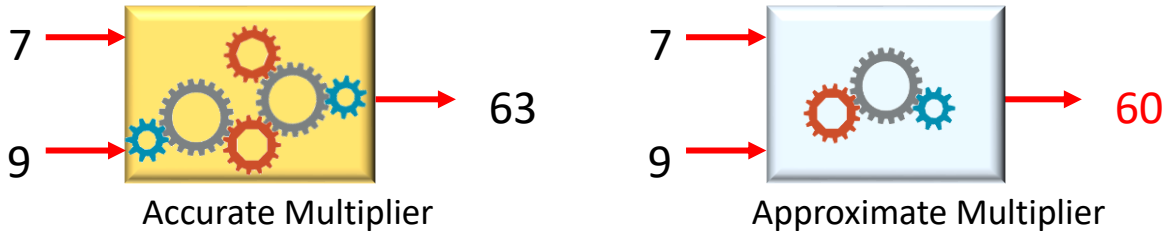
- Delay: 6 Full-adders



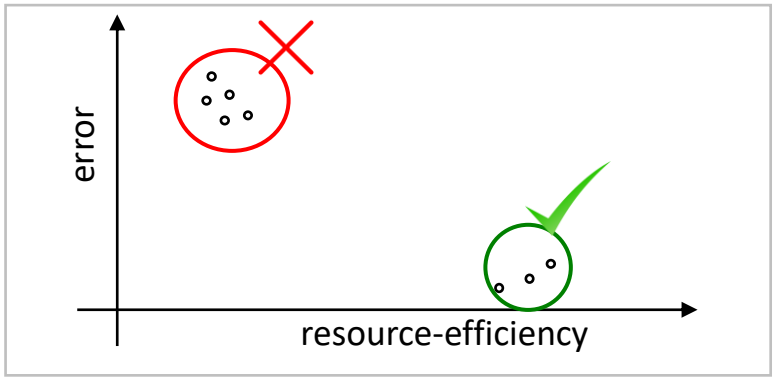
- Errorneous output for some input combinations (Max Error = 2^3)
- Delay: 3 Full-adders → Twice as fast.. always !!

Approximate Arithmetic Units

- Resource-efficient approximate arithmetic units
  - Hardware logic design is modified
    - Simple and efficient design ↔ Erroneous output



- Primary aim when designing approximate arithmetic units
  - Small error for more resource-efficiency
  - A designs that can offer us the best trade-offs



Approximate Log-based Multipliers

Approximate Log based Multipliers

- Multiplier circuits are more resource-hungry than adders

- Lets recall the log property
  - Log(AxB) = log (A) + log (B)

- So we can use

$$AxB = \text{antilog}(\log(A) + \log(B))$$

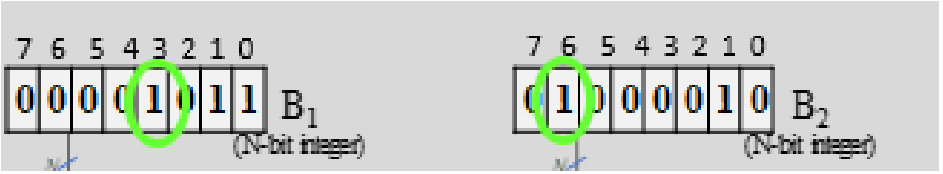
  - Reduces to addition ☺
  - But, need to compute log and antilog ☹
  - Solution: compute log-antilog approximately !!
    - Results in approximate log based multiplier

Approximate Log based multiplier

- Basic scheme

Mitchell's Multiplier (MA)

1. Approximate log of inputs
   - Integer part: Position of leading-one (k_1, k_2)
   - Fractional part: Rest of the bits (x_1, x_2)

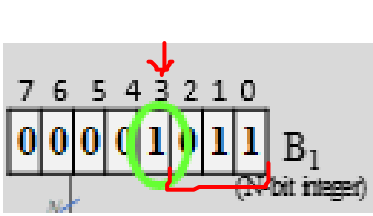


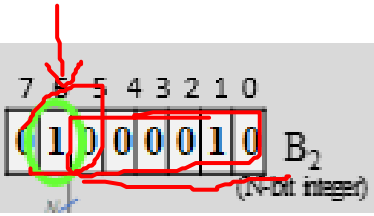
Approximate Log based multiplier

- Basic scheme

Mitchell's Multiplier (MA)

1. Approximate log of inputs
   - Integer part: Position of leading-one (k_1, k_2)
   - Fractional part: Rest of the bits (x_1, x_2)





$k_1 = 11$

$x_1 = 0.011$

$k_2 = 110$

$x_2 = 0.000010$

$\log(B_1) = k_1 + x_1$

$= 11.011$

$\log(B_2) = k_2 + x_2$

$= 110.000010$

Approximate Log based multiplier

- Basic scheme

Mitchell's Multiplier (MA)

1. Approximate log of inputs
   - Integer part: Position of leading-one (k_1, k_2)
   - Fractional part: Rest of the bits (x_1, x_2)
2. Add

$\log(B_1) = k_1 + x_1$
$= 11.011$

$\log(B_2) = k_2 + x_2$
$= 110.000010$

1001 .0110100

Approximate Log based multiplier

- Basic scheme

Mitchell's Multiplier (MA)

1. Approximate log of inputs
   - Integer part: Position of leading-one (k_1, k_2)
   - Fractional part: Rest of the bits (x_1, x_2)
2. Add
3. Approximate antilog of the sum
   - Append 1 to the left of fractional part
   - Scale w.r.t the integer part

$\log(B_1) = k_1 + x_1 = 11.011$

$\log(B_2) = k_2 + x_2 = 110.000010$

1001 .0110100

1.0110100

1011010000 = $(720)_{10}$
Accurate: 66x11 = $(726)_{10}$

Approximate Log based multiplier

- Basic scheme

Mitchell's Multiplier (MA)

1. Approximate log of inputs
   - Integer part: Position of leading-one (k_1, k_2)
   - Fractional part: Rest of the bits (x_1, x_2)
2. Add
3. Approximate antilog of the sum
   - Append 1 to the left of fractional part
   - Scale w.r.t the integer part

Mathematically

$$approx\_product = \begin{cases} 2^{k_1+k_2}(1 + x_1 + x_2), & x_1 + x_2 < 1 \\ 2^{k_1+k_2+1}(x_1 + x_2), & x_1 + x_2 \geq 1 \end{cases}$$

k_1, k_2: position of leading-one in B_1 and B_2

x_1 and x_2: fractional part of the log values

Approximate Log based multiplier

Based on the elementary log-multiplication property: $\log_b(AB) = \log_b A + \log_b B$

Mitchell's Multiplier (MA)

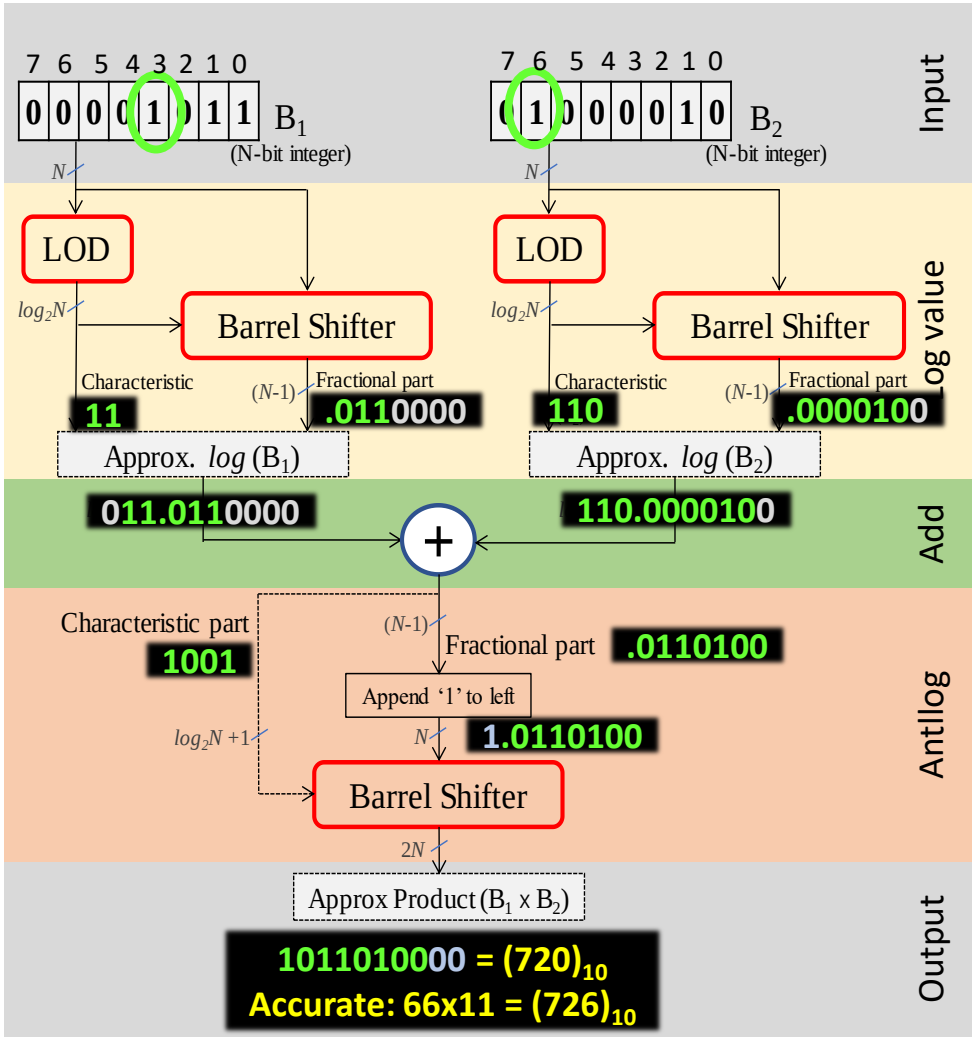
1. Approximate log of inputs
   - Integer part: Position of leading-one (k_1, k_2)
   - Fractional part: Rest of the bits (x_1, x_2)
2. Add
3. Approximate antilog of the sum
   - Append 1 to the left of fractional part
   - Scale w.r.t the integer part

Mathematically

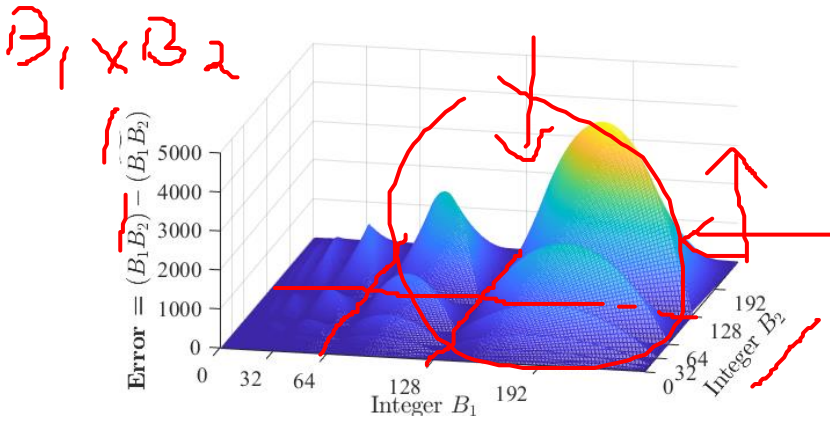
$$approx\_product = \begin{cases} 2^{k_1+k_2}(1 + x_1 + x_2), & x_1 + x_2 < 1 \\ 2^{k_1+k_2+1}(x_1 + x_2), & x_1 + x_2 \geq 1 \end{cases}$$

k_1, k_2: *position of leading-one in* B_1 *and* B_2

x_1 *and* x_2: *fractional part of the log values*



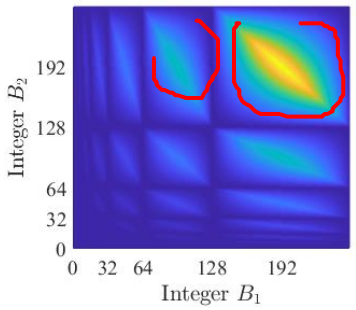
Problem in MA: High and Biased Error



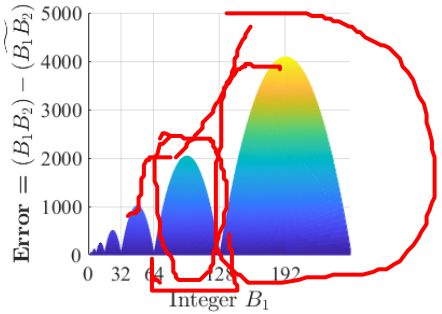
$$Error = \begin{cases} 2^{k_1+k_2}(x_1x_2), & x_1 + x_2 < 1 \\ 2^{k_1+k_2}(1 + x_1x_2 - x_1 - x_2), & x_1 + x_2 \geq 1 \end{cases}$$

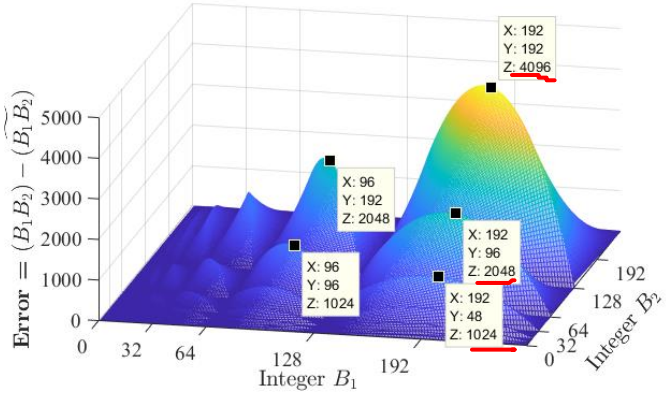
- Error always has the same sign
  - Can we do error-correction with minimal overhead?
- Observations
  1. Error profile is replicated in every power-of-two (pair of k_1 and k_2)
  2. Error profile is scaled by $2^{k_1+k_2}$

Top View



Side View



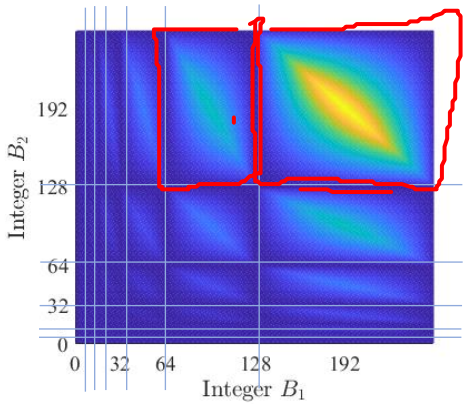


UNSW SYDNEY

Proposed Error Correction: Idea

- Compute mean of error within each interval (k_1 and k_2)

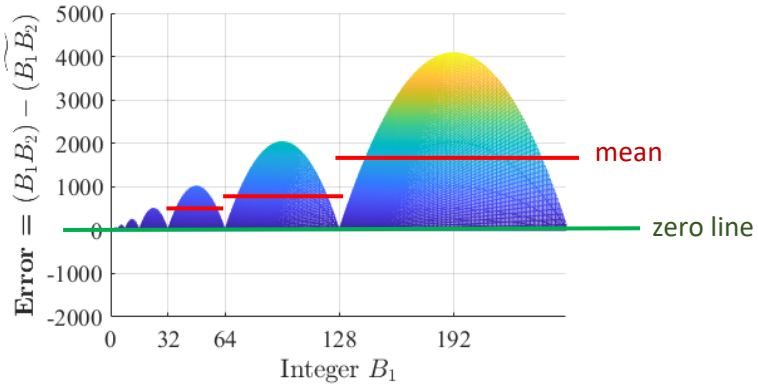
$$average_{error} = \frac{1}{(1-0)}\frac{1}{(1-0)}\int_0^1\int_0^1 Error.dx_2.dx_1 \quad = (0.08333) \times 2^{k_1+k_2}$$



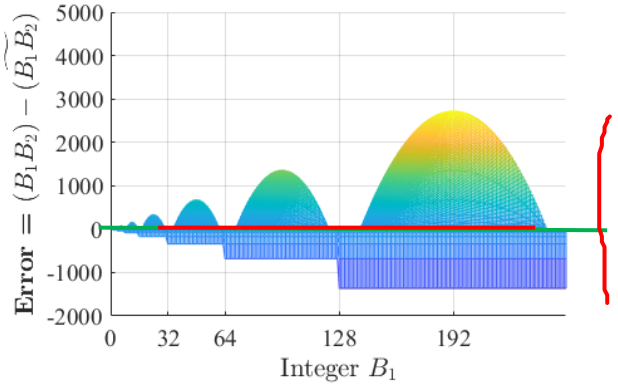
- Add to the approximate product

$$apx\_product = \begin{cases} 2^{k_1+k_2}(1+x_1+x_2), & x_1+x_2 < 1 \\ 2^{k_1+k_2+1}(x_1+x_2), & x_1+x_2 \geq 1 \end{cases} + (0.08333) \times 2^{k_1+k_2}$$

Before Correction

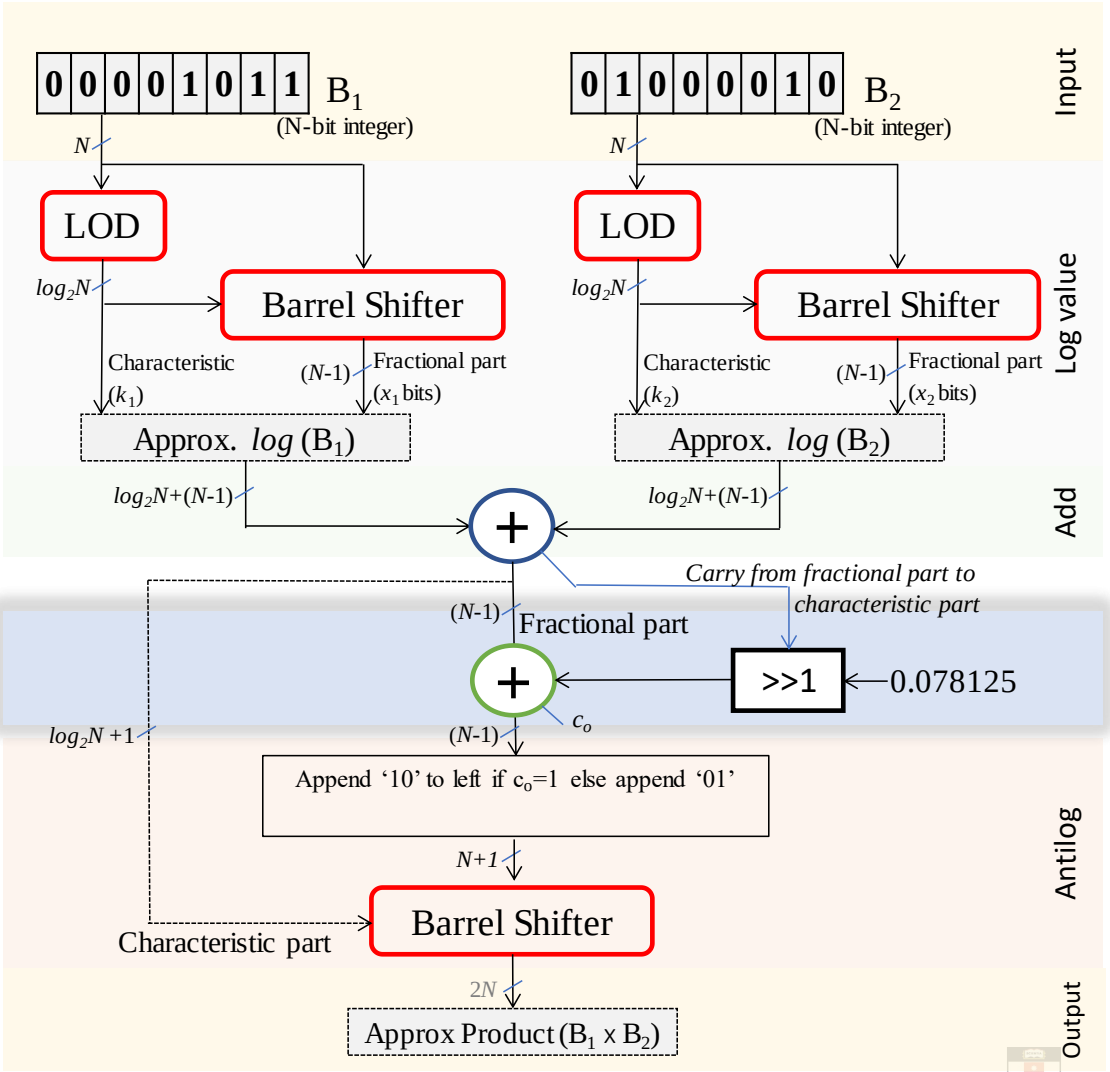


After Correction



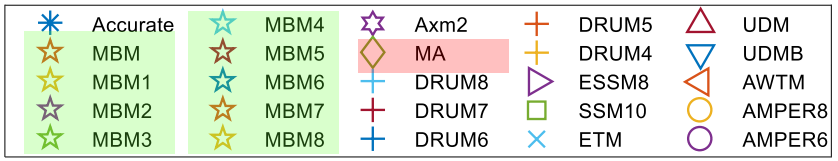
Improved Design - MBM

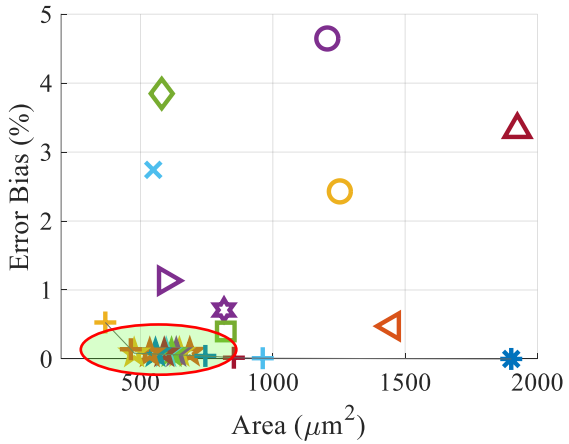
- Key features
  - Needs only
    - A 7-bit adder
    - A 2x1 mux
  - Single error correction term
  - Independent of the value of k
  - Independent of integer size N

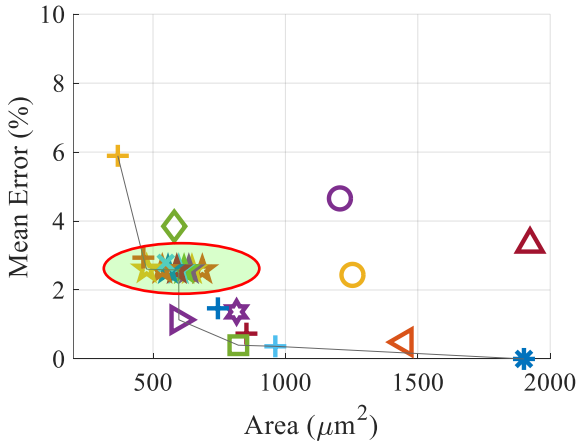


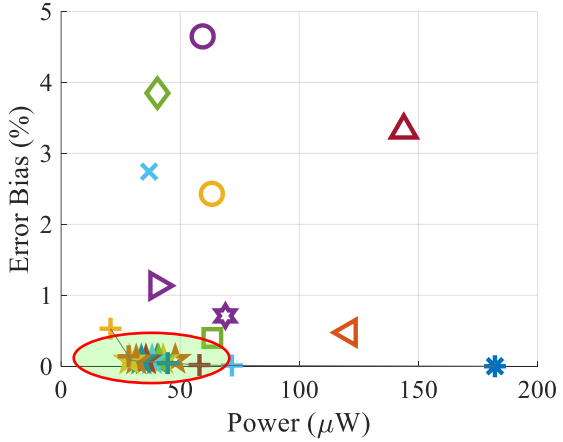
Results – 16-bit Multipliers

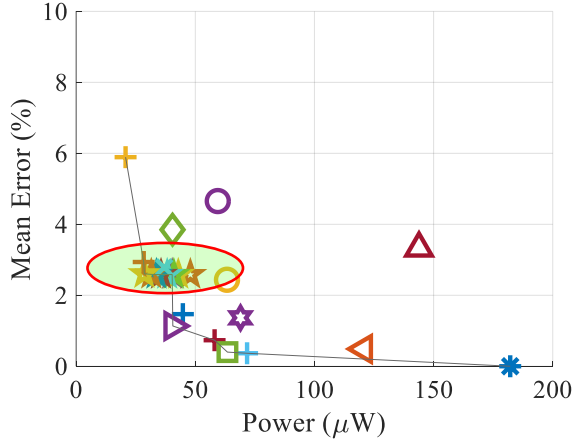
- Comparison with state-of-the-art approximate integer multipliers
- Grey line outlines optimal points
- MBM-t gives Pareto optimal points in all!!











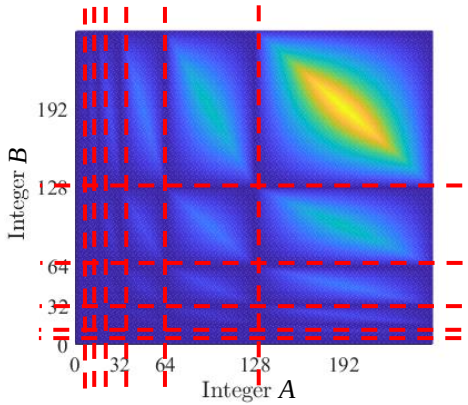
Experiment Details:

- RTL synthesis using TSMC 45nm at 1GHz
- Error analysis using Monte Carlo simulations
  - Error metrics – compared with accurate integer multiplier
    1) Error Bias → Mean of Relative Errors
    2) Mean Error→ Mean of Absolute Relative Errors

Approximate Multiplier: REALM

Quick Overview and Results

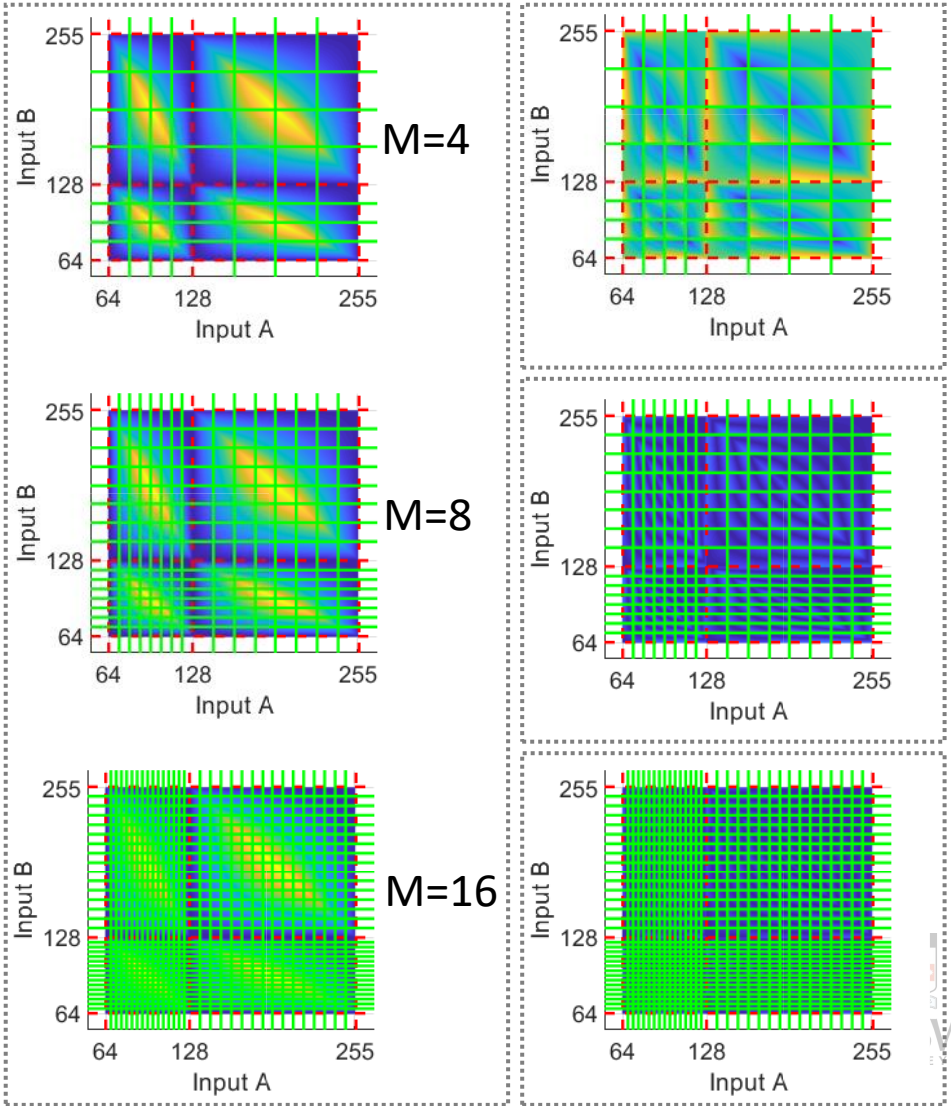
REALM Overview



- Number of partitions M affects the error as well as the hardware cost
- M is an error-configuration knob

Partition each power-of-two-interval into MxM segments

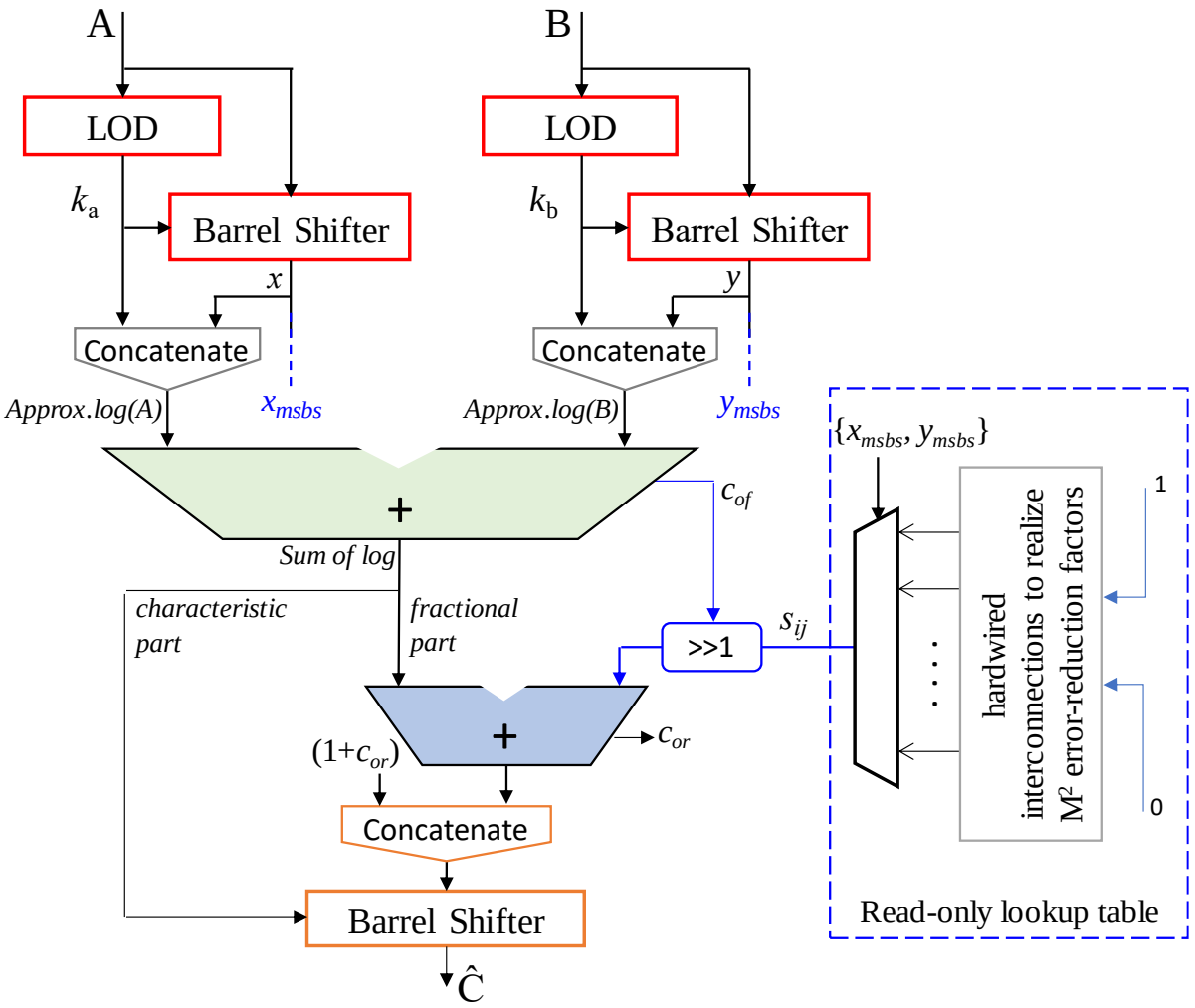
Apply error-correction on each segment separately



REALM – Hardware design

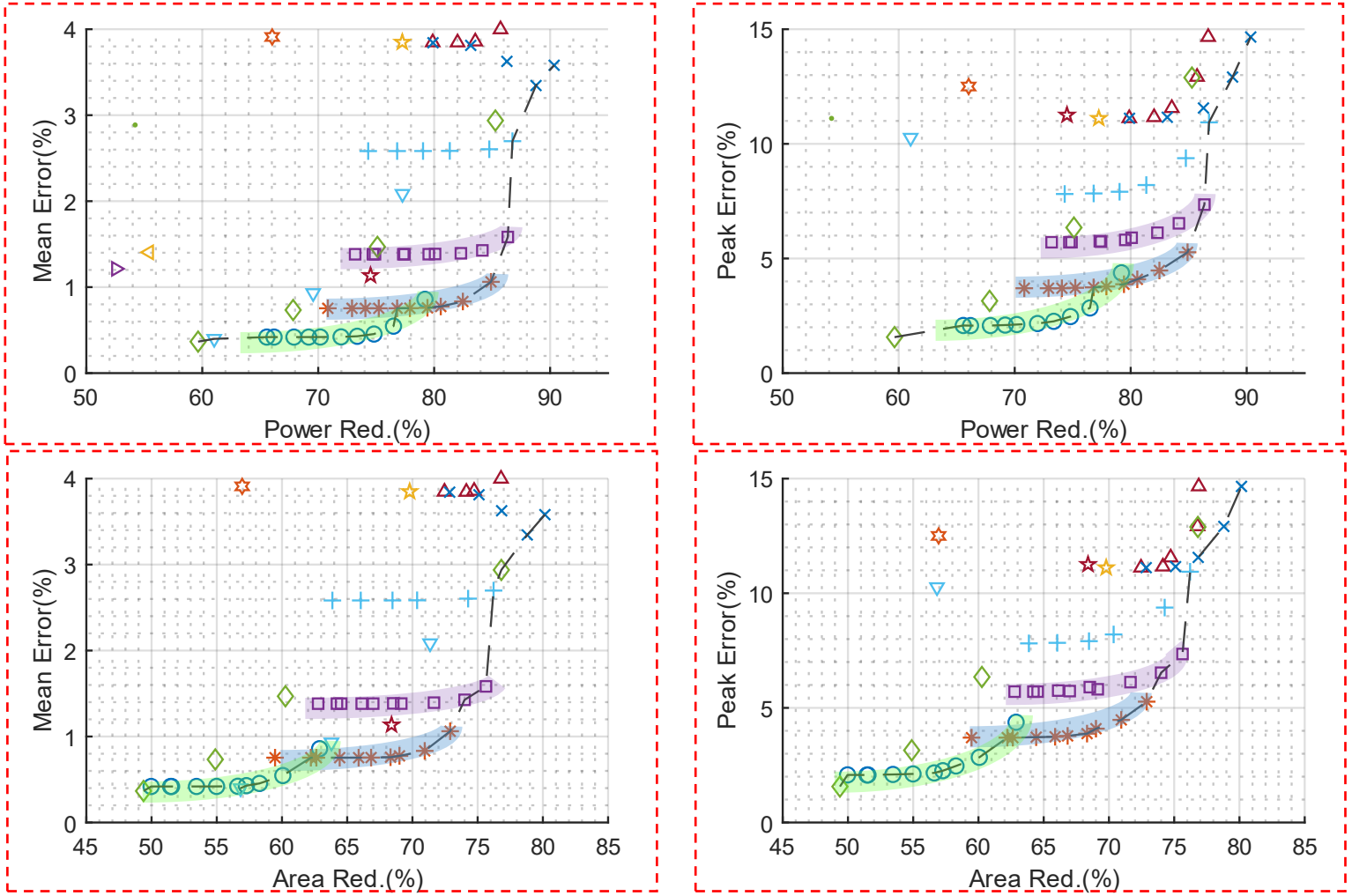
Proposed REALM

$$\hat{C} = \begin{cases} 2^{k_a+k_b}(1+x+y+s_{ij}), & x+y<1 \\ 2^{k_a+k_b+1}(x+y+s_{ij}/2), & x+y \geq 1 \end{cases}$$



REALM Results

REALM gives Pareto-optimal design points



References

- **Hassaan Saadat**, Haseeb Bokhari, Sri Parameswaran, "Minimally Biased Multipliers for Approximate Integer and Floating-Point Multiplication," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018. DOI: https://doi.org/10.1109/TCAD.2018.2857262

- **Hassaan Saadat**, Haris Javaid, Aleksandar Ignjatovic, Sri Parameswaran, "REALM: Reduced-Error Approximate Log-based Integer Multiplier," 2020 Design, Automation and Test in Europe Conference and Exhibition (DATE), 2020. DOI: https://doi.org/10.23919/DATE48585.2020.9116315

UNSW SYDNEY

Questions

- Can approximate multipliers be used in LWE cryptography?

- Can you come up with an approximation scheme better suited for this?