BaseJump STL:

SystemVerilog Needs a Standard Template Library for Hardware Design

Michael Bedford Taylor

Bespoke Silicon Group

University of Washington

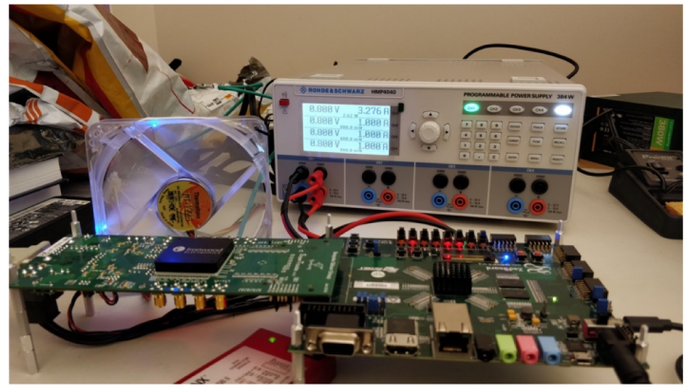
http://bjump.org/stl

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

The Celerity Open Source Tiered Accelerator Fabric

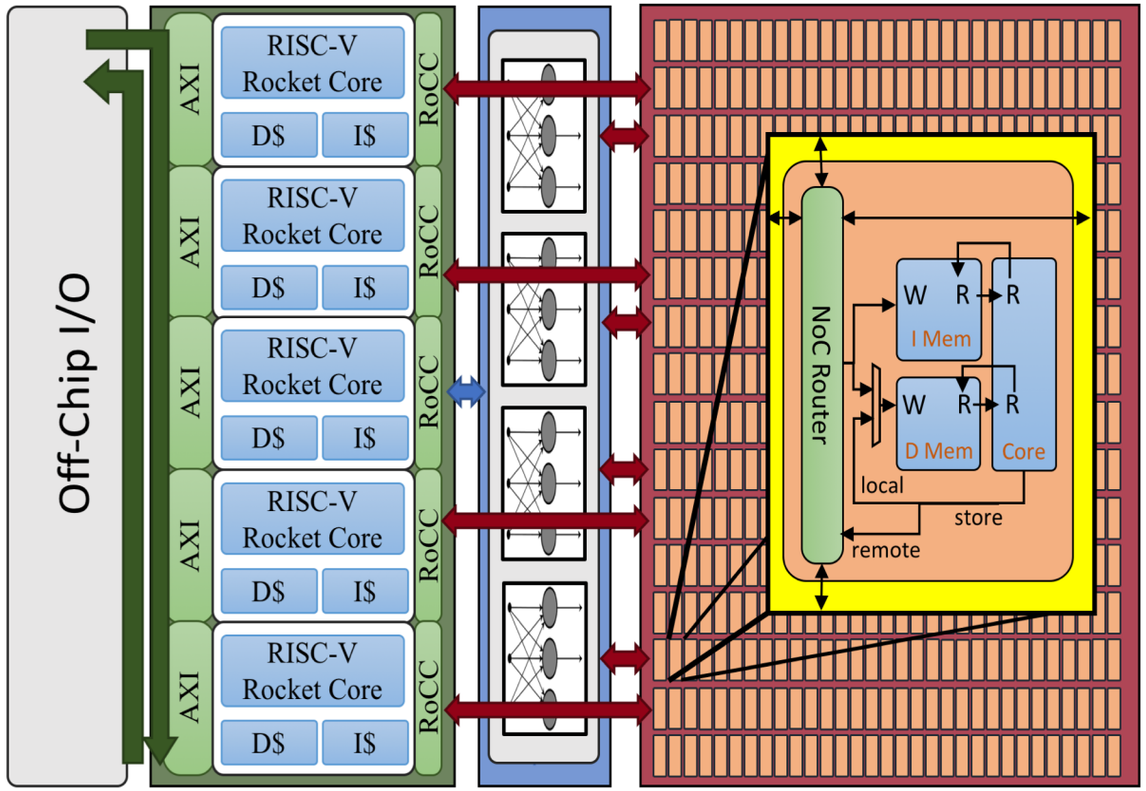
Celerity: 9-months total time to tapeout, w/ Cornell & Mich

TSMC 16FFC 5x5mm



511 RISC-V Cores

- 496-core manycore (BaseJump Manycore)
- High Speed I/O & SoC Fabric
- 10-core always on manycore (“)
- 5 Linux-capable cores (Berkeley Rocket)
- Binarized Neural Network in HLS
- 1 GHz Frequencies



[IEEE Micro, Mar/Apr 2018]

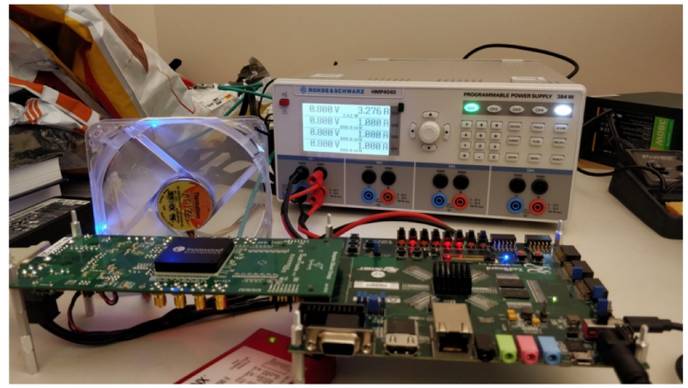
[Hotchips 2017]

DARPA CRAFT

The Celerity Open Source Tiered Accelerator Fabric

Celerity: 9-months total time to tapeout, w/ Cornell & Mich

TSMC 16FFC 5x5mm



511 RISC-V Cores

- 496-core manycore (BaseJump Manycore)
- High Speed I/O & SoC Fabric
- 10-core always on manycore (“)
- 5 Linux-capable cores (Berkeley Rocket)
- Binarized Neural Network in HLS
- 1 GHz Frequencies

500B RISC-V Instructions Per Second

World record for RISC-V performance

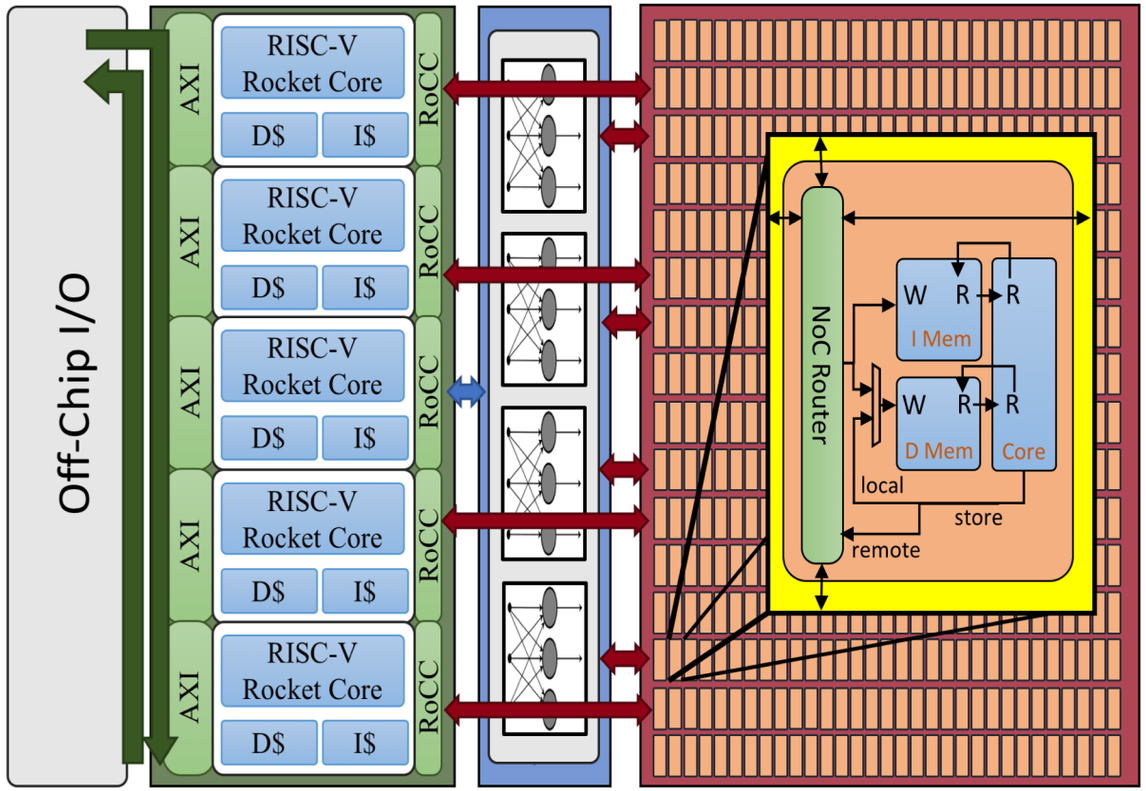
Beats prior record by 100X

100% Open Source (http://opencelerity.org)

[IEEE Micro, Mar/Apr 2018]

[Hotchips 2017]

DARPA
CRAFT



BaseJump: Open Source DNA For ASICs

Accelerating the creation of discrete accelerators

You Do This Part

HDL Design

http://bjump.org

BASEJUMP: *A Fully Open-Source ASIC Stack*

BaseJump: Open Source DNA For ASICs

Accelerating the creation of discrete accelerators

You Do This Part

HDL Design

***BaseJump STL*:** (the focus of this talk)

Standard Template Library For SystemVerilog [DAC 2018]

http://bjump.org

BASEJUMP: *A Fully Open-Source ASIC Stack*

BaseJump: Open Source DNA For ASICs

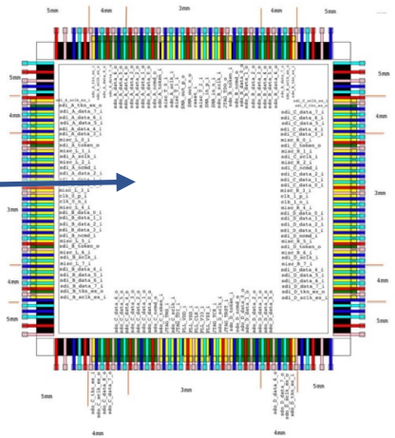
Accelerating the creation of discrete accelerators

You Do This Part

HDL Design

***BaseJump STL*:** (the focus of this talk)

Standard Template Library For SystemVerilog [DAC 2018]



***BaseJump Socket*:**
Standardized IO Padring

http://bjump.org

BASEJUMP: *A Fully Open-Source ASIC Stack*

BaseJump: Open Source DNA For ASICs

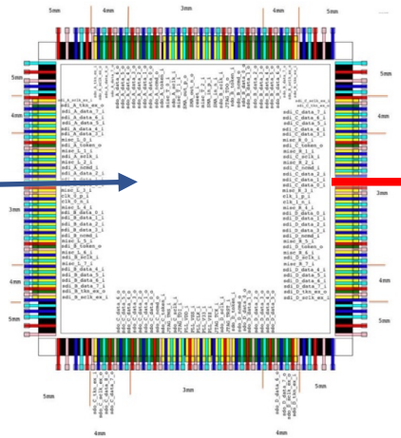
Accelerating the creation of discrete accelerators

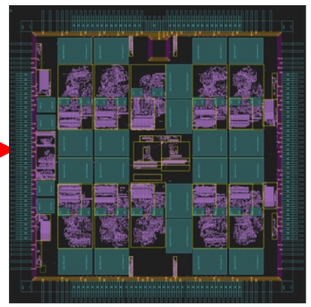
You Do This Part

HDL Design

***BaseJump STL*:** (the focus of this talk)

Standard Template Library For SystemVerilog [DAC 2018]





***BaseJump Socket*:**
Standardized IO Padring

http://bjump.org

BASEJUMP: *A Fully Open-Source ASIC Stack*

BaseJump: Open Source DNA For ASICs

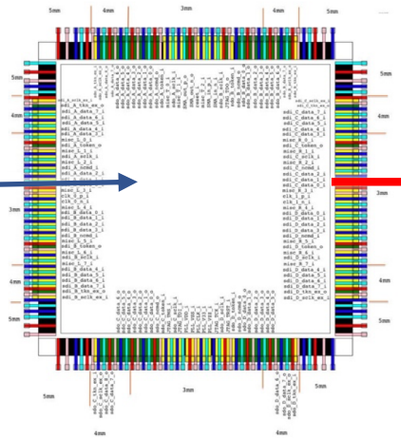
Accelerating the creation of discrete accelerators

You Do This Part

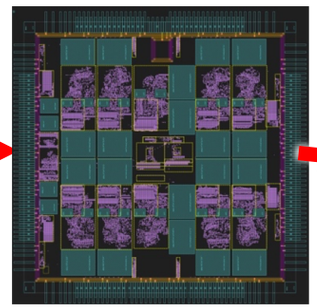
HDL Design

***BaseJump STL*:** (the focus of this talk)

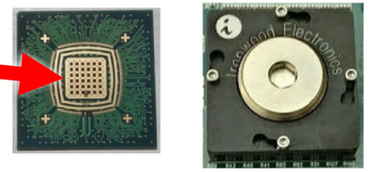
Standard Template Library For SystemVerilog [DAC 2018]



***BaseJump Socket*:**
Standardized IO Padring



***BaseJump Socket*:**
Standardized BGA Package



http://bjump.org

BASEJUMP: *A Fully Open-Source ASIC Stack*

BaseJump: Open Source DNA For ASICs

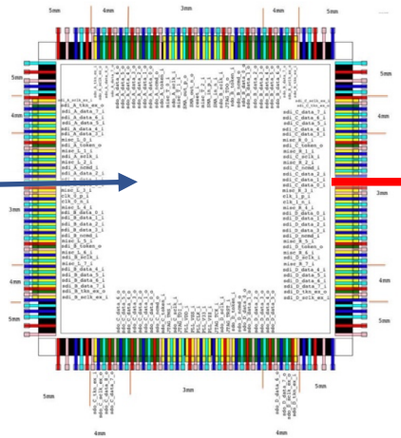
Accelerating the creation of discrete accelerators

You Do This Part

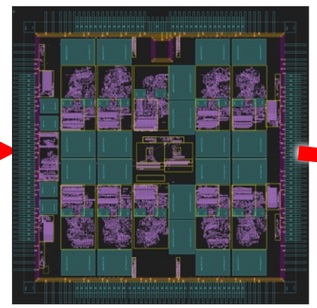
HDL Design

***BaseJump STL*:** (the focus of this talk)

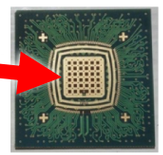
Standard Template Library For SystemVerilog [DAC 2018]



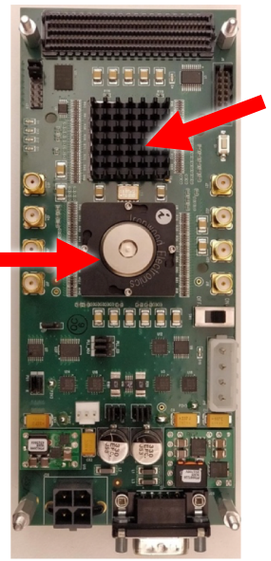
***BaseJump Socket*:**
Standardized IO Padring



***BaseJump Socket*:**
Standardized BGA Package



Click!!!



***BaseJump*:**
Open FPGA Firmware

BaseJump Motherboard

http://bjump.org

BASEJUMP: *A Fully Open-Source ASIC Stack*

BaseJump: Open Source DNA For ASICs

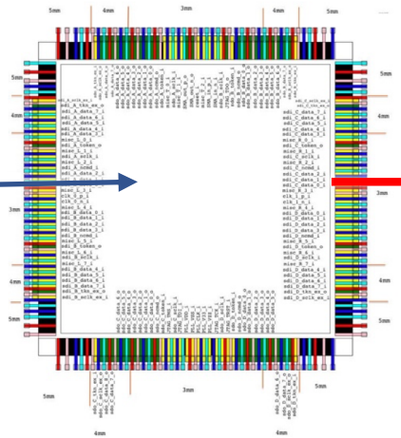
Accelerating the creation of discrete accelerators

You Do This Part

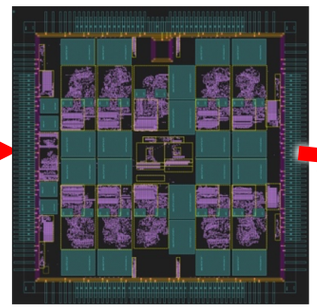
HDL Design

***BaseJump STL*:** (the focus of this talk)

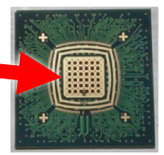
Standard Template Library For SystemVerilog [DAC 2018]



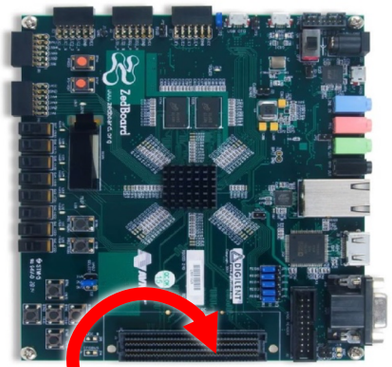
***BaseJump Socket*:**
Standardized IO Padring



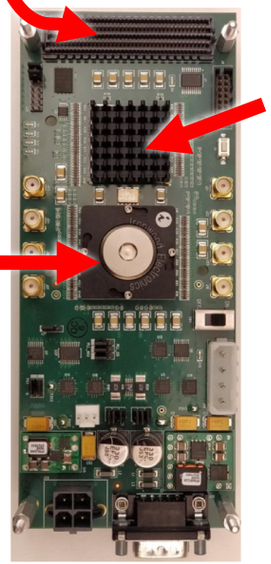
***BaseJump Socket*:**
Standardized BGA Package



ZedBoard:
Arm Core + Xilinx IP



Click!!!



***BaseJump*:**
Open FPGA Firmware

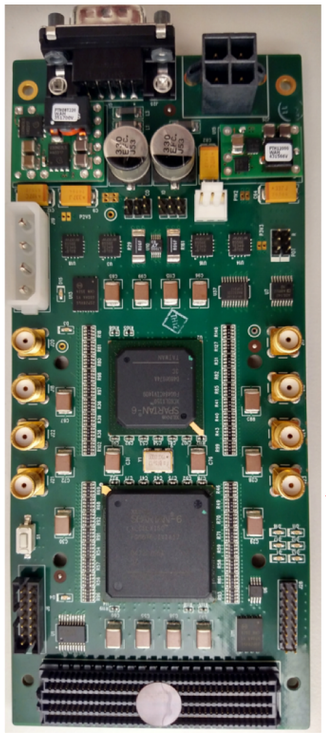
BaseJump Motherboard

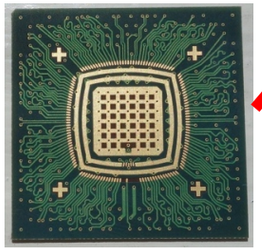
http://bjump.org

BASEJUMP: *A Fully Open-Source ASIC Stack*

bjump.org

Basejump: Early Adopters





PRINCETON UNIVERSITY

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Massachusetts Institute of Technology

(DARPA PERFECT)

bjump.org

UNIVERSITY OF CAMBRIDGE

UCSB

UCSD

NSF SaTC Med (Security)

UCSD

Ucla

DARPA CRAFT (16nm)

UNIVERSITY OF MICHIGAN

CORNELL UNIVERSITY FOUNDED A.D. 1865

UNIVERSITY of VIRGINIA

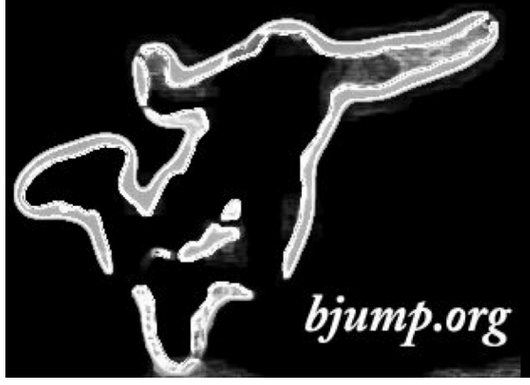
NYU TANDON SCHOOL OF ENGINEERING

NSF SaTC Large (Crypto)

BaseJump STL:

SystemVerilog Needs a Standard Template Library for Hardware Design

80% of the code for Celerity was implemented using BaseJump STL



PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Introduction: A Thought Experiment

```
If (width == 32)
   fft_32()
else if (width == 16)
   fft_16()
```

What if we translate this code into software or hardware,
But we never actually use the fft_32 functionality?

Comparing the cost of unused functionality: HW vs SW

In software, the unused code:

- occupies cheap storage
- barely affects performance, power, cost, or energy

```
If (width == 32)
   fft_32()
else if (width == 16)
   fft_16()
```

Comparing the cost of unused functionality: HW vs SW

In software, the unused code:

- occupies cheap storage
- barely affects performance, power, cost, or energy

In hardware, the unused code:

- occupies die area, routing ($)
- creates new critical paths (ns)
- has static power (W)
- dissipates dynamic energy through accidental toggles (pJ)

```
If (width == 32)
   fft_32()
else if (width == 16)
   fft_16()
```

Unused Code is Way More Costly in HW than SW!

The Need for HW Metaprogramming

Programming: Writing code to specify the functionality of computation

```
If (width == 32)
   fft_32()
else if (width == 16)
   fft_16()
```

Functionally correct even if width *is never 32.*

Metaprogramming: Writing code to specify the code for a computation

```
If (width == 32)
   fft_32()
else if (width == 16)
   fft_16()
```

If width is never 32, this is the hardware I want to generate.

Let's use the term ***HW Metaprogramming*** for this.

But Why Not Just Write What We Need?

fft_16()

Just code-to-fit what you need; the time honored tradition.

It works! But this is also why HW is in the dark ages.

Why?

But Why Not Just Write What We Need?

fft_16()

Just code-to-fit what you need; the time honored tradition.

It works! But this is also why HW is in the dark ages.

Why?

No code reuse! Everything is exactly fit to the current situation and must be re-written to the new situation.

But Why Not Just Write What We Need?

fft_16()

Just code-to-fit what you need; the time honored tradition.

It works! But this is also why HW is in the dark ages.

Why?

No code reuse! Everything is exactly fit to the current situation and must be re-written to the new situation.

In order for hardware to scale like software, we need to make code reuse the common case. We need to be able to build great libraries.

But we need metaprogramming to make those libraries efficient so that the implementations are just as efficient as code-to-fit, and there is near-zero unused HW.

Enter the Generator: Metaprogramming-First HDLs

Level 1: Verilog + vpp, Perl – *printf code*

Level 1.1: Verilog + python – *printf code*

Level 2: Stanford Genesis2

Perl Metaprogramming Composition Language

SystemVerilog leaf hardware

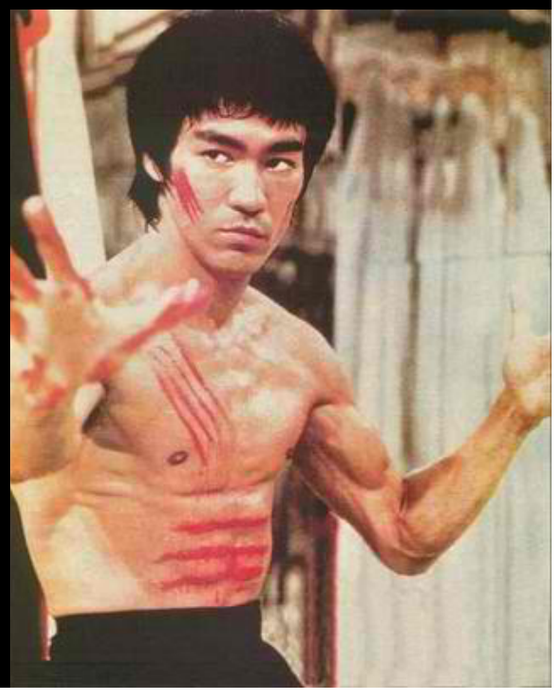
Level 3: MyHDL [Linux Journal 2004]

Python-embedded HDL

Level 3: Chisel [DAC 2012]

Scala-embedded HDL

PyMTL, Pyrope & many others



But SystemVerilog Has SuperPowers Too

Concise; no language embedding artifacts

Pervasive Tool Support for Verification and Backend Design

Active Language Development (e.g. IEEE 1800-2017)

Lots of Users; Widely Taught

Does not require functional programming language experience

How far can we push SystemVerilog for HW Metaprogramming and Library support?

Metaprogramming Support In SystemVerilog

Generate For

- Construction of variable amounts of hardware

Generate If

- Conditional Construction of Hardware

Parameter

- Construction of variable widths of hardware

Structs & Interfaces

- Named bundles of wires → same code, swap in different struct

Macros (eek!)

- Generate-by-text-manipulation; like C

Inspiration from Software

C++'s Standard Template Library (STL)

- Implements common software datastructures
- Best-of-class data structure & algo implementations specialized for:
  - your data type
  - particular input sizes
- Tight language integration
- Before this, everybody had to recode the same data structure code

Inspiration from Software

C++'s Standard Template Library (STL)

- Implements common software datastructures
- Best-of-class data structure & algo implementations specialized for:
  - your data type
  - particular input sizes
- Tight language integration
- Before this, everybody had to recode the same data structure code

If SystemVerilog supports metaprogramming, why doesn't it have a standard library for HW design?

BaseJump STL

~~C++'s~~ A Standard Template Library (STL) for System Verilog

- Implements common ~~software datastructures~~ hardware structures
- Best-of-class ~~data~~ HW structure & algo implementations specialized for:
  - your data type
  - particular input sizes
- Tight language integration
- Before this, everybody had to recode the same ~~data~~ HW structures

What else should we add?

Requirements for a HW STL

1. Portability Interface

   The STL shall provide interfaces that allow a design to be moved unchanged between ASIC process nodes and vendors, as well as to different FPGA vendors.

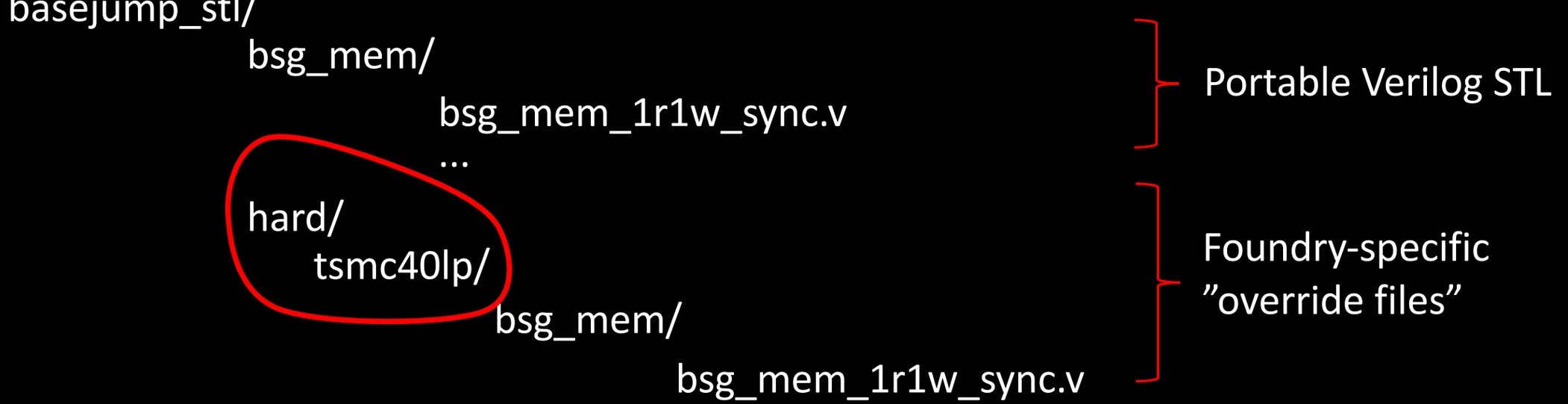
```
basejump_stl/
        bsg_mem/
                bsg_mem_1r1w_sync.v
                ...
        bsg_math/
                ...
```

Portable Verilog STL

Requirements for a HW STL

1. Portability Interface

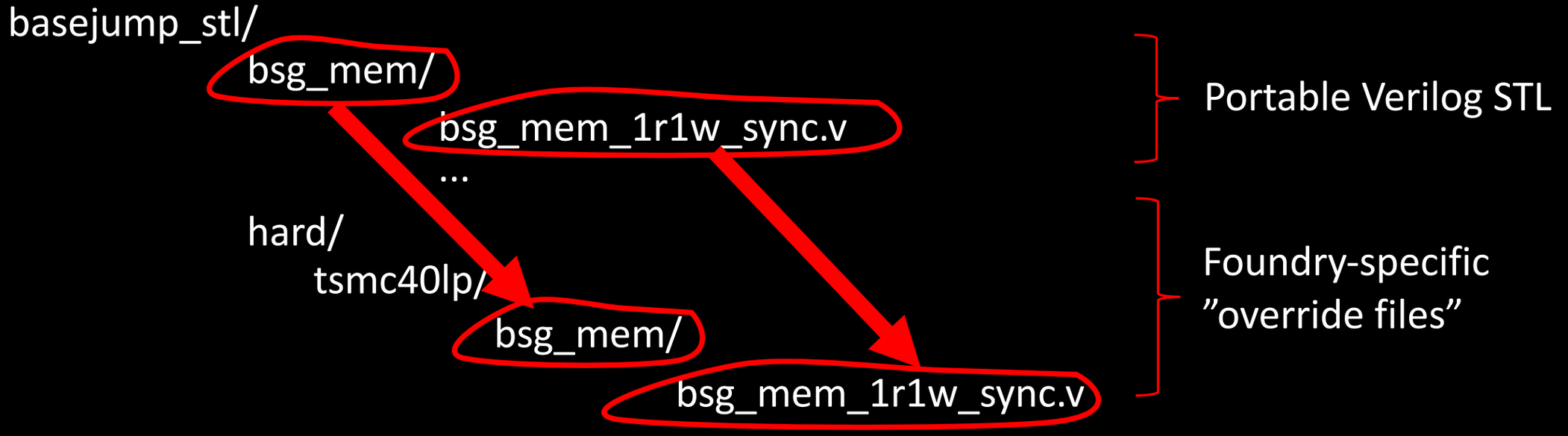
   The STL shall provide interfaces that allow a design to be moved unchanged between ASIC process nodes and vendors, as well as to different FPGA vendors.



Requirements for a HW STL

1. Portability Interface

The STL shall provide interfaces that allow a design to be moved unchanged between ASIC process nodes and vendors, as well as to different FPGA vendors.

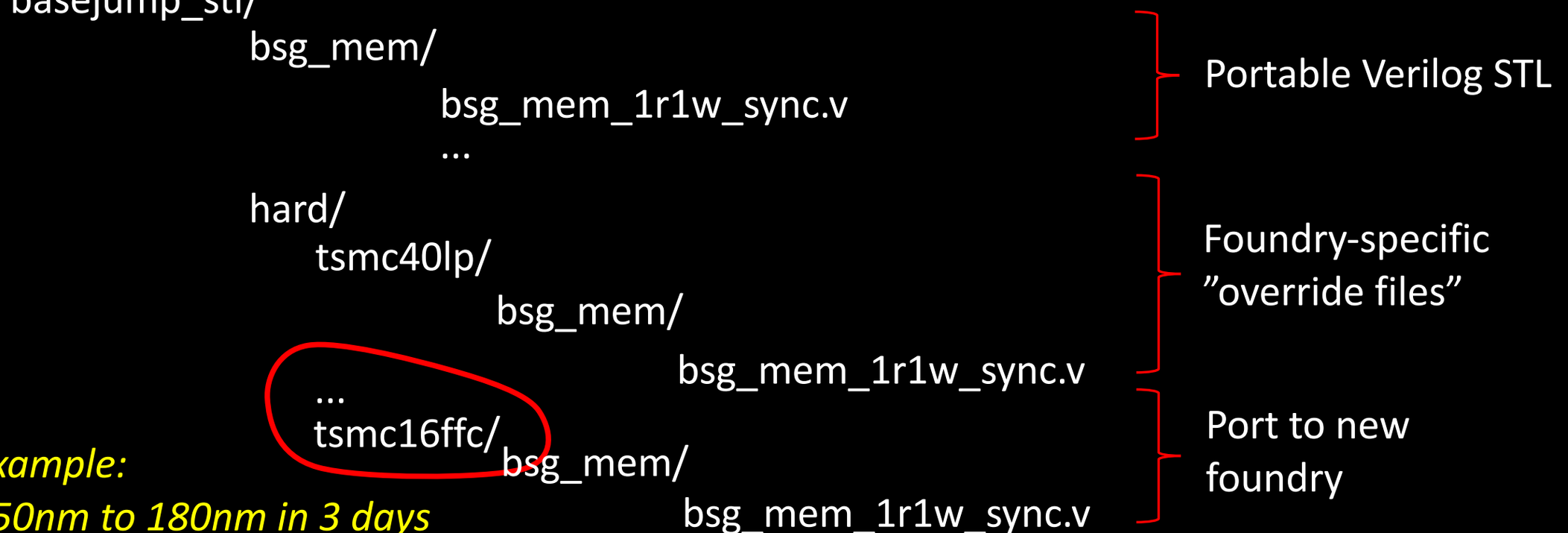


Only a subset of files need to be customized for a process.

Requirements for a HW STL

1. Portability Interface

The STL shall provide interfaces that allow a design to be moved unchanged between ASIC process nodes and vendors, as well as to different FPGA vendors.



Requirements for a HW STL

2. Portable Leaf Building Blocks (Corollary to #1)

The STL shall provide portable interfaces to leaf building blocks:

- SRAMs
- Synchronizers
- RFs
- Clock Generators
- I/Os

See the paper (or the code) for the nuances.

Requirements for a HW STL

3. Efficient Hardware Primitives

   The STL shall provide efficient implementations for all commonly used hardware primitives.

BaseJump STL Components

Package	Example HW Primitives
bsg_dataflow	FIFOs, stream mergers, round-robin arbitrators, serial-to-parallel converters
bsg_math	Floating point Add, Multiply, compare, CORDIC functions
bsg_noc	network-on-chip building blocks RISC-V interface logic
bsg_async	asynchronous fifos and interfaces
bsg_clk_gen	synthesizable digital clock generator
bsg_comm_link	high-speed I/O source-synchronous interface
bsg_fsb	front side bus (high-speed bridge between off-chip and on-chip worlds)
bsg_mem	portability layer for SRAMs
bsg_misc	popcount, flop trays, decoders, lfsr, multiplies, flexible muxes, transposers crossbars, gray_to_binary, priority encoder, thermometer encoders, counters
bsg_tag	SoC configuration interface (like SPI or JTAG)
bsg_test	Test bench blocks; reset generators, delay lines, clock gens

Several Hundred Modules, All Parameterized

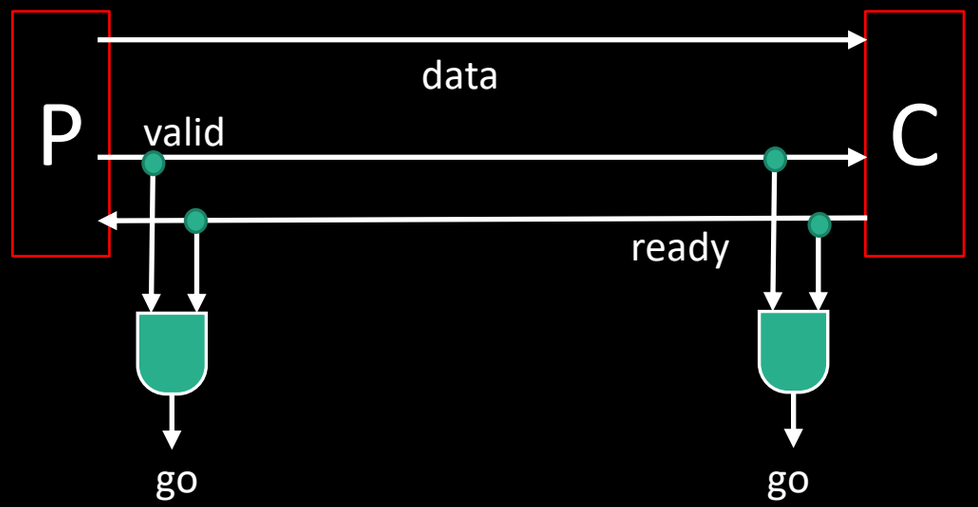
Requirements for a HW STL

4. Latency Insensitive Design

To support interfacing of modules that have internal state, the STL shall:

- provide the right set of latency-insensitive interfaces that allow hardware to be composed correctly without considering the internals of composed blocks.
  (e.g., no waveform diagrams)
- This overhead of these interfaces, in terms of area, power, or performance shall be near zero.
- The interfaces shall allow wire delay to be managed in a portable way

Typical “Valid and Ready” interface



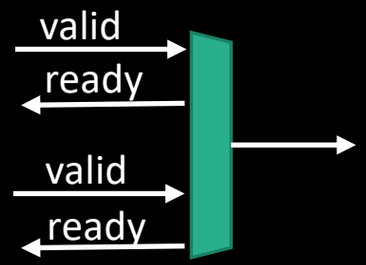
Valid and Ready cannot depend on each other.

Nice, because valid and ready signals have almost an entire cycle to traverse the distance between P and C, or to be computed internally.

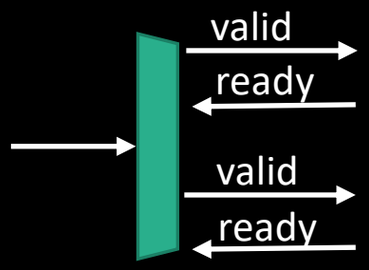
Guaranteed no combinational loops.

Good design practice for top-level, long-distance SOC connections that are almost a cycle of wire delay.

But it's too weak to describe very simple hardware primitives.



I can only take one element. How do I commit to taking an element without examining the valids?



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.

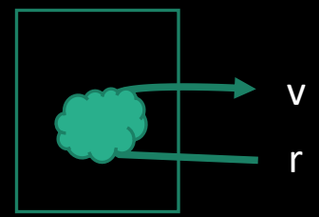
These are "demanding" interfaces because they require up-front information before asserting their output.

Two interfaces are sufficient for short-range connections; and the third for long-range connections.

ready -> valid "r->v"

producer: I **demand** r up front and then will decide v

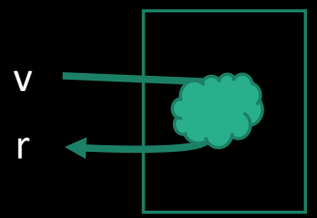
consumer: I **volunteer** r up front and will wait for v



valid -> ready "v->r"

producer: I **volunteer** v up front and wait for r

consumer: I **demand** v up front and then will decide r



valid & ready "v&r" (for long range connections)

producer: I **volunteer** v up front

consumer: I **volunteer** r up front

Pairing Producers and Consumers

Producer \ *Consumer*	Helpful	Demanding
Helpful	rv->&	v->r
Demanding	r->v	Use FIFO (universal converter)

Simple rule: **Demanding interfaces must be paired with Helpful ones.**

Given the choice, what should I make a producer or consumer interface?

As helpful as possible .... without adding new logic:

This is the most natural interface for that module.

If you have to add more logic to make the module more helpful, the module may be connecting to another helpful module, and you wasted logic.

Requirements for a HW STL

5. Parameterization

The STL primitives shall be elegantly parameterized to allow them to be reusable.

Parameters should be used to:

- capture slight variations in common modes of usage
- Improve code factoring
- Reduce bugs
- Reduce module count

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 under time constraints.

Requirements for a HW STL

6. Efficient Plumbing

   The STL should provide hardware primitive implementations that support efficient data movement.

   Primitives to support efficient, minimalist, higher-level primitives like FIFOs (all common varieties), virtual channels, credit-counters, crossbars, and network routers...

Support for Metaprogramming

7. Bug-reducing coding style

No low-true signals.

No non-synthesizable code (except for assertions and display statements)

Consistent naming:

_i – input; _o – output; _p – parameter, _r – register

width_p – standard width param name

els_p – standard elements param name

Non-synthesizable modules for testing clearly marked in module names with _nonsynth_

Requirements for a HW STL

8. Support for Composable Metaprogramming

Some support already there – see my *SystemVerilog wishlist* later in this talk!

See the paper for how we handled it without these features.

Requirements for a HW STL

9. A Testing Suite (!)

Must test all combinations of inputs and all combinations of supported parameters.

Although performance is the main draw, the true savings in the STL is that the code has already been tested and debugged, and you will save development time and NRE!*

[ASPLOS 2018, Khazraee et al.]

SystemVerilog Wishlist for Metaprogramming

1. Synthesis support for arrays of interfaces
   Currently have to split up interfaces into arrays of structs

SystemVerilog Wishlist for Metaprogramming

1. Synthesis support for arrays of interfaces

   Currently have to split up interfaces into arrays of structs

2. Tolerance for Zero-width signals

   With els_p as an input param, when els_p = 1, the width of a pointer is 0 bits.

SystemVerilog Wishlist for Metaprogramming

1. Synthesis support for arrays of interfaces

   Currently have to split up interfaces into arrays of structs

2. Tolerance for Zero-width signals

   With els_p as an input param, when els_p = 1, the width of a pointer is 0 bits.

3. Make Default Parameters Optional

   Often there is no good default; we want to force user to specify it.

SystemVerilog Wishlist for Metaprogramming

1. Synthesis support for arrays of interfaces

   Currently have to split up interfaces into arrays of structs

2. Tolerance for Zero-width signals

   With els_p as an input param, when els_p = 1, the width of a pointer is 0 bits.

3. Make Default Parameters Optional

   Often there is no good default; we want to force user to specify it.

4. True type polymorphism

   Pass struct or type as a parameter

SystemVerilog Wishlist for Metaprogramming

5. Declaration of bit widths with [0+:width_p] notation

SystemVerilog Wishlist for Metaprogramming

5. Declaration of bit widths with [0+:width_p] notation
6. Simple Bit Width Inference

   With an easy way to see inferred widths

SystemVerilog Wishlist for Metaprogramming

5. Declaration of bit widths with [0+:width_p] notation
6. Simple Bit Width Inference

   With an easy way to see inferred widths
7. Better Generate Statement Debugging

   i.e. a preprocessor option to see expanded version.

SystemVerilog Wishlist for Metaprogramming

5. Declaration of bit widths with [0+:width_p] notation
6. Simple Bit Width Inference

   With an easy way to see inferred widths
7. Better Generate Statement Debugging

   i.e. a preprocessor option to see expanded version.
8. Language Construct to indicate signal is unused

   Allows uniform interfaces to leaf blocks without tool complaining.

   e.g. some memories may require a reset line and some may not.

Future Steps

Can the EDA Community move forward with the standardization of an STL for SystemVerilog?

Thanks! `http://bjump.org/stl`

We welcome you to use and improve our STL prototype!

Special Thanks:

BaseJump STL Users

(40+ users, 4 tape-outs, 40+ tape-ins, 2 FPGAs)

Linton Salmon & CRAFT Program

Berkeley - Chisel & RISC-V

Celerity Team (Cornell, Michigan, UCSD, UW)

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.