BitSAD: A Domain-Specific Language for Bitstream Computing

Your Brain is a Unary Computer '19

Kyle Daruwalla, **Heng Zhuo**, and Mikko Lipasti

22 June 2019

University of Wisconsin - Madison, Dept. of Elec. and Comp. Eng.

Motivation

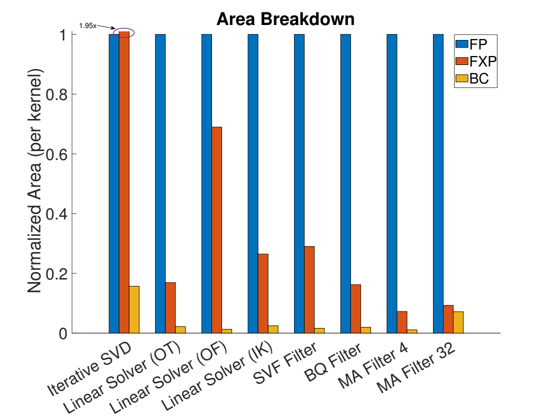


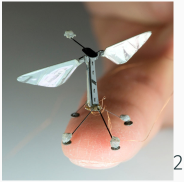
Figure 1: The resource (LUTs + FFs) consumption of Bitstream Computing (BC) implementations are much lower than floating point (FP) and fixed point (FXP) designs.

1. Why do we need BitSAD
2. What is BitSAD
3. What does it bring
4. What is coming next

Motivation and Background

Limitations of Current Approaches

RoboBee has ultra-low resources constraints:
($<35\,\text{mW}$ compute power) [1]


[2]

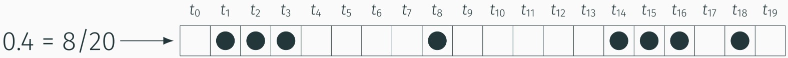
Traditional computing paradigms cannot meet these constraints!

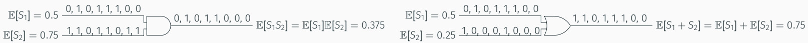
- Prior system connected to desktop computer
- Current on-device chip only supports basic stationary flight [3]

[1] Duhamel et al. 2011
[2] Ma 2015
[3] Zhang et al. 2017

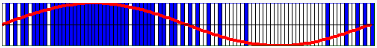
What is Bitstream Computing?

Stochastic Bitstreams:





Deterministic Bitstreams:

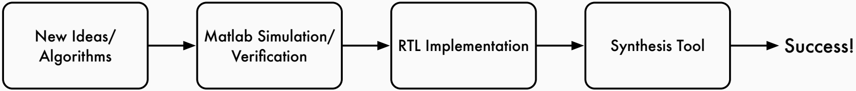


Density of “1” ⇒ Higher amplitude

- Sequence is *deterministic*
- Oversampled audio data
- Leads to efficient filters

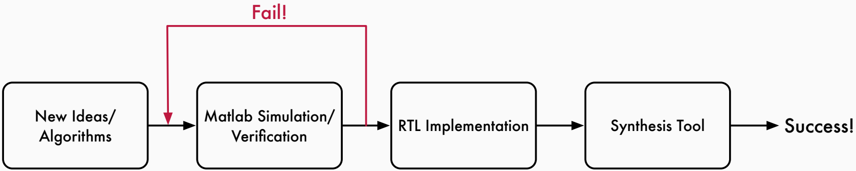
Design flow

Traditional design flow turnover is slow:



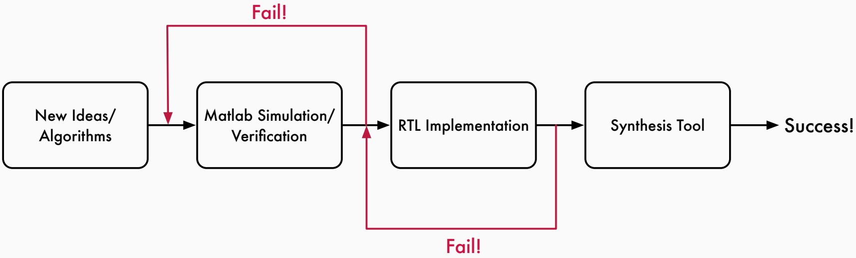
Design flow

Traditional design flow turnover is slow:



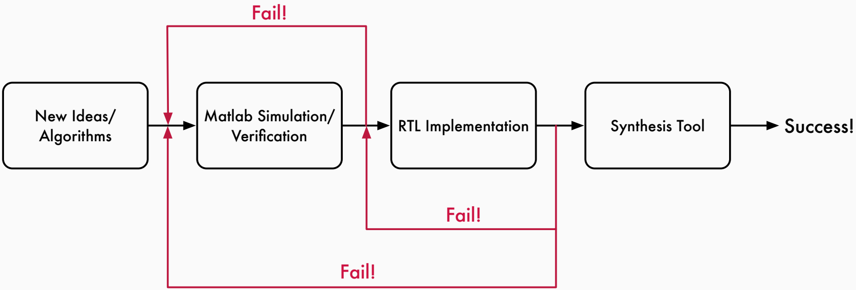
Design flow

Traditional design flow turnover is slow:



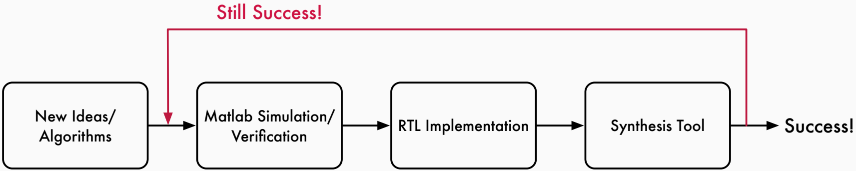
Design flow

Traditional design flow turnover is slow:



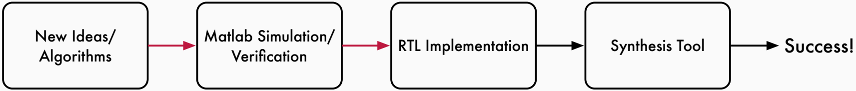
Design flow

Traditional design flow turnover is slow:



Design flow

Where is the problem coming from:



Design flow

What can we do about it:

BitSAD Bitstream Synthesizer and Designer

Based on Scala, which:

1. is a general-purpose, functional programming language
2. runs on Java virtual machine
3. uses IDE or command line build tool (`sbt`)
4. allows plugin in arbitrary compiler phases

Data Types

Listing 1: Example code on SBitstreams.

```
var a = SBitstream(0.5)
var b = SBitstream(0.25)
var c = SBitstream(-0.5)

var d = a + b
var e = a * c // handles sign
```

Listing 2: Example code on DBitstreams.

```
var buff = DelayBuffer(32) // delay buffer of length 32
var sdm = SDM() // Sigma-Delta modulator FXP -> DBitstream

var y = 2 * buff.pop + x.pop // x is a pre-loaded DBitstream
var z = sdm.evaluate(y)
```

Basic Operators

Operator	Description	SBitstream	DBitstream
+	Addition	Y	Y
-	Subtraction	Y	Y
*	Multiplication	Y	Y
/	Division	Y	N
:/	Fixed-Gain Div.	Y	N

Table 1: Operators defined on the SBitstream and DBitstream data type.

Listing 3: Operator Examples with mixed types.

```
var x = SBitstream(0.5)
var y = x - 0.1
var z = y * 0.2 + 1
```

Matrix Support

Similar to Matlab code:

Listing 4: Creating a `Matrix[A]`.

```
var a = Matrix(Array(
  Array(0.1, 0.2),
  Array(0.3, 0.4)
)
```

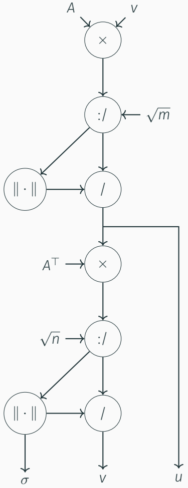
Listing 5: Working with `Matrix[A]`.

```
var a = rand[SBitstream](2, 2) // generate some random matrices
var b = rand[SBitstream](2, 1) // could be any Numeric type
var c = a * b                  // c is a 2x1 vector
var d = norm(0.25 * c)         // takes L2-norm of vector c
```

Hardware Generation

Listing 6: Example BitSAD program.

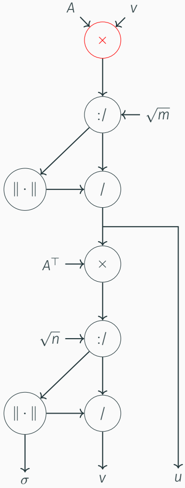
```
case class Module (params: Parameters) {

  // Define outputs
  val outputList = List(("v", params.n, 1),
                        ("u", params.m, 1),
                        ("sigma", 1, 1))

  def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
      (Matrix[SBitstream], Matrix[SBitstream], SBitstream) = {
    // Update right singular vector
    var w = A * v
    var wScaled = w :/ math.sqrt(params.m)
    var u = wScaled / Matrix.norm(wScaled)

    // Update left singular vector
    var z = A.T * u
    var zScaled = z :/ math.sqrt(params.n)
    var sigma = Matrix.norm(zScaled)
    var _v = zScaled / sigma

    (u, _v, sigma)
  }

}//Singular value decomposition
```



Hardware Generation

Listing 7: Example BitSAD program.

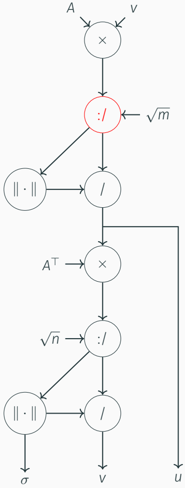
```
case class Module (params: Parameters) {

  // Define outputs
  val outputList = List(("v", params.n, 1),
                        ("u", params.m, 1),
                        ("sigma", 1, 1))

  def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
      (Matrix[SBitstream], Matrix[SBitstream], SBitstream) = {
    // Update right singular vector
    var w = A * v
    var wScaled = w :/ math.sqrt(params.m)
    var u = wScaled / Matrix.norm(wScaled)

    // Update left singular vector
    var z = A.T * u
    var zScaled = z :/ math.sqrt(params.n)
    var sigma = Matrix.norm(zScaled)
    var _v = zScaled / sigma

    (u, _v, sigma)
  }

}//Singular value decomposition
```



Hardware Generation

Listing 8: Example BITSAD program.

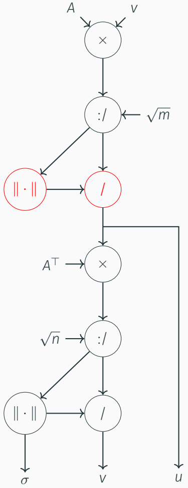
```
case class Module (params: Parameters) {

  // Define outputs
  val outputList = List(("v", params.n, 1),
                        ("u", params.m, 1),
                        ("sigma", 1, 1))

  def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
      (Matrix[SBitstream], Matrix[SBitstream], SBitstream) = {
    // Update right singular vector
    var w = A * v
    var wScaled = w :/ math.sqrt(params.m)
    var u = wScaled / Matrix.norm(wScaled)

    // Update left singular vector
    var z = A.T * u
    var zScaled = z :/ math.sqrt(params.n)
    var sigma = Matrix.norm(zScaled)
    var _v = zScaled / sigma

    (u, _v, sigma)
  }

}//Singular value decomposition
```



Hardware Generation

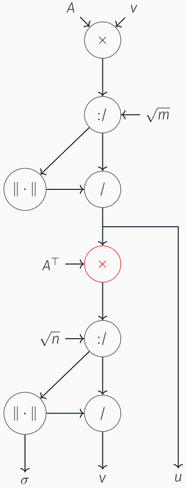
Listing 9: Example BITSAD program.

```
case class Module (params: Parameters) {

  // Define outputs
  val outputList = List(("v", params.n, 1),
                        ("u", params.m, 1),
                        ("sigma", 1, 1))

  def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
      (Matrix[SBitstream], Matrix[SBitstream], SBitstream) = {
    // Update right singular vector
    var w = A * v
    var wScaled = w :/ math.sqrt(params.m)
    var u = wScaled / Matrix.norm(wScaled)

    // Update left singular vector
    var z = A.T * u
    var zScaled = z :/ math.sqrt(params.n)
    var sigma = Matrix.norm(zScaled)
    var _v = zScaled / sigma

    (u, _v, sigma)
  }

}//Singular value decomposition
```



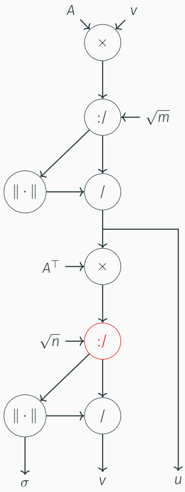
Listing 10: Example BitSAD program.

```
case class Module (params: Parameters) {

  // Define outputs
  val outputList = List(("v", params.n, 1),
                        ("u", params.m, 1),
                        ("sigma", 1, 1))

  def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
      (Matrix[SBitstream], Matrix[SBitstream], SBitstream) = {
    // Update right singular vector
    var w = A * v
    var wScaled = w :/ math.sqrt(params.m)
    var u = wScaled / Matrix.norm(wScaled)

    // Update left singular vector
    var z = A.T * u
    var zScaled = z :/ math.sqrt(params.n)
    var sigma = Matrix.norm(zScaled)
    var _v = zScaled / sigma

    (u, _v, sigma)
  }

}//Singular value decomposition
```



Listing 11: Example BitSAD program.

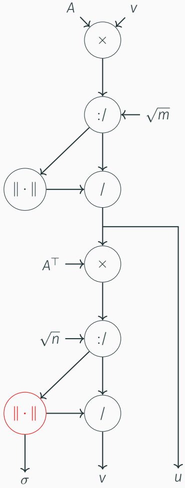
```
case class Module (params: Parameters) {

  // Define outputs
  val outputList = List(("v", params.n, 1),
                        ("u", params.m, 1),
                        ("sigma", 1, 1))

  def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
      (Matrix[SBitstream], Matrix[SBitstream], SBitstream) = {
    // Update right singular vector
    var w = A * v
    var wScaled = w :/ math.sqrt(params.m)
    var u = wScaled / Matrix.norm(wScaled)

    // Update left singular vector
    var z = A.T * u
    var zScaled = z :/ math.sqrt(params.n)
    var sigma = Matrix.norm(zScaled)
    var _v = zScaled / sigma

    (u, _v, sigma)
  }

}//Singular value decomposition
```



Hardware Generation

Listing 12: Example BitSAD program.

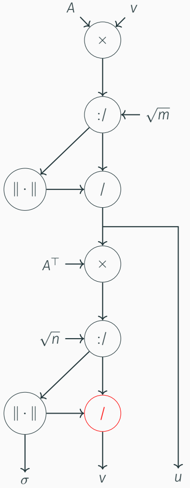
```
case class Module (params: Parameters) {

  // Define outputs
  val outputList = List(("v", params.n, 1),
                        ("u", params.m, 1),
                        ("sigma", 1, 1))

  def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
      (Matrix[SBitstream], Matrix[SBitstream], SBitstream) = {
    // Update right singular vector
    var w = A * v
    var wScaled = w :/ math.sqrt(params.m)
    var u = wScaled / Matrix.norm(wScaled)

    // Update left singular vector
    var z = A.T * u
    var zScaled = z :/ math.sqrt(params.n)
    var sigma = Matrix.norm(zScaled)
    var _v = zScaled / sigma

    (u, _v, sigma)
  }

}//Singular value decomposition
```



Hardware Generation

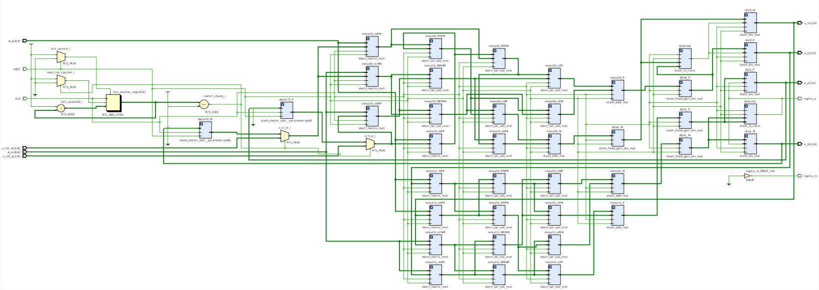
Listing 13: Example BitSAD program.

```
case class Module (params: Parameters) {

  // Define outputs
  val outputList = List(("v", params.n, 1),
                        ("u", params.m, 1),
                        ("sigma", 1, 1))

  def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
      (Matrix[SBitstream], Matrix[SBitstream], SBitstream) = {
    // Update right singular vector
    var w = A * v
    var wScaled = w :/ math.sqrt(params.m)
    var u = wScaled / Matrix.norm(wScaled)

    // Update left singular vector
    var z = A.T * u
    var zScaled = z :/ math.sqrt(params.n)
    var sigma = Matrix.norm(zScaled)
    var _v = zScaled / sigma

    (u, _v, sigma)
  }

}//Singular value decomposition
```



But, wait, why not just write Verilog then?

But, wait, why not just write Verilog then?



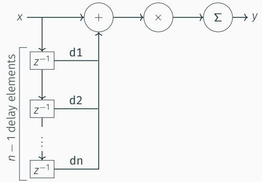
Warning

It is a bad idea!

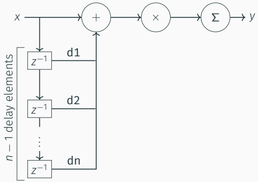
Subtleties of Bitstream Computing

Example: Moving Average Filter

Consider the following equivalent expressions for a moving average filter of length 4:



Example: Moving Average Filter



Consider the following equivalent expressions for a moving average filter of length 4:

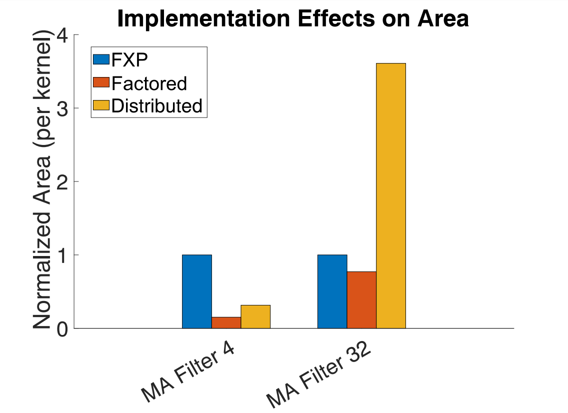
Factored:

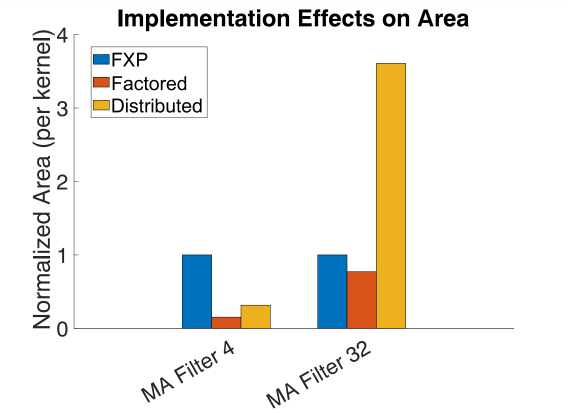
```
0.25 * (d1 + d2 + d3 + x)
```

Distributed:

```
0.25 * d1 + 0.25 * d2 + 0.25 * d3 + 0.25 * x
```

Example: Moving Average Filter



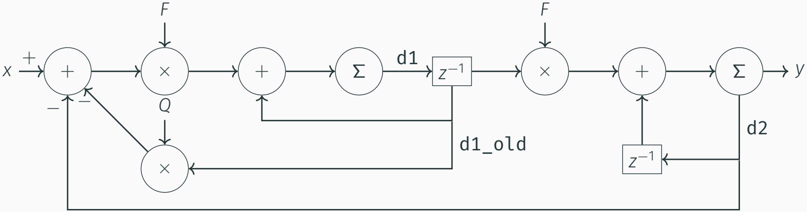


Remark

BitSAD allows users to effectively explore these trade-offs

Work in Progress

Constant coefficient: F, Q, input: x, output: y

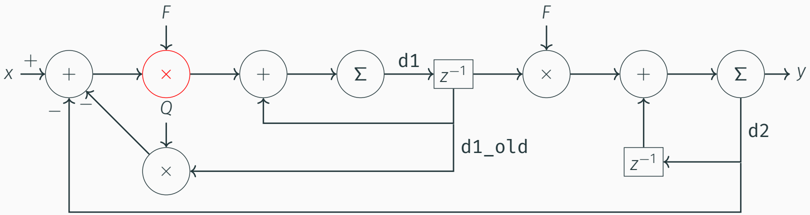


Baseline:

```
d1 = F * (x - d2 - Q * d1_old) + d1_old
```

Example: State Variable Filter

Constant coefficient: F, Q, input: x, output: y



Baseline:

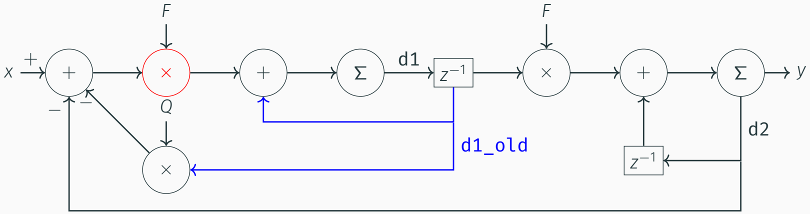
```
d1 = F * (x - d2 - q * d1_old) + d1_old
```

with strength reduction:

```
d1 = F * x - F * d2 - F * Q * d1_old + d1_old
```

Example: State Variable Filter

Constant coefficient: F, Q, input: x, output: y



Baseline:

```
d1 = F * (x - d2 - q * d1_old) + d1_old
```

with strength reduction:

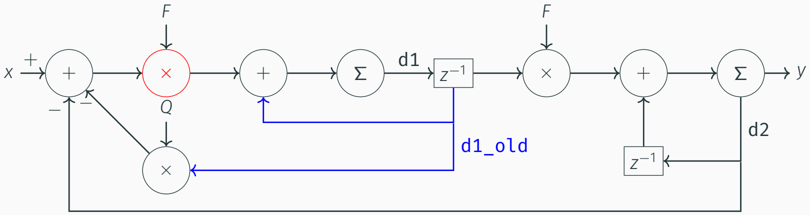
```
d1 = F * x - F * d2 - F * Q * d1_old + d1_old
```

with strength reduction and algebraic simplification:

```
d1 = F * x - F * d2 + (1 - F * Q) * d1_old
```

Example: State Variable Filter

Constant coefficient: F, Q, input: x, output: y



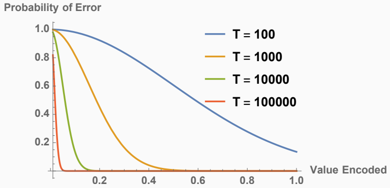
Let BITSAD handle this

- try different combinations
- compare systhesized results
- choose the best implementation

Latency Issue

Bitstreaming Computing latency is high!

$$p = \lim_{T \to \infty} \frac{1}{T} \sum_{t=1}^{T} X_t$$

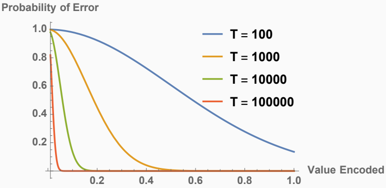


Error Bound for varying $p \in [0.01, 1]$ and varying T as well

Latency Issue

Bitstreaming Computing latency is high!

$$p = \lim_{T \to \infty} \frac{1}{T} \sum_{t=1}^{T} X_t$$

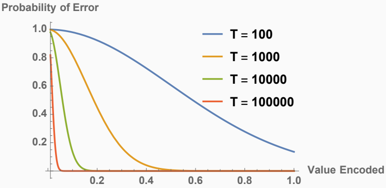


How do we trade off between latency and accuracy?

Latency Issue

Bitstreaming Computing latency is high!

$$p = \lim_{T \to \infty} \frac{1}{T} \sum_{t=1}^{T} X_t$$



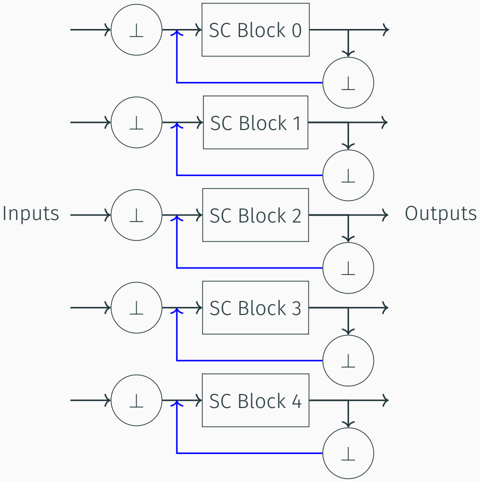
How do we trade off between latency and accuracy?

Bit-level, cycle-accurate software simulation in BitSAD

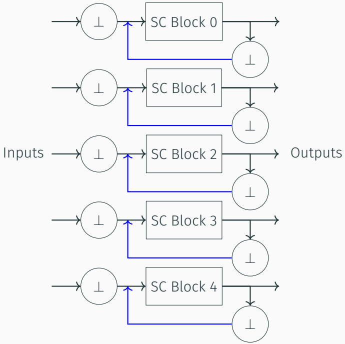
Latency Issue

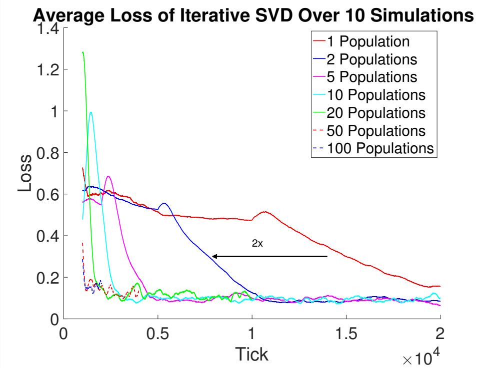
Population coding,
inspired from biology:

$$\overline{p} = \frac{1}{T/N} \sum_{t=1}^{T/N} \frac{1}{N} \sum_{i=1}^{N} X_{i,t}$$



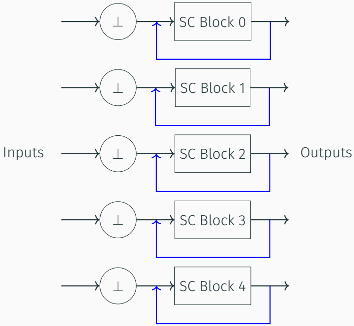
Population Coding: good?

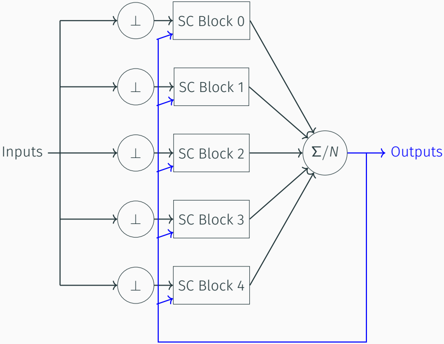




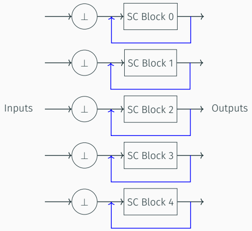
Experiments done with iterative SVD.

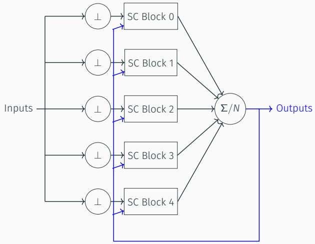
Population Coding: how to do better?

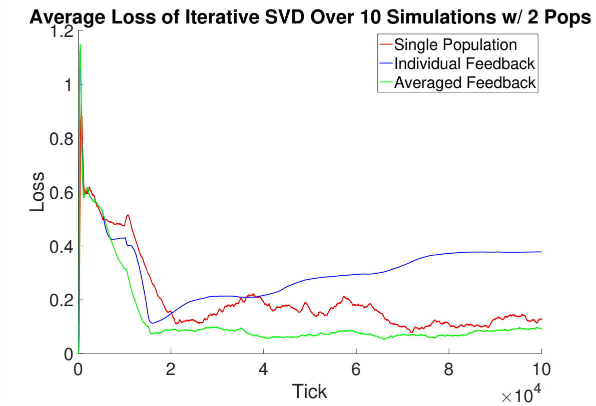




Population Coding: better!







Optimization

Design optimization based on:

- Design details (RTL)
- Design requirements (Timing, Area, Power)

Design optimization iteration:

- Strength reduction and algebraic simplification
- Population coding
- more to explore

Conclusions

Ultra-low power/resource constrained applications require new computing paradigm

BitSAD allows:

- software algorithm testing
- Verilog generation automation
- fast design turnovers

What is more:

- BitBench, tomorrow at LCTES'19 (`https://github.com/UW-PHARM/BitBench`)
- BitSAD v2, in progress (`https://github.com/UW-PHARM/BitSAD`)

Dont be bit sad, use BitSAD!

Questions?

Duhamel, Pierre Emile et al. (2011). "Hardware in the loop for optical flow sensing in a robotic bee". In: *IEEE International Conference on Intelligent Robots and Systems*, pp. 1099–1106. ISSN: 2153-0858. DOI: 10.1109/IROS.2011.6048759.

Ma, Kevin Y. (2015). *RoboBee*. URL: http://www.aboutkevinma.com/index.html#publications (visited on 04/01/2018).

Zhang, Xuan et al. (2017). "A Fully Integrated Battery-Powered System-on-Chip in 40-nm CMOS for Closed-Loop Control of". In: *IEEE Journal of Solid-State Circuits* 52.9, pp. 2374–2387. DOI: 10.1109/JSSC.2017.2705170.

What is Bitstream Computing?

Given a floating point number, p, as the mean of a Bernoulli distribution, can be encoded with a stochastic bitstream, the value of which, X is:

$$\mathbb{P}(X_t = 1) = p \qquad \mathbb{P}(X_t = 0) = 1 - p \tag{1}$$

With timesteps T, p can be estimated as:

$$p = \mathbb{E}X_t \approx \frac{1}{T}\sum_{t=1}^{T} X_t \tag{2}$$