BitSAD

A Domain-Specific Language for Bitstream Computing

Kyle Daruwalla
daruwalla@wisc.edu
University of Wisconsin - Madison
Madison, Wisconsin, USA

Heng Zhuo
hzhuo2@wisc.edu
University of Wisconsin - Madison
Madison, Wisconsin, USA

Mikko Lipasti
mikko@engr.wisc.edu
University of Wisconsin - Madison
Madison, Wisconsin, USA

Abstract

Advances in manufacturing and fabrication have enable "pico-sized" robots. These devices show promise in areas such as search-and-rescue, synthetic pollination, stealth surveillance, etc., but enabling these applications requires evaluating advanced computer vision and machine learning algorithms in real-time while adhering to a *strict* power budget. Taking a cue from biology, bitstream computing realizes this goal by employing a unary, streaming data format. Computing under this paradigm allows complex operations such as decimal multiplication to be reduced to a simple AND gate. Yet, at the system-level, the implementation details and techniques expose complex trade-offs. Understanding and exploring these dynamics is a time-consuming task for designers. Thus, we introduce a domain-specific language (DSL), BitSAD, to make bitstream computing as programmatically efficient as Matlab or Python. Furthermore, the DSL can directly generate synthesizable Verilog. Using BitSAD, we will show that (1) bitstream computing can perform **complex algorithms with low resource consumption**, (2) seemingly minor implementation details can greatly influence the resulting design, and (3) BitSAD allows for quick **generation of synthesizable bitstream computing circuits in Verilog.**

Keywords bitstream, stochastic computing, domain-specific language (DSL), pulse density modulation, deterministic bitstream

1 Introduction

Sensors in embedded systems are increasingly relying on *bitstreams*: oversampled, sigma-delta modulated data streams. Typical computing platforms require power-hungry ADCs and DACs to convert between this format and binary. By processing the bitstream directly, we automatically save on energy by removing the data converters from the pipeline. Moreover, using techniques from stochastic computing (SC) [1], we can reduce the resource complexity of common operators. And, in applications such as filtering, we can leverage the oversampled input bit rate to tune our filter coefficients to be hardware-friendly.

In particular, robotic systems with real-time deadlines have historically relied on optimized fixed-point algorithms running on low-power microprocessors, digital signal processors, ASICs, or FPGAs. But, due to recent advancements in fabrication, miniature robots, known as pico-aerial vehicles (PAVs), are possible. Keeping these robots in flight requires complicated computer vision algorithms, and initial proof-of-concept demonstrations of such devices have relied on a desktop computer [2] [3]. Moreover, current solutions are only capable of performing the necessary computations to keep the robot in flight [2] [4]. This motivates the need for an alternate computing paradigm that is *resource efficient and flexible*. Fig. 1 displays the area consumption for several "kernel" applications typical for a PAV with floating point (FP), fixed point (FXP), and bitstream computing (BC) implementations (these designs are mapped to a ZYNQ-7000 FPGA). It is clear from the graph that the BC designs have a significant advantage in resource constrained applications.

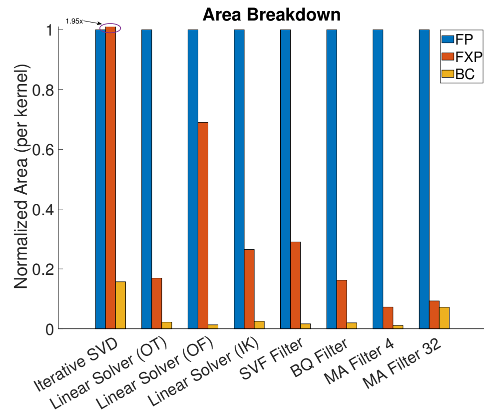

Figure 1. The resource (LUTs + FFs) consumption of bitstream computing (BC) implementations are much lower than floating point (FP) and fixed point (FXP) designs.

Unfortunately, developing an algorithm for bitstream computing can be cumbersome and time-consuming, and this discourages designers. On the other hand, writing software for a general purpose microprocessor seems simple and feasible. In order to make the savings in Fig. 1 accessible to algorithm developers, we created BitSAD. Our domain-specific language (DSL) allows users to write algorithms in a Matlab-like syntax. They can enjoy all the features of a high-level programming language that are not available in

Verilog. Furthermore, we provide custom data **types for bitstream computing** and libraries to perform **convenient linear algebra matrix operations**. Any algorithm developed in BitSAD can be **simulated at the software-level** to detect bit-level errors quickly, and the code automatically **generates synthesizable Verilog** that implements the algorithm.

2 BitSAD: Overview

BitSAD, or Bitstream Synthesizer and Designer, is a domain-specific language written in Scala. It provides custom types, classes, operators to make designing machine learning, computer vision, or generic linear algebra based algorithms for bitstream computing easy. Now, we will describe the structure and syntax of the DSL at a high-level.

2.1 Data Types

A bitstream is a sequence of single bit values in time. The value at a given timestep is either *stochastic* or *deterministic*. Stochastic bitstreams are the variety found in spiking neural networks and stochastic computing. Deterministic bitstreams are typical in audio applications, commonly known as pulse density modulation (PDM). An algorithm in BitSAD operates on either stochastic or deterministic bitstreams.

2.1.1 Stochastic Bitstreams

A stochastic bitstream encodes a floating point number, p, as the mean of a Bernoulli distribution. In other words, at each timestep, the value of the bitstream, X_t, is given by

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

Thus, the value of each bit in time is *stochastic* (i.e. it is not uniquely determined by p and t). If we wait T timesteps, we can estimate p by

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

Fig. 2 illustrates an example of a stochastic bitstream. Simple logic gates can be used to represent complex operators [1]. Special care must be taken to make sure all stochastic bitstreams represent a floating point number < 1 (violation of this condition is called saturation). This is required by Eq. 2. Similarly, stochastic bitstreams (as defined by Eq. 1) can only represent positive numbers. So, computation must be split into positive and negative data paths.

BitSAD provides the `SBitstream` type to represent stochastic bitstreams. Table 1 illustrates the operators defined for `SBitstreams`. Notice that we support special operations, such as fixed-gain division, in which a stochastic bitstream is divided by a positive constant larger than one. At the hardware-level, this operator is more efficient that dividing two bitstreams.

Furthermore, BitSAD handles issues like saturation and splitting computation into multiple data paths. Consider the code in Lst. 1. On Line 5, the result of `a + b` is greater than one. In this case, the compiler will throw a warning alerting the programmer about possible saturation. On Line 6, signed multiplication is being performed, which requires many separate data paths to determine the final signed result. The programmer does not need to be aware of this, and BitSAD handles this under the hood.

Listing 1. Example code on `SBitstreams`.

```
1 var a = SBitstream(0.5)
2 var b = SBitstream(0.75)
3 var c = SBitstream(-0.5)
4
5 var d = a + b // throws warning
6 var e = a * c // handles sign
```

2.1.2 Deterministic Bitstreams

Deterministic bitstreams is the BitSAD nomenclature for PDM audio streams. In this data format, the density of ones is proportional to the value being represented. While the computation on deterministic bitstreams is still done with fixed point units, we can leverage the encoding to use much lower precision hardware than the common binary representation, pulse coded modulation (PCM). This is because audio filters are designed with the sample rate as a parameter, which is fixed for PCM data. In contrast, PDM data is oversampled (3 MHz vs. 44.1 kHz), so we can choose our effective sample rate to achieve low-precision filter coefficients. For example, the coefficients for an state-variable filter (SVF) are given by

$$f = 2\sin\left(\frac{\pi F_c}{F_s}\right) \qquad q = \frac{1}{Q} \tag{3}$$

where F_c is the center frequency, F_s is the sample rate, and Q is the quality factor [5]. For a given center frequency, there is only one choice of f. With deterministic bitstreams, we can choose f to be a low-precision number (e.g. 3 bits), then adjust F_s to satisfy Eq. 3. In this way, we can design low resource filters with the same fidelity as their PCM counterparts.

BitSAD provides the `DBitstream` type for deterministic bitstreams. Table 1 shows the operators defined on this data type.

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

Table 1. Operations defined on the `SBitstream` and `DBitstream` data type.

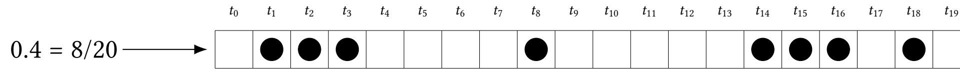

Figure 2. The value 0.4 is represented using a stochastic bitstream. To represent the value 0.4 over 20 time ticks, there should be 8 random occurrences of a high bit.

When working with DBitstreams, the resulting output will be a fixed point number. Since the overall filter output is also a bitstream, a sigma-delta modulator (SDM) must be used to convert the FXP number to a bitstream. BitSAD provides a simple class called SDM to do this. Similarly, many filters require delay buffers, which is provided by the DelayBuffer class. Lst. 2 illustrates how these classes are used.

Listing 2. Example code on DBitstreams.

```
1 var buff = DelayBuffer(32) // delay buffer of length 32
2 var sdm = SDM()
3
4 var y = 2 * buff.pop + x.pop // x is a pre-loaded DBitstream
5 var z = sdm.evaluate(y)
```

2.1.3 Type Hierarchy

Scala provides a rich type hierarchy that allows for implicit conversion and operations between different types. It also alleviates the burden of specifying type from the programmer, but its compile-time type inference system allows for the performance of statically typed languages.

BitSAD takes advantage of this by inheriting from the appropriate numerical classes. So, to the type system, SBitstream and DBitstream are Numeric types. This means that our types work out-of-the-box for any code whose sole requirement is a Numeric type. Moreover, code like Lst. 3 makes sense.

Listing 3. Example operations with mixed types.

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

2.2 Matrix Support

Most computer vision or machine learning algorithms are expressed using linear algebra constructs. So, in BitSAD, we provide the Matrix[A] class to facilitate these constructs. A can be any Numeric type (so SBitstream and DBitstream plug-in easily). You can create a Matrix[A] using Arrays like in Lst. 4.

Listing 4. Creating a Matrix[A].

```
1 var a = Matrix(Array(
2         Array(0.1, 0.2),
3         Array(0.3, 0.4)
4 ))
```

Typically, users won't want to hardcode matrices. Instead, they would start from known formats, such as all zeros, all ones, random, or identity matrices, then manipulate them to provide the structure they desire. Table 2 lists all the functions that can be used to construct a matrix. Table 3 lists all the functions that can be used to manipulate matrices in complex ways. Additionally, matrices automatically support all the operators defined on A element-wise as well as matrix multiplication.

Function	Description
ones[A](rows, cols)	All-Ones Matrix
zeros[A](rows, cols)	All-Zeros Matrix
eye[A](rows, cols)	Identity Matrix
rand[A](rows, cols)	Uniform Random Matrix

Table 2. Function for creating Matrix[A] variables.

Function	Description
norm[A](m, type)	Matrix norm
conv2d[A](m, kernel)	2D convolution
reshape[A](m, r, c)	Reshape matrix
horzConcat[A](m1, m2)	Horizontal concatenate
vertConcat[A](m1, m2)	Vertical concatenate
tile[A](m, gridSize)	Split into array of matrices

Table 3. Function for manipulating Matrix[A] variables (show as m above).

Finally, Lst. 5 illustrates how simple it is work with matrices.

Listing 5. Working with Matrix[A].

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

2.3 Software Emulation

Under the hood, SBitstream and DBitstream operators are bit-aware. In other words, operators like addition are performed on streams of bits. Thus, at a software-level, BitSAD automatically emulates bitstream hardware. This allows designers to identify issues such as saturation *before* building a hardware system. Moreover, a flag can be used to force floating point computation so that SBitstream and DBitstream behave like floating point numbers. This allows a designer to verify an algorithm's correctness without the effects of bitstream computing.

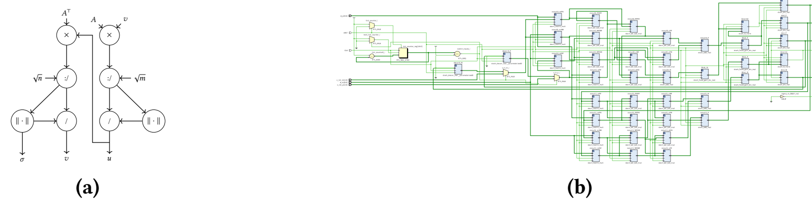

Figure 3. (a) A DFG of the example code. (b) Top-level RTL schematic generated by BitSAD compiler.

2.4 Hardware Generation

BitSAD ships with a Scala compiler plugin that generates synthesizable Verilog. Operators, like `:/`, are synthesized to pre-built modules also provided with the language. Users simply need to define the module they wish to synthesize, and the section of synthesize code in a `loop` function. Lst. 6 shows the iterative SVD from Fig. 1 as a BitSAD program.

Listing 6. Example BitSADprogram.

```
package IterativeSVD

import bitstream.types._
import bitstream.simulator.units._
import math._

trait Parameters {
  val m: Int
  val n: Int
}

object DefaultParams extends Parameters {
  val m = 2
  val n = 2
}

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)
  }

}
```

Fig. 3 illustrates how much work is taken from the programmer and placed on the compiler. Fig. 3a is a directed-flow graph of the `loop` body. This is the abstraction level at which the programmer designs an algorithm. Fig. 3b is the RTL schematic of the top-level Verilog generated by the compiler.

3 Subtleties of Bitstream Computing

As shown in Fig. 1, bitstream computing can greatly reduce the resource consumption of designs. Yet, this is not a "free lunch." Consider the following equivalent expressions for a moving average filter of length 4:

$$y_t = 0.25 \sum_{i=t}^{t-3} x_i \tag{4}$$

$$y_t = \sum_{i=t}^{t-3} 0.25 x_i \tag{5}$$

In Eq. 4, the last four input samples are added, then multiplied by 0.25. In Eq. 5, the samples are scaled by 0.25, then added. For normal FP or FXP arithmetic, it is clear that Eq. 5 requires more multipliers and is more expensive. But with bitstream computing, x_i is a single bit, so $0.25x_i$ can be implemented with a simple mux. On the other hand, $\sum_{i=t}^{t-3} x_i$ is a FXP number, so multiplying outside the sum requires a fixed point multiplier. Clearly, the fixed point multiplier is more expensive than four muxes, so it would be reasonable to favor Eq. 5.

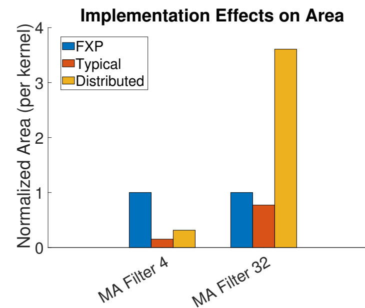

Figure 4. Effects of implementation details on area. "Typical" designates an stochastic bitstream design of Eq. 4 and "Distributed" designates a stochastic bitstream design of Eq. 5. In the latter case, the bitstream design consumes more area than the FXP design for an 32-wide MA filter.

Fig. 4 illustrates that this is not so straightforward. Notice that for a moving average filter of length 32, the bitstream design is more costly than the FXP design. This is because x_i is a single bit, so at each node in the sum, we add the accumulated result (an FXP number) to another sample (a bit). Suppose we are using 8-bit FXP numbers. Then each node is an 8-bit adder with 9 bits of unknown (to the synthesizer)

input. In contrast, $0.25x_i$ is an FXP number. So each node in this style sum is an 8-bit adder with 16 bits of unknown input. In the former case, the synthesizer is able to utilize resource sharing and LUT compression to create a smaller design that in the latter case.

Thus, even for the simple example of a moving average filter, bitstream computing exposes subtle trade-offs that vary as the algorithm is scaled. This behavior is even more complex for advanced algorithms such as the iterative singular value decomposition (SVD) in Fig. 1. BitSAD is designed to allow users to *effectively* explore these trade-offs *quickly*.

4 Conclusion and Future Work

This work has highlighted the potential opportunity of bitstream computing, as well as the pitfalls and subtleties of bitstream designs. Under this motivation, we designed BitSAD to allow users to quickly implement algorithms using bitstream data types, test those algorithms in software, then generate synthesizable Verilog that implements those algorithms. Our goal is to continue to develop the DSL to identify trade-off scenarios (such as in Sec. 3) and optimize expressions.

We hope that the creation of BitSAD will lead to a greater focus on bitstream computing as a viable alternative in resource constrained systems, enabling a slew applications otherwise limited by overbearing compute components.

References

[1] B. R. Gaines. Stochastic computing. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 149–156, New York, NY, USA, 1967. ACM.

[2] Xuan Zhang, Mario Lok, Tao Tong, Sae Kyu Lee, Brandon Reagen, Simon Chaput, Pierre-Emile J Duhamel, Robert J Wood, David Brooks, and Gu-Yeon Wei. A Fully Integrated Battery-Powered System-on-Chip in 40-nm CMOS for Closed-Loop Control of. *IEEE Journal of Solid-State Circuits*, 52(9):2374–2387, 2017.

[3] Xuan Zhang, Mario Lok, Tao Tong, Simon Chaput, Sae Kyu Lee, Brandon Reagen, Hyunkwang Lee, David Brooks, and Gu-yeon Wei. A Multi-Chip System Optimized for Insect-Scale Flapping-Wing Robots.

[4] Taylor S. Clawson, Silvia Ferrari, Sawyer B Fuller, and Robert J Wood. Spiking Neural Network (SNN) Control of a Flapping Insect-scale Robot. In *Conference on Decision and Control, IEEE*, number 55, pages 3381–3388, 2016.

[5] Nigel Redmond. The digital state variable filter, 2003.