INVITED: BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design

Michael Bedford Taylor
Bespoke Silicon Group
University of Washington
http://bjump.org/stl

ABSTRACT

We propose a Standard Template Library (STL) for synthesizeable SystemVerilog that sharply reduces the time required to design digital circuits. We overview the principles that underly the design of the open-source BaseJump STL, including light-weight latency-insensitive interfaces that yield fast microarchitectures and low bug density; thin handshaking rules; fast porting of hardened chip regions across nodes; pervasive parameterization and specialization, and static error checking. We suggest extensions to SystemVerilog that will make it a more functional design language, and discuss our validation, including with the DARPA CRAFT-sponsored 16nm TSMC Celerity SoC with 511 RISC-V cores and 385M transistors. 80% of the modules for the design were instantiated directly from BaseJump STL, reducing verification time, accelerating development, and showing the promise of the approach.

CCS CONCEPTS

• **Hardware → Very large scale integration design**; **VLSI system specification and constraints**;

KEYWORDS

Hardware Design

1 INTRODUCTION

Improvements in hardware design productivity have significantly lagged corresponding improvements in software design productivity, despite the rising importance of specialization [5, 6, 8–11] to address the dark silicon problem [12, 13]. Modern software programming languages have extensive libraries of useful routines that are tested, reusable, and highly composable. These libraries act as a multiplier on programmer productivity, allowing them to rapidly connect together large portions of pre-written code into large, powerful applications. In the C/C++ software world, one of the most transformative such libraries was the C++ Standard Template Library (STL), which originated outside of the C++ language but was later folded in. Early C/C++ libraries focused on providing I/O calls and a portability interface, but, much like IP cores today in the FPGA/ASIC world, these were "leaf" level routines, rather than building blocks that can be composed to implement new classes of computation and data movement. The STL, in contrast, focused on four classes of composable blocks: containers, algorithms, functions, and iterators. Containers implemented high-performance data structures that employed asymptotically optimal algorithms for insertion, retrieval, and removal and which through the use of templates, could be applied to arbitrary datatypes. Algorithms provided ways of manipulating this data, for example searching, sorting, union, intersection, and merging. Iterators provided an uniform interface for code to access containers regardless of their implementation. Functions could be used to customize the behavior of algorithms, for example, specifying a comparison operator for a sort algorithm. A final key feature of the STL was its focus on performance; in many cases, the code generated by the STL was more efficient than typical hand-written programmer code, since its use of templates allowed customized code to be generated for a particular data type and particular size parameters, and the libraries were written by a small number of world-class programmers, and in open source versions, is continuously improved via crowd-sourcing.

In this paper, we describe an experiment in which we generalize these insights from the C++ STL and apply it the task of creating a library to accelerate *synthesizeable* hardware design in the SystemVerilog hardware design language. The result is an open source library named *BaseJump STL*. BaseJump STL has been used by 40+ designers. BaseJump has been used to design both FPGA and ASIC HW. Four distinct chips designed with BaseJump STL have been taped out, fabricated, and tested, including a 511-core heterogeneous RISC-V core, a ten-core RISC-V array, and a high-speed source synchronous communication chip. Two of these chips were in TSMC 180nm, and two were in TSMC 16nm. Additionally, 8 unique RTL-to-GDS tapeins were performed in 90nm technology by third-party users. Finally, BaseJump STL has been used in FPGA designs that span Virtex-6, Spartan-6, and Zynq FPGA lines. BaseJump has a unique portability interface that allows designs to be rapidly retargeted across FPGA generations, between process nodes and between different foundry providers; for example BaseJump STL enabled a team at UC San Diego to do an emergency port of a 250nm chip design to a 180nm tapeout without changing any of the design's source code.

Of course, other approaches have been proposed to increase design productivity, such as high-level synthesis (HLS). Much as C/C++ has not been obsoleted by high-productivity languages like Python or TensorFlow due to the needs of lower-level systems programming, so too will SystemVerilog continue to play a role for highly tuned, crafted, hardware.

DAC '18, June 24–29, 2018, San Francisco, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3199848

2 WHAT SHOULD BE IN A HARDWARE DESIGN STL?

The mapping of the concept of the high-level idea of a C++ STL to the equivalent concept in SystemVerilog is not a straight-forward question. Informed by our construction of many previous systems, we settled upon a number of key features:

(1) **Portability Interface.** The STL should provide interfaces that allow a design to be moved, unchanged, between ASIC process nodes and vendors, as well as to different FPGA vendors.

(2) **Leaf Building Blocks.** The STL shall provide a portable interface to leaf building blocks, such as SRAMs, register files, clock generators, and high-speed I/O interfaces.

(3) **Efficient Hardware Primitives.** The STL should provide hardware primitive implementations commonly used by expert designers to create efficient hardware, for example, parallel prefix circuits.

(4) **Latency Insensitive Design.** To support interfacing of modules that have internal state, the STL should provide the right set of Latency-Insensitive Interfaces [4] that allow hardware to be composed correctly without considering the internals of the composed blocks, and without introducing power, performance, or area overhead. Moreover, these interfaces should allow wire delay to be managed in a portable way.

(5) **Parameterization.** The STL primitives should be elegantly parameterized to allow them to be reusable. Moreover, the implementations shall be pervasively specialized based on the input parameters, to allow code that is more efficient than can be reasonably written by humans. Parameters should also be used to capture slight variations in common modes of operation (e.g. count leading zeros from high to low versus from low to hi), improving code factoring, reducing bugs, and reducing module count.

(6) **Efficient Plumbing.** The STL should provide hardware primitive implementations that support efficient data movement. It should provide a spectrum of implementations, to cover both functionality and usage differences. For example, FIFOs stage data but there are a small number of fundamental ways to do it based on the demands of the surrounding logic. We need well-designed, bug-free asynchronous FIFOs, as well as implementations of FIFOs that are optimized, for example, based on whether reads and writes are performed every cycle, or less frequently. Moreover, these components should support efficient, minimalist, higher-level primitives like virtual channels, credit-counters, crossbars, and network routers.

(7) **Coding Style.** The STL shall use consistent coding styles and helper macros that reduce bugs and increase code understanding. For example, the use of low-true signals is avoided, since it is a hold-over from older days of logic design and leads to bugs. Non-synthesizable code shall be avoided generally, but when they are necessary for testing libraries (e.g. a clock generation module), they shall be clearly marked in the module name (e.g. `bsg_nonsynth_clock_gen`).

(8) **Metaprogramming.** Metaprogramming refers to code that generates code. The primary interface to meta-programmed routines shall be SystemVerilog, as opposed to using Python- or Scala- embedded DSLs, which are less succinct for leaf code and have harder learning curves. The STL shall make judicious use of metaprogamming constructs, such as generate statements. Where SystemVerilog generate statements become inefficient or awkward, python scripts shall be used to generate a SystemVerilog function that exhaustively lists the hardware generated for each combination of parameters. This generated code shall look like it is written by humans. This generated code is what is used by users of BaseJump STL, although the python scripts are available to verify correctness.

(9) **Testing Suite.** The STL shall have unit tests for each module.

3 PORTABILITY

Portability across ASIC and FPGA implementations is a desirable goal. The two most important aspects for portability with respect to ASIC versus FPGA flow are the treatment of reset logic and the use of process-specific hard cells like memories or synchronizer flops and level-shifters. Regarding reset signals, while FPGA flows allow registers to be initialized to a constant value via bitstream, ASICs require proper reset signals. In BaseJump STL, we address the reset issue by having explicit reset lines for logic in the interfaces of modules that *may* require reset logic, for example some memory block implementations may need a reset wire because they will be accompanied by some flops in a wrapper.

Synchronizer flops and level shifters are fairly uniform across process nodes, so generic module interfaces are easy to define. In ASIC and FPGA flows, SRAMs and register files are typically instantiated using memory generators. The interfaces of the memories have idiosyncratic interfaces and properties, seldom agreeing on whether signals are low or high true, how mask bits are specified, or behavior on simultaneous reads and writes. These factors need to be aggregated and unified into usable interfaces for portability. Another portability constraint is the need to specify standard cells, with specific placement constraints. For example in the Synopsys IC Compiler flow, `rp_groups` are used to indicate groups of standard cells that are placed contiguously, and these groups can be easily placed at particular locations in the chip.

To achieve portability, we introduce the concept of a `hard` shadow directory that contains process technology tuned replacements for BaseJump STL modules (e.g. `bsg_mem/bsg_mem_1r1w_sync.v`'s shadow is `hard/tsmc/16/bsg_mem/bsg_mem_1r1w_sync.v`). These files subsume the standard portable file of the same name in the standard BaseJump STL directory structure but have exactly the same interface. So for example, for TSMC 16, we use a `bsg_mem_1r1w_sync` module which based on `widths_p` and `els_p`, instantiates a hardened memory-generator-created SRAM, but on other combinations, calls back to the standard synthesizeable code in `bsg_mem_1r1w_sync`, allowing fine grained substitution according to whichever implementation is most optimal for that configuration of SRAM. Typically, these modules take a `harden_p` flag which allows the tools to flag if a hardened version has not been substituted, and also allows the user to override the hardening of a module and to use the synthesizeable version instead.

Memory portability using BaseJump STL is achieved by enforcing a coding style that requires explicit instantiation of a set of BaseJump STL memory primitives instead of using SystemVerilog-inferred memories, or using standard flip-flops as synchronizers. Here is an example interface for a memory:

```
module
bsg_mem_1r1w_sync_mask_write_bit
#(parameter width_p=-1
, parameter els_p=-1
, parameter write_then_read_same_addr_p=0
, parameter addr_width_lp=`BSG_SAFE_CLOG2(els_p)
, parameter harden_p=0
)
(input    clk_i
, input reset_i

, input                     w_v_i
, input [width_p-1:0]       w_mask_i
, input [addr_width_lp-1:0] w_addr_i
, input [width_p-1:0]       w_data_i

, input                      r_v_i
, input [addr_width_lp-1:0]  r_addr_i

, output logic [width_p-1:0] r_data_o
);

bsg_mem_1r1w_sync_mask_write_bit_synth
#(.width_p(width_p)
,.els_p (els_p  )
,.write_then_read_same_addr_p(write_then_read_same_addr_p)
,.harden_p(harden_p)
) synth
(.*);

/* assertions checking for read/write address collision */
...
/* initial begin that prints out memory parameters */
endmodule
```

The above example shows the interface for a memory with 1 read port and 1 write port, both synchronous, which can be an arbitrary width and contain an arbitrary number of words, and allows for individual bits to be selected for writing via a mask. A critical correctness issue for ASIC portability are the semantics of simultaneous read and writes to SRAMs. In some memories, a simultaneous read and write destroys the contents of that memory cell. In other memories, a simultaneous read and write results in correct writes of the data, but incorrect read values are returned. In other memories, the side effects are that the write occurs and then the read occurs. The parameter `write_then_read_same_addr_p` indicates the intent of the designer as to whether they expect the ram to be able to successfully perform a simultaneous write and read of the same address, with the write appearing to go first[1]. The use of this parameter is critical to avoid portability even when memories have different semantics. The implementation of the module thunks down onto a synthesizeable implementation (or a hard block with synthesized logic to adjust the semantics) if the behavior is unsupported directly by the hard block.

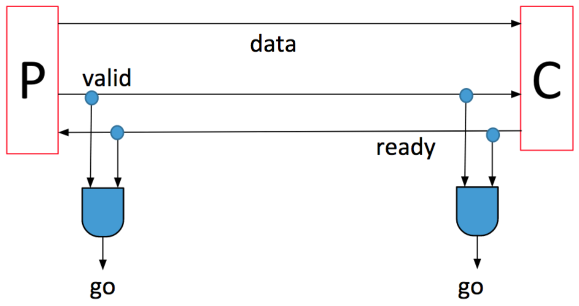

Figure 1: *ready-valid-and* (rv->&) synchronization.

In some cases, analog or mixed signal IP blocks may have highly idiosyncratic interfaces for which there is no conceivable general interface. In most of these cases, these items are best placed at the top-level of the design, which conveniently sequesters non-portable code into the one top-level file of the design.

4 LATENCY-INSENSITIVE HARDWARE DESIGN

Latency-Insensitive Hardware Design [4] is powerful design technique that drives designers to decompose their systems into a set of connected modules that have simple standardized handshake interfaces. These interfaces allow modules to be connected in new ways without having to reason about the timing inside the modules. Additionally, they allow the system to be correct even if the number of cycles that a downstream module takes can vary. Effectively latency-insensitive interfaces localize the control logic within a module, eliminating the need to have timing diagrams and datasheets for modules to convey correct usage. (See [7] for a survey of the diversity of interfaces found across a broad corpus of SystemVerilog code.)

Of course, some logic does benefit from cycle-accurate reasoning, for example, a heavily bypassed processor pipeline with complex interactions between instructions. In these cases, it makes sense to compose the system so that the latency insensitivity property applies to the interface of the module but not necessarily the internals.

A typical interface between two Latency-Insensitive modules is shown in Figure 1. In this paper, we term this *ready-valid-and*, or `rv->&`, indicating that only the & gate depends on the ready and valid signal. At the beginning of the cycle, both Producer (P) and Consumer (C) send their willingness to send data and to receive it, respectively. During the cycle, the signals propagate across the chip. At the end of the cycle, the Producer and Consumer independently

[1]We chose this approach because a hardened 1R1W SRAM that does not support this can be easily modified with some additional standard cells to do write-to-read bypassing, where as read-then-write is much more expensive.

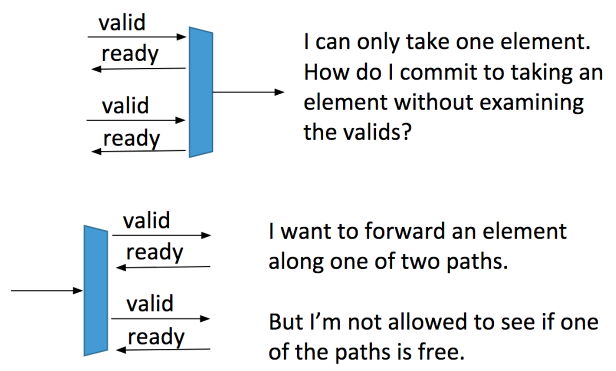
valid
ready
valid
ready
I can only take one element. How do I commit to taking an element without examining the valids?
valid
ready
valid
ready
I want to forward an element along one of two paths.
But I'm not allowed to see if one of the paths is free.

Figure 2: `rv->&` is not universal.

AND the signals together to determine whether the exchange of data has occurred. There are a number of desirable properties of this arrangement. First, there are no combinational loops in the system. Second, the ready and valid signals have almost an entire cycle to transfer between producer and consumer, which means that producer and consumer can be placed almost an entire clock cycle of wire delay apart, making them a good fit for communication between blocks at the top-level of the design, and even more so when hierarchical physical design flows are used. In cases where a greater wire latency is implied, an intermediate node can be inserted that also speaks the same protocol and forwards data packets on, typically using a 2-element buffer.

A challenge with `rv->&`, however, is that it is not universal. There are simple pieces of hardware that cannot be built with these interfaces. The problem is, as shown in Figure 2 in some cases, a producer or consumer needs to combine an input ready or valid signal with other information before committing to an action, which violates the invariants of `rv->&`.

For universality, BaseJump STL adds two more interfaces, shown in Figure 3. We call the interface where a producer computes a valid signal based on the consumer's ready signal *ready-then-valid*, or `r->v`, and we say that the producer is *demanding*, because it requires up-front information and the consumer is *helpful*, as it offers the information up-front. We call the interface where a consumer computes a ready signal based on the producer's valid signal *valid-then-ready*, or `v->r`, and we say that the consumer is *demanding*, because it requires up-front information and the producer is *helpful*, because it offers the information up-front. These two interfaces enable more conditional behavior on the part of the producer and

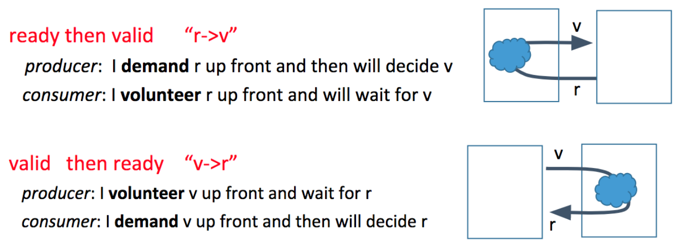
ready then valid "r->v"
producer: I demand r up front and then will decide v
consumer: I volunteer r up front and will wait for v
v
r
valid then ready "v->r"
producer: I volunteer v up front and wait for r
consumer: I demand v up front and then will decide r
v
r

Figure 3: Two interfaces for universality: ready-then-valid `r->v` and valid-then-ready `v->r`.

		Consumer Demanding	**Consumer** Helpful
Producer	Demanding	Use a FIFO	`r->v`
Producer	Helpful	`v->r`	`rv->&`

Figure 4: Taxonomy of producer-consumer interfaces based on if producer and consumer are *helpful* or *demanding*.

consumer, and make our library of handshakes more universal. However, they carry a few limitations. First, because a round trip is required for the signal to go from the producer to the consumer and back, or visa versa, the modules can be at most half a cycle of wire latency away, and often less, since the computation of the dependent signal usually requires at least a few gates. Inherently these interfaces are more local, and should not be placed at the top-level of the chip between two distant modules. So `rv->&` is better suited for long links, because in that interface, both sides of the link are *helpful*.

But what if both the producer and the consumer are *demanding*? In this case, they are not directly compatible. We solve this by inserting a FIFO; typical FIFOs are *helpful* on both ends, telling the producer side at the beginning of the cycle if it has free space, and telling the consumer side at the beginning of the cycle if it has data. So a FIFO is a universal interfacer and automatically converts between incompatible interfaces.

As shown in Figure 4, we can, based on a given consumer-producer pair, easily determine which interface makes the most sense. However, in practice, we will not be designing producers and consumers together, since we want modules of many different types to inter-operate. Instead, we will have whatever interface it is that a module was designed with. The good news is, there is only one case that does not work with a simple converter module: the case where the producer is `r->v` and the consumer is `v->r`. When these two demanding interfaces are paired, a FIFO would have to be added.

Which of these interfaces should we use for a module? Generally speaking, the first rule is that we want an interface that requires no additional logic to be added, one that is the most natural for the module. The second rule is, all else being equal, we should chose interfaces that are helpful over those that are demanding. This, way, we will only be adding FIFOs when absolutely needed, and the overhead of latency insensitive design is minimized.

5 A WISHLIST FOR SYSTEMVERILOG

In the design of the several hundred modules for BaseJump STL, we discovered a number of shortcomings of SystemVerilog (and the synthesis tools) that could potentially be improved (apologies if some of these have been addressed at the time of publication!):

(1) Arrays of interfaces are not widely supported across tools, but arrays of structs are. We resorted to splitting bidirectional interfaces into structs to get around this issue, because arrays of structs are very useful.

(2) Better support for zero-width signals would greatly help parameterized modules. In many cases, a module can scale

from N items downto one item, and a pointer to one item requires zero wires. In many cases, we had to use a macro `‘define BSG_SAFE_CLOG2(x) ( ((x)==1) ? 1 : $clog2((x)))` to use widths of one instead of zero to allow the full spectrum of sizes to work with a module.

(3) Better support for forcing users to define parameters rather than requiring users to set defaults. For many modules, there is no sense of a reasonable default, and any default that is actually used is the wrong one. Currently, we define these (mostly numerical) parameters as the string "inv", to get the tools to fail fast if there is no good default value.

(4) True type polymorphism, where we can pass a struct or other type into a module. Useful for containers, etc. Leads to true C++ style flexibility.

(5) Declaration of bit widths using [0+:width_p] notation.

(6) The ability for modules to have output parameters, so that they may compute and return the widths of their outputs.

(7) Bitwidth inference with an easy way to see the inferred width.

(8) Better generate statement debugging (i.e. a preprocessor that outputs the expanded version.)

(9) A language construct to indicate in a module that a signal is unused, reducing spurious warnings from tools. This is useful to maintain uniform interfaces while porting across different architectures, for example for leaf-level process-specific blocks that sometimes require a reset line and sometimes do not.

6 EFFECTIVE HARDWARE PRIMITIVES

We list here a representative subset of the hardware primitives that we organically added to BaseJump STL. All modules are parameterized with data path widths, number of inputs, and number of internal storage items, when possible.

(1) **bsg_fpu_add, bsg_fpu_cmp, bsg_fpu_mul**.
Floating point operators.

(2) **bsg_idiv_iterative,bsg_imul_iterative**:
Iterative Divider and Multiplier.

(3) **bsg_crossbar_o_by_i**:
Arbitrary crossbar generator. Any number of inputs, outputs, and variable width.

(4) **bsg_mux**:
Variable els_p -input mux, with width_p bits per input.

(5) **bsg_transpose** :
Transpose 2D bit vector. a[i][j] -> a[j][i].

(6) **bsg_reduce**, **bsg_scan** :
Parallel Prefix Scan and Reduce Operations.

(7) **bsg_gray_to_binary**, **bsg_binary_plus_one_to_gray**:
Gray code support.

(8) **bsg_mesh_router**, **bsg_mesh_router_buffer**, **bsg_mesh_stitch**, **bsg_noc_links**:
Efficient Network on Chip.

(9) **bsg_tag_client**, **bsg_tag_master**, **bsg_tag_trace_reply**:
JTAG/SPI style remote state setter with Clock Domain Crossing.

(10) **bsg_async_fifo**, **bsg_async_credit_counter**, **bsg_async_ptr_gray**, **bsg_launch_sync_sync**, **bsg_sync_sync**:
Clock domain crossing logic.

(11) **bsg_decode**, **bsg_decode_with_v**, **bsg_encode_one_hot**, **bsg_priority_encode_one_hot_out**, **bsg_priority_encode**, **bsg_thermometer_count** :
Decoders & Encoders.

(12) **bsg_level_shift_up_down_sink**, **bsg_level_shift_up_down**:
Level shifters.

(13) **bsg_dff**, **bsg_dff_en**, **bsg_dff_negedge_reset**, **bsg_dff_reset**, **bsg_dff_reset_en**.
Flop trays.

(14) **bsg_tiehi**, **bsg_tielo**:
For safely tying input pins to logical 0 or 1; avoids ESD issues. (Example use: Allows controlled placement of tie cells attached to I/O Pads to avoid crosstalk.)

(15) **bsg_mem_1r1w**, **bsg_mem_1r1w_sync**, **bsg_mem_1r1w_mask_write_bit**, **bsg_mem_1r1w_mask_write_var**, **bsg_mem_1r1w_sync_synth**, **bsg_mem_1r1w_synth**, **bsg_mem_1r1w_sync**:
1-read port, 1-write port memory.

(16) **bsg_mem_1rw_sync**, **bsg_mem_1rw_sync_mask_write_bit**, **bsg_mem_1rw_sync_mask_write_byte**, **bsg_mem_1rw_sync_mask_write_byte_synth**, **bsg_mem_1rw_sync_mask_write_var**, **bsg_mem_1rw_sync_synth**:
1-read-or-write port memory.

(17) **bsg_mem_2r1w**, **bsg_mem_2r1w_sync**, **bsg_mem_2r1w_sync_synth**, **bsg_mem_2r1w_sync**:
2-read port, 1-write port memory.

(18) **bsg_mem_3r1w**:
3 read-port, 1 write-port memory.

(19) **bsg_mem_banked_crossbar**:
Variable banked memory with crossbar on each side.

(20) **bsg_mem_multiport**:
Flexible # of ports memory.

(21) **bsg_ascii_to_rom.py**:
Converts from ASCII to SystemVerilog case statement.

(22) **bsg_fifo_1r1w_large**, **bsg_fifo_1r1w_large_banked**, **bsg_fifo_1r1w_narrowed**, **bsg_fifo_1r1w_pseudo_large**, **bsg_fifo_1r1w_small**, **bsg_fifo_1r1w_small_credit_on_input**, **bsg_fifo_1rw_large**:
FIFOs.

(23) **bsg_two_fifo, bsg_relay_fifo, bsg_fifo_shift_datapath, bsg_fifo_tracker, bsg_fifo_shift_datapath**:
More FIFOs and components.

(24) **bsg_channel_tunnel**:
Virtualize many ready/valid streams over a single ready/valid stream. Use to tunnel multiple streams of traffic over a NOC, or off-chip.

(25) **bsg_round_robin_n_to_1, bsg_round_robin_1_to_n, bsg_round_robin_fifo_to_fifo**:
Channel multiplexing, demultiplexing, data redistribution.

(26) **bsg_serial_in_parallel_out, bsg_parallel_in_serial_out**:
Widening / Narrowing logic.

(27) **bsg_hypoteneuse**:
CORDIC-based Euclidean distance calculator.

(28) **bsg_adder_cin, bsg_popcount, bsg_rotate_right**:
Combinational operators.

(29) **bsg_mux_bitwise, bsg_muxi_bitwise** :
Bitwise muxing.

(30) **bsg_mul, bsg_mul_pipelined**:
Signed / Unsigned, Pipelined / Unpipelined Radix-4 Booth-encoded Multiplier.

(31) **bsg_circular_ptr, bsg_counter_clear_up, bsg_counter_clock_downsample, bsg_counter_en_overflow, bsg_counter_up_down, bsg_counter_w_overflow, bsg_wait_after_reset, bsg_wait_cycles**:
Counters.

(32) **bsg_lfsr**:
Variable size LFSR.

(33) **bsg_and, bsg_inv, bsg_nor2, bsg_nor3,**

(34) bsg_nand, bsg_xnor, bsg_xor, bsg_clkbuf:
Hardened gatestacks for placement in ICC or Cadence.

7 CONCLUSIONS

Our initial version of BaseJump STL has been released to the public, and we are looking to expand the breadth of hardware modules implemented, and also the performance, specialization and flexibility of the library. Our eventual hope is to, with the help of others, incorporate an evolved version of the library into the SystemVerilog language standard.

We encourage you to download the code from `http://bjump.org/stl`, and welcome your additions to the open source project!

ACKNOWLEDGMENTS

We thank the members of the Bespoke Silicon Group, who have used BaseJump STL to design four chips from 180nm down to 16nm, and including a 511-core RISC-V chip [1], demonstrating its usability and portability. We thank the developers of our BaseJump FPGA Firmware, who exercised BaseJump STL for Virtex-6, Spartan-6, and Zynq FPGAs.

We thank Pulkit Bhatnagar, Bandhav Veluri, Scott Davidson, Shaolin Xie, Luis Vega, Chun Zhao, Paul Gao, Tommy Jung, Moein Khazraee, Richard Park, Christopher Torng, Shengye Wang, and Lu Zhang who contributed to BaseJump STL.

We thank Linton Salmon, the DARPA CRAFT Program, and the members of the Michigan/UCSD/Cornell CERTUS/Celerity team, including Christopher Batten, Ronald G. Dreslinski, Rajesh K. Gupta, Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Ian Galton, Patrick P. Mercier, Mani Srivastava, and Zhiru Zhang.

We thank also our chip design friends on the RISC-V [2] and Chisel [3] projects, including Krste Asanovic, Yunsup Lee, Andrew Waterman, Jonathan Bachrach, Borivoje Nikolic and the UC Berkeley team.

We thank our UW EE 477 Winter 2018 class, that built 8 unique accelerator-based chips, from RTL to GDS, in 90nm, using BaseJump STL, in 5 weeks.

We thank Stephen Twigg, Derek Lockhart and Richard Ho of Google for feedback on our latency-insensitive approach.

REFERENCES

[1] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Ian Galton, Rajesh K. Gupta, Patrick P. Mercier, Mani Srivastava, Michael Bedford Taylor, and Zhiru Zhang. 2017. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. In *HOTCHIPS*.

[2] Krste Asanović and David A. Patterson. 2014. *Instruction Sets Should Be Free: The Case For RISC-V.* Technical Report UCB/EECS-2014-146. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC*.

[4] L. P. Carloni. 2015. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proc. IEEE* 103, 11 (Nov 2015), 2133–2151.

[5] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Taylor, and Steven Swanson. 2010. GreenDroid: A Mobile Application Processor for a Future of Dark Silicon. In *HOTCHIPS*.

[6] Nathan Goulding-Hotta, Jack Sampson, Qiaoshi Zheng, Vikram Bhatt, Steven Swanson, and Michael Taylor. 2012. GreenDroid: An Architecture for the Dark Silicon Age. In *Asia and South Pacific Design Automation Conference (ASPDAC)*.

[7] Chintan Kaur, Ravi Narayanaswami, and Richard Ho. 2016. EASI2L: A Specification Format for Automated Block Interface Generation and Verification. In *DVCon2016*.

[8] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Taylor. 2017. Moonwalk: NRE Optimization in ASIC Clouds or, accelerators will use old silicon. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[9] Ikuo Magaki, Moein Khazraee, Luis Vega, and Michael Taylor. 2016. ASIC Clouds: Specializing the Datacenter. In *International Symposium on Computer Architecture (ISCA)*.

[10] Michael Taylor. 2014. A Landscape of the New Dark Silicon Design Regime. In *Design Automation and Test in Europe*.

[11] Michael Taylor. 2017. The Evolution of Bitcoin Hardware. *Computer, IEEE* (Sept-Oct. 2017).

[12] Michael B. Taylor. 2012. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference (DAC)*.

[13] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation cores: reducing the energy of mature computations. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.