



# BSV Training

Our training session is structured around a series of complete, runnable examples.  
(in the Example\_Programs/ directory)

For each example, we study the code, build it, run it, and analyze it.

卷之三

## Module 6: Layout & Design

第16课

- ASW

四百四十一

## Conclusion

五

Page 10

Digitized by srujanika@gmail.com

四

www.FT.com

Unit 10

Digitized by srujanika@gmail.com

FORMS

• 100

inbound lead

• 100 •

[www.mathsrevision.com](http://www.mathsrevision.com)

BSV constructs in the code, we'll take excursions into understand them in more detail and in more generality.

(in the Reference/ directory)

[www.bluespec.com](http://www.bluespec.com)

These training materials (examples, lecture slides)  
are available at:

[https://github.com/rsnikhil/Bluespec\\_BSV\\_Tutorial.git](https://github.com/rsnikhil/Bluespec_BSV_Tutorial.git)

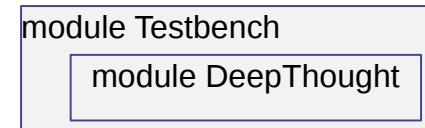
Please download a copy for yourself.

# Introduction

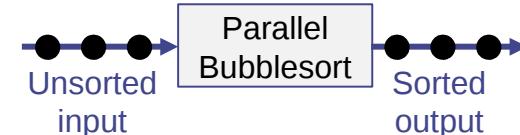
- BSV is a modern Hardware Design Language (HDL), in use since about 2000 in many major companies and universities worldwide
- BSV embodies a fundamental rethink of what an HDL should be:
  - Circuit generation, not just circuit description, employing the full power of a modern functional programming language (Haskell-like expressivity and types)
  - Atomic Transactional Rules as the fundamental behavioral abstraction, instead of synchronous clocks (scalable, compositional)
- BSV is not like classical “HLS” (High Level Synthesis), where the source language (C/C++) has a quite different computation model (and algorithmic cost structure) from the target (hardware)
  - BSV is architecturally transparent: you are in full control of architecture and there are no architectural surprises
  - With BSV you think hardware, you think about architectures, you think in parallel
- BSV is “universal” in applicability (like traditional HDLs). BSV has been used for CPUs, caches, coherence engines, DMAs, interconnects, memory controllers, DMA engines, I/O devices, security devices, RF and multimedia signal processing, and all kinds of accelerators.

# Overview of examples

(2a-2c) Basic look-and-feel, tool usage  
("Hello World")



(3a-3e) Basic concurrency and modularity  
(Sort 5 integers; generalize to sort 'n' items of type 't')



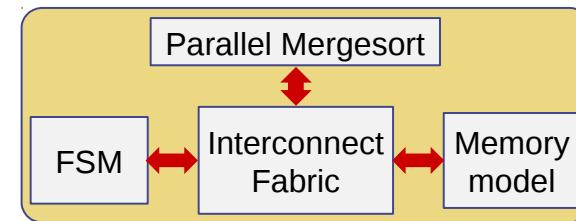
(4a-4b) Microarchitectures: FSMs and Pipelines  
(iterative computation, rigid pipelines, elastic pipelines)



(5a-5b) Greater concurrency with CRegs  
(Pipeline and Bypass FIFOs)



(6a-6c) Parallel Mergesort  
(Standalone IP; generalize to accelerator in SoC)



(9a) Interfacing with existing HW and protocols



(Non-consecutive example numbers are due to correspondence with forthcoming book chapters)

Note: all BSV code is synthesizable to FPGA/ASIC

# A little more detail on the examples

## Eg02\_HelloWorld:

- Eg02a\_HelloWorld/ Simple "Hello World" program as a single module
- Eg02b\_HelloWorld/ Splits into two separately compiled modules, Testbench and DeepThought
- Eg02c\_HelloWorld/ Adds some state-machine functionality to DeepThought

## Eg03\_Bubblesort:

- Eg03a\_Bubblesort/ sequential sort of 5 values, each of type Int #(32)
- Eg03b\_Bubblesort/ parallel sort, using 'maxBound' special value
- Eg03c\_Bubblesort/ generalize '5' to 'n'
- Eg03d\_Bubblesort/ generalize 'Int #(32)' to 't'
- Eg03e\_Bubblesort/ uses 'Maybe' types instead of 'maxBound' to represent +infinity

## Eg04\_MicroArchs:

- Eg04a\_MicroArchs/
  - Iterative shifter: Bit #(8) << Bit #(3)
  - Rigid pipelined shifter: Bit #(8) << Bit #(3)
  - Elastic pipelined shifter: Bit #(8) << Bit #(3)
- Eg04b\_MicroArchs/ generalizes '8' to 'n'
  - Iterative shifter: Bit #(n) << Bit #(TLog#(n))
  - Rigid pipelined shifter: Bit #(n) << Bit #(TLog#(n))
  - Elastic pipelined shifter: Bit #(n) << Bit #(TLog#(n))

## Eg05\_CRegs\_Greater\_Concurrency:

- Eg05a\_CRegs\_Greater\_Concurrency/ Example with FIFOs
- Eg05b\_CRegs\_Greater\_Concurrency/ Example with Up/Down counters

## Eg06\_Mergesort:

- Eg06a\_Mergesort/ System: mkConnection (mergesort, mem); 1 merge engine, 1 mem port)
- Eg06b\_Mergesort/ System: SoC; same mergesort module (1 merge engine, 1 mem port)
- Eg06c\_Mergesort/ System: SoC; n merge engines, n mem ports; reorder buffer

## Eg09a\_AXI4\_Stream/

- Simple example showing interfacing BSV to existing buses

# Training plan

These slides (“START\_HERE.pdf”). It describes the main thread of the training, which is to go through a series of complete, working examples, which are all provided in the directory “Example\_Programs/”. All examples are executable and synthesizable.

Please go through Examples serially: for each example (e.g., Example 1),

- Go through the corresponding slide deck: Eg02\_HelloWorld.pdf  
It will explain the example, with its possibly multiple versions.
- It will take you through a tour of each version, and show you how to build and run it. In a classroom setting, we study and discuss the source code extensively.
  - We show how to build and run the examples in both Bluesim and Verilog simulation, using command-line Makefiles. Some examples also demonstrate the use of BDW (the Bluespec Development Workstation GUI interface).
  - With both Bluesim and Verilog simulation, we generate VCD waveform files, view the waveforms, and analyze them.

As we go through the examples, we reinforce concepts by looking at the topic-based lectures slides called “Lec\_topic.pdf” in the “Reference/” directory.

# Lecture slide decks reading guide

The topic-based lecture slide decks in the “Reference/” directory are intended as a reference, and need not be read sequentially.

However, people learning BSV on their own for the first time may wish to read them in the following order:

- Lec01\_Intro  
General intro to the Bluespec approach, and some comparisons to other Hardware Design Languages and High Level Synthesis.
- Lec02\_Basic\_Syntax  
Gets you familiar with the “look and feel” of BSV code.
- Lec03\_Rule\_Semantics, Lec04\_CRegs  
These two lectures describe BSV’s concurrency and parallelism semantics (based on rules and methods). This is the KEY feature distinguishing BSV from other hardware and software languages.
- Lec05\_Interfaces\_TLM, Lec06\_StmtFSM  
These two lectures describe slightly advanced constructs: more abstract interfaces, and more abstract rule-based processes.
- Lec07\_Types, Lec08\_Typeclasses  
These two lectures describe BSV’s type system, which is essentially identical to that of the Haskell functional programming language.
- Lec09\_BSV\_to\_Verilog  
Describes how BSV is translated into Verilog by the bsc tool. Read this only if you are curious about this, or if you need to interface to other existing RTL modules.
- Lec10\_Interop\_RTL  
How to import Verilog/VHDL code into BSV, and how to connect BSV into existing Verilog/VHDL.
- Lec11\_Interop\_C  
How to import C code into BSV (for simulation only). How to export a BSV subsystem as a SystemC module (for use in a SystemC program).
- Lec12\_Multiple\_Clock\_Domains  
How to create BSV designs that use multiple clocks or resets.
- Lec13\_RWires  
Some facilities typically used in interfacing to external RTL. These are similar in spirit to CRegs, but lower level.

# Orientation around the training materials

All the training materials are provided to you inside a single top-level directory.

Inside this directory, you will see sub-directories and files. Briefly:

<i>File/directory</i>	<i>Comments</i>
START_HERE.pdf	This slide deck
Reference/Lec_*.pdf	Lecture slide decks, by topic
Example_Programs/	Sub-directory
.../Common/	Common codes used by many/most of the examples
.../Eg02_HelloWorld.pdf	Slides for Eg02
.../Eg02a_HelloWorld/	sub-directory for Eg02a code
.../Eg02b_HelloWorld/	sub-directory for Eg02b code
<i>... similarly, other examples ...</i>	<i>... similarly, other examples ...</i>

# Orientation around each example sub-directory

Each example sub-directory (such as Eg03a\_Bubblesort/ ) typically contains:

<i>File/directory</i>	<i>Comments</i>
.../src_BSV/	Sub-directory containing BSV source files
.../dump.vcd	File with waveform data, created during Bluesim and/or Verilog simulation. Can be viewed in any waveform viewer
.../Waves.tiff	Screenshot of waveform viewer displaying waves

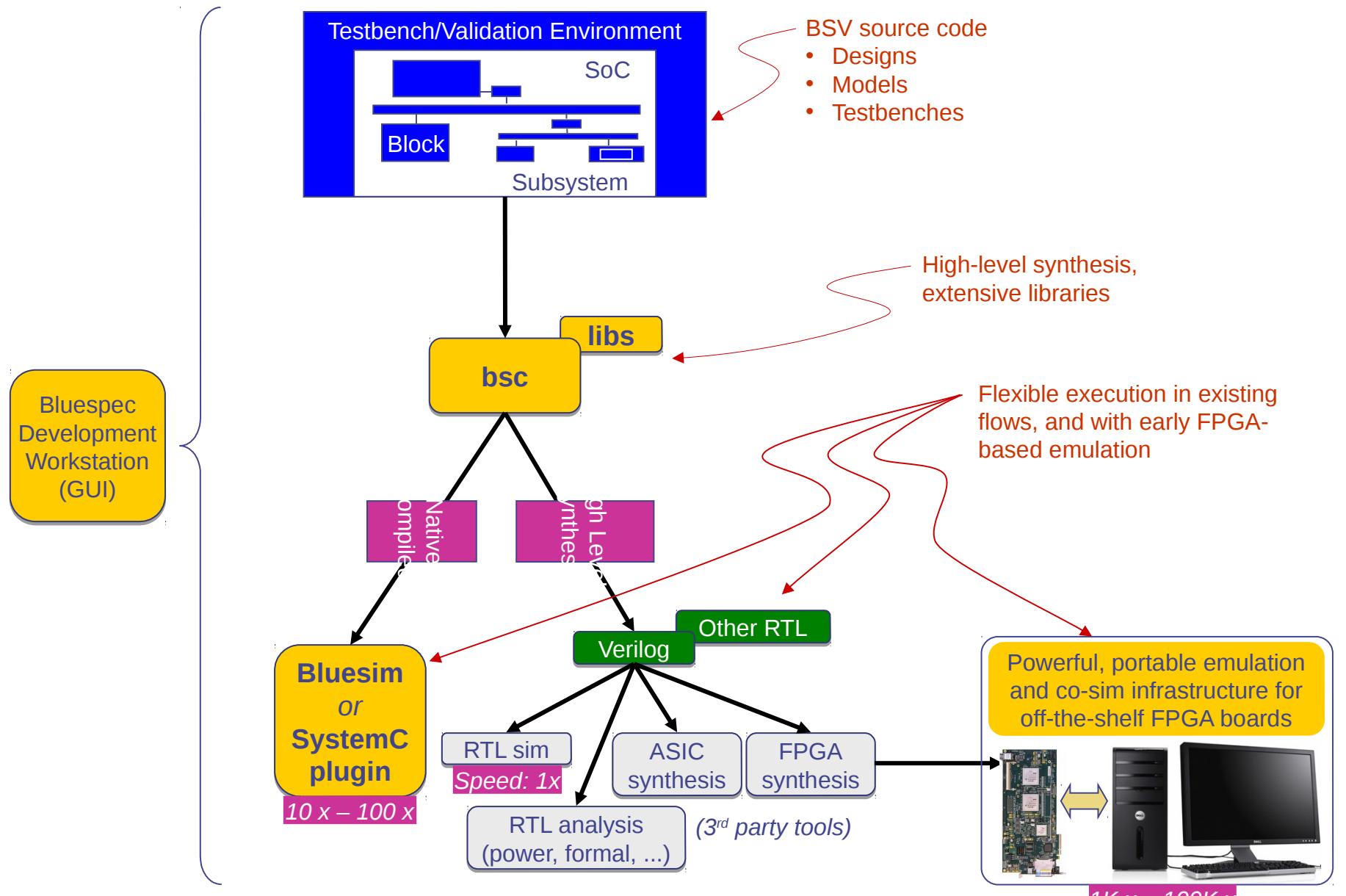
# Cheat-sheet to build and run the example programs

## To build and run from ‘make’ commands on the command line:

- ‘cd’ to the Build/ directory, which contains a Makefile
  - (You may wish to make a copy of the Build directory for each example you build)
  - Edit the top few lines of the Makefile to specify which example you are building, the location of your source files (distribution accompanying this training), and your Verilog simulator
- ‘make compile’, ‘make link’, and ‘make simulate’ to build and run in Bluesim
  - The created executable is usually called ‘out’ (with an ‘out.so’ shared object)
- ‘make v\_compile’ to generate Verilog files in the ‘verilog\_dir/’ directory
- If you have a Verilog simulator installed, then:
  - ‘make v\_link’, and ‘make v\_simulate’ to build and run in Verilog simulation
- ‘make clean’ and ‘make full\_clean’ to clean up directories

Full details on the commands in the Makefiles, and also on using the BDW GUI (Bluespec Development Workstation) are given in the Bluespec User Guide (see “Resources” slide later in this slide deck).

# Bluespec HLS tools and flow



# What you'll learn about BSV during these exercises

By going through these examples, and with excursions into lectures for deeper discussion, you'll learn about most of the BSV language and its capabilities:

- BSV modules, interfaces, module hierarchies, interfaces
- The core semantics of BSV:
  - Rules, methods, and rule concurrency, scheduling
    - Tighter concurrency using CRegs and RWires
    - Parallel (“instantaneous”) Actions within rules/methods
- Types: structured types, polymorphism, strong typing
- User-extensible overloading: typeclasses, instances, provisos
- Numeric types and constraints using typeclasses to establish relationships between sizes of components
- Importing C (for Bluesim and Verilog simulation)
- Interfacing with existing hardware (Verilog, VHDL, etc.)
- BSV packages and separate compilation

Things we are not planning to cover during this training (please ask for separate sessions, if you wish):

- Higher-order parameterization: parameterizing functions and modules with other functions and modules
- Multiple clock domains
- Facilities for quick deployment and debugging on FPGAs, controlled by host software

# Resources

- These examples and lecture slides
  - Also, the `$BLUESPEC_HOME/training/BSV` directory has more examples, tutorials, and labs
- Language reference guide: `$BLUESPEC_HOME/doc/BSV/reference-guide.pdf`
  - Complete reference on the BSV language (syntax, semantics, all language constructs, scheduling annotations, importing C and Verilog, extensive libraries)
- Tool usage guide: `$BLUESPEC_HOME/doc/BSV/user-guide.pdf`
  - How to use BDW (Bluespec Development Workstation), how to compile and link using `bsc`, how to simulate using Bluesim and Verilog sim, how to generate and view waveforms, etc.
- BSV-by-Example book (authors: Nikhil and Czeck):
  - Around 60 examples, each focusing on one topic, with ready-to-run source code
  - Hardcopy version: purchase at Amazon.com
  - Free PDF of book: `$BLUESPEC_HOME/doc/BSV/bsv_by_example.pdf`
  - All the example code: `$BLUESPEC_HOME/doc/BSV/bsv_by_example_appendix.tar.gz`
- General questions about BSV, the tools, anything:
  - User Forums at [bluespec.com](http://bluespec.com) (free, after registration)
  - E-mail to '[support@bluespec.com](mailto:support@bluespec.com)'

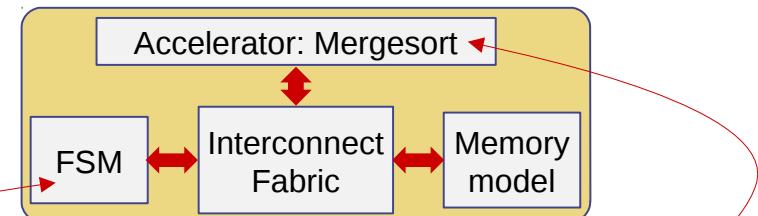
# Resources: Follow-on Examples

We can provide some follow-on larger examples for later self-study (they are not covered during the training itself). They build on the SoC structure of Example 6:

## (6a-6c) Parallel Mergesort

(Standalone IP; generalize to accelerator in SoC)

(20) CPU: replace the *FSM* by a simple implementation of a complete CPU  
(Berkeley RISC-V instruction set)



(30) Accelerator: Vector Add

(31) Accelerator: Vector Inner Product

(32) Accelerator: Matrix Multiplication

(33) Accelerator: Blocked Matrix Multiplication

*Note: all BSV code is synthesizable to FPGA/ASIC*

# 3<sup>rd</sup>-party Resources

*Note: Bluespec has no official relationship with the organizations mentioned below (other than providing Bluespec tools through its standard University program), and is not responsible for the content posted by them. These links are provided here for information only, for your convenience. Please contact these organizations directly with any questions about this content.*

- MIT Courseware: “Complex Digital Systems”
  - FPGA project-oriented digital design course
  - Courseware (lectures, labs, ...): [http://csg.csail.mit.edu/6.375/6\\_375\\_2013-www/index.html](http://csg.csail.mit.edu/6.375/6_375_2013-www/index.html)
- MIT Courseware: “Computer Architecture: A Constructive Approach”
  - Teaching processor architectures with executable BSV code
  - Courseware (lectures, labs, ...): <http://csg.csail.mit.edu/6.S195/index.html>
- Univ. of Cambridge (UK) BSV examples (Prof. Simon Moore)
  - <http://www.cl.cam.ac.uk/~swm11/examples/bluespec/>



End

```

import POF3d;

typedef! Evt3d(Evt3d) Evt3d;

model ex_Jet_cenR_lowphi;

Integer file_depth = 15;

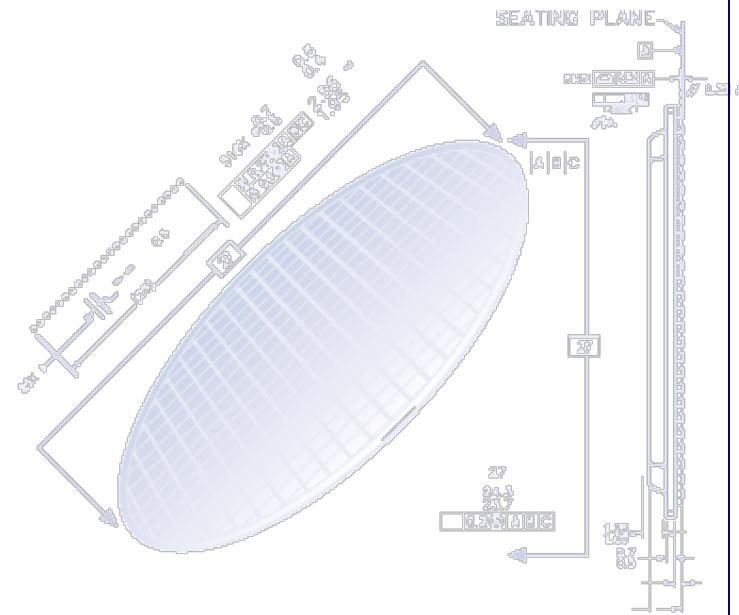
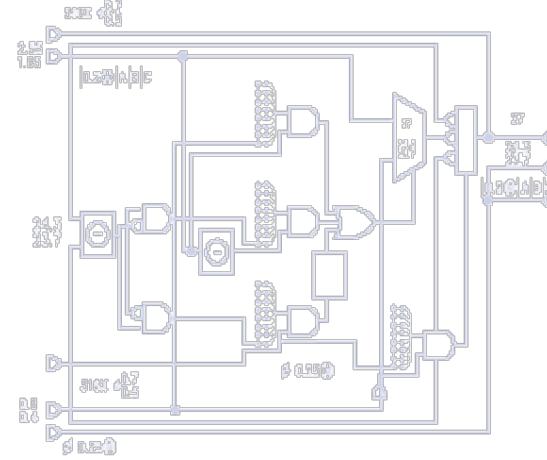
function Int3d determine_gauge(Evt3d);
  return 0;
endfunction

POF3d(Evt3d) mbound3d;
mbound3d.PT_mb3d(momentum) the_mb3d(momentum);
POF3d(Evt3d) mbound4d;
mbound4d.PT_mb4d(momentum) the_mb4d(momentum);
POF3d(Evt3d) mbound5d;
mbound5d.PT_mb5d(momentum) the_mb5d(momentum);

rule end (rule);
  Evt3d h_jet = mbound3d;
  POF3d(phi) cut_phi = 0;
  determine_gauge(phi) == 0 ? mbound1 : mbound2;
  mbound1.PT_mb1(phi) <= phi;
  mbound2.PT_mb2(phi) >= phi;
  mbound3d = 0;
endrule

mbound1 : ex_Jet_cenR_lowphi;

```





# BSV Training

## Lec\_Intro

Context-setting overview: A word about Bluespec, Inc., the company. BSV's approach to raising the level of abstraction as an HDL (Hardware Design Language). Comparison with other approaches. BSV use models (modeling, design, verification, prototyping, virtual platforms, etc.). Overview of tool flow. Links to more training resources.



These training materials (examples, lecture slides)  
are available at:

[https://github.com/rsnikhil/Bluespec\\_BSV\\_Tutorial.git](https://github.com/rsnikhil/Bluespec_BSV_Tutorial.git)

Please download a copy for yourself.

# Bluespec, Inc. company overview

- Founded in 2003
  - HQ and engineering in Framingham, Massachusetts; worldwide presence
  - Patented technology: proven, mature, and shipping since 2005
    - Technology roots in MIT research (atomic rule synthesis) and Haskell, a modern functional programming language (for types, parameterization, static elaboration)

# Customer examples

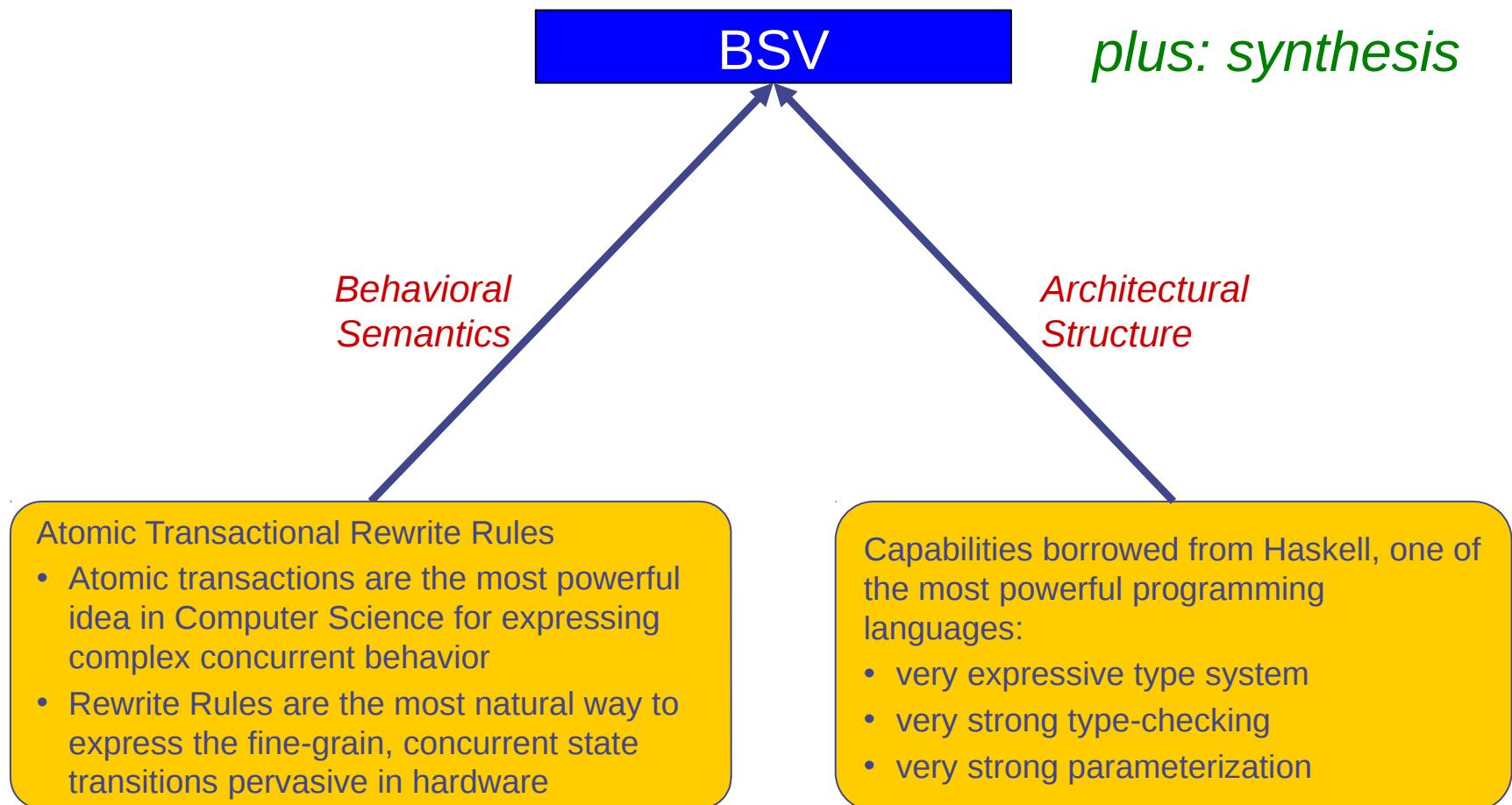


And,  
many  
more!

# Vibrant research partnerships



# BSV: based on advanced Computer Science innovations



# BSV: fundamentally different approach to HW design

## Structural expressiveness:

- Unlike most HDLs (other than Lava and Chisel), BSV is based on *circuit generation* rather than merely *circuit description*.
- 'Generate' is not an afterthought but an organic part of the language.
- 'Generate' is powerful—a full higher-order functional language with very powerful parameterization (~ Haskell)
- Expressive polymorphic types with strong static type-checking (~ Haskell)

## Behavioral expressiveness:

- Unlike other HDLs (even Lava and Chisel), BSV is based on *atomic transactional rules* instead of a globally synchronous view of the world
- Atomic Transactional Rules
  - “Event centric”, “Reactive”, “elastic”, “Method-based module communication”
  - Almost ``asynchronous'' in mindset (even though compiled to synchronous logic)
  - Scalable reasoning, even across module boundaries
  - Compositional
  - Amenable to formal reasoning, proofs
- vs. Purely Synchronous view:
  - “State centric”, signal-based module communication
  - Globally synchronous reasoning: difficult to scale robustly, fragile composition

- BSV is not like classical “HLS” (High Level Synthesis), where the source language (C/C++) has a quite different computation model (and algorithmic cost structure) from the target (hardware)
  - BSV is architecturally transparent: you are in full control of architecture and there are no architectural surprises
  - With BSV you think hardware, you think about architectures, you think in parallel
- BSV is “universal” in applicability (like traditional HDLs). BSV has been used for CPUs, caches, coherence engines, DMAs, interconnects, memory controllers, DMA engines, I/O devices, security devices, RF and multimedia signal processing, and all kinds of accelerators.
  - Since 2000, in several major companies and universities worldwide

# Learning effort is comparable to other modern languages

The screenshot shows the Amazon product page for 'BSV by Example' (Paperback). At the top, there's a greeting for 'Hello, Rishiyur S. Nikhil.' and links for recommendations, Today's Deals, Gifts & Wish Lists, and Gift Cards. The main navigation bar includes 'Shop All Departments', 'Search Books', 'Books', 'Advanced Search', 'Browse Subjects', 'New Releases', and 'Bestsellers'. A red arrow points to the 'Click to LOOK INSIDE!' button next to the book cover. The book cover features a yellow background with a stylized microchip design. The title 'BSV BY EXAMPLE' is prominently displayed. Below the cover, it says 'RISHIYUR S NIKHIL AND KATHY R CZECK'. The product details include: 'BSV by Example [Paperback]', 'Rishiyur S Nikhil (Author), Kathy R Czeck (Author)', 'Be the first to review this item | Like (0)', 'Price: \$26.00 & this item ships for FREE with', 'In Stock.', 'Ships from and sold by Amazon.com. Gift-wrap available.', and 'Want it delivered Thursday, May 26? Order it in the next 11 hours and 37 minutes, and choose one-day delivery at checkout. Details'.

Tutorial book for self-learning, with small, fully executable examples.

(PDF version + all source codes available free from Bluespec)

The article is titled 'MIT Prof Uses ESL Tools, FPGAs to Teach System Architecture' and is categorized under 'XPERT OPINION'. It is written by Clive (Max) Maxfield, President of Maxfield High-Tech Consulting, with an email address max@CliveMaxfield.com. The background of the article section features a blurred image of a computer screen displaying a circuit diagram or simulation results.



BSV is in use in over 50 universities worldwide, including top-tier universities.

(See article in Xilinx Xcell Journal 3Q 2011 on remarkable projects accomplished by students in a 1-semester MIT course.)

# Comparing BSV's approach to other HDLs

<i>Behavioral Semantics</i>	BSV	Synthesizable RTL (Verilog, VHDL, SystemVerilog, SystemC)	“HLS” from C/C++/Matlab
Behavior	Rules (atomic)	Synchronous circuits	Sequential programming
Interfaces	Object-oriented methods (atomic)	Wires (few TLM interfaces)	N/A (few predefined interfaces for top-level)

<i>Structural abstractions</i>	BSV	Synthesizable RTL (Verilog, VHDL, SystemVerilog, SystemC)	“HLS” from C/C++/Matlab
Architectural transparency	Strong	Strong	Weak
Type-checking	Strong	Weak/Medium	Medium
Types	Powerful user-defined types	Bits, weak user-defined types	Weak user-defined types
Parameterization	Powerful	Weak	Weak

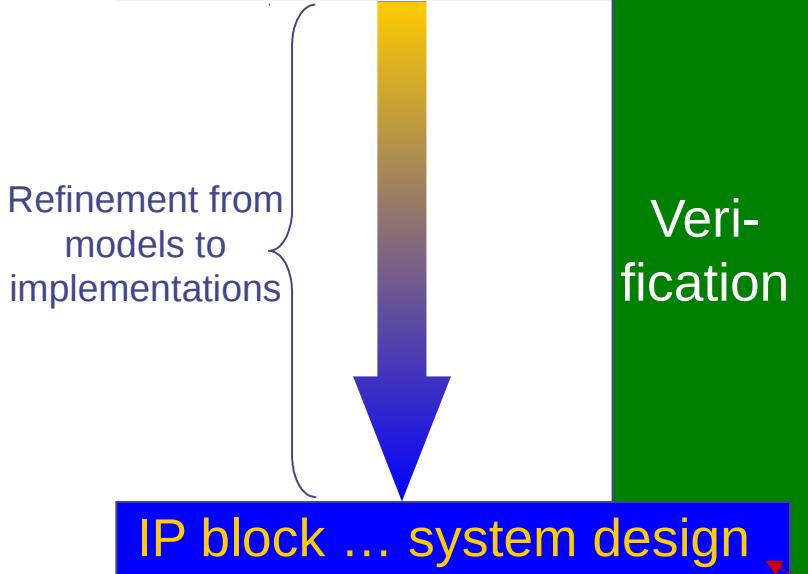
Note: Last two columns only consider *synthesizable subsets* (e.g., not SystemC TLM). Higher-level constructs are available if you are only interested in simulation.

# BSV use models

*BSV is used for modeling:*

- E.g., processor architecture models in BSV include MIPS, Sparc, x86, Itanium, ARM, PowerPC, TenSilica, RISC-V, JVM, and some more
- All of them synthesized and running on FPGAs
- Many of them executing real apps and OSs (Linux, ...)

## High-level Modeling



*BSV is used for verification of complex IPs:*

- E.g., transactors for PCIe Gen 3, multi-core cache-coherent processor interconnect and AXI
- All synthesized, and running on FPGAs

*BSV is used for complex IPs:*

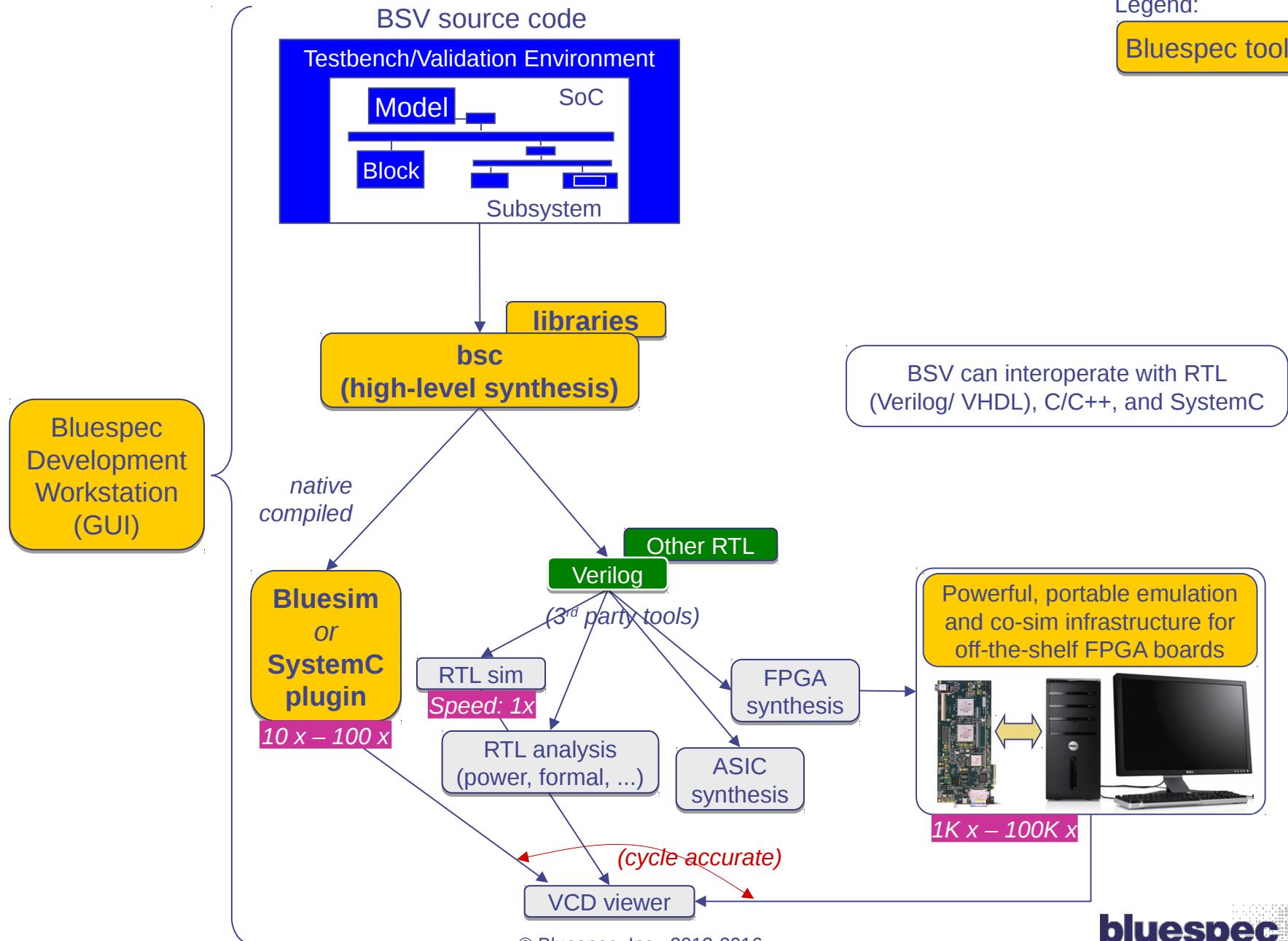
- E.g., IPs in commercial mobile devices (phones/tablets) and set-top boxes, involving both high-speed datapaths and complex control

*Note: unlike BSV's broad range of use models, other high-level synthesis tools are only used for IP design, and only for signal-processing IPs (datapath, not control)*

## BSV tool flow: core tools

## Legend:

## Bluespec tools



# Resources

- Language reference guide: `$BLUESPEC_HOME/doc/BSV/reference-guide.pdf`
  - Complete reference on the BSV language (syntax, semantics, all language constructs, scheduling annotations, importing C and Verilog, extensive libraries)
- Tool usage guide: `$BLUESPEC_HOME/doc/BSV/user-guide.pdf`
  - How to use BDW (Bluespec Development Workstation), how to compile and link using `bsc`, how to simulate using Bluesim and Verilog sim, how to generate and view waveforms, etc.
- Examples, lecture slides, training materials:
  - [https://github.com/rsnikhil/Bluespec\\_BSV\\_Tutorial.git](https://github.com/rsnikhil/Bluespec_BSV_Tutorial.git)
- BSV-by-Example book (authors: Nikhil and Czeck):
  - Around 60 examples, each focusing on one topic, with ready-to-run source code
  - Hardcopy version: purchase at Amazon.com
  - Free PDF of book: `$BLUESPEC_HOME/doc/BSV/bsv_by_example.pdf`
  - All the example code: `$BLUESPEC_HOME/doc/BSV/bsv_by_example_appendix.tar.gz`
- General questions about BSV, the tools, anything:
  - User Forums at [bluespec.com](http://bluespec.com) (free, after registration)
  - E-mail to ‘[support@bluespec.com](mailto:support@bluespec.com)’

# Lecture slide decks reading guide

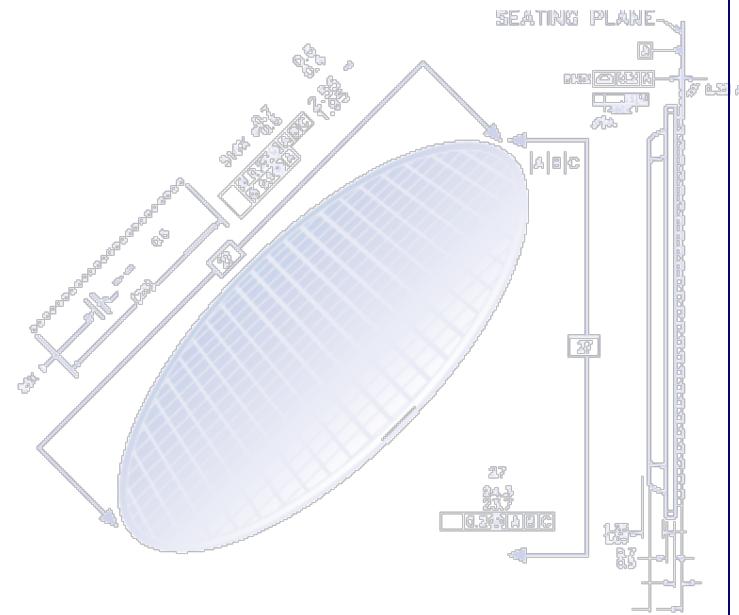
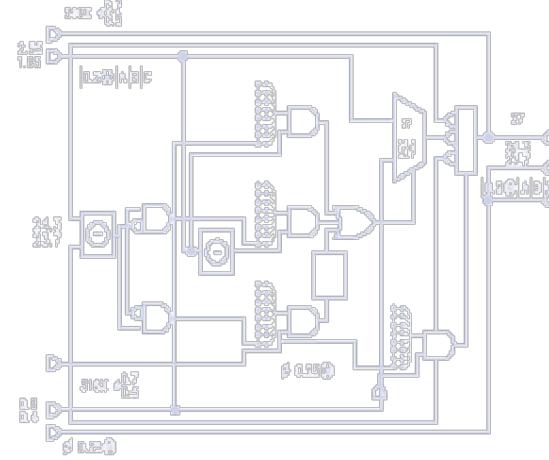
The topic-based lecture slide decks in the "Reference/" directory are intended as a reference, and need not be read sequentially.

However, people learning BSV on their own for the first time may wish to read them in the following order:

- Lec\_Intro
- General intro to the Bluespec approach, and some comparisons to other Hardware Design Languages and High Level Synthesis.
- Lec\_Basic\_Syntax  
Gets you familiar with the "look and feel" of BSV code.
- Lec\_Rule\_Semantics, Lec\_CRegs  
These two lectures describe BSV's concurrency semantics (based on rules and methods). This is the KEY feature distinguishing BSV from other hardware ands of software languages.
- Lec\_Interfaces\_TLM, Lec\_StmtFSM  
These two lectures describe slightly advanced constructs: more abstract interfaces, and more abstract rule-based processes.
- Lec\_Types, Lec\_Typeclasses  
These two lectures describe BSV's type system, which is essentially identical to that of the Haskell functional programming language.
- Lec\_BSV\_to\_Verilog  
Desribes how BSV is translated into Verilog by the bsc tool. Read this only if you are curious about this, or if you need to interface to other existing RTL modules.
- Lec\_Interop\_RTL  
How to import Verilog/VHDL code into BSV, and how to connect BSV into existing Verilog/VHDL.
- Lec\_Interop\_C  
How to import C code into BSV (for simulation only). How to export a BSV subsystem as a SystemC module (for use in a SystemC program).
- Lec\_Multiple\_Clock\_Domains  
How to create BSV designs that use multiple clocks or resets.
- Lec\_RWires  
A follow-up to Lec\_CRegs, showing lower-level and stateless primitives for greater concurrency



End





# BSV Training

# Lec\_Basic\_Syntax

General syntactic structure of BSV programs: Identifiers, types, packages, scoping, interfaces, modules, module instantiation, module hierarchy, interface methods (Action, ActionValue and value methods), single-assignment (“`=`”), “let”, functions.

`typedef struct {`  
    `char *label;`  
    `char *value;`  
    `} Label;`

`int main()`  
{  
    Label l;  
    l.label = "val";  
    l.value = "1234567890";  
  
    printf("%s\n", l.value);  
    return 0;  
}

**Note: this lecture is explained in other slides.**

erequisite to understand behavior (dynamic semantics), which is  
lectures (Lec\_Rule\_Semantics, Lec\_CRegs, Lec\_StmtFSM)

© Bluespec, Inc., 2015-2016

# Basic syntax elements, and identifiers

Basic syntax elements (identifiers, comments, whitespace, strings, integer constants, infix operators, etc.) all follow standard Verilog and SystemVerilog syntax.

Standard “static scoping” rules for identifier visibility and shadowing in nested scopes.

BSV identifiers are case sensitive.

The case of the first letter in an identifier is significant:

- Constants and Types begin with an uppercase letter: Int, UInt, Bool, True, False, ...
- Variables and type variables begin with a lowercase letter (type1, x, y, w, mkMult, ...)

*Two exceptions, for legacy sake: the types ‘int’ and ‘bit’*

- *These are familiar types from Verilog*
- *We recommend the equivalent standard BSV syntax Int#(32) and Bit#(1) instead*

Module naming convention: in all our examples we use names like  
“mkTestbench” and “mkMult”

- The “mk” prefix is pronounced “make” and reinforces the idea that, as in Verilog, a module declaration is a generator, i.e., the module can be *instantiated* multiple times, each time providing a fresh instance.
- This is just a convention, for style and readability (not a syntax rule)

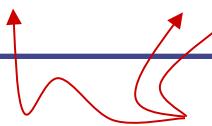
# Syntax of types: Type Expressions

BSV uses SystemVerilog's notation for parameterized types

```
Type ::= TypeConstructor #(Type1, ..., TypeN)
      | TypeConstructor           // special case when N=0)
```

i.e., a type expression is a type constructor applied to zero or more other types. In the special case where it is applied to zero other types, the #() part can be omitted.  
Examples:

Type	Comments
Integer	Unbounded signed integers (static elaboration only)
Int#(18)	18-bit signed integers Note: 'int' is a synonym for Int#(32)
UInt#(42)	42-bit unsigned integers
Bit#(23)	23-bit bit vectors Note: 'bit[15:0]' is a synonym for Bit#(16)
Bool	Booleans, with constants True and False
Reg#(UInt#(42))	Interface of register that contains 42-bit unsigned integers
Mem#(A,D)	Interface of memory with address type A and data type D
Server#(Rq,Rsp)	Interface of server module with request type Rq and response type Rsp

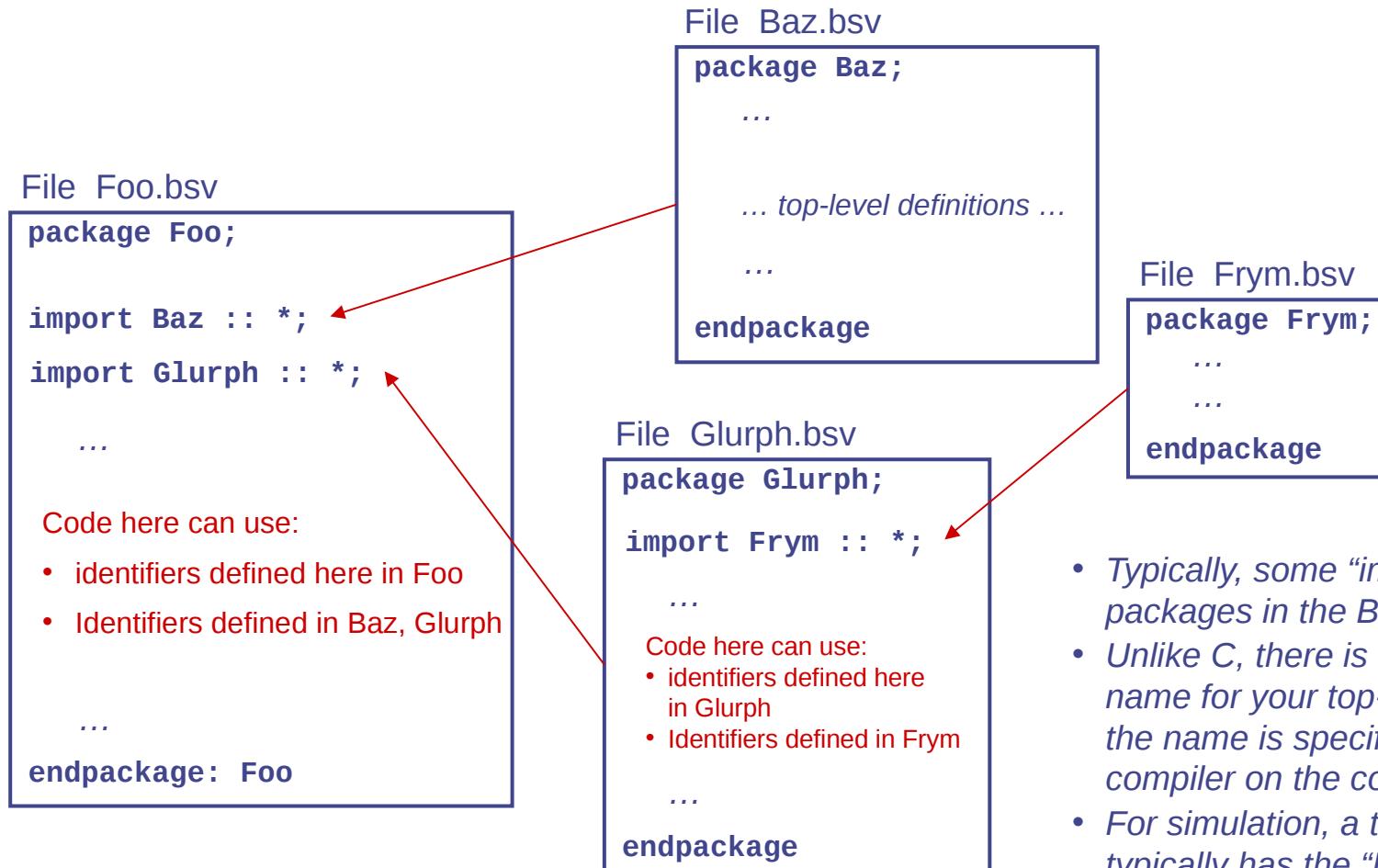


Note uppercase first letter in type names

© Bluespec, Inc., 2015-2016

# Overall syntactic structure of a BSV program

A complete BSV program is a collection of files (each file is a BSV “package”). A package Foo may “import” another package Baz, making the top-level identifiers defined in Baz visible (and usable) in Foo.



- Typically, some “imports” refer to packages in the BSV library
- Unlike C, there is no special “main” name for your top-level module; the name is specified to the **bsc** compiler on the command line
- For simulation, a top-level module typically has the “Empty” interface

# What's in a package?

```
File Foo.bsv
package Foo;
import Baz :: *;
export a, b;
typedef struct {...} S;
interface Foo_IFC;
...
endInterface
UInt #(16) a = 23;
function int f (int x);
...
endfunction
module mkFoo (...);
...
endmodule
endpackage: Foo
```

File name and package name must match; *bsc* will complain if they do not.

The package/endpackage lines are optional; *bsc* will use the filename if they are absent.

*import/export statements*

*type declarations*

*interface declarations*

*value (constant) declarations*

*function declarations*

*module declarations*

Trailing “: Foo” is optional, and  
must match the name at the top

# Namespace control with imports and exports

File Foo.bsv

```
package Foo;  
  
import Baz :: *; ←  
import Glurph :: *; ←  
  
...  
  
Code here can use:  
• identifiers defined here in Foo  
• B_ifc, mkB and Baz::x from Baz  
• Glurph::x from Glurph  
• identifiers defined Frym  
  
...  
  
endpackage: Foo
```

Baz only exports the three identifiers B\_ifc, mkB and x; all other identifiers in Baz are *private* to Baz.

File Baz.bsv

```
package Baz;  
  
export B_ifc, mkB, x;  
...  
interface B_ifc;  
...  
endinterface  
...  
module mkB (...);  
...  
endmodule  
...  
UInt #(16) x = 42;  
...  
endpackage
```

Glurph exports its own identifier x, and also *re-exports* everything it imports from Frym.

File Glurph.bsv

```
package Glurph;  
  
import Frym :: *; ←  
export Frym :: *, x;  
...  
Bool x = False;  
...  
endpackage
```

With no export statement, *all* of Frym's identifiers are exported.

File Frym.bsv

```
package Frym;  
...  
...  
endpackage
```

# Separate compilation

The *bsc* tool processes each file separately:

- Parsing, typechecking, name resolution, and certain other things
- It generates code separately (for Bluesim or Verilog generation) only for modules marked with the “(\* synthesize \*)” attribute (or, equivalently, named with the “-g” flag on the command line)

We recommend using the “(\* synthesize \*)” attribute liberally, i.e., on as many modules as possible:

- It can speed up compilation since modules are analyzed and compiled separately, and because it enables incremental compilation (*bsc* does not have to recompile already compiled modules whose source files have not changed)
- The greater retention of structure into the compiled code provides more clarity when debugging and viewing waveforms

# What's in an interface declaration?

The diagram shows a code snippet for a Bluespec interface named `Foo_IFC`. The code includes type parameters, action method declarations, an ActionValue method declaration, value method declarations, and sub-interface declarations.

```
interface Foo_IFC #(numeric type n, type t);
    method Action m1 (int x, Bool y);
    method ActionValue #(int) m2 (... args ...);
    method int m3 (... args ...);
    interface Put #(int) i4;
    ...
endInterface
```

- interface name:** Points to the identifier `Foo_IFC`.
- type parameters:** Points to the type parameters `#(numeric type n, type t)`.
- Action method declarations (methods can have arguments):** Points to the `method Action` declaration.
- ActionValue method declaration:** Points to the `method ActionValue` declaration.
- Value method declarations (return type is not Action or ActionValue):** Points to the `method int` declaration.
- sub-interface declarations:** Points to the `interface Put` declaration.

# What's in a module declaration?

The diagram illustrates the components of a Bluespec module declaration with the following annotations:

- module name**: Points to the identifier `Foo_IFC`.
- module parameters**: Points to the parameter `#(int n)`.
- module interface type**: Points to the interface type `(Foo_IFC)`.
- Value (constant) declarations and definitions**: Points to the variable declaration `UInt #(16) a = 23;`.
- Module instantiations**: Points to the module instantiations `Reg #(int) x <- mkReg (10);` and `Switch #(4,4) switch <- mkSwitch;`.
- Function declarations**: Points to the function declaration `function int f2 (...);` and its endfunction block.
- Rules**: Points to the rule declaration `rule rl_r1 (...);` and its endrule block.
- Method definitions**: Points to the method declaration `method Action m1 (int x, Bool y);` and its endmethod block.
- Sub-interface definitions**: Points to the sub-interface definition `interface i4;` and its endinterface block.

```
module Foo_IFC #(int n) (Foo_IFC);
    UInt #(16) a = 23;

    Reg #(int) x <- mkReg (10);
    Switch #(4,4) switch <- mkSwitch;

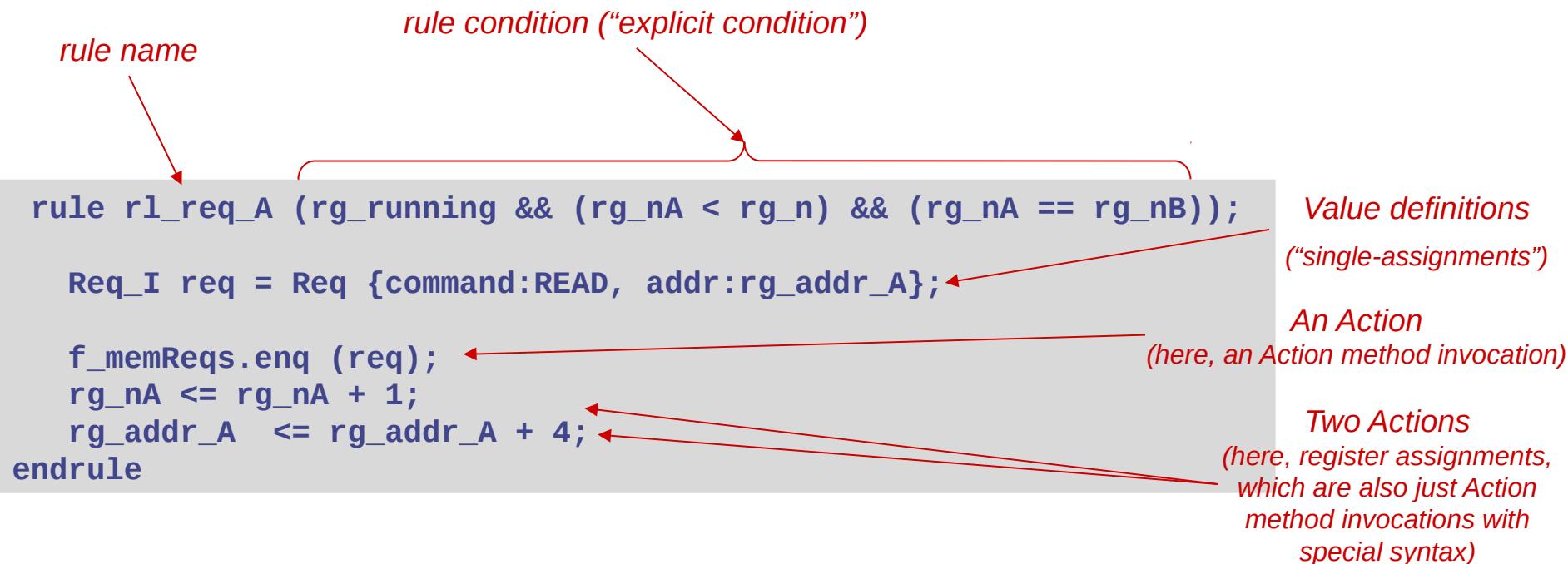
    function int f2 (...);
        ...
    endfunction

    rule rl_r1 (...);
        ...
    endrule

    method Action m1 (int x, Bool y);
        ...
    endmethod

    interface i4;
        ...
    endinterface
endmodule
```

# What's in a rule?



*Note: rule and method bodies can also contain conditionals and generative loops.*

- A rule condition is always an expression of type `Bool`
  - (Therefore, by BSV strong type-checking rules, it cannot have a side-effect, i.e., it cannot contain an Action!)
- The rule body is an expression of type `Action`
  - The overall Action of a rule body may, in turn, be consist of smaller Actions, which, in turn, may contain smaller Actions, and so on (Action is a recursively defined type)

# What's in a method definition?

The diagram shows a Bluespec method definition with various annotations:

- method name**: Points to the identifier `ActionValue#(t)`.
- method condition ("implicit condition")**: Points to the expression `xs[0]==1 && (rg_inj == n)`.
- Value definitions ("single-assignments")**: Points to the assignment statements `new_xs = shiftInAtN`, `writeVReg`, and `rg_inj <= 0`.
- Actions**: Points to the conditional statement `if (xs[1] == 1)`.
- return statements (only in Value and ActionValue methods; not in Action methods)**: Points to the `return x0;` statement.

```
method ActionValue#(t) get () if (xs[0]==1 && (rg_inj == n));
  let new_xs = shiftInAtN (readVReg (xs), tagged Invalid);
  writeVReg (xs, new_xs);
  if (xs[1] == 1)
    rg_inj <= 0;
  return x0;
endmethod
```

*Note: rule and method bodies can contain conditionals and generative loops.*

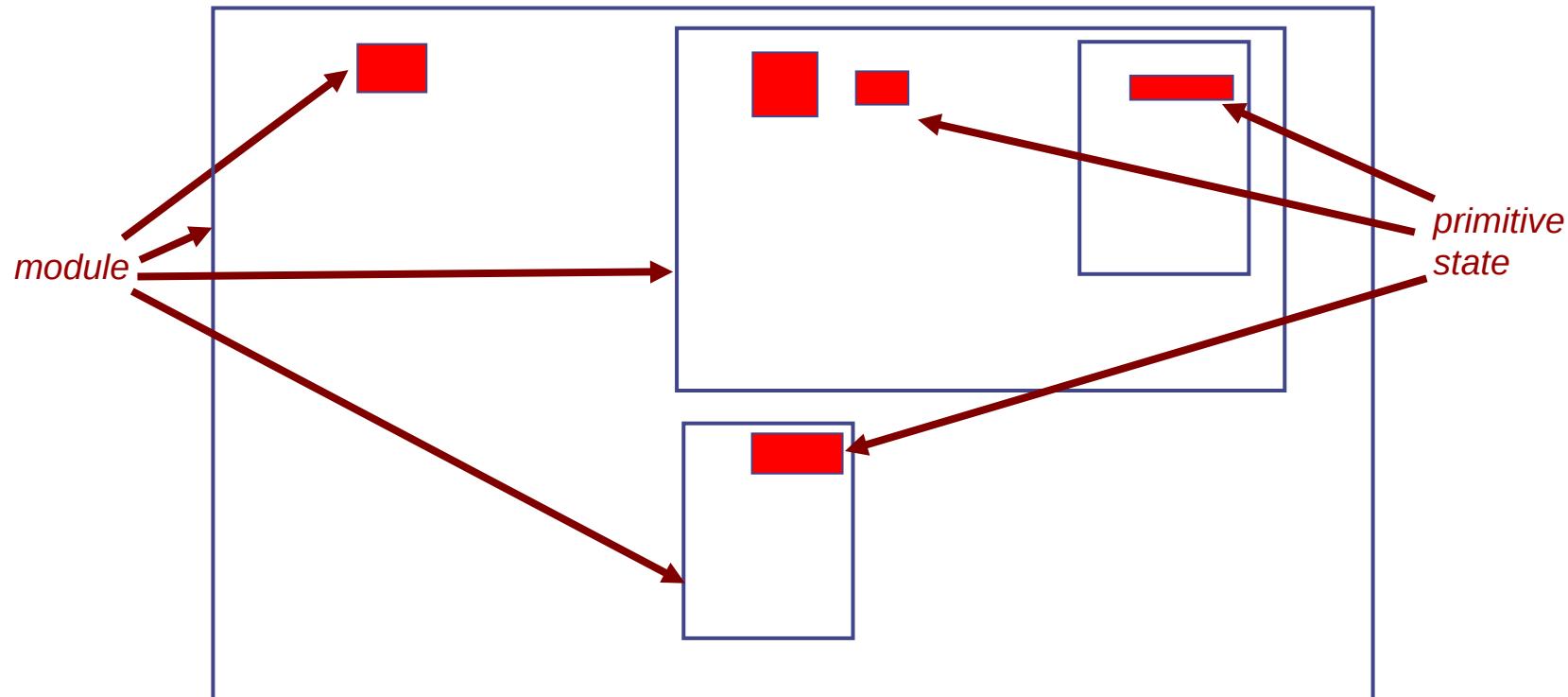
- A *method condition* is always an expression of type *Bool*
  - (Therefore, by BSV strong type-checking rules, it cannot have a side-effect, i.e., it cannot contain an Action!)
- For *ActionValue* methods, the method body contains *Actions* (like a rule body), and a 'return' statement for the return value
- For *Action* methods, the method body contains *Actions* (like a rule body)
- For *value* methods, the method body is a begin-end block with a 'return' statement

# Circuit structure: module hierarchy and state

A BSV design consists of a *module hierarchy* (just like in Verilog, SystemVerilog and SystemC)

The leaves of the hierarchy are “primitive” state elements, including registers, FIFOs, etc.

Even registers are (semantically) modules (unlike in Verilog, SystemVerilog, ...).



All “primitives” in BSV are in fact implemented in Verilog and “imported” using BSV’s standard import mechanism. Hence, you can easily create new primitives or import existing Verilog IP.

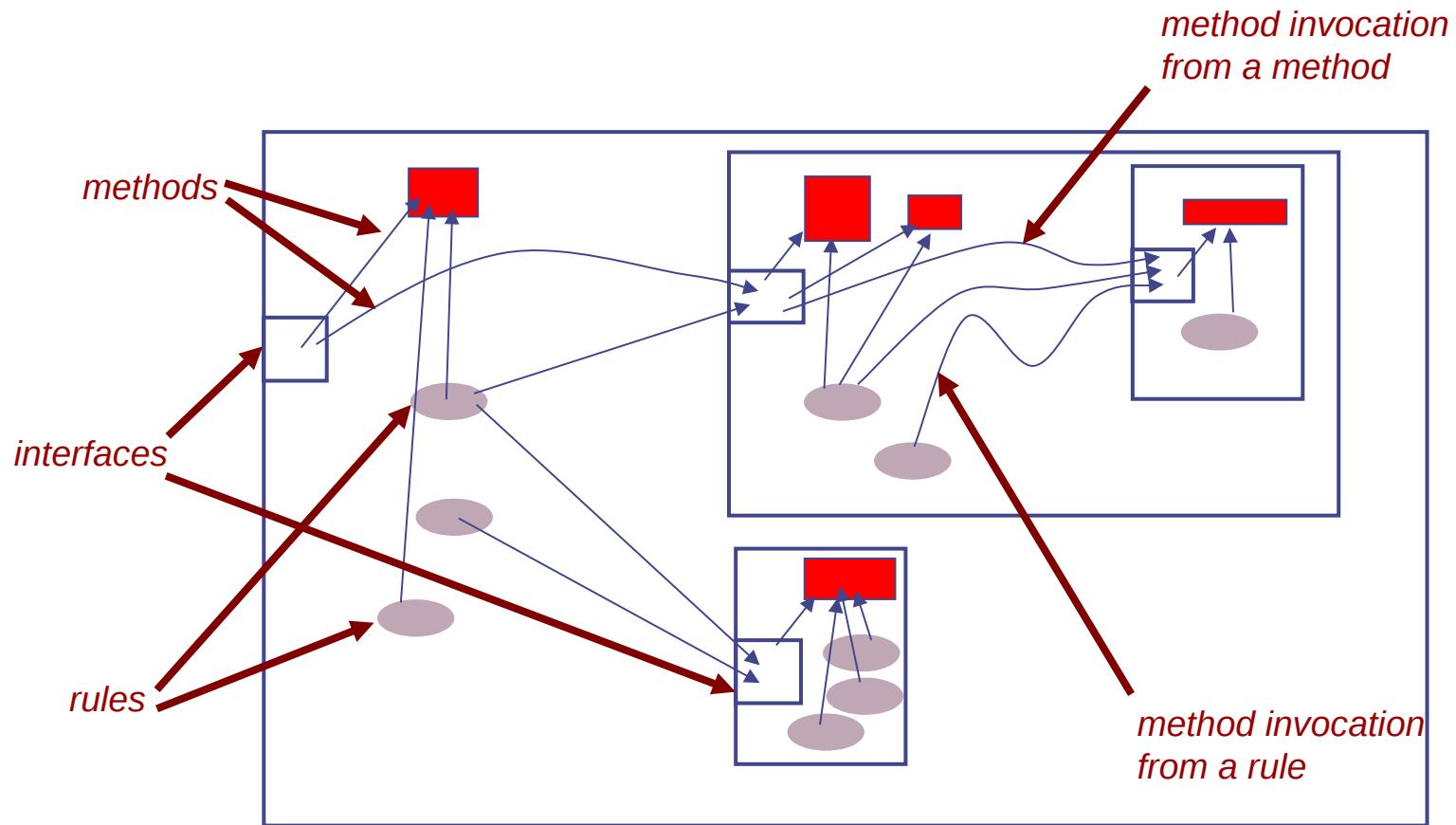
# Rules and interface methods

Modules provide interfaces, which contain *interface methods*.

Modules contain rules, which use methods in other modules.

All inter-module communication is via methods (object-oriented)

A method can itself use methods of other modules.



# Even registers are modules ...

Registers are just modules with the following interface:

```
interface Reg #(t);
    method Action _write (t v);
    method t      _read  ();
endinterface: Mult_ifc
```

Where “t” is “int” or “Bool” or some other type

Following standard BSV syntax, a register update would look like this:

```
x._write (x._read () << 1);
```

But, for convenience, the BSV compiler allows you to omit “.\_read()” from register reads, and will insert it for you. So, our update becomes:

```
x._write (x << 1);
```

Further, for convenience, BSV provides special syntax using the conventional “ $\leq$ ” for register assignment for “.\_write()”. So, our update becomes:

```
x <= x << 1;
```

# Module instantiation

Module instantiation has the following syntax:

```
interface_type instance_name <- module_name ( module_parameters );
```

“( *module\_parameters* )” can be omitted if a module has no parameters.

Examples:

```
Mult_ifc m <- mkMult;
```

```
Reg#(int) w      <- mkRegU;  
Reg#(Bool) got_x <- mkReg (False);
```

**mkRegU** is a module with no parameters; the register’s initial (reset) value is unspecified.  
**mkReg** is a module with a parameter for the register’s initial (reset) value.

# Interfaces: introduction

- All inter-module communication in BSV is expressed using *interface methods*
  - BSV is completely “transactional”, or “object-oriented”
  - (Traditional “in” and “out” signal ports are just a special case!)
- Every module *provides* an interface, i.e., it has an interface whose methods are invoked by other modules that need to communicate with it
  - This is always specified as an *interface type*
- Methods can be viewed just as rule fragments—they have the same features as rules, with the same semantics:
  - Conditions and bodies (which can be Actions)
  - Thus, interfaces and methods are a way to organize a large design into smaller, manageable, reusable modules

# Modules and interfaces: example

```
interface FIFO #(type any_t);
    method Action enq (any_t x);
    method any_t first;
    method Action deq;
    method Action clear;
endinterface: FIFO
```

Interface declaration

(*FIFO#(t)* is a standard interface in the BSV library)

Module that provides  
a FIFO interface

```
module mkFIFO (FIFO#(some_t));
    ...
endmodule
```

Instantiating modules  
with FIFO interfaces

Using FIFO interface methods

```
module mkBaz (...);
    FIFO#(int) f1 <- mkFIFO;
    FIFO#(int) f2 <- mkFIFO;
    ...
    ...
    rule r1 (f1.first() > 8);
        f2.enq (f1.first() + 2);
        f1.deq;
    endrule
endmodule
```

# Three kinds of methods, depending on return type

```
interface FIFOwPop #(type t);
    method Action           enq (t x);
    method t                first;
    method ActionValue#(t)  pop;   // combines first and deq
endinterface: FIFOwPop
```

- *Value* methods: Takes 0 or more arguments and has a return type other than Action or ActionValue. E.g., ‘first’.
  - BSV’s type discipline guarantees that it cannot have a side-effect
  - It is always purely combinational
- *Action* methods: Takes 0 or more arguments and has Action type. Represents a pure side-effect inside the module. E.g., ‘enq’
- *ActionValue* methods: Takes 0 or more arguments and has ActionValue type. Represents a side-effect inside the module, and also returns a value. E.g., ‘pop’

Rule and method conditions cannot have side-effects (they’re “pure”).

Thus, Action and ActionValue methods can only be used in rule bodies, and bodies of other Action and ActionValue methods. Value methods can be used anywhere, including in rule and method conditions.

# Using ActionValue methods

```
module mkFIFOwPop (FIFOwPop#(int));  
    ...  
endmodule
```

```
module mkFoo (...);  
    FIFOwPop#(int) f1 <- mkFIFOwPop;  
    FIFOwPop#(int) f2 <- mkFIFOwPop;  
    ...  
    rule r1 (...);  
        int x <- f1.pop;  
        f2.enq (x + 2);  
    endrule  
endmodule
```

Note this assignment symbol

In both cases, the right-hand side of the assignment has a side-effect and returns a value:

- The first one is a side-effect *during static elaboration*: it creates a module and returns its interface
- The second one is a side-effect *during dynamic execution*: it pops an element from the FIFO and returns it

# Defining methods in modules

```
module modName [ #(type arg,...) ] ( IfcType );
  ...
  ...
  method [ type ] methodName1 (arg, ..., arg) [ if ( cond ) ];
    ... method body ...
  endmethod

  ...
  method [ type ] methodNameN (arg, ..., arg) [ if ( cond ) ];
    ... method body ...
    return expr           // if it is a value method
  endmethod
endmodule
```

- All the method definitions of a module are written at the end of the module (after the sub-module instantiations, rules, etc.)
- The ‘if’ conditions become the implicit conditions of the methods

# Interfaces are a means to *modularize* rules

- Action and ActionValue methods become a part of any rule from which they are invoked
  - The method condition (implicit condition) becomes a part of the rule condition
  - The method's actions become a part of the rule's actions
- Value methods also become a part of any rule from which they are invoked
  - The method condition (implicit condition) becomes a part of the rule condition

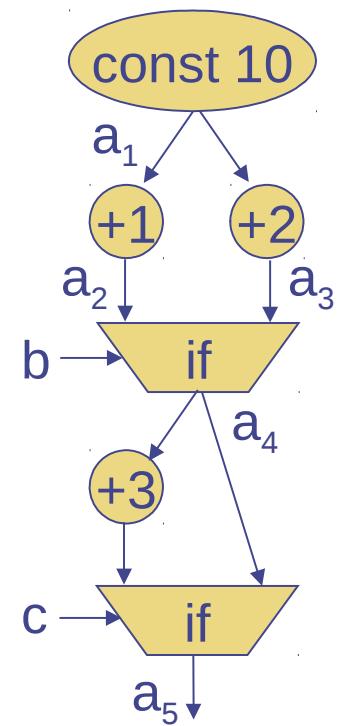
Thus,

- Modules, interfaces and methods can be viewed as a way to organize a large piece of behavior into smaller, localized chunks
- Interfaces do not have any separate semantics—they seamlessly integrate into standard rule semantics

# Variables and “single-assignment” in BSV

- Variables in BSV represent immutable values (as in mathematics)
  - Note: in C, C++, Verilog, etc., a variable represents a storage location that can be updated dynamically and repeatedly by assignment. In BSV, dynamic updates are *only* expressed with Action (including register assignment “`<=`”).
- Repeated syntactic assignment in BSV with “`=`” is just a notational device for incrementally describing more complex expressions
  - It’s like a new variable (e.g. with a new subscript) from that point on
  - I.e., it represents a new value over the “rest of the program text”, and is not associated with “time”
  - This concept is well-known as “static single-assignment” or (SSA) in functional programming languages (and inside most modern compilers)

```
int a = 10;  
  
if (b) a = a + 1;  
else a = a + 2;  
  
if (c) a = a + 3;
```

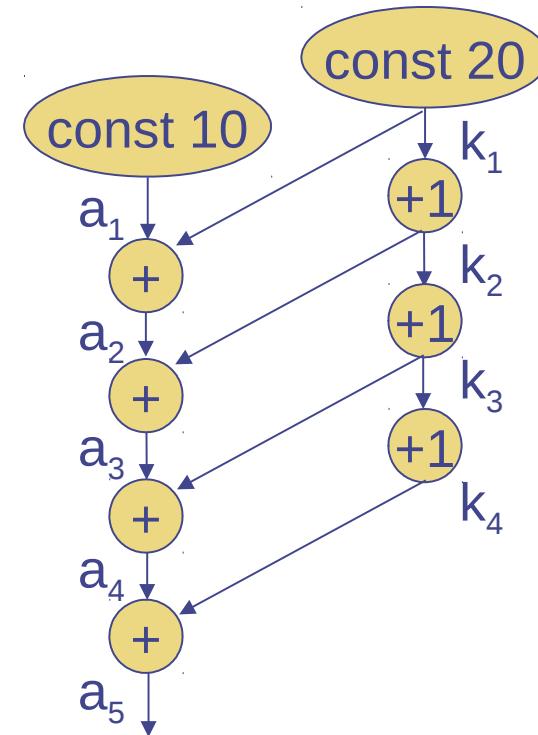


# Ordinary variable assignment: =

Another example, involving a statically elaborated loop

- [Compiler gurus: this is just “SSA form” (static single assignment)]

```
int a = 10;  
for (int k = 20; k < 24; k = k+1)  
    a = a + k;
```



# Assignment symbols: =, <- and <=

'=' is used just for variable assignment. It is purely static. Example:

```
Int#(124) b = myMemory.lookup(addr);
```

'<=' is used as a syntactic shorthand for \_write method invocation.

Example:

```
rule doIt;
    myRegA <= 32;
endrule
```

Note: "<=" is also used inside expressions for the "less than or equal to" operator. This is always unambiguous due to context.

'<-' is used at the top-level of modules for module instantiation:

```
Reg#(Bit#(32)) myRegA <- mkRegA(0);
```

'<-' is used in rules and methods to invoke an ActionValue and assign its returned value:

```
rule doIt;
    let a <- myFIFO.pop;
    ...
endrule
```

# “let”

- BSV has a “let” statement by which you can declare a variable and initialize it, with the compiler deducing the type of the variable based on the initial value expression
- It can reduce clutter, especially when the type is “obvious” from the init value

```
let var = init;           // syntax  
  
let x = 24'h9BEEF;  
  
let y = x + 3;  
  
let z = MyStruct {memberA: exprA, memberB: exprB};
```

*type var;  
var = init;*

Equivalent to declaring and defining a variable, except that the type is automatically inferred by the compiler from the “init” expression;

Compiler infers type: **Bit#(24)**

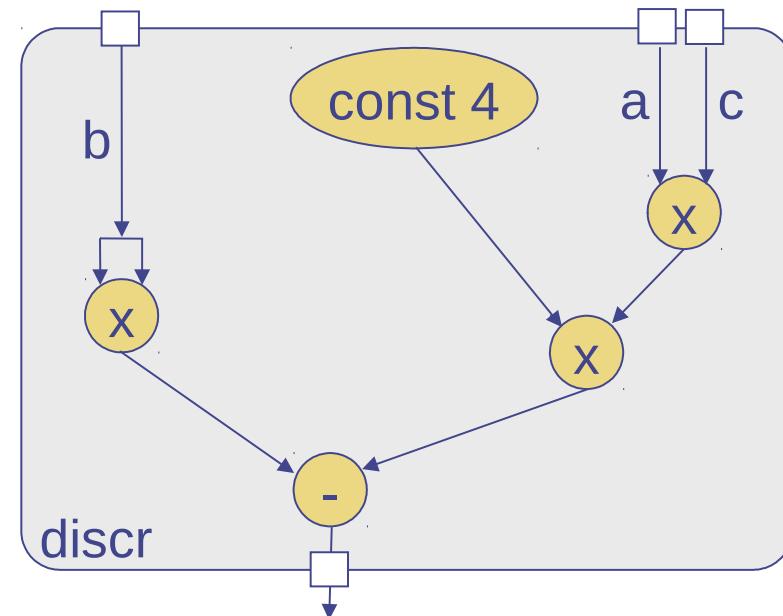
Compiler infers type: **Bit#(24)**

Compiler infers type: **MyStruct**

# Functions

- Functions in BSV have no run-time semantics. They are purely a static syntactic convenience, and are expanded in-line during static elaboration
  - No “stack frame”, no “call and return”
  - Think of them as circuits plugged into place wherever used

```
function int discr (int a, int b, int c);
    return b*b - 4*a*c;
endfunction
```

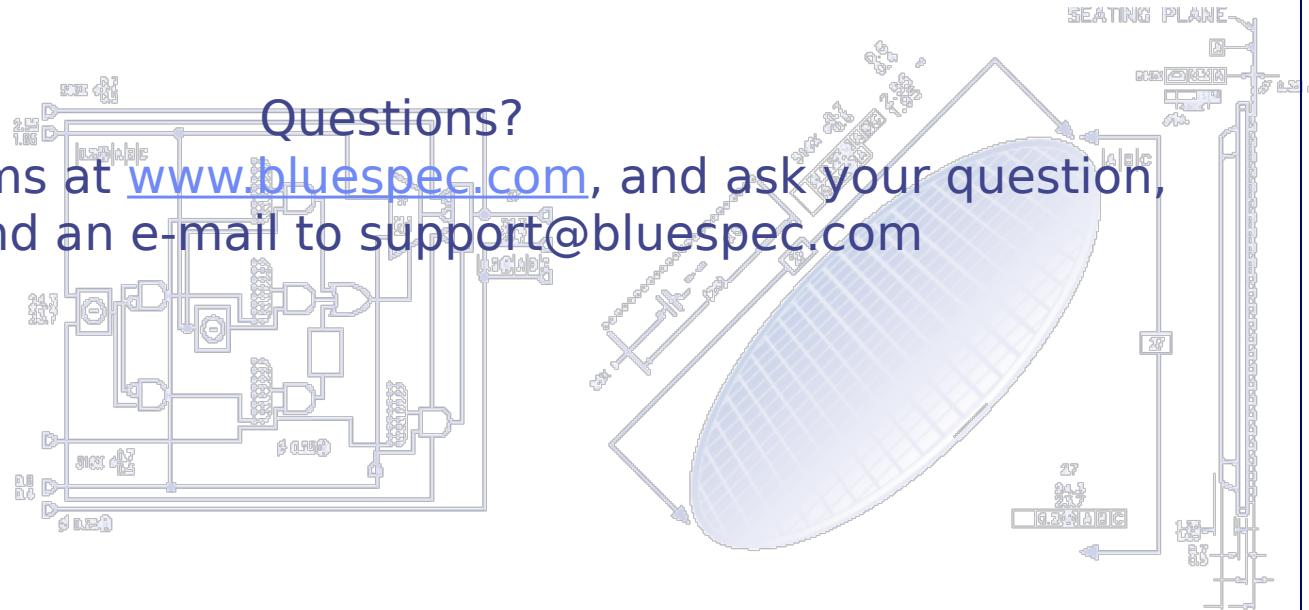




End

# Questions?

Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

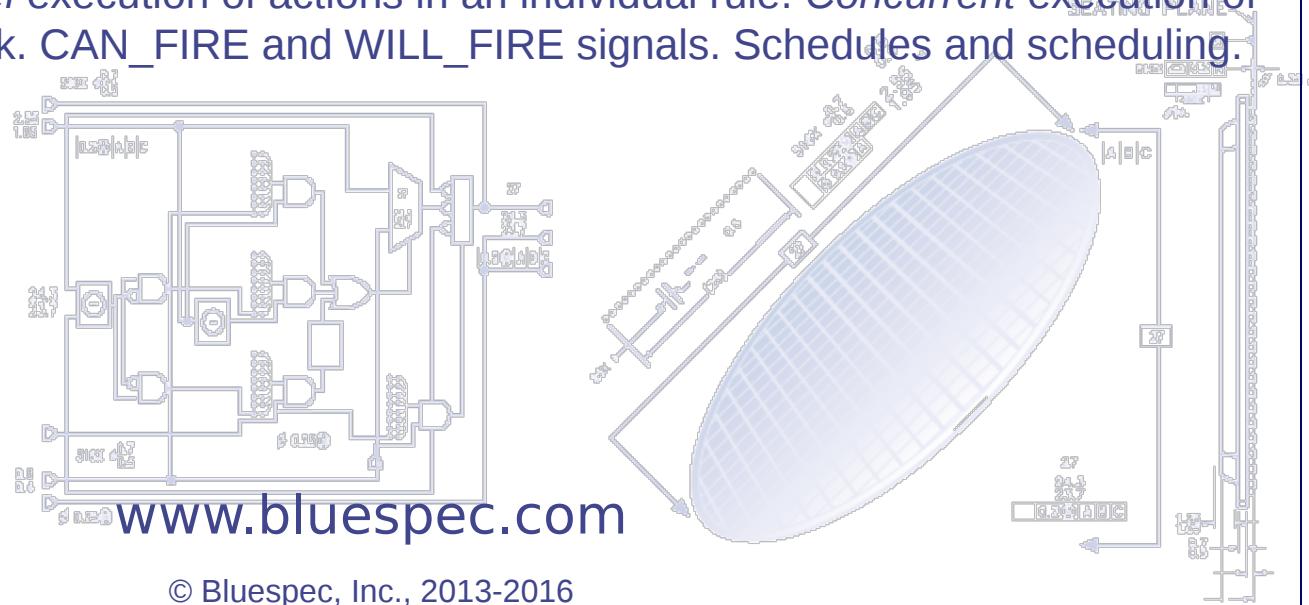


## BSV Training

### Lec\_Rule\_Semantics

Semantics of rules. *Parallel* execution of actions in an individual rule. *Concurrent* execution of multiple rules within a clock. CAN\_FIRE and WILL\_FIRE signals. Schedules and scheduling.

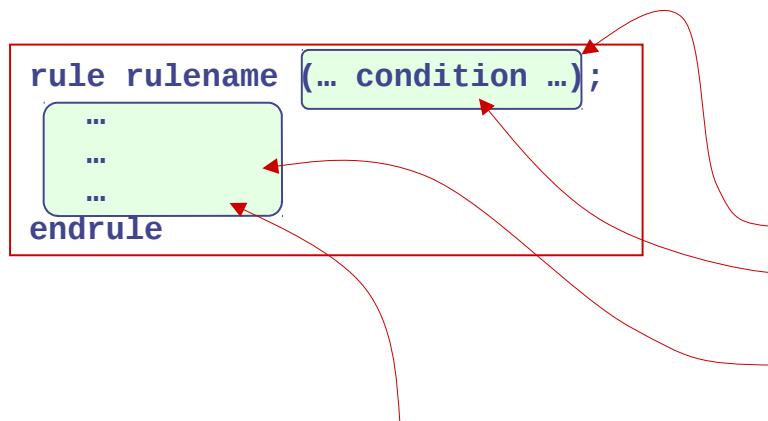
```
typeparam Bit#(2) Default;
module Action Rule#(Action);
    Integer fifo_depth = 15;
    function Bit#(1) determine_pwr(Bit#(1) val);
        return {val[0],val[1]}; // extension
    endfunction
    action
        Action
            if(val == 1) the_ifound();
            else if(val == 2) the_ofound();
            else if(val == 3) the_cifound();
            else if(val == 4) the_cifound();
            else if(val == 5) the_stored();
            else if(val == 6) the_stored();
        endaction
    endaction
endaction
endrule
endmodule
```



# Rule execution semantics, first approximation

*(This is just a first approximation; more detail to follow)*

Every rule has a rule name, and two semantically significant parts:



Composite Action: the collection of all the Actions invoked in the rule-body.

CAN\_FIRE condition: a value of type **Bool**, representing the conjunction (**&&**) of:

- the explicit rule-condition
- the method-conditions of any method invoked in the rule-condition, and
- the method-conditions of all methods invoked in the rule-body.

To first approximation, a BSV program can be simulated by:

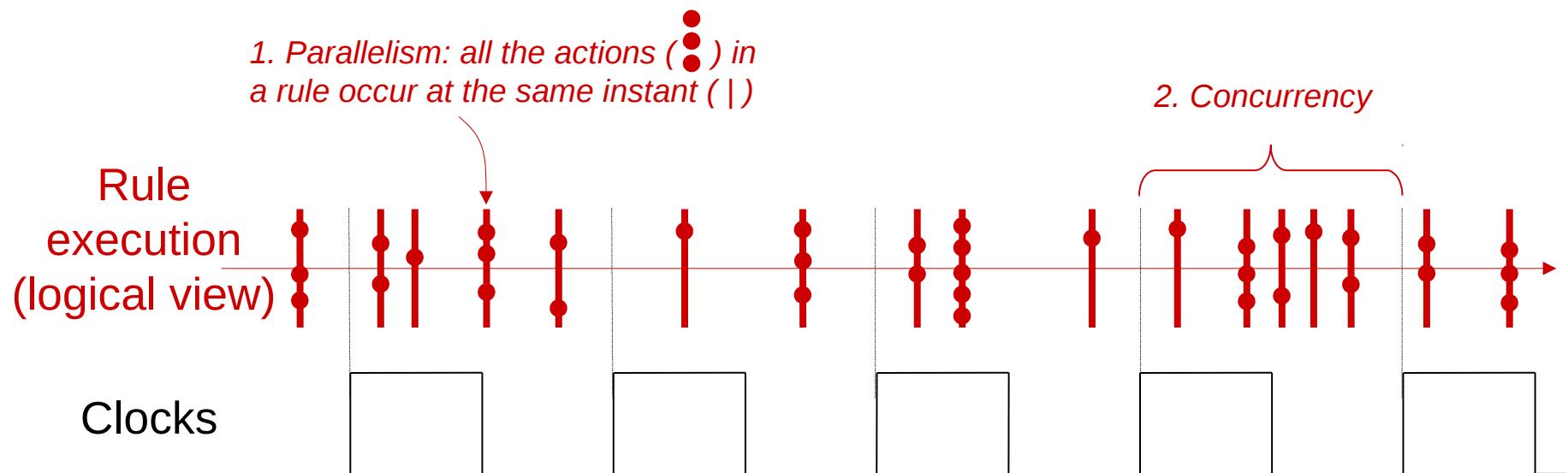
whenever a rule's CAN\_FIRE is True,  
do the rule-body actions

# Rule semantics

Rule semantics are explained in two parts:

1. Individual rule: logically, a rule fires in a single instant. All the actions in the rule (and any methods it calls) happen at the same instant. We call this *parallelism of actions*.
2. Multiple rules within a clock: logically, they execute in sequence, so the overall state change in a clock can be understood as the simple sequential composition of the state changes of the individual rules. We call this the *concurrency* of rules.

Visualization:



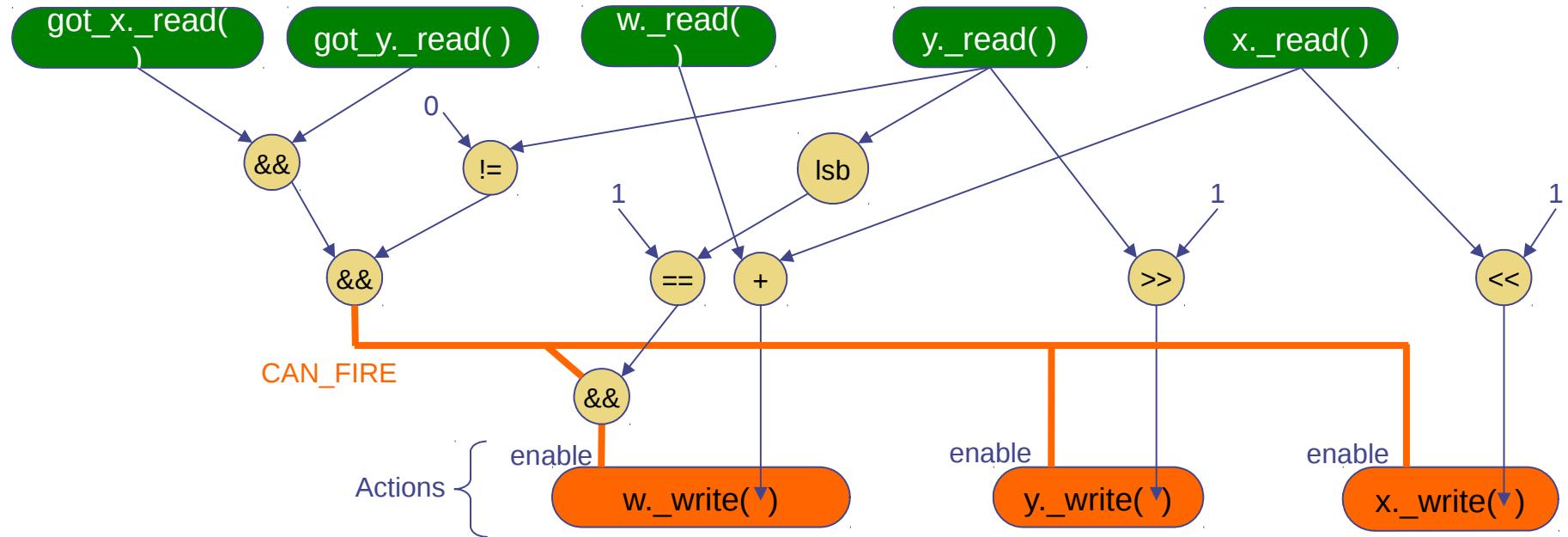
*1. Parallelism:* semantics of an individual rule, involving the simultaneous, instantaneous execution of all its “Actions”

# Semantics of a single rule (parallelism of its actions)

Every rule can be regarded as a flow of data from constants and outputs of methods, through functions and operators, ending in inputs to Action methods (this flow is always a combinational circuit).

Example: below is a visualization of the flow for the rule shown at right

```
rule compute ((y != 0) && got_x && got_y) ;
  if (lsb(y) == 1) w <= w + x;
  x <= x << 1;
  y <= y >> 1;
endrule
```



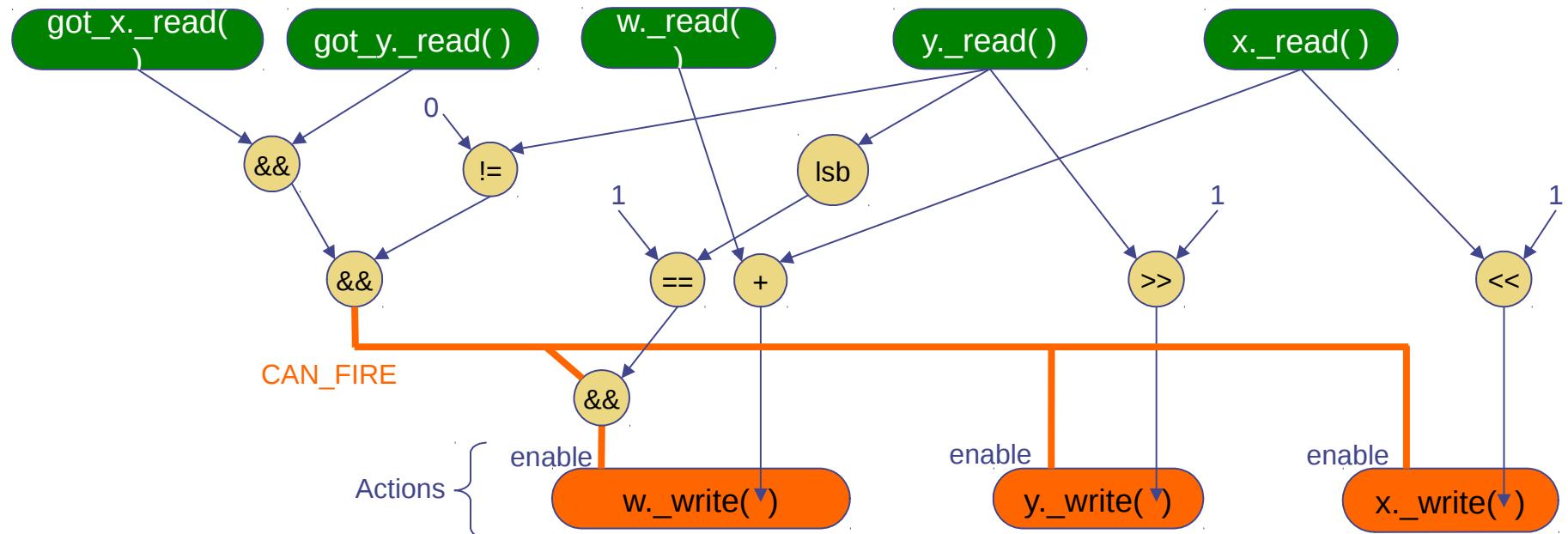
# Semantics of a single rule (parallelism of its actions)

Every Action method has an “enable” input: when True, the action is performed, using the argument values on its remaining inputs (if any).

The semantics of an individual rule are simple:

- (Simultaneously) for each of the rule’s actions whose “enable” is True, do the action

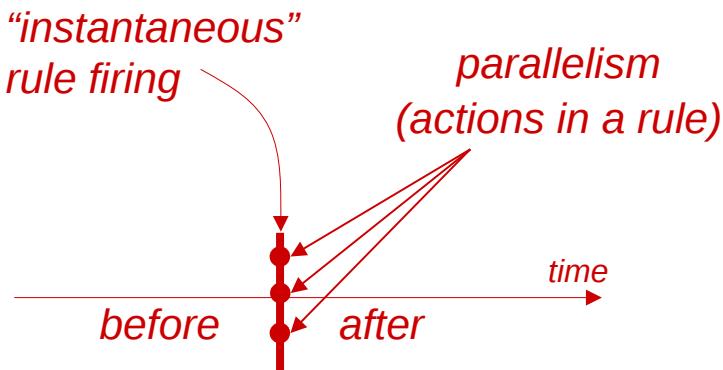
Note: as illustrated below, if the CAN\_FIRE condition of the rule is False, no Action will be enabled.  
Further, for w.write() to be enabled, the condition `(1sb(y) == 1)` must also be true.



# Semantics of a single rule (parallelism of its actions)

Some important points to observe and remember:

- There is no ordering among the actions (their textual order in the rule is not relevant). We truly think of the actions as “simultaneous”
- Because of conditionals inside a rule, not all its actions may be performed (e.g., w.\_write)
- Any values “read” by the rule are from “before the firing instant” (from previous rules)
- Any values “written” by the rule are only visible “after the firing instant” (for later rules)



# All actions in a rule are simultaneous

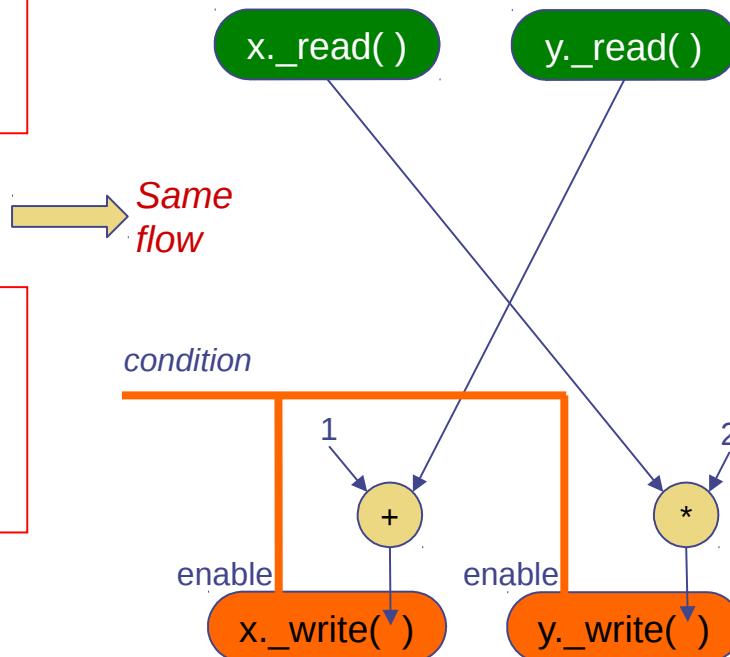
To emphasize that all actions in a rule are simultaneous, note:

- The textual ordering of actions in a rule is not important
- You can even exchange values in two registers

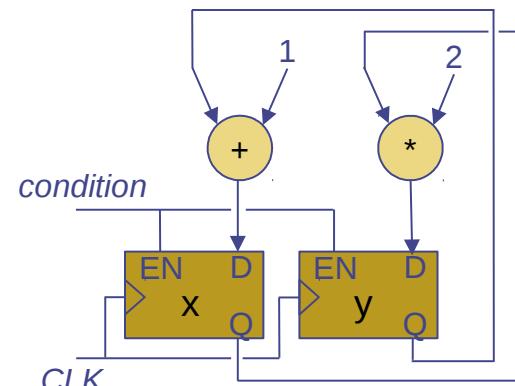
```
rule rule2a (condition);  
  x <= y + 1;  
  y <= x * 2;  
endrule : rule2a
```

```
rule rule2b (condition);  
  y <= x * 2;  
  x <= y + 1;  
endrule : rule2b
```

*Textual order of  
actions is irrelevant*



*Some HW intuition  
(for this simple rule)*



# Not all actions can be combined within a rule

```
rule rule3 (...);  
  valuea <= expr1;  
  valuea <= expr2;  
  ...  
endrule : rule3
```

*Cannot  
instantaneously  
update a register  
with two values*

```
rule rule4 (...);  
  fifo.enq (23);  
  fifo.enq (34);  
  ...  
endrule : rule4
```

*Cannot  
instantaneously enq  
two values into one  
enq port of a FIFO*

```
rule rule5 (...);  
  let x = regFile.read (5);  
  let y = regFile.read (7);  
  ...  
endrule : rule5
```

*Cannot instantaneously  
read two registers from  
one read port of a  
register file*

- The *bsc* compiler will flag such errors
  - “Cannot compose certain actions in parallel”
- Note: it is of course possible to have different register, FIFO or regFile modules that have *multiple* ports that can be accessed simultaneously, in which case those simultaneous accesses can be used in a single rule.

## 2. Concurrency: semantics of multiple rules within a clock (while preserving a logical sequential order)

# Overview of rule concurrency

Define a *schedule* as some linear ordering of all rules in a program: r1 r2 ... rN

Then, the semantics of multiple rules in a clock (logical concurrency) is simple:

For each clock:

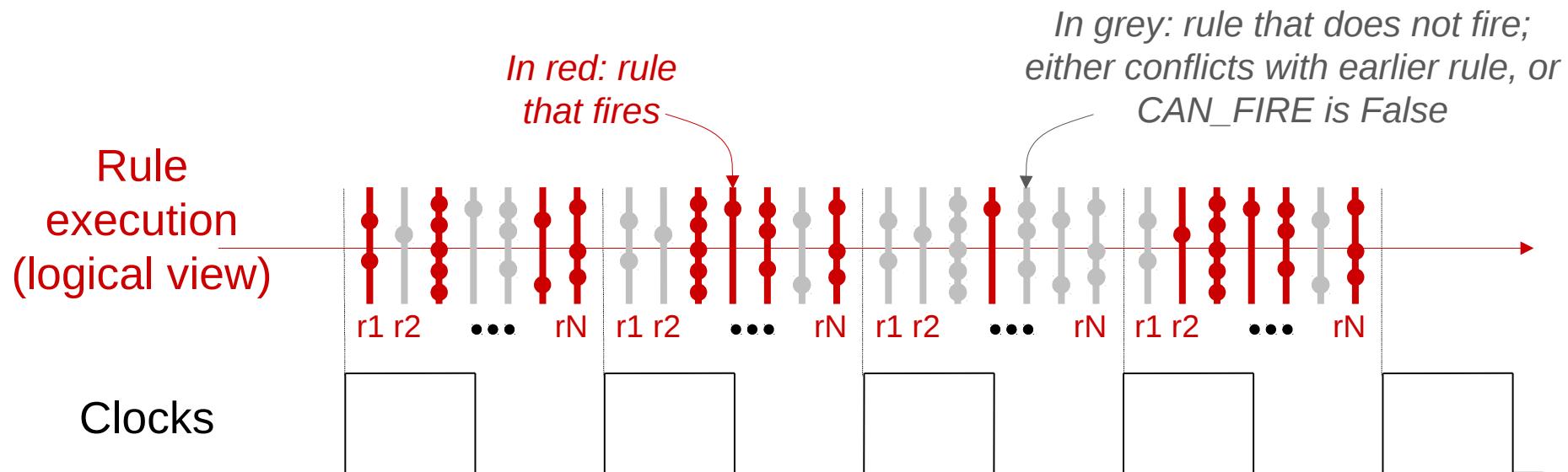
Consider each rule *in order*

If rJ does not conflict with earlier rules (r1..rJ-1)

Execute rJ according to the per-rule semantics

(i.e., if its CAN\_FIRE is true, do its actions)

(we'll discuss conflicts shortly)



# Method Ordering Constraints (inducing *conflicts*)

Consider two rules rule1 and rule2, in that order.



Consider two method calls x.mA in rule1, and x.mB in rule2, on a common module x. These methods can be either in the condition or body or rule1 and rule2.

For every module (x), the compiler knows certain ordering constraints on its methods:

Constraint	Meaning
mA <i>conflict_free</i> mB	Rules invoking mA and mB can fire concurrently (either order)
mA < mB	Rules invoking mA and mB can fire concurrently, provided the rule invoking mA is earlier than the rule invoking mB
mB < mA	Rules invoking mA and mB can fire concurrently, provided the rule invoking mB is earlier than the rule invoking mA
mA <i>conflict</i> mB	Rules invoking mA and mB cannot fire concurrently (neither order)

For primitive modules the ordering constraints on its interface methods are built-in (“given”).

For a user-defined module, the ordering constraints on its interface methods are inferred by *bsc*, the compiler, when the module is compiled.

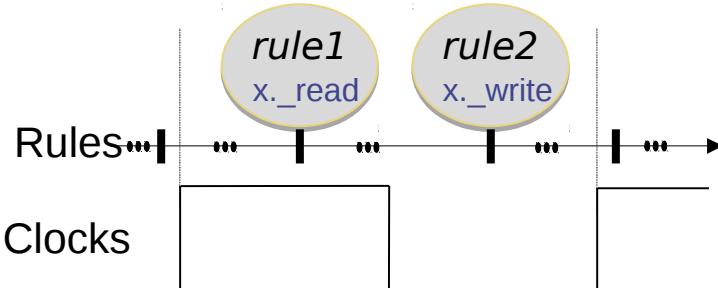
# Example: method ordering constraints on registers

For the mkReg register primitive, a method ordering constraint is: `_read < _write`

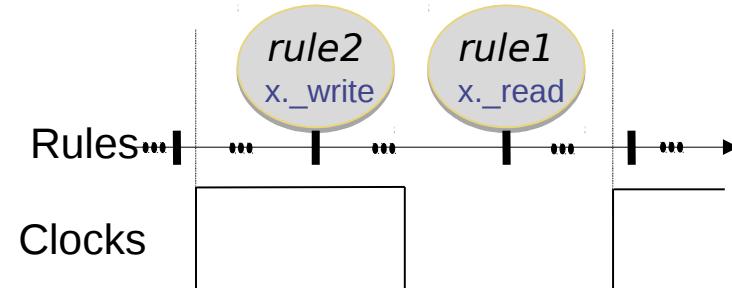
(One can see that this maps easily into hardware: during a clock, we can only read the old value (from the last clock edge) and when we write, it is only visible after the next clock edge.)

These constraints may result in a conflict for some rule schedules.

*No conflict: rule order is consistent with method ordering constraint (and maps naturally to HW semantics)*



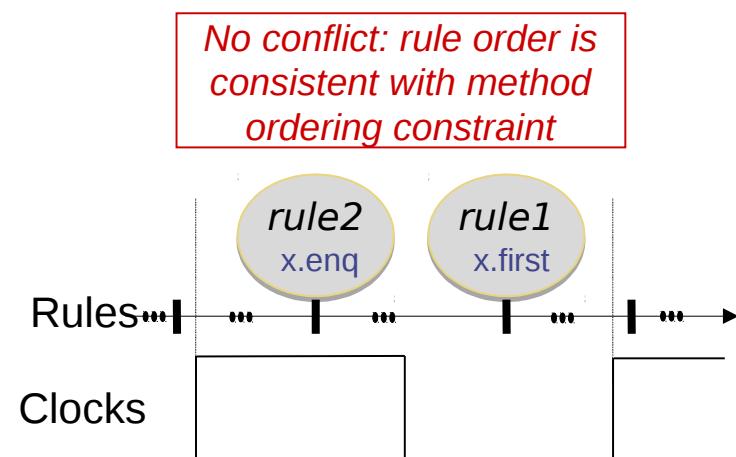
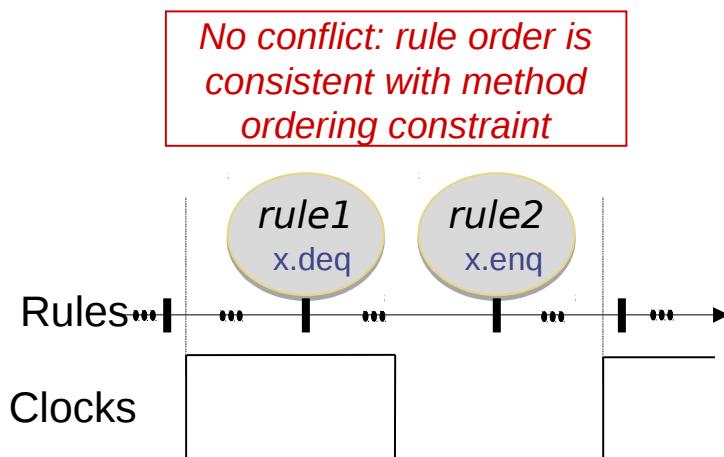
*Conflict: rule order is inconsistent with method ordering constraint*



# Example: method ordering constraints for FIFOs

The mkFIFO primitive has this method ordering constraint:

{deq, first} *conflict\_free* enq

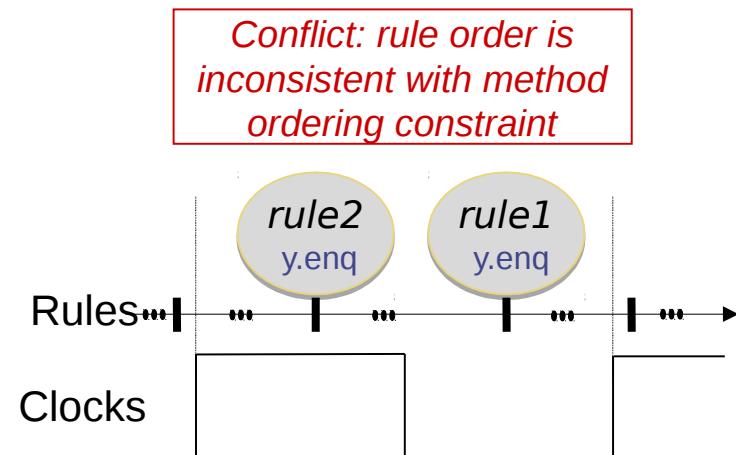
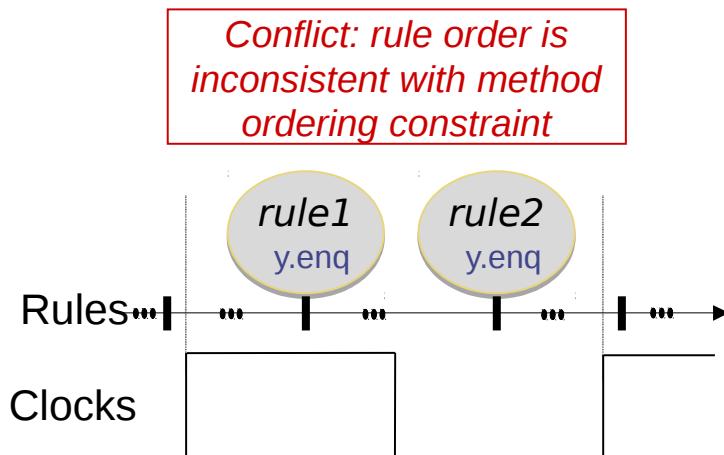


# Example: method ordering constraints for FIFOs

The mkFIFO primitive also has this method ordering constraint:

enq *conflict* enq

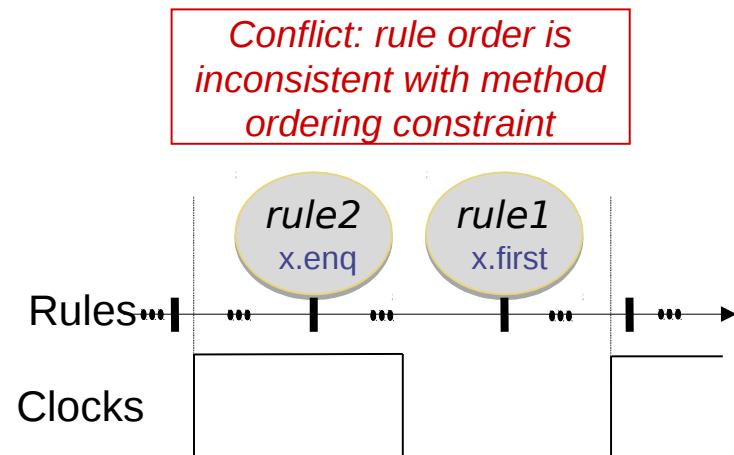
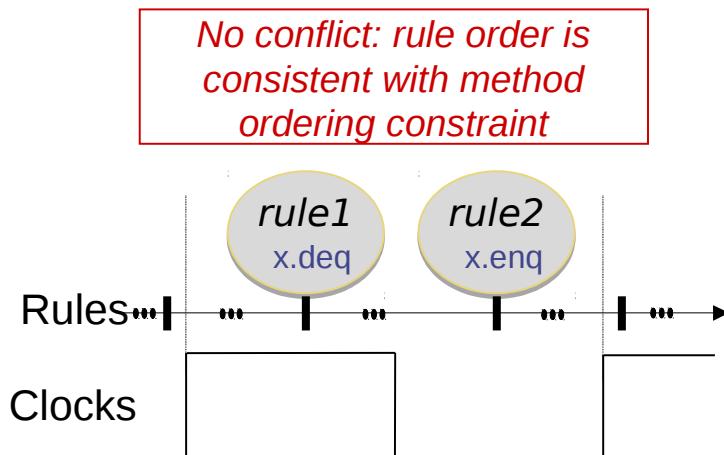
(Again, one can imagine the hardware considerations underlying this constraint: in a single-port FIFO, at most one item can be enqueued in each clock.)



# Example: method ordering constraints for FIFOs

The mkPipelineFIFO primitive (which provides the same interface as mkFIFO, but has a different implementation) has this method ordering constraint:

```
{deq, first} < enq
```

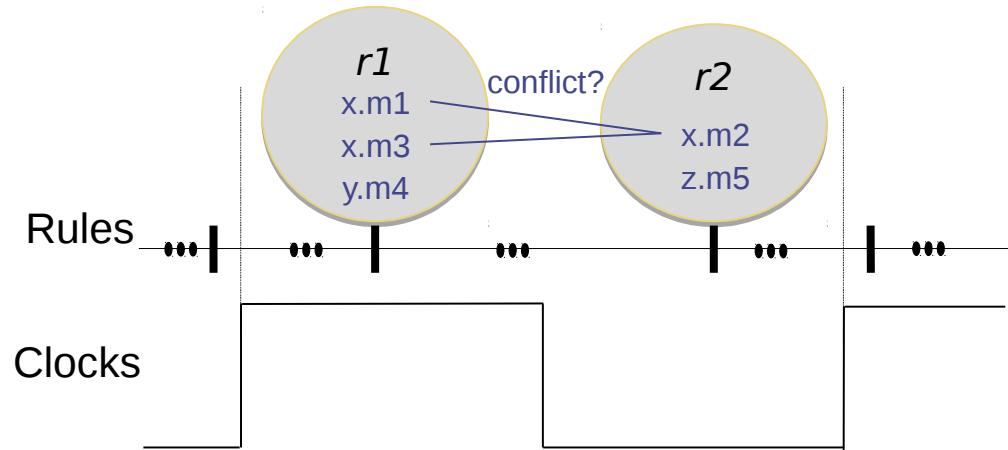


# From method ordering to rule ordering

In the past few slides, we have focused on some particular pair of methods in two rules that are arranged in some particular order.

Of course, rules typically invoke several methods, often on several sub-modules.

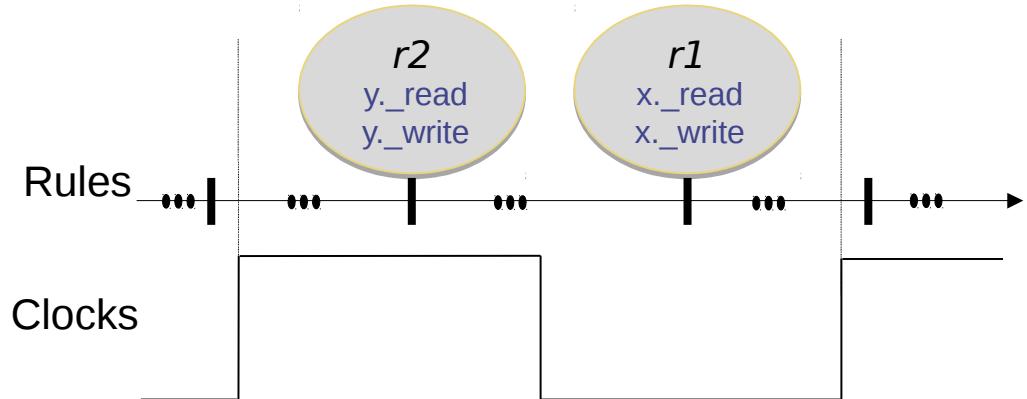
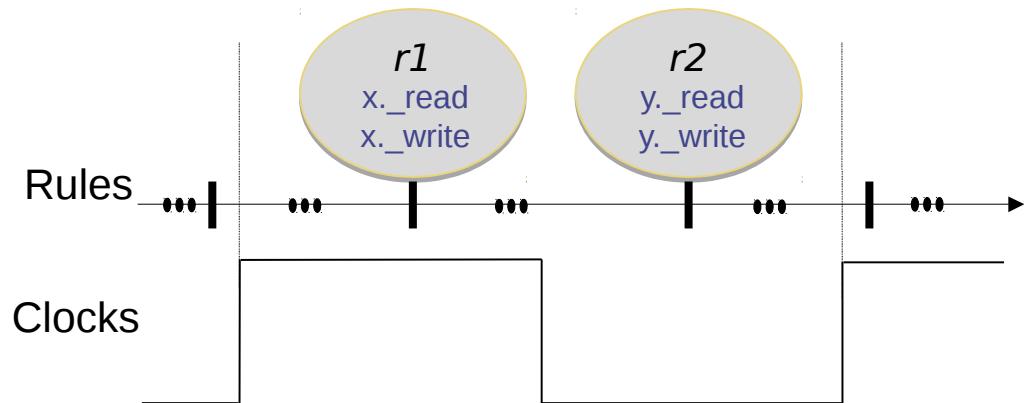
We say that there is a conflict between an ordered pair of rules  $r_1$  and  $r_2$  if there is a conflict between *any* pair of methods  $x.m_1$  in  $r_1$  and  $x.m_2$  in  $r_2$ .



Note: conflicts and orderings only arise between methods of the *same* module (like  $x.m_1$  and  $x.m_2$ ), not between methods of different modules (like  $y.m_4$  and  $x.m_2$ )

# Example: no conflict

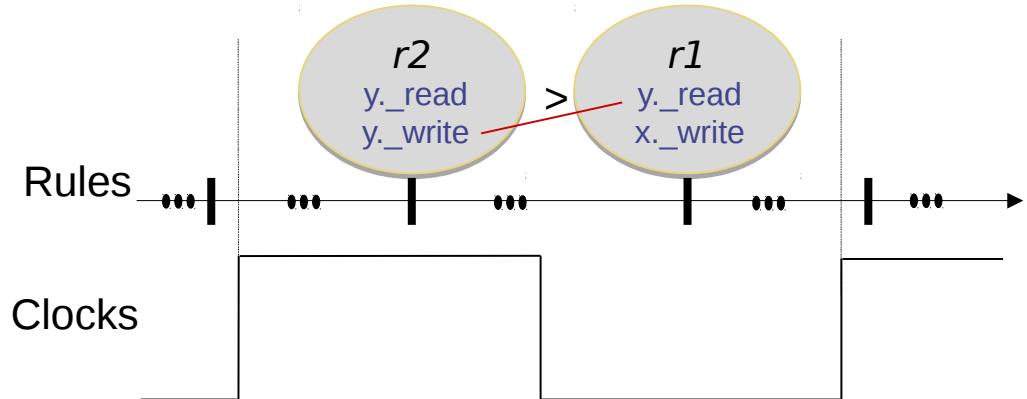
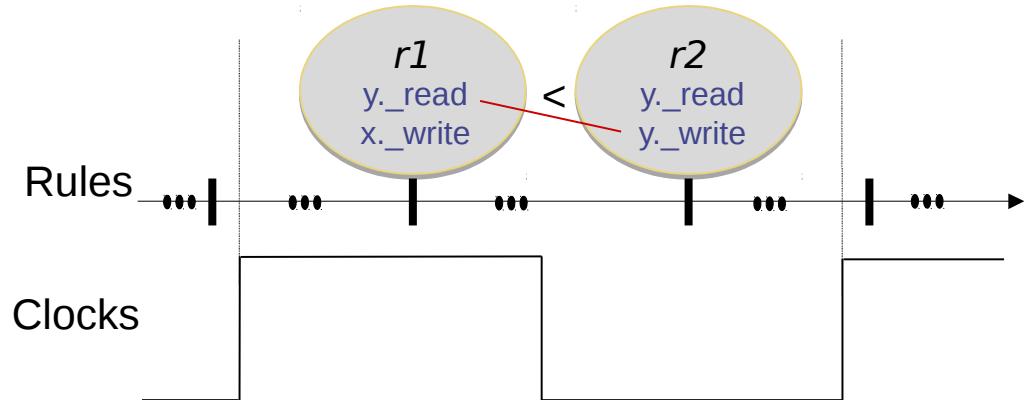
```
rule r1;  
    x <= x + 1;  
endrule  
  
rule r2;  
    y <= y + 2;  
endrule
```



There are no constraints between a method of *x* and method of *y*, so both rule orderings are legal (no conflict).

# Example: potential conflict

```
rule r1;  
    x <= y + 1;  
endrule  
  
rule r2;  
    y <= y + 2;  
endrule
```

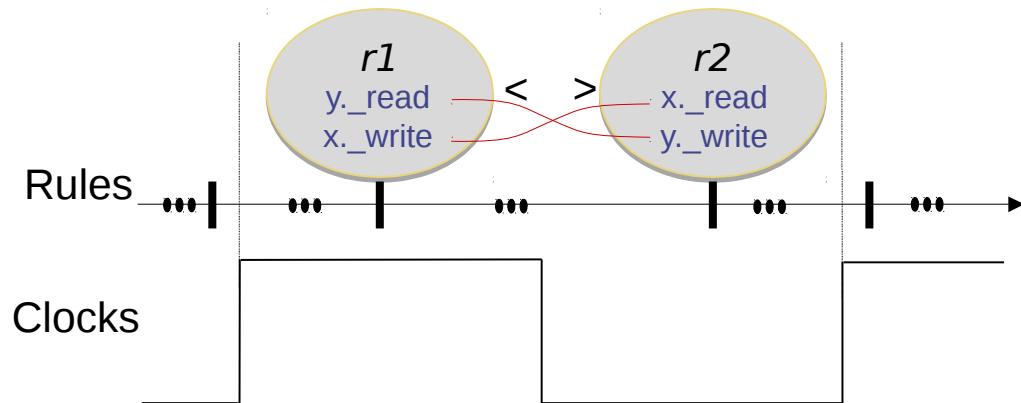


The only relevant constraint is: *y.\_read* < *y.\_write*  
(there are no constraints between methods of different registers *x* and *y*).

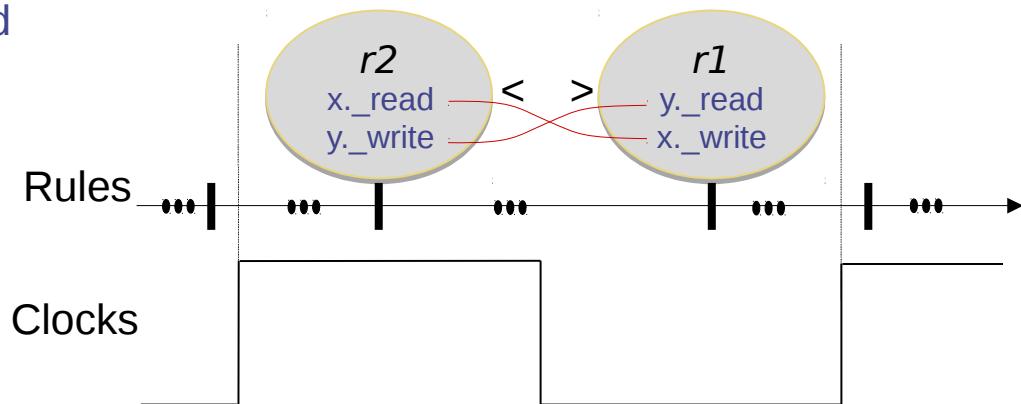
The upper ordering (or schedule) is consistent with this (no conflict).  
The lower ordering (or schedule) is inconsistent with this (has a conflict).

# Example: conflict

```
rule r1;  
    x <= y + 1;  
endrule  
  
rule r2;  
    y <= x * 2;  
endrule
```



In both possible orderings, a method constraint is violated. This is a conflict.

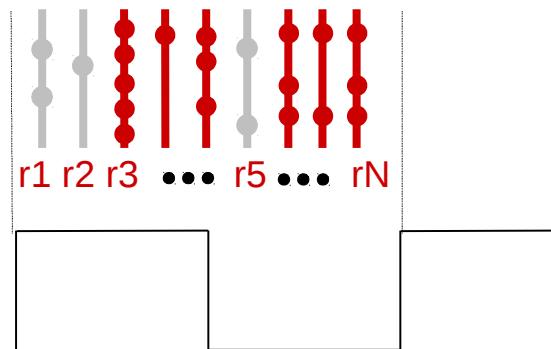


Compare slide 8: we had the same two actions in a single rule, and it was ok!  
This illustrates that some Actions may be simultaneous (in a single rule)  
but not concurrent (logically sequential in two different rules).

# For conflicts, the earlier rule disables the later rule

A schedule (i.e., a particular linear ordering of rules) can contain conflicts.

A conflict between two rules in a schedule just means that the latter rule must be suppressed (not allowed to fire) if the earlier one fires.



Suppose the compiler has picked this schedule of rules.

In this schedule, suppose there is a conflict between r3 and r5.

Then it generates logic like this: `WILL_FIRE_r5 = (! WILL_FIRE_r3) && CAN_FIRE_r5`

i.e., on those cycles that r3 fires, r5 is disabled.

Rule r5 can fire only on those cycles when r3 does not fire.

# Choice of schedule

Recall:

Define a *schedule* as some linear ordering of all rules in a program: r1 r2 ... rN

There are  $N!$  (factorial N) possible linear orderings of all the rules: which one does bsc pick?

bsc chooses a schedule trying to maximize rule concurrency (minimize conflicts).

This is based on a sophisticated analysis of the rules, the methods they invoke, the ordering constraints on the methods, and the boolean expressions representing rule conditions, method conditions, conditional statements and conditional expressions.

Occasionally you will see a compile-time informational message from bsc when it has to make an “arbitrary” ordering choice between two rules.

# Logical semantics and implementations

We have described Rule Semantics purely in BSV source-code terms, i.e., independent of any particular implementation, whether in Bluesim, Verilog simulation, or actual silicon.

*This is how we always think about BSV programs, and this is how we debug them.*

This separation of *logical* view and *implementation* view is present throughout Computer Science and Engineering. Example:

- *Logical* view: an assembly language (x86, ARM, ... your favorite processor). Semantics are defined as a sequential execution, 1 instruction at a time. Also, this is how we debug, for example using gdb.
- *Implementation* view: what happens in a real CPU: pipeline parallelism, out-of-order execution, superscalar execution, branch prediction and speculation, ... Further, these techniques will vary from one implementation to the next.

Similarly, rule orderings may not be evident in the Verilog generated by *bsc* from a BSV program. However, the implementation behavior will be identical to the logical semantics.

# Rule semantics summary

All rules in a program are considered in a linear order called “the schedule”

On each clock, a rule fires if its CAN\_FIRE condition is true, and if it does not conflict\* with any earlier rule.

When a rule fires, it is logically at a single instant. All actions within the rule take place at that same instant.

(\*) A conflict between two rules exists if the rule order violates any ordering constraint between methods called in the two rules.

## Controlling rule scheduling

# Controlling rule scheduling

Recall:

There are  $N!$  (factorial  $N$ ) possible linear orderings of all the rules: which one does *bsc* pick?

*bsc* chooses an ordering trying to maximize rule concurrency (minimize conflicts).

The user can influence *bsc*'s choices with various *attributes* to control scheduling

- BSV attribute syntax (same as SystemVerilog) : *(\* attribute = “rule and method names” \*)*
- These are written in a module, typically just before the rules mentioned in the attribute.
- Since methods are just rule fragments, these attributes can mention methods as well

# Controlling scheduling: rule urgency

```
(* descending_urgency = "r1, r2" *)  
  
rule r1 (c1);  
    fifo.enq (e1); // one enq per cycle  
endrule  
  
rule r2 (c2);  
    fifo.enq (e2); // one enq per cycle  
endrule
```

- Urgency is the order/priority in which WILL\_FIRE conditions are computed
- In this example, r1 and r2 conflict because of 'fifo.enq()' If r1 WILL\_FIRE, we will suppress r2 (even if it CAN\_FIRE)
- If the user does not specify urgency between conflicting rules, the compiler will pick an urgency order and notify the user

# Controlling scheduling: rule preempts

```
(* preempts = "r1, r2" *)  
  
rule r1 (upA);  
    x <= x + 3;  
endrule  
  
rule r2;  
    y <= y + 1;  
endrule
```

- This forces one rule's firing to suppress another rule's firing even if there is no conflict between the rules
  - This is equivalent to *forcing* a conflict between two rules
- In this example, whenever r1 fires the scheduler will suppress r2 (even if it CAN\_FIRE)
  - For example, here y effectively counts “idle cycles” of r1

# Controlling scheduling: rules mutually exclusive

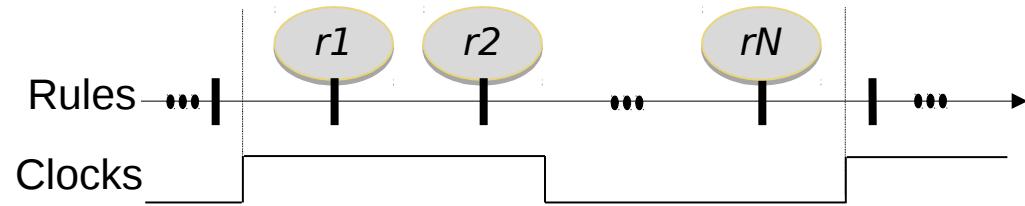
```
(* mutually_exclusive = "updateBit0, updateBit1" *)  
  
rule updateBit0 (oneHotNumber[0] == 1);  
    x[0] <= 1;  
endrule  
  
rule updateBit1 (oneHotNumber[1] == 1);  
    x[1] <= 1;  
endrule
```

- Asserts to the compiler that two rules' conditions are mutually exclusive (will never simultaneously be true in any clock)
- When rules are mutually exclusive, the compiler can generate better HW
  - E.g., simple muxes instead of priority muxes
- The compiler does sophisticated Boolean analysis to try to prove that two rule conditions are mutually exclusive
  - However, this question is undecidable in general. E.g.,
    - The conditions depend on external inputs
    - Mutual exclusivity depends on application-specific semantic knowledge (such as “one-hotness” of a bit-vector)
  - This assertion helps the compiler, when it is unable detect mutual exclusivity
  - For simulation, the compiler also generates code to verify mutual exclusivity

The last topic in this lecture (subtle distinction between “urgency” and “earliness”) can be skipped on first reading.  
It is a subtlety that matters only rarely.

# A refinement on rule ordering: urgency and earliness

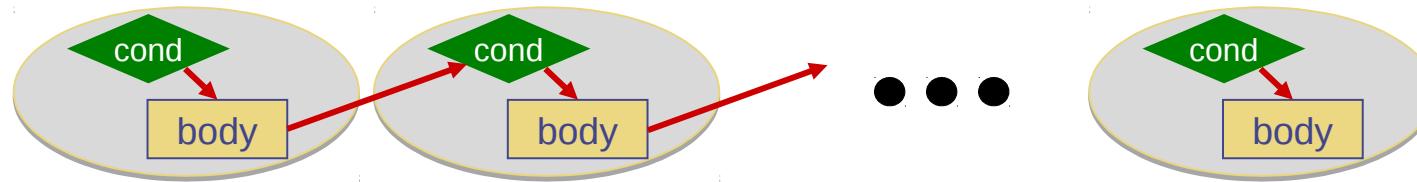
Consider a set of concurrent rules (rules enabled because their rule conditions are true, and that fire in a clock in a logical order):



Executing each rule involves two separate activities:

- evaluate its condition
- evaluate its body

Thus, when executing a rule sequence, we're conceptually alternating these activities:



But note:

- Rule conditions are pure boolean expressions. They have no side effects. Thus, just evaluating the rule condition of rule  $rA$  can never affect another rule  $rB$  (neither evaluating  $rB$ 's condition, nor what  $rB$ 's body does).
- Many rule bodies do not affect other rule conditions

Thus, we can reorder condition and body evaluations (provided we preserve required orderings)

Thus, rule orderings can be refined into two orderings:

- *Urgency order*: order in which we evaluate rule conditions
- *Execution order*: order in which we evaluate rule bodies (technically this is original rule ordering), also called *Earliness*

# Controlling scheduling: rule execution order

```
(* execution_order = "r1, r2" * )  
  
rule r1;  
    x <= 5;  
endrule  
  
rule r2;  
    y <= 6;  
endrule
```

- Forces an ordering in the logical semantics
  - Execution order is also called “earliness”
- In this example, forces “r1 < r2”

# Urgency and Execution orders may be different

```
(* descending_urgency="enq_item, enq_bubble" *)
rule enq_item;
    outfifo.enq(infifo.first); infifo.deq;
    bubbles <= 0;
endrule

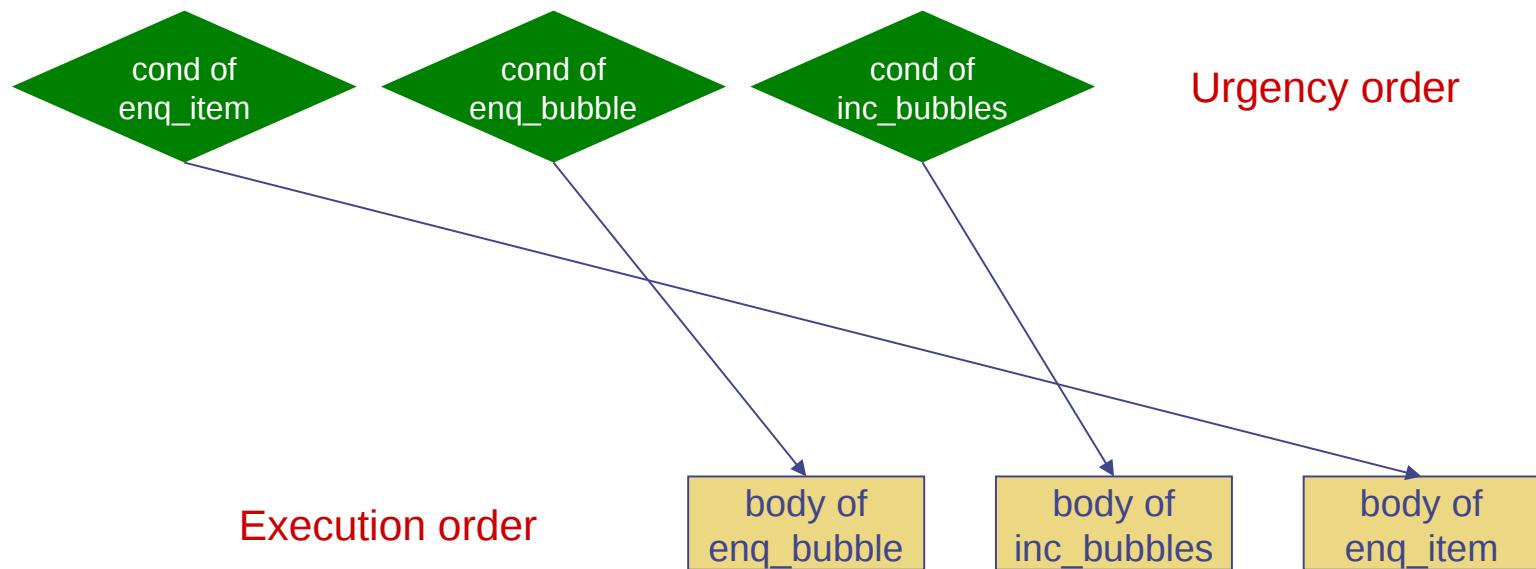
rule inc_bubbles;
    bubbles <= bubbles + 1;
endrule

rule enq_bubble;
    outfifo.enq(bubble_value);
    max_bubbles <= max (max_bubbles, bubbles);
endrule
```

- This code enqueues items from the infifo into the outfifo, if available; otherwise, it enqueues a ‘bubble\_value’
- It also computes the maximum stretch of bubbles
- The execution order is: `enq_bubble < inc_bubbles < enq_item` because reads of ‘bubbles’ must precede writes of ‘bubbles’
- However, we have forced the urgency to be: `enq_item < enq_bubble`

# Urgency and Execution orders may be different

Pictorially:



# Hands-on

- BSV-by-Example book: Examples in Chapter 7



End

# Questions?

Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

# BSV Training

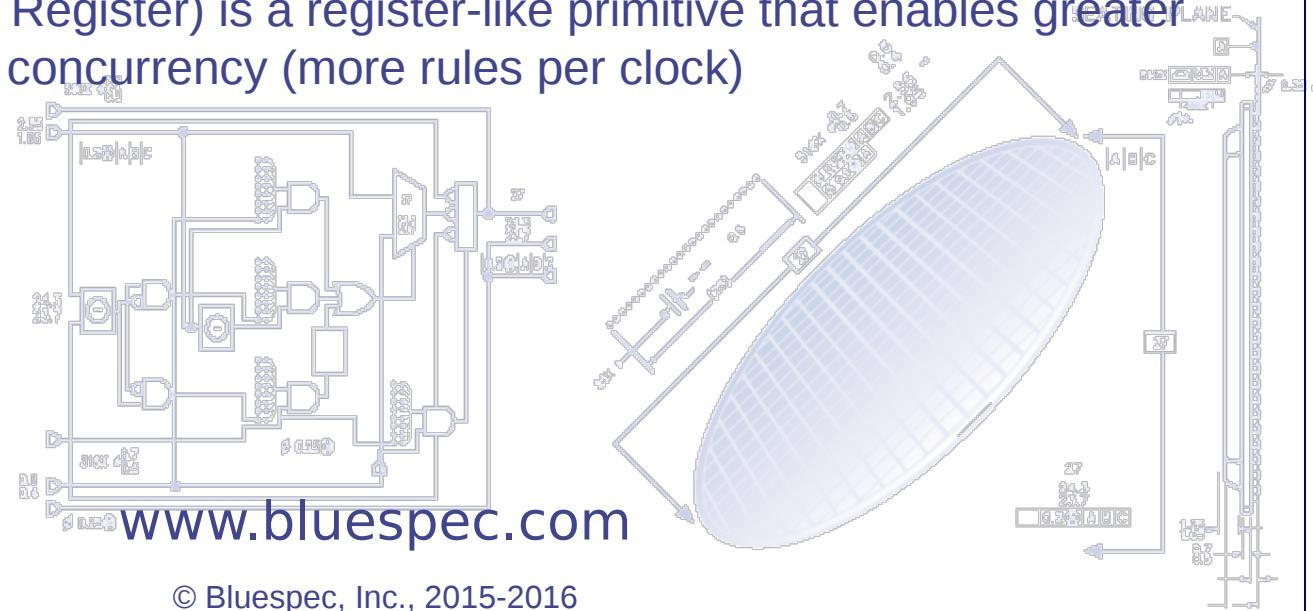
## Lec\_CRegs

A CReg (Concurrent Register) is a register-like primitive that enables greater concurrency (more rules per clock)

```

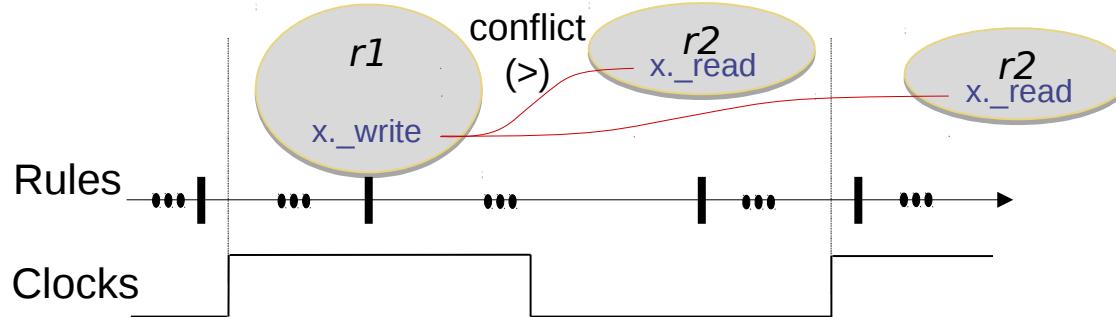
import FIFO#(T);
typedef FIFO#(T) Tmplt;
module mkCReg#(T)(T)
  Integer fifo_depth = 15;
  function UInt#(1) determine_pwr(Bit#(T));
    return (T == 1) ? 1 : 2;
  endfunction
  FIFO#(T) fifo();
  rule add (true);
    Bit#(T) b;
    if (fifo_depth >= 1) begin
      Tmplt t;
      t.enq(b);
    end
  endrule
endmodule

```



# Enabling greater concurrency

With a primitive module like mkReg, the effect of a rule's Action (`_write`) is not visible until the next clock (via `_read`), because of its method ordering constraint (`<`)



For greater rule concurrency, we need another primitive whose method ordering constraints allow an Action's effect to be visible in the same clock.

In BSV, we use a primitive called the `CReg`<sup>1</sup>.

<sup>1</sup> “CReg” = Concurrent Register. These are based on the “Ephemeral History Register” which was researched by Daniel Rosenband at MIT in 2004.

# A motivating example

- Suppose we want to build a two-port, saturating, up/down counter of 4-bit signed integer values, with the following interface:

```
interface UpDownSatCounter_Ifc;  
    method ActionValue #(Int #(4)) countA (Int #(4) delta);  
    method ActionValue #(Int #(4)) countB (Int #(4) delta);  
endinterface
```

- The “two ports” are the two identical methods countA and countB
- A module implementing this interface has internal state holding the current value of the counter (Int #(4) type, so range is -8 to +7)
- When either method is called,
  - The internal state is incremented by delta (range: -8 to +7), but saturates at +7 on overflow and at -8 on underflow
  - The old value of the counter is returned as the result of the method

*Note: because of finite precision and saturation, “count” operations are not commutative like in conventional arithmetic; so, the order of these operations matters here!*

# An implementation using ordinary registers (v1)

```
module mkUpDownSatCounter (UpDownSatCounter_Ifc);
  Reg #(Int #(4)) ctr <- mkReg (0);

  function ActionValue #(Int #(4)) fn_count (Int #(4) delta);
    actionvalue
      // Extend the precision to avoid over/under flows
      Int #(5) new_val = extend (ctr) + extend (delta);
      if (new_val > 7) ctr <= 7;
      else if (new_val < -8) ctr <= -8;
      else ctr <= truncate (new_val);

      return ctr; // note: returns old value
    endactionvalue
  endfunction

  method countA (Int #(4) deltaA) = fn_count (deltaA);
  method countB (Int #(4) deltaB) = fn_count (deltaB);
endmodule
```

Since both methods do the same thing, we abstract their common behavior into a function fn\_count()

BSV notes:

- “extend (e)” sign-extends for Int#(n), and zero-extends for Bit#(n) and UInt#(n)
- “truncate (e)” drops MSBs, taking care of sign bits etc.
- The number of bits extended/truncated depends on the input and output type widths

# A testbench to drive the up/down counter module

```
module mkTest (Empty);
    UpDownSatCounter_Ifc ctr <- mkUpDownSatCounter;
    Reg #(int) step <- mkReg (0);
    Reg #(Bool) flag0 <- mkReg (False); Reg #(Bool) flag1 <- mkReg (False);

    function Action count_show (Integer rulenumber, Bool a_not_b, Int #(4) delta);
        action
            let x <- (a_not_b ? ctr.countA (delta) : ctr.countB (delta));
            $display ("cycle %0d, r%0d: is %0d, count (%0d)", cur_cycle, rulenumber, x, delta);
        endaction
    endfunction

    // Rules 0-9 are sequential, just testing one method at a time
    rule r0 (step == 0); count_show (0, True, 3); step <= 1; endrule
    rule r1 (step == 1); count_show (1, True, 3); step <= 2; endrule
        ... and similarly, sequentially feed deltas of 3,3, -6,-6,-6, -6, 7, 3,
    // Concurrent execution
    rule r10 (step == 10 && !flag0); count_show (10,True, 6); flag0 <= True; endrule
    rule r11 (step == 10 && !flag1); count_show (11,False, -3); flag1 <= True; endrule

    // Show final value
    rule r12 (step == 10 && flag0 && flag1); count_show (12,True, 0); $finish; endrule
endmodule: mkTest
```

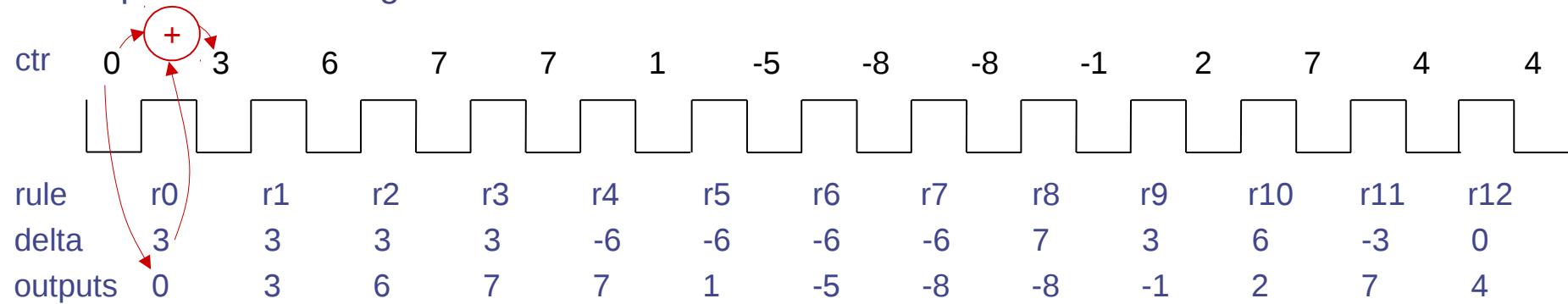
In rules 0-9, we call either countA or countB with deltas: 3,3,3,3, -6,-6,-6,-6, 7, 3  
The rule conditions and step assignments force them to fire 1 rule per clock (and so it doesn't matter whether we call countA or countB in these rules).

Rules 10 and 11 could potentially fire concurrently (if scheduling permits).

Rule 12 just displays the final counter value and exits.

# Expected behavior and outputs for v1

We expect the following behavior if r10 fires in the clock before r11:



We expect the following behavior if r11 fires in the clock before r10:



# Actual output for v1

When we compile the program (v1), bsc produces the following message:

```
Warning: "Test.bsv", line 16, column 8: (G0010)
  Rule "r10" was treated as more urgent than "r11". Conflicts:
    "r10" cannot fire before "r11": calls to ctr.countA vs. ctr.countB
    "r11" cannot fire before "r10": calls to ctr.countB vs. ctr.countA
```

This is saying:

- r10 and r11 conflict; they cannot be scheduled in the same clock  
(countA and countB conflict because they both read and write the “ctr” register inside mkUpDownSatCounter, thus both possible rule orders will violate a “\_read < \_write” ordering constraint)
- bsc has chosen to give priority to r10, i.e., if both r10 and r11 are enabled in the same clock, the scheduling logic will allow r10 to fire and will suppress r11  
(r11 could fire, and indeed it does, in the next clock, when r10 is no longer enabled)
- Note: you can force the opposite priority by adding a “descending\_urgency” attribute to the module

When we run the program (v1), we see:

(per first schedule in previous slide)

```
cycle 1, r0: is 0, count (3)
cycle 2, r1: is 3, count (3)
cycle 3, r2: is 6, count (3)
cycle 4, r3: is 7, count (3)
cycle 5, r4: is 7, count (-6)
cycle 6, r5: is 1, count (-6)
cycle 7, r6: is -5, count (-6)
cycle 8, r7: is -8, count (-6)
cycle 9, r8: is -8, count (7)
cycle 10, r9: is -1, count (3)
cycle 11, r10: is 2, count (6)
cycle 12, r11: is 7, count (-3)
cycle 13, r12: is 4, count (0)
```

2 cycles

## v1 is not really a “2-port” counter

v1 of our mkUpDownSatCounter may be functionally correct, but it's hardly a “2-port” counter!

When we say “2-port”, we are making a performance characterization, i.e., we expect both ports to be operable in the same clock.

For this, we need to replace the Reg in mkUpDownSatCounter with an CReg, a different primitive that allows “multiple reads and writes” within a clock.

# First: specifying the semantics of the two ports

Before we worry about implementations and CRegs, we must first specify the *desired semantics* of the two ports! Specifically:

When both countA and countB are operated in the same clock,

- what should be the final value of the counter?
- what should be the “old” values returned by each method?

In light of the finite precision arithmetic, and the saturating behavior, there is no obvious unique answer! It is a design choice!

In RTL designs, this is typically where you'll see an *ad hoc* choice made by the designer

- Which is (hopefully!) implemented correctly
- Which is (hopefully!) documented clearly and fully in English text in the datasheet
- Which may contain usage rules the user of the IP must follow, and which therefore need verification

In BSV, method orderings give us a formal and precise way to specific the semantics. By specifying that we want “countA < countB” or “countA > countB”, we give precise answers to the above two semantic questions, because when operated in the same clock, there is a well-defined *logical* ordering that specifies the behavior exactly.

Further, *bsc* always verifies correct usage because it's in the semantics, not *ad hoc* English.

# CRegs (Concurrent Registers)

A CReg provides a *vector* of standard Reg interfaces that can be operated concurrently:

```
interface CReg #(numeric type n, type t);
    interface Vector #(n, Reg #(t)) ports;
endinterface
```

BSV notes:

- “Vector” is a standard importable BSV library
- The parameter n is the number of elements in the vector; t is the data type stored in the CReg

The ports of an CReg can be operated concurrently, with the following ordering constraints:

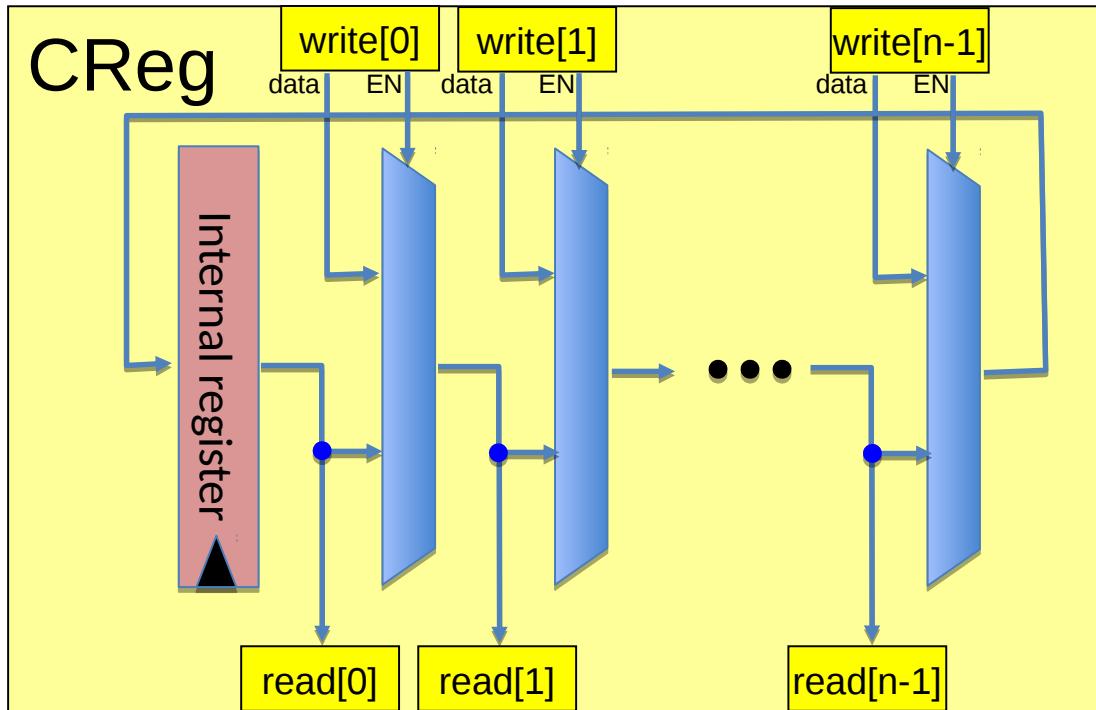
```
ports [0]._read <= ports [0]._write <
ports[1]._read <= ports[1]._write <
ports[2]._read <= ports[2]._write <
...
ports[n-1]._read <= ports[n-1]._write
```

*This is the same as the standard register method-ordering constraint*

*But note that a value written in port 0 can be read concurrently on port 1 (by a logically later rule in the same clock), unlike an ordinary register where a write can only be read in the next clock*

# A possible implementation of an CReg

This figure shows a possible circuit implementation of a CReg:



But note, this is not a *definition* of an CReg, it is merely shown to strengthen intuition. It is important, as usual, to keep separate the logical semantics from any implementation semantics. When using CRegs in BSV, one only needs to consider its method-ordering constraints (shown in the previous slide).

# Implementing our counter using CRegs (v2)

```
module mkUpDownSatCounter (UpDownSatCounter_Ifc);
    CReg#(2, Int #(4)) ctr <- mkCReg(0); // 2 ports

    function ActionValue #(Int #(4)) fn_count (Integer p, Int #(4) delta);
        actionvalue
            // Extend the precision to avoid over/under flows
            Int #(5) new_val = extend (ctr.ports [p]) + extend (delta);
            if (new_val > 7) ctr.ports [p] <= 7;
            else if (new_val < -8) ctr.ports [p] <= -8;
            else ctr.ports [p] <= truncate (new_val);

            return ctr.ports [p]; // note: returns old value
        endactionvalue
    endfunction

    method countA (Int #(4) delta) = fn_count (0, delta);
    method countB (Int #(4) delta) = fn_count (1, delta);
endmodule
```

Change Reg to CReg

Add CReg port selections to reads and writes

For "countA < countB".

To implement "countB < countA", change to:

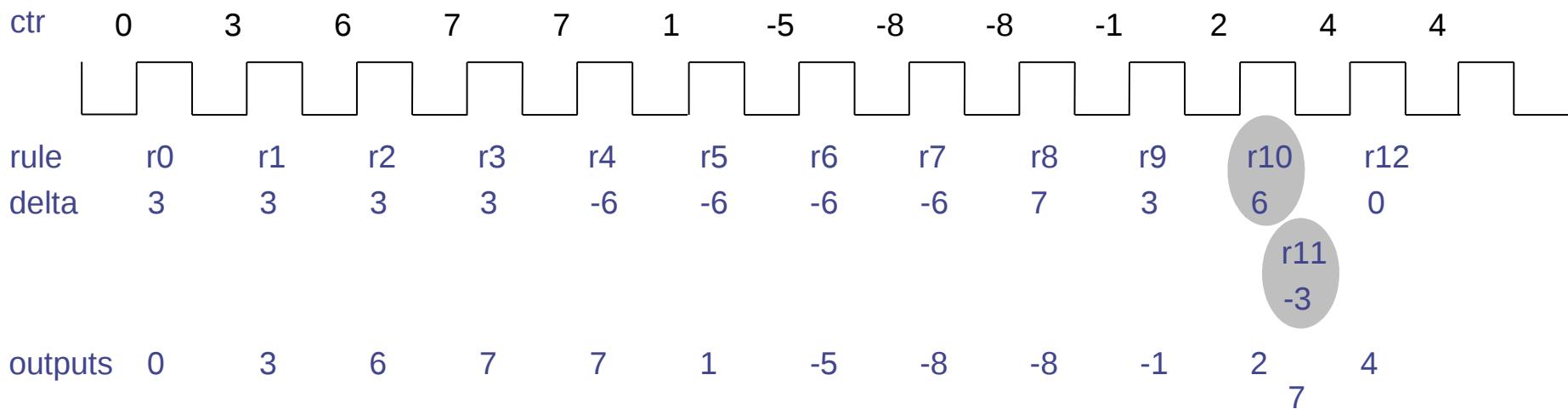
```
... = fn_count (1, delta);
... = fn_count (0, delta);
```

This is only a slight change to v1:

- The internal “ctr” is now a 2-port CReg instead of a Reg
- fn\_count is now parameterized by the CReg port “p” it should use
- countA and countB call this function with ports 0 and 1, respectively, thereby implementing the ordering semantics “countA < countB”

# Behavior and outputs for v2

We expect the following behavior ( $r_{10} < r_{11}$  in same clock):



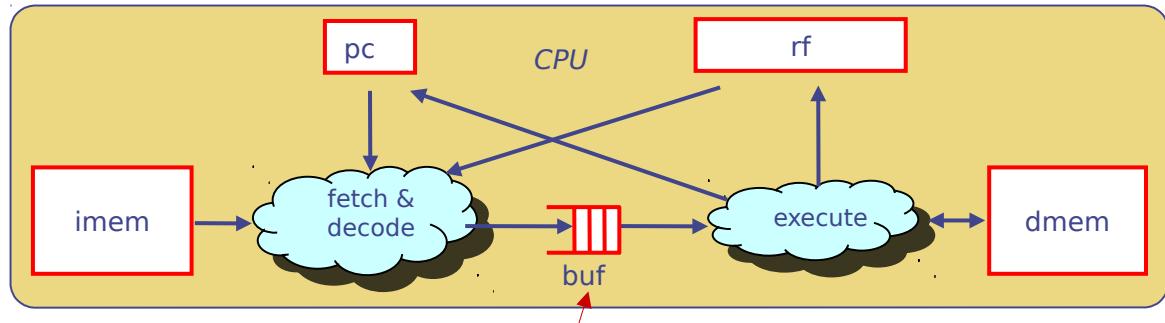
When we run the program (v2), we see:

```
cycle 1, r0: is 0, count (3)
cycle 2, r1: is 3, count (3)
cycle 3, r2: is 6, count (3)
cycle 4, r3: is 7, count (3)
cycle 5, r4: is 7, count (-6)
cycle 6, r5: is 1, count (-6)
cycle 7, r6: is -5, count (-6)
cycle 8, r7: is -8, count (-6)
cycle 9, r8: is -8, count (7)
cycle 10, r9: is -1, count (3)
cycle 11, r10: is 2, count (6)
cycle 11, r11: is 7, count (-3)
cycle 12, r12: is 4, count (0)
```

same cycle

# A second example

Consider a 2-stage CPU pipeline:



Let us focus on the FIFO connecting the stages:

Usually this is just a 1-element FIFO (we call it a “PipelineFIFO”).

a.k.a. “pipeline register with interlock” (the interlock is just the extra valid bit that allows the execute stage to stall if there is nothing in the pipeline register, and allows the fetch/decode stage to stall if there is already something in the register which has not been consumed by the execute stage).

# An implementation using ordinary registers (v1)

```
module mkFIFOF1 (FIFOF #(t));
    Reg #(t)           rg          <- mkRegU;      // data storage
    Reg #(Bit #(1))   rg_count   <- mkReg (0);  // # of items in FIFO (0 or 1)

    method Bool notEmpty = (rg_count == 1);
    method Bool notFull  = (rg_count == 0);

    method Action enq (t x) if (rg_count == 0); // can enq if not full
        rg <= x;
        rg_count <= 1;
    endmethod

    method t first () if (rg_count == 1); // can see first if not empty
        return rg;
    endmethod

    method Action deq () if (rg_count == 1); // can deq if not empty
        rg_count <= 0;
    endmethod

    method Action clear;
        rg_count <= 0;
    endmethod
endmodule
```

But: enq and {first, deq} could never be concurrent, with mutually exclusive conditions:  $rg\_count == 0$  and  $rg\_count == 1$

Implication  the fetch/decode stage and the execute stage in the 2-stage CPU pipeline could never execute in the same clock (it isn't really a pipeline!)

# First: specify desired semantics of concurrent methods

Before we worry about implementations, we must first specify the desired *semantics* of concurrency on FIFO methods. In BSV we commonly use the following two kinds of FIFOs:

## PipelineFIFOs:

- When empty, only enq is enabled
- When full, enq, first and deq are enabled, with:  $\{first,deq\} < enq$   
i.e., if both methods are enabled, logically it is like  $\{first,deq\}$  followed by enq,  
i.e., data currently in the FIFO is returned for  $\{first,deq\}$ , and new data is enqueueued.

## BypassFIFOs:

- When full, only  $\{first,deq\}$  is enabled
- When empty, enq, first and deq are enabled, with:  $enq < \{first,deq\}$   
i.e., if both methods are enabled, logically it is like enq followed by  $\{first,deq\}$ ,  
i.e., the newly enqueueued value is “bypassed” through to  $\{first,deq\}$ .

# An implementation of Pipeline FIFOs using CRegs

```
module mkPipelineFIFOF (FIFOF #(t));
    CReg #(3, t)          crg      <- mkCRegU;      // data storage
    CReg #(3, Bit #(1))  crg_count <- mkCReg (0); // # of items in FIFO

    method Bool notEmpty = (crg_count.ports[0] == 1);
    method Bool notFull  = (crg_count.ports[1] == 0);

    method Action enq (t x) if (crg_count.ports[1] == 0);
        crg.ports[1] <= x;
        crg_count.ports[1] <= 1;
    endmethod

    method t first () if (crg_count.ports[0] == 1);
        return crg.ports[0];
    endmethod

    method Action deq () if (crg_count.ports[0] == 1);
        crg_count.ports[0] <= 0;
    endmethod

    method Action clear;
        crg_count.ports[2] <= 0;
    endmethod
endmodule
```

This is only a slight change to v1:

- notEmpty, first and deq use CReg port 0
- notFull and enq use CReg port 1
- clear uses CReg port 2

# An implementation of BypassFIFOs using CRegs

```
module mkBypassFIFOF (FIFOF #(t));
    CReg #(3, t)          crg      <- mkCRegU;      // data storage
    CReg #(3, Bit #(1))  crg_count <- mkCReg (0); // # of items in FIFO

    method Bool notEmpty = (crg_count.ports[1] == 1);
    method Bool notFull  = (crg_count.ports[0] == 0);

    method Action enq (t x) if (crg_count.ports[0] == 0);
        crg.ports[0] <= x;
        crg_count.ports[0] <= 1;
    endmethod

    method t first () if (crg_count.ports[1] == 1);
        return crg.ports[1];
    endmethod

    method Action deq () if (crg_count.ports[1] == 1);
        crg_count.ports[1] <= 0;
    endmethod

    method Action clear;
        crg_count.ports[2] <= 0;
    endmethod
endmodule
```

This is only a slight change to v1:

- notFull and enq use CReg port 0
- notEmpty, first and deq use CReg port 1
- clear uses CReg port 2

# CReg summary

The CReg is a highly concurrent primitive, i.e., it has multiple methods that can be invoked by multiple rules within a clock in a well-defined logical sequential order.

When using a CReg to communicate between rules that you want to be concurrent (i.e., able to fire in the same clock),

- first, be clear about what semantics you want, by thinking about what logical ordering of rules you want
- then, use CRegs to implement that ordering
  - (ascending CReg port indexes directly correspond to ordering)

Note: a design using CRegs will be functionally correct with any schedule, even one rule per clock. In the extreme schedule of one rule per clock, an CReg is exactly equivalent to an ordinary register (using mkReg).

In practice, we more often directly use concurrent library modules like PipelineFIFO and BypassFIFO. If necessary, we use CRegs to implement a concurrent module that is not available in the library.

# Hands-on

- BSV-by-Example book: Examples in Chapter 8



End

# Questions?

Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

# BSV Training

## Lec\_Interfaces\_TLM

Transaction-Level Modeling (TLM)

Get/Put, Client/Server, interface transformers, mkConnection, Connectable typeclass

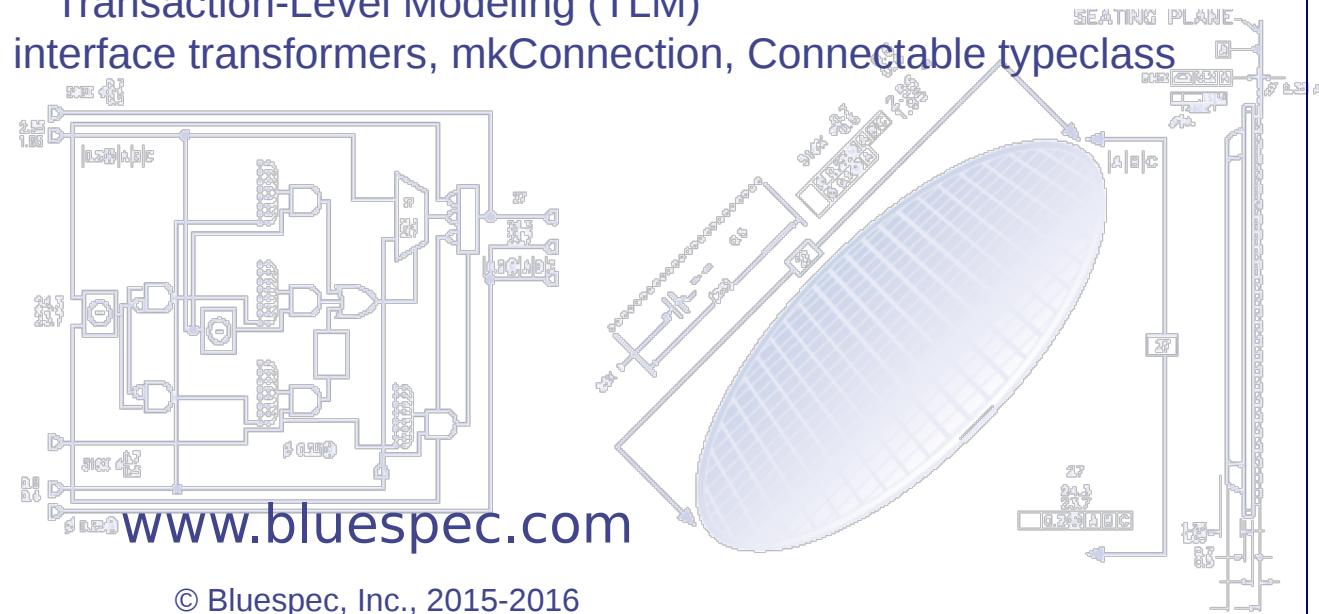
```

import PFR#(T);
typeif(Elm#(T) default);
  module end;
endinterface

integer fifo_depth = 15;
function UInt#(1) determine_psize(DataT);
  return {1,pw};
endfunction

interface Action
  method Action the_ifc.fifo.enq(DataT data);
    FIFO#(DataT) enqueue();
    action
      FIFO#(DataT) enq(data);
      FIFO#(DataT) enq(data);
      FIFO#(DataT) enq(data);
      FIFO#(DataT) enq(data);
      FIFO#(DataT) enq(data);
    endaction
  endaction
endinterface

```



# Introduction

- BSV's 'method' construct in interfaces allows you to define arbitrary methods for a module, specifying arguments and their types, and results and their types.
- However, as a matter of style, readability, documentation, succinctness and maintainability, we often reuse certain standard interfaces from the BSV library, such as Get/Put and Client/Server, along with standard definitions from the BSV library for connecting modules with such interfaces
- In the SystemC world, use of such stylized interfaces is often called TLM (for "Transaction Level Modeling"). It's the same idea here (BSV has had this capability since 2000!), and it's more than just for modeling—we use them routinely in production synthesized code.

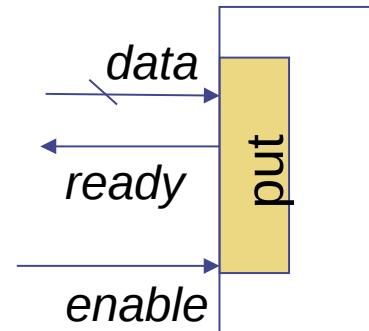
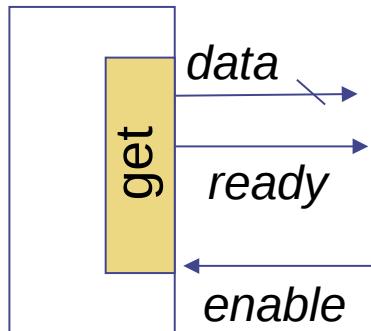
# Get/Put interfaces

Instead of defining *ad hoc* methods in interfaces, one often uses interfaces already defined in the BSV library that capture certain common design patterns. Example:

```
// For getting a value out of a module
interface Get#(type t);
    method ActionValue#(t) get();
endinterface: Get

// For putting a value into a module
interface Put#(type t);
    method Action put(t x);
endinterface: Put
```

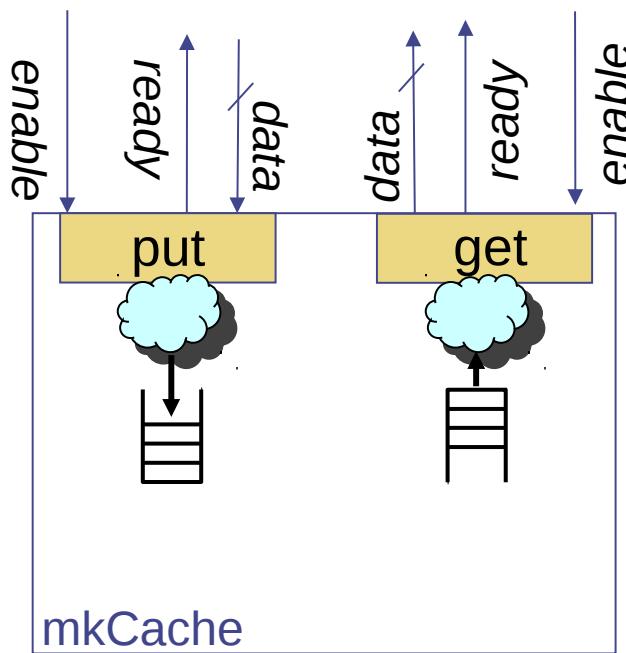
By themselves, these do not imply anything more than the interface wires and protocol; what's in the module depends on how you to define these methods.



# Get/Put example

An interface provided by a cache towards a processor

```
interface Cachelfc;
  interface Put#(Req_t) p2c_request;
  interface Get#(Resp_t) c2p_response;
  ...
endinterface
```



```
module mkCache (Cachelfc);
  FIFO#(Req_t) p2c <- mkFIFO;
  FIFO#(Resp_t) c2p <- mkFIFO;

  ... rules expressing cache logic ...

  interface p2c_request;
    method Action put (Req_t req);
      p2c.enq (req);
    endmethod
  endinterface

  interface c2p_response;
    method ActionValue#(Resp_t) get ();
      let resp = c2p.first; c2p.deq;
      return resp;
    endmethod
  endinterface

endmodule
```

# Standard interface transformers

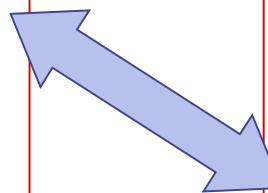
A FIFO ‘enq’ operation can be seen as a ‘put’ operation.

FIFO ‘first’ and ‘deq’ operations can be seen as a ‘get’ operation.

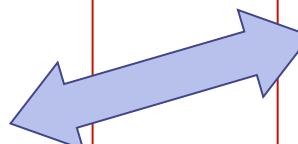
These ideas can be captured as functions that transform interfaces.

(These two examples already exist in the BSV library)

```
function Put#(Req_t) toPut (FIFO#(Req_t) fifo);
return (
  interface Put;
    method Action put (a);
      fifo.enq (a);
    endmethod
  endinterface);
endfunction
```



```
function Get#(Resp_t) toGet (FIFO#(Resp_t) fifo);
return (
  interface Get;
    method ActionValue#(Resp_t) get ();
      let a = fifo.first;
      fifo.deq;
      return a;
    endmethod
  endinterface);
endfunction
```



```
module mkCache (Cachefc);
  FIFO#(Req_t) p2c <- mkFIFO;
  FIFO#(Resp_t) c2p <- mkFIFO;

  ... rules expressing cache logic ...

  interface p2c_request;
    method Action put (Req_t req);
      p2c.enq (req);
    endmethod
  endinterface

  interface c2p_response;
    method ActionValue#(Resp_t) get ();
      let resp = c2p.first; c2p.deq;
      return resp;
    endmethod
  endinterface

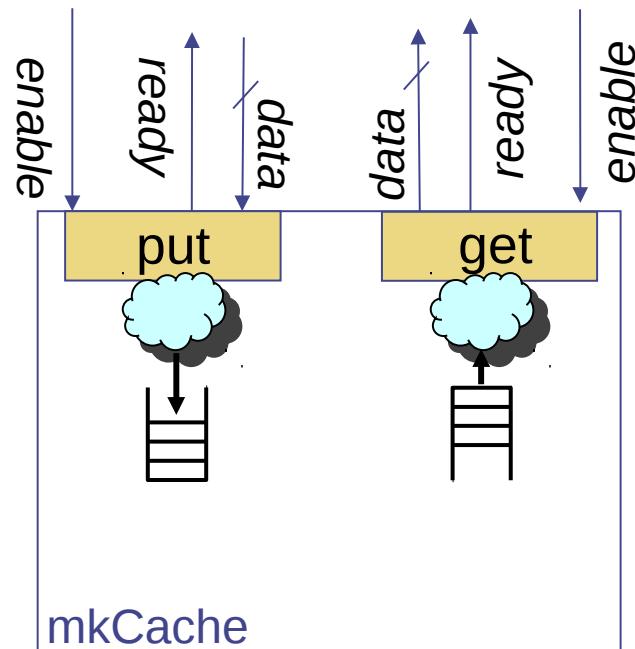
endmodule
```

# Using standard interface transformers

This simplifies the interface definition in the module.

There is no HW cost to this (it statically elaborates to the same HW).

```
interface Cachelfc;  
  interface Put#(Req_t) p2c_request;  
  interface Get#(Resp_t) c2p_response;  
  ...  
endinterface
```



```
module mkCache (Cachelfc);  
  FIFO#(Req_t) p2c <- mkFIFO;  
  FIFO#(Resp_t) c2p <- mkFIFO;
```

*... rules expressing cache logic ...*

```
interface p2c_request = toPut (p2c);  
  
interface c2p_response = toGet (c2p);  
endmodule
```

Note: toGet and toPut are actually overloaded functions from ToPut and ToGet typeclasses. The BSV library supplies instances for many interfaces, including FIFOs.

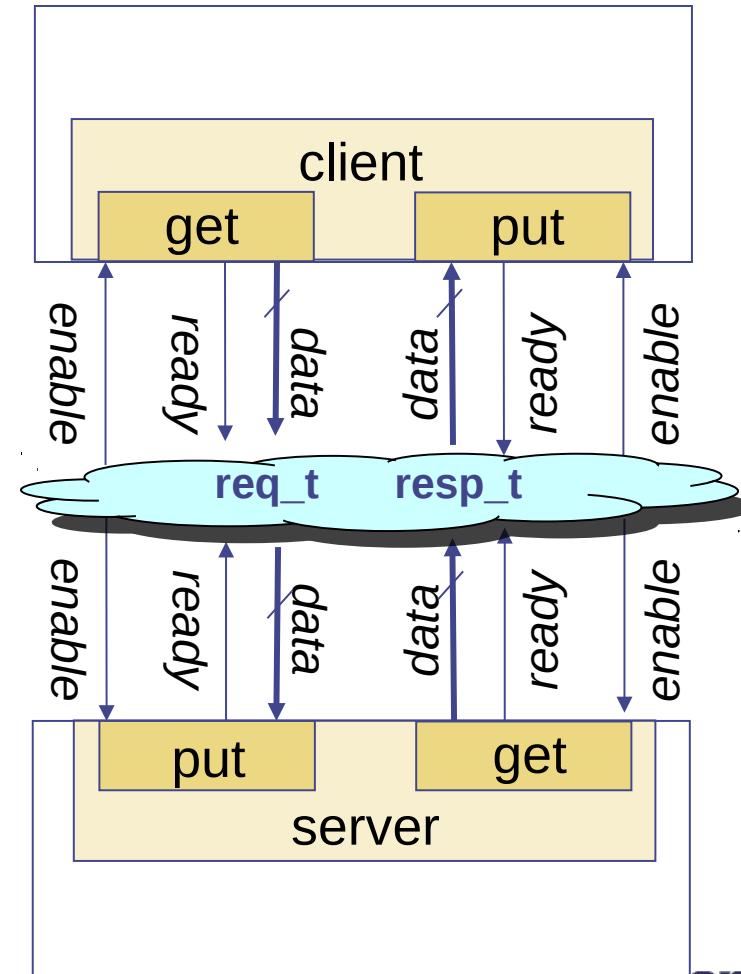
# Client/Server interfaces

Interfaces can be nested.

I.e., inside an interface declaration, instead of declaring methods, you can use an already-defined interface. Here is another example from the BSV library.

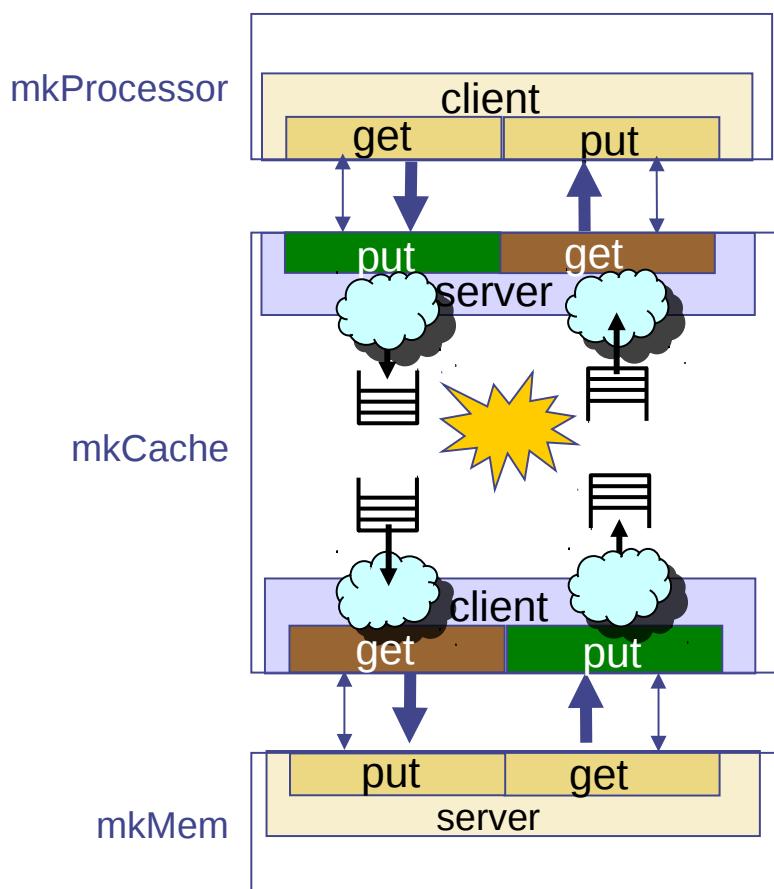
```
interface Client #(req_t, resp_t);
  interface Get#(req_t) request;
  interface Put#(resp_t) response;
endinterface
```

```
interface Server #(req_t, resp_t);
  interface Put#(req_t) request;
  interface Get#(resp_t) response;
endinterface
```



# Example: using Client/Server for the cache

```
interface Cachelfc;
  interface Server#(Req_t, Resp_t) ipc;
  interface Client#(Req_t, Resp_t)
    icm;
endinterface
```



```
module mkCache (Cachelfc);
  FIFO#(Req_t) p2c <- mkFIFO;
  FIFO#(Resp_t) c2p <- mkFIFO;

  FIFO#(Req_t) c2m <- mkFIFO;
  FIFO#(Resp_t) m2c <- mkFIFO;

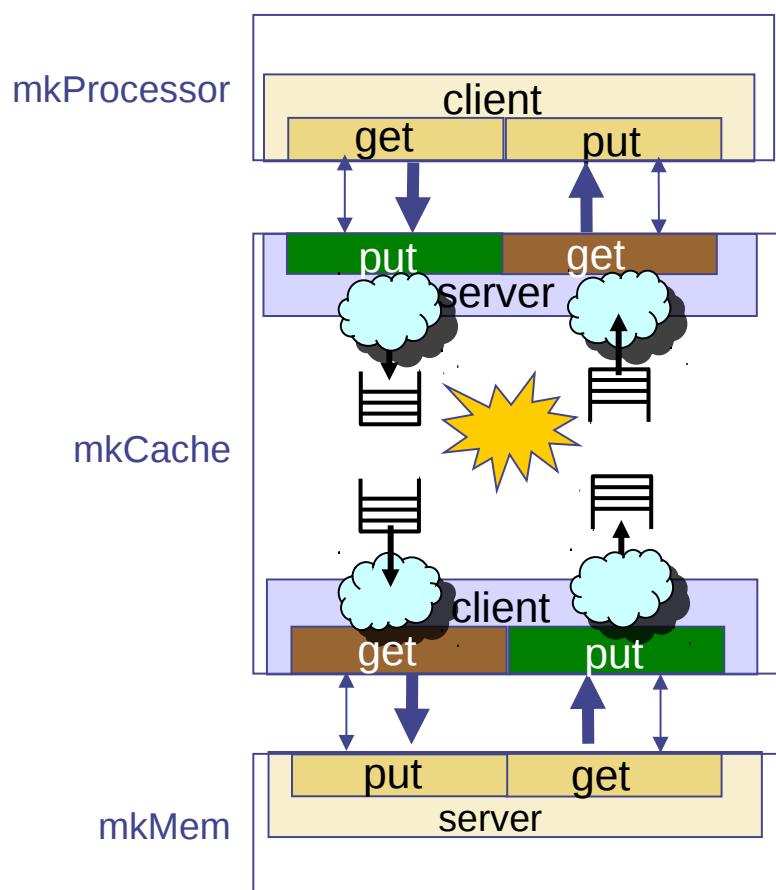
  ... rules expressing cache logic ...

  interface Server ipc;
    interface Put request = toPut (p2c);
    interface Get response = toGet (c2p);
  endinterface

  interface Client icm;
    interface Get request = toGet (c2m);
    interface Put response = toPut (m2c);
  endinterface
endmodule
```

# Example: using interface transformers

```
interface Cachelfc;  
  interface Server#(Req_t, Resp_t) ipc;  
  interface Client#(Req_t, Resp_t)  
    icm;  
endinterface
```



This can be further simplified with another common interface transformer

```
module mkCache (Cachelfc);  
  FIFO#(Req_t) p2c <- mkFIFO;  
  FIFO#(Resp_t) c2p <- mkFIFO;  
  
  FIFO#(Req_t) c2m <- mkFIFO;  
  FIFO#(Resp_t) m2c <- mkFIFO;  
  
  ... rules expressing cache logic ...  
  
  interface Server ipc =  
    toGPServer (p2c, c2p);  
  
  interface Client icm =  
    toGPClient (c2m, m2c);  
endmodule
```

# In general, interface transformers are modules

In the examples so far, `toGet`, `toPut`, `toGPClient` and `toGPServer` were simple functions.

Functions in BSV are “pure”—they cannot contain any internal state, and therefore can represent only “instantaneous” (combinational) computation.

In general, an interface transformer may need state and temporal computation

- E.g., a transformer that “serializes” from wide data to narrow data

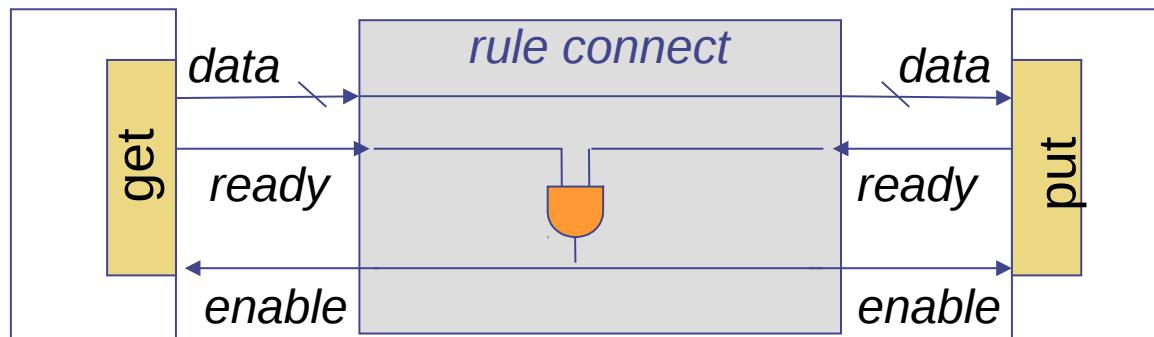
Such transformers will have to be modules, not just functions.

They’re often written using the “connections” methodology, discussed next.

# Connecting Get and Put

A Get and a Put interface (carrying the same type of data) can be connected with an explicit rule.

```
module mkTop (...)  
  Get#(int) m1 <- mkM1;  
  Put#(int) m2 <- mkM2;  
  
  rule connect;  
    let x <- m1.get();  m2.put (x);      // note implicit conditions  
  endrule  
endmodule
```



But, as we will see in the next few slides, even this design pattern can be captured with an abstraction.

# Capturing the design pattern

We can define a parameterized module that captures the design pattern.

```
module mkConnectionGetPut #(Get#(t) g, Put#(t) p) (Empty);
  rule connect;
    let x <- g.get(); p.put (x);
  endrule
endmodule
```

```
module mkTop (...)

  Get#(int) m1 <- mkM1;
  Put#(int) m2 <- mkM2;
```

```
  mkConnectionGetPut (m1, m2);
endmodule
```

// Technically: Empty e <- mkConnection (m1, m2);  
// Replaces:  
// rule connect;  
// let x <- m1.get(); m2.put (x);  
// endrule

# Further generalization of the connection pattern

Similarly, we could create abstractions for other types of connections:

`mkConnectionPutGet(p,g)`

`mkConnectionClientServer (c,s)`, `mkConnectionServerClient (s,c)`

`mkConnectionAXIMasterAXISlave (am, as)`

`mkConnectionTLMMasterTLMSlave (tm,ts)`

....

Instead of inventing new names for each such connection between pairs of related interface types, we can use BSV's "*overloading*" mechanism to use a common name, "mkConnection", for all of them.

Using *overloading resolution*, the compiler will figure out the correct module to be used for the connection, based on the interface argument types.

The concepts related to overloading in BSV are:

- "typeclass"
- "instance"
- "deriving"

(automatic creation of certain instances)

*(Typeclasses and overloading are discussed in more detail in another lecture)*

# The “Connectable” typeclass

```
typeclass Connectable #(type t1, type t2);
    module mkConnection #(t1 m1, t2 m2) (Empty);
endtypeclass
```

This declares a “type class”, which is a set of types on which certain “overloaded” identifiers can be declared. (This declaration is already in the BSV library.)

This can be read as: “two types t1 and t2 are in the Connectable typeclass when an overloaded identifier mkConnection has been defined for them, with the module type shown”.

We populate a typeclass explicitly using “instance” declarations:

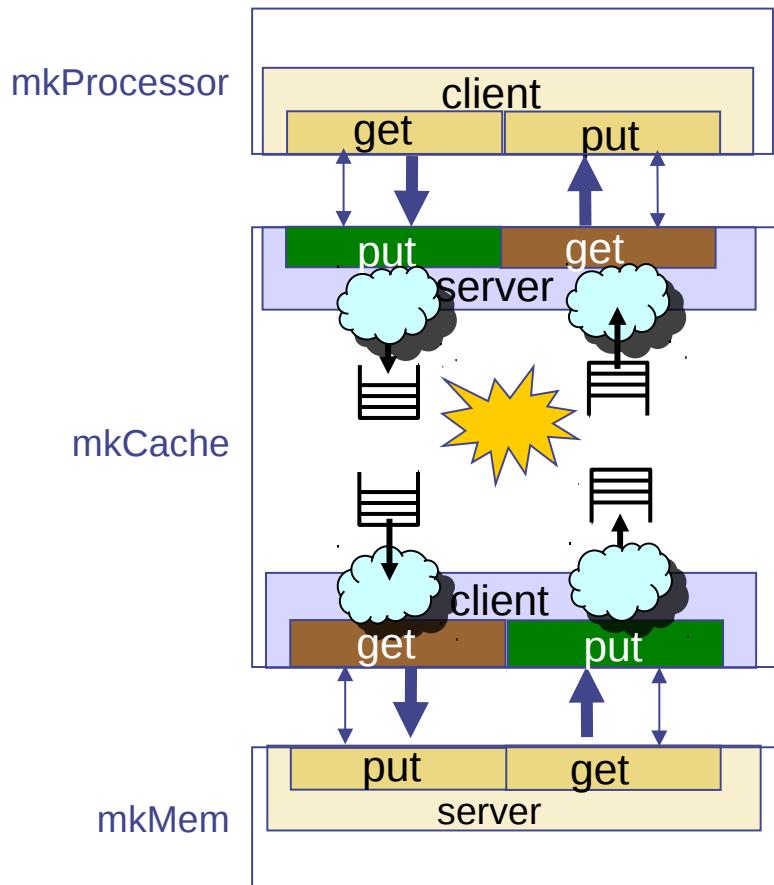
```
instance Connectable #(Get #(t), Put#(t));
    module mkConnection #(Get#(t) m1, Put#(t) m2) (Empty);
        rule r;
            let x <- m1.get; m2.put (x);
        endrule
    endmodule
endinstance
```

The BSV library provides instances for Get/Put, Client/Server, and many other types

[ C++ gurus: Connectable is like a “virtual class”, with “virtual member” mkConnection.  
The Get/Put pair of types “inherits” from this virtual class by providing a definition for mkConnection. ]

# Example: using mkConnection

The top-level of our processor-cache-memory system (mkTopLevel) reduces to 5 lines of code:



```
interface Cachelfc;
  interface Server#(Req_t, Resp_t) ipc;
  interface Client#(Req_t, Resp_t) icm;
endinterface

module mkTopLevel (...)

  // instantiate subsystems
  Client #(Req_t, Resp_t) p <- mkProcessor;
  Cache_Ifc #(Req_t, Resp_t) c <- mkCache;
  Server #(Req_t, Resp_t) m <- mkMem;

  // instantiate connects
  mkConnection (p, c.ipc);
  mkConnection (c.icm, m);

endmodule
```



End

# Questions?

Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

```

import PFGT;
import DataT;
import ListT;
import module ext_Std_Logic_Standard;

Integer file_depth;
function Bit4 determine_pmen(DataT);
return Bit4(0);
endfunction

PFGT(DataT) bmem2D;
Bit4 PFGT(Bit4[16] the_bmem1(the_bmem2));
PFGT(DataT) critmem1();
PFGT(Bit4[16] the_crit1(the_crit2));
PFGT(DataT) critmem2();
Bit4 PFGT(Bit4[16] the_crit2(the_crit3));
end

rule end (true);
    DataT In_Data = bmem2D[1];
    PFGT(Bit4[16]) crit1mem1 =
        $signed(critmem1[DataT]);
    $signed(crit1mem1[DataT]) = 0 ? crit1mem1 : crit1mem1 +
        $signed(critmem2[DataT]);
    $signed(crit1mem1[DataT]) <= 1;
    endrule
endrule
endmodule

```



# BSV Training

# Lec\_ StmtFSM

This is a facility for describing structured rule-based processes.  
StmtFSM, mkFSM, composing FSMs, controlling FSMs, mkFSMServer

```

import FIFOF#(bit);
typedef FIFOF#(bit) bitFIFO;
module ex_UltimateFSM();
    Integer fifo_depth = 16;
    function bit#(1) determine_parity(Bit#(1) b);
        return {b[0]}; //odd parity
    endfunction

    FIFOF#(Bit#(1)) libounds();
    bitFIFO#(Bit#(1)) the_Lbound(libraries);
    FIFOF#(Bit#(1)) enbounds();
    bitFIFO#(Bit#(1)) the_Cbound(jibraries);
    FIFOF#(Bit#(1)) libounds();
    bitFIFO#(Bit#(1)) the_Ubound(jibraries);

    rule end (true);
        Bit#(1) lib_bound = libounds.first();
        Bit#(1) en_bound = enbounds.first();
        Bit#(1) the_bound = the_Lbound.first();
        Bit#(1) the_Ubound = the_Ubound.first();
        Bit#(1) the_Cbound = the_Cbound.first();
        libounds.pop();
        enbounds.pop();
        the_Lbound.pop();
        the_Ubound.pop();
        the_Cbound.pop();
    endrule
endmodule

```



# Motivations

Finite State Machines (FSMs) are very common in hardware design.

With BSV rules, you can encode arbitrary FSMs.

For example, a simple FSM involving some sequencing and a loop:

```
typedef enum { S0, S1, S2, ... } State deriving (Bits, Eq);

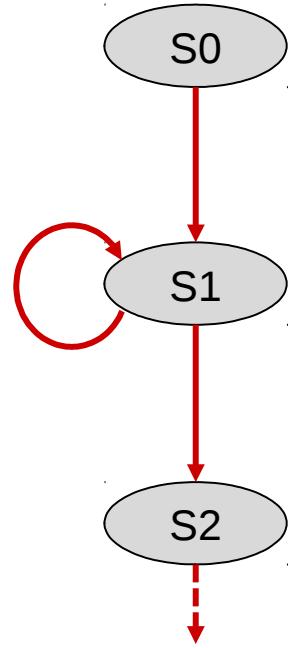
module mkFoo (...);
  Reg #(State) state <- mkReg (S0);

  rule r0 (state == S0);
    ... do state S0 actions ...
    state <= S1;                                // next state
  endrule

  rule r1 (state == S1);
    ... do state S1 actions ...
    state <= (some cond ? S1 : S2);      // loop back to S1 state, or exit loop
  endrule

  rule r2 (state == S2);
    ... do state S2 actions ...
    ... transition to next state ....
  endrule

endmodule
```



# Structured FSMs can be expressed more succinctly

There are common structured design patterns in FSMs:

- sequences, conditionals, loops, parallel threads, etc.

BSV provides a powerful FSM sub-language to express these more succinctly than having to write out the rules explicitly:

- But note that the semantics are *identical* to rules
- In fact, the compiler expands the FSM spec into the rules that you would have written by hand if you were to express them directly as rules

To use this facility:

- Import the “StmtFSM” package: `import StmtFSM :: *;`
- Create an FSM specification (an expression of type Stmt)
- Create an FSM module by giving it the specification
  - This returns an “FSM” interface with “start” and “done” methods
- Operate the FSM using the “start” and “done” methods

# The FSM interface

```
interface FSM;  
    method Action start;  
    method Bool done;  
    method Action waitTillDone;  
    method Action abort;  
endinterface
```

*FSM interface*

```
module mkFSM #( Stmt s )( FSM );
```

*mkFSM module*

Note:

- ‘done’ and ‘waitTillDone’ are just alternative ways for knowing when the FSM is done. You can either test the boolean ‘fsm.done’, or you can execute the Action statement ‘fsm.waitTillDone’, whose implicit condition is the same as fsm.done.

In different situations, one of the other may be more convenient.

- ‘abort’ allows an external agent to force the FSM to reset to its initial state, no matter what state it is in, and no matter how deeply nested it is (more about nesting later)

# Example revisited, with FSMs instead of rules

```
import StmtFSM :: *; // Import the "StmtFSM" package

module mkFoo (...);
    Stmt stmt = seq
        ... do state S0 actions ...
        ... do state S1 actions ...
        while (some cond) ... do state S1 actions ...
        ... do state S2 actions ...
    endseq;

    FSM fsm <- mkFSM (stmt); // Create an FSM specification (an expression of type Stmt)
                            // Create an FSM module by giving it the specification
                            // This returns an "FSM" interface with "start" and "done" methods

    rule init (...);
        fsm.start; ...
    endrule

    rule done (fsm.done);
        ...
    endrule
endmodule
```

- Import the “StmtFSM” package:
- Create an FSM specification (an expression of type Stmt)
- Create an FSM module by giving it the specification
  - This returns an “FSM” interface with “start” and “done” methods
- Operate the FSM using the “start” and “done” methods

**while (some cond) ... do state S1 actions ...**

can also be  
written as

**while (True) seq**  
... do state S1 actions ...  
**if (some cond) break;**  
**endseq**

# Composing FSMs

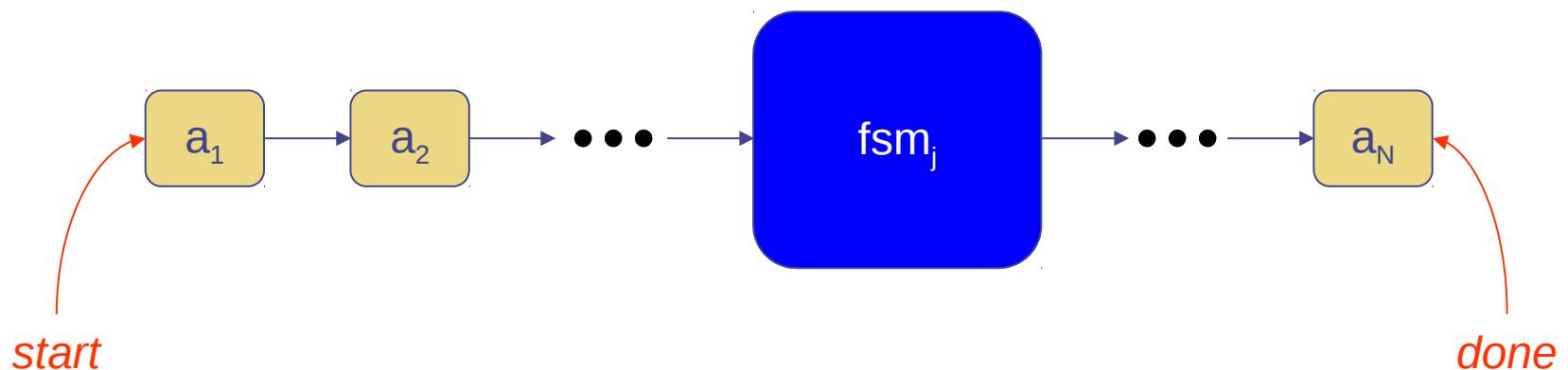
FSMs can be composed—individual FSMs can be combined systematically into larger FSMs.

The compositional principle is:

- Every FSM has a *start* and *done* state, embodied in the FSM interface methods
- When composing a larger FSM, the *start* and *done* for the larger FSM is systematically derived from the *start* and *done* of the component FSMs

Example: Linear sequencing

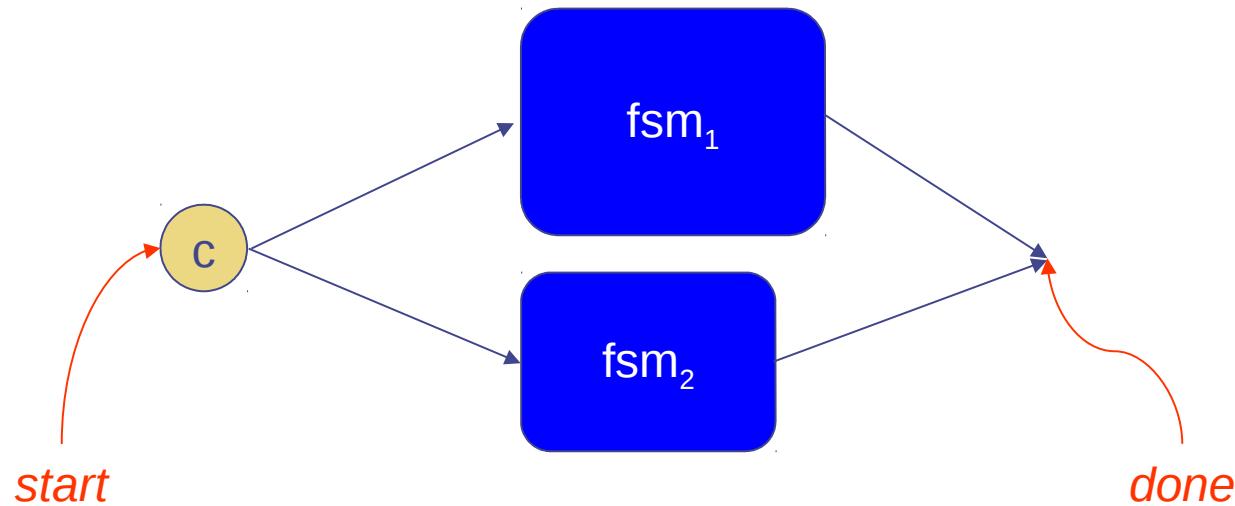
- Syntax: **seq** a<sub>1</sub>; a<sub>2</sub>; ... ; a<sub>N</sub>; **endseq**
- Where each a<sub>j</sub> is either an expression of type Action, or itself a sub-fsm (expression of type Stmt)



# Conditionals

## Syntax

- **if** (Bool expr) fsm1
- **if** (Bool expr) fsm1 **else** fsm2

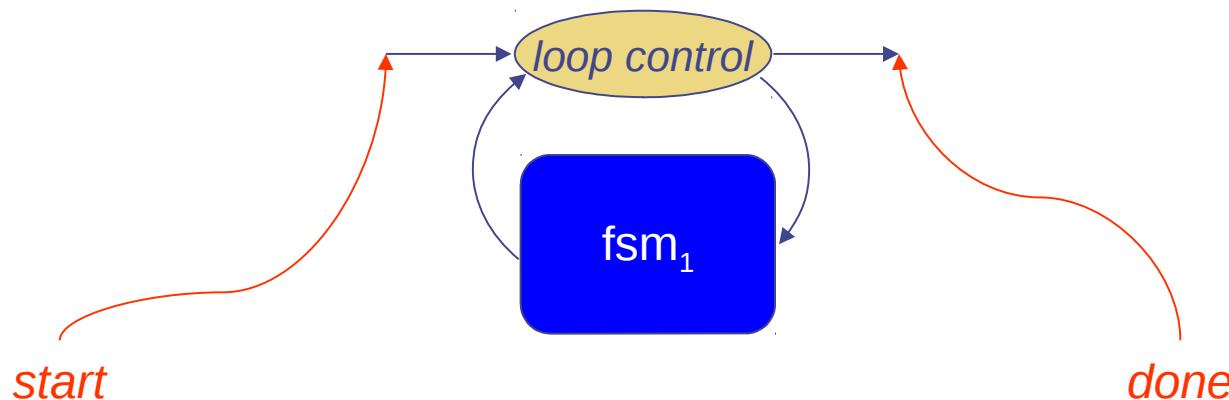


# Iteration (loops)

## Syntax

- **for** (loop control) fsm1
- **while** (Bool expr) fsm1
- **repeat** (Integer expr) fsm1

Loop bodies can contain **break** and **continue** keywords, with the usual meaning



# Parallel composition (fork-join)

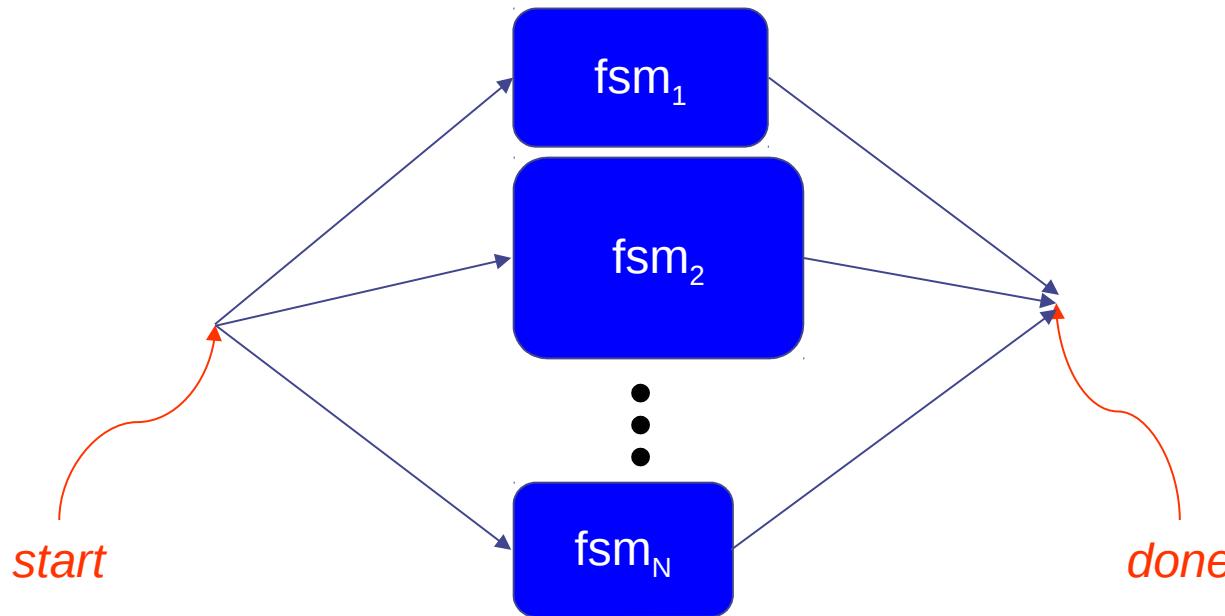
## Syntax

- `par fsm1; fsm2; ...; fsmN; endpar`

The sub-FSMs start at the same time and proceed independently.

The FSM is done when the all sub-FSMs are done (they need not complete simultaneously; we wait for the last one to be done).

Note: the sub-FSM can contain Actions that conflict with each other—these are resolved automatically using standard rule scheduling.



# Another FSM example

```
Stmt specfsm =  
  seq  
    write( 15, 51 );  
    read( 15 ) ;  
    ack ;  
    ack ;  
    write( 16, 61 );  
    write( 17, 71 );  
    // a memory operation and an  
    // acknowledge can occur  
    // simultaneously  
    action  
      read( 16 ) ;  
      ack ;  
    endaction  
    action  
      read( 17 ) ;  
      ack ;  
    endaction  
    ack ;  
    ack ;  
  endseq ;
```

```
FSM testfsm <- mkFSM (specfsm);
```

- action/endaction blocks compose larger entities of type Action. Since an Action is always within a rule, it guarantees that the sub-actions will be simultaneous (atomic).

```
rule run ( True );
```

```
  testfsm.start ;
```

```
endrule
```

```
rule done (testfsm.done);
```

```
  $finish(0);
```

```
endrule
```

# FSMs are often used as testbench stimulus generators

Example:

```
// Specify an FSM generating a test sequence
Stmt test_seq =
  seq
    for (i <= 0; i < NI; i <= i + 1)           // each source
      for (j <= 0; j < NJ; j <= j + 1) action   // each destination
        let pkt <- gen_packet ();
        send_packet (i, j, pkt);                 // test i-j path in isolation
      endaction

    // then, test arbitration by sending packets simultaneously to same dest
    action
      send_packet (0, 1, pkt0);                // to dest 1
      send_packet (1, 1, pkt1);                // to dest 1 (so, collision)
    endaction
  endseq;

mkAutoFSM (test_seq); // Generate the FSM and code to run it automatically
```

mkAutoFSM is another module provided in the library:

- It has an Empty interface
- Internally, it uses mkFSM to create the FSM, and it creates rules
  - to automatically start the FSM
  - to invoke \$finish when the FSM is done

# Revisiting our testbench from the EHRs lecture ...

```
module mkTest (Empty);
    UpDownSatCounter_Ifc ctr <- mkUpDownSatCounter;
    Reg #(int) step <- mkReg (0);
    Reg #(Bool) flag0 <- mkReg (False); Reg #(Bool) flag1 <- mkReg (False);

    function Action count_show (Integer rulenumber, Bool a_not_b, Int #(4) delta);
        action
            let x <- (a_not_b ? ctr.countA (delta) : ctr.countB (delta));
            $display ("cycle %0d, r%0d: is %0d, count (%0d)", cur_cycle, rulenumber, x, delta);
        endaction
    endfunction

    // Rules 0-9 are sequential, just testing one method at a time
    rule r0 (step == 0); count_show (0, True, 3); step <= 1; endrule
    rule r1 (step == 1); count_show (1, True, 3); step <= 2; endrule
        ... and similarly, sequentially feed deltas of 3,3, -6,-6,-6,-6, 7, 3,
    // Concurrent execution
    rule r10 (step == 10 && !flag0); count_show (10,True, 6); flag0 <= True; endrule
    rule r11 (step == 10 && !flag1); count_show (11,False, -3); flag1 <= True; endrule

    // Show final value
    rule r12 (step == 10 && flag0 && flag1); count_show (12,True, 0); $finish; endrule
endmodule: mkTest
```

*These parts just constitute a structured FSM.  
On the next slide we use StmtFSM instead of explicit rules*

# Revisiting our testbench from the EHRs lecture ...

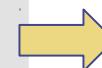
```
module mkTest (Empty);
    UpDownSatCounter_Ifc ctr <- mkUpDownSatCounter;

    function Action count_show (Integer rulenumber, Bool a_not_b, Int #(4) delta);
        action
            let x <- (a_not_b ? ctr.countA (delta) : ctr.countB (delta));
            $display ("cycle %0d, r%0d: is %0d, count (%0d)", cur_cycle, rulenumber, x, delta);
        endaction
    endfunction

    mkAutoFSM (
        seq
            count_show (0, True, 3);
            count_show (1, True, 3);
            ... and similarly, sequentially feed deltas of 3,3, -6,-6,-6,-6, 7, 3,
        par
            count_show (10,True, 6);
            count_show (11,False, -3);
        endpar
        count_show (12,True, 0);
    endseq);
endmodule: mkTest
```

*mkAutoFSM* is just a BSV library function that

- takes a Stmt argument (here, “seq...endseq”),
- creates an FSM from the Stmt,
- runs it once,
- and calls \$finish



```
module mkAutoFSM #(Stmt s)(Empty);
    FSM fsm <- mkFSM (s);
    rule rA;
        fsm.start;
    endrule
    rule rB (fsm.done);
        $finish;
    endrule
endmodule: mkAutoFSM
```

# Suspendable FSMs

The library provides another FSM constructor:

```
module mkFSMWithPred #(Stmt s, Bool b) (FSM);
```

An external agent can “asynchronously” start/stop the FSM by controlling the boolean predicate b.

[ *Language gurus: With parallel composition, nesting, abort and suspend, you get similar expressive power to FSMs in the language Esterel. The Esterel literature characterizes this power—it allows FSM descriptions to be exponentially smaller than descriptions without these capabilities.* ]

# FSM “servers”

Another useful composition facility is the FSM Server.

Normally, to perform a multi-cycle request to a server, your FSM would have to express it in a “split-phase” fashion:

```
Stmt s = seq
    mem.request.put (Req {op: Load, addr: a});
    let response <- mem.response.get;
endseq
```

With FSM servers, you can express this in a more traditional “procedure call” style.

```
function RStmt #(Data) fn_memServer (Req req);
    seq
        ... do the work for reading mem data ...
        return data;      // RStmt is a generalization of Stmt with 'return'
        values
    endseq
endfunction
```

```
FSMserver #(Addr, Data) memServer <- mkFSMServer (fn_memServer);
```

```
Stmt s = seq
    response <- callServer (memServer, Req {op: Load, addr:a });
endseq
```

# More info on FSMs

Section C.6.1 in the Reference Guide goes into a lot more detail on FSMs, including describing further functions, modules, etc. and providing many more examples.

# Hands-on

- BSV-by-Example book: Examples in Chapter 14



End

# Questions?

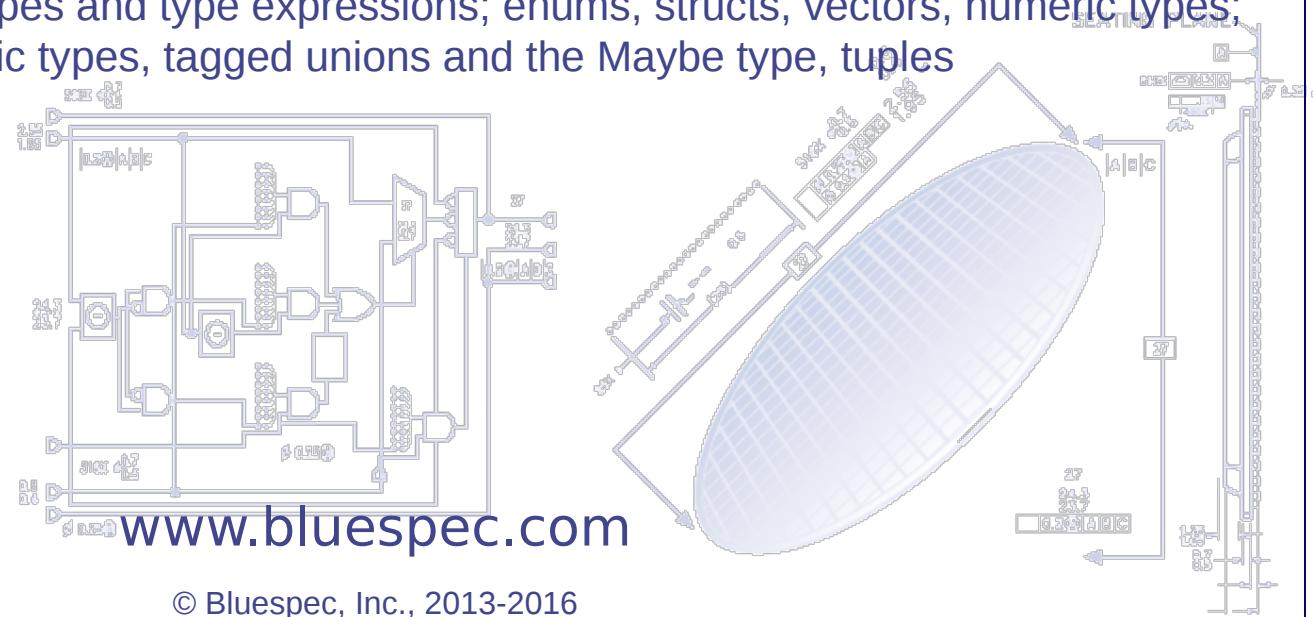
Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com



# BSV Training

## Lec\_Types

Role of types; syntax of types and type expressions; enums, structs, vectors, numeric types; polymorphic types, tagged unions and the Maybe type, tuples



# The role of types in modern programming languages

Most modern programming languages make strong use of *data types* to raise the level of abstraction and for correctness

- Types are an abstraction: ultimately, all computation, whether in SW or HW, is done on bits, but it is preferable to think in terms of integers, fixed point numbers, floating point numbers, booleans, symbolic state names, ethernet packets, IP addresses, employee records, vectors, and so on
- *Strong type-checking* is used to eliminate unintentional “mis-interpretation” of bits, such as taking the square of symbolic state, subtracting two IP addresses, indexing into an employee record, and so on

Note: historically, and even in some modern programming texts, types are explained in terms of bit representations. The more modern (and mathematical view), is to view each type as a set of abstract values along with the operations that can be performed on those values (i.e., an algebra). This view totally divorces a type from its representation; indeed the same type can have many possible representations. This is the view embraced in BSV.

# Types in BSV

- BSV has basic scalar types just like Verilog
- BSV has SystemVerilog type mechanisms like `typedefs`, `enums`, `structs`, `tagged unions`, `arrays` and `vectors`, `interface types`, `type parameterization`, `polymorphic types`
- BSV also has types for static entities like functions, modules, interfaces, rules, and actions
  - (so you can write static-elaboration functions that compute with such entities)
- BSV has very powerful *systematic user-defined overloading*—`typeclasses` and `instances` (more powerful than C++)
  - This is used heavily by advanced users
- Type-checking in BSV is *very strict*
  - Even registers are strongly-typed
  - No silent extensions and truncations
  - Typical anecdote (observed by BSV and Haskell programmers, which have the same type system):  
*“if it gets through the type-checker, it just works”*

# Syntax of types: Type Expressions

BSV uses SystemVerilog's notation for parameterized types

```
Type ::= TypeConstructor #(Type1, ..., TypeN)
      | TypeConstructor           // special case when N=0)
```

i.e., a type expression is a type constructor applied to zero or more other types. In the special case where it is applied to zero other types, the #() part can be omitted.  
Examples:

Type	Comments
Integer	Unbounded signed integers (static elaboration only)
Int#(18)	18-bit signed integers Note: 'int' is a synonym for Int#(32)
UInt#(42)	42-bit unsigned integers
Bit#(23)	23-bit bit vectors Note: 'bit[15:0]' is a synonym for Bit#(16)
Bool	Booleans, with constants True and False
Reg#(UInt#(42))	Interface of register that contains 42-bit unsigned integers
Mem#(A,D)	Interface of memory with address type A and data type D
Server#(Rq,Rsp)	Interface of server module with request type Rq and response type Rsp



Note uppercase first letter in type names

© Bluespec, Inc., 2013-2016

# Typedefs, enums, and structs

```
typedef Bit #(32) Word;
typedef Bit #(32) Addr;
typedef Bit #(32) Data;

typedef Bit#(4) RegName;
```

*Simple typedefs (left) are just synonyms for readability; all these types are equivalent.*

*Enum and struct typedefs (below) define new types, not equivalent with any other type.  
(So, type-checking prevents misuse.)*

```
typedef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq);
```

```
typedef struct {
    Opcode    op;
    RegName   dest;
    RegName   src1;
    RegName   src2;
} Instr
deriving (Bits);
```

*"deriving (Eq)" tells bsc to pick a "natural" equality comparison operator for this type.*

```
typedef struct {
    Opcode    op;
    RegName   dest;
    Bit #(32) v1;
    Bit #(32) v2;
} DecodedInstr
deriving (Bits);
```

*"deriving (Bits)" tells bsc to pick a "natural" bit-representation for this type.*

*(detailed treatment in a later section)*

# structs

Enum and struct types are “first class” types. They can be stored in state elements. They can be passed as method and function arguments and results, etc. (unlike C, but like C++).

```
Reg #(Opcode) rg_op <- mkReg (Noop);  
FIFO #(DecodedInstr) buf <- mkFIFO;
```

*Strongly typed: they can never contain values of any other type, even if they are represented in the same number of bits.*

Like C/C++, you can declare a variable with a struct type, and incrementally assign its members (fields). However, we often directly build entire struct values using struct expressions:

```
rule fetch (buf.notStall (instr));  
    let di = DecodedInstr { op: instr.op,  
                           dest: instr.dest,  
                           v1: rf.sel1 (instr.src1),  
                           v2: rf.sel2 (instr.src2) };  
  
    buf.enq (di);  
    pc <= pc + 1;  
endrule: fetch
```

# Vectors

We commonly use Vectors to express repeated structures.

In any package where we use them, we must first import the Vector package:

```
import Vector :: *;
```

Vectors are just type constructors, like any other. In particular, they can contain any types (including other Vectors):

```
interface EHR #(numeric type n, type t);
    interface Vector #(n, Reg #(t)) ports;
endinterface

typedef Vector #(10, Vector #(5, Int #(16))) Matrix;
```

Vectors are indexed with the usual square bracket “[ ]” notation:

```
Int #(5) new_val = extend (ctr.ports [p]) + extend (delta);
if (new_val > 7) ctr.ports [p] <= 7;
else if (new_val < -8) ctr.ports [p] <= -8;
else ctr.ports [p] <= truncate (new_val);
```

# Numeric types

- Some type constructors take *numeric types* in certain type-parameter positions.  
Examples:
  - 18-bit signed integers
  - Vector of sixteen 42-bit unsigned integers
- In a position where a numeric type is expected, you can provide:
  - Literal numeric values: 18, 16, 42, ....
  - Numeric type expressions: `TAdd #(18,16), TMul #(2,32), TLog #(19)`
- Although these have superficial similarity to numeric values and numeric value expressions:  
 $18 \quad 16 \quad 42 \quad 18+16 \quad 2*32 \quad \log2(19), \dots$   
they are not the same!
- Specifically, numeric types and type expressions are much weaker than full-blown numeric values and arithmetic expressions because:
  - Type-checking is performed in a separate, earlier phase of the compiler before any numeric value expression evaluation
  - Type-checking needs to be resolved statically

# Numeric types vs. numeric values

Numeric types are used in certain positions in type expressions and in type classes.  
Examples:

```
Bit #(23)
Int #(Tadd #(n, 16))
Vector #(8, sometype )
MyFifo #(4, sometype )
Bit #(Sizeof ( sometype ))
```

Numeric *types* are completely distinct from numeric *values*  
(even though we use the same literals, and we do similar ops like Add, Log, ...).

The central reason for this is that *bsc* does type-checking early, and must resolve types completely during compile-time. The numeric types sub-language is carefully designed to allow this. It would be undecidable if it could express arbitrary arithmetic, and thus we cannot use ordinary numeric values in types.

However, there is no such danger in going in the opposite direction, i.e., to use a numeric type where we need an ordinary numeric value. For this, BSV provides a pseudo-function (see Reference Guide Sec. B.3.3) suggested by this prototype (which is of course not legal syntax because BSV functions take values as arguments, not types):

```
function Integer valueof ( a numeric type );
```

# Polymorphic types

- Any type-parameter in a type expression can be a *type variable* (identifier beginning with a lower-case letter). Examples:
  - $n$ -bit signed integers **Int #(n)**
  - Vector of  $m$  elements, each of type  $t$  **Vector #(m, UInt #(t))**
- This allows for writing highly parameterized designs
- [C++ users: BSV type variables are like *template types*]

# The “Maybe” type

BSV (and SystemVerilog) have kind of type called “tagged unions”. One tagged union type frequently used in BSV is the “Maybe” type. (Reference Manual section B.2.10)

*The type declaration*

```
typedef union tagged {
    void Invalid;
    t    Valid;
} Maybe #(type t)
deriving (Eq, Bits);
```

A “*Maybe#(t)*” value is

- either “*invalid*” (with no associated value, i.e., *void*)
- or “*valid*” with an associated value of type “*t*”  
i.e., a “*valid*” bit along with a value

*Creating values of this type*

```
tagged Invalid
```

*creates 0 (invalid) along with a don't care value*

```
tagged Valid expression
```

*creates 1 (valid) along with the value of the expression*

*Using values of this type, with “pattern matching”*

```
if (value matches tagged Valid .x)
    ... here you can use x, the valid associated value ...
else
    ... here you handle the “invalid” case ...
```

Tagged unions are similar to “unions” in C/C++, except that tagged unions are type-safe, whereas unions are not. There is no way to examine the value when the valid bit is “invalid”.

# “Tuple” types

A 2-tuple is just a pair of values; a 3-tuple is a triple, ... and so on

*The 2-tuple type:*

**Tuple2 #(t1, t2)**

*A pair of values, the first one of type t1, and the second of type t2*

*Creating values of this type*

**tuple2 (expression1,  
expression2)**

*creates a value with two components, the value of expression1 and the value of expression2*

*Using values of this type: functions to extract components:*

**tpl\_1 (expression), tpl\_2 (e), ...**

*extract the j'th component of tuple that is value of expression*

*Using values of this type, with “pattern matching”*

**match { .x, .y } = expression;**

*declares new variables x and y, and binds them to the components of the 2-tuple value of expression*

2-tuples are just structs with 2 fields (in general, an  $n$ -tuple is a struct with  $n$  fields).

But tuples are so useful and common that they're pre-defined in BSV.

# Exporting abstract types

File Foo.bsv

```
package Foo;
```

```
import Bar :: *; ←
```

```
...
```

Code here can declare identifiers of type Request, can declare registers holding values of type Request, etc. It can retrieve values of type Request from method m1, send arguments to method m2, etc.

But it can never examine the fields (members) of such values, since the member names are not available here. Thus, within Foo, Request is an opaque (abstract) type.

```
...
```

```
endpackage: Foo
```

Bar exports mkBar and the type Request, but not the fieldnames (members) addr and data.

File Bar.bsv

```
package Bar;
```

```
export Request, mkBar;
```

```
...
```

```
typedef struct {
    Bit #(16) addr;
    Bit #(32) data;
} Request;
```

```
...
```

```
module mkBar (...);
```

```
    ...  
    method Request m1 (...);
```

```
    method Action m2 (Request r);
```

```
...
```

```
endpackage
```

# Hands-on

- BSV-by-Example book: Examples in Chapter 3 and 10



End

# Questions?

Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

```

import PFGT;
import DataT;
import ListT;
import module ext_Std_Logic_Standard;

Integer file_depth;
function Bit4 determine_pmen(DataT);
return Bit4(0);
endfunction

PFGT(DataT) bmem2D;
Bit4 PFGT(Bit4[16] the_bmem1(the_bmem2));
PFGT(DataT) critmem1();
PFGT(Bit4[16] the_crit1(the_crit2));
PFGT(DataT) critmem2();
Bit4 PFGT(Bit4[16] the_crit2(the_crit3));
end

rule end (true);
    DataT In_Data = bmem2D[1];
    PFGT(Bit4[16]) crit1mem1 =
        $signed(critmem1[DataT]);
    $signed(crit1mem1[DataT]) = 0 ? crit1mem1 : crit1mem1 +
        $signed(critmem2[DataT]);
    $signed(crit1mem1[DataT]) <= 1;
    endrule
endrule
endmodule

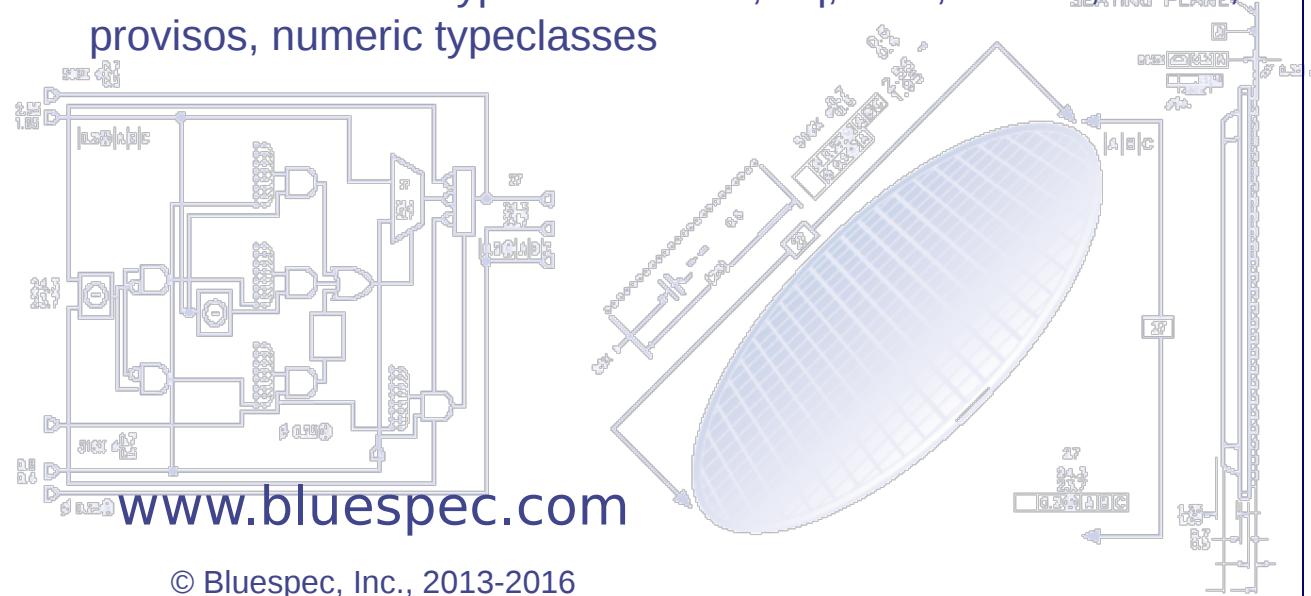
```



# BSV Training

## Lec\_Typeclasses

Overloading: Typeclasses and instances. Built-in typeclasses Bits, Eq, Arith, FShow, Literal,  
provisos, numeric typeclasses



# Type Parameterization: Polymorphism and Overloading

- BSV, like C++ and many advanced programming languages, has two kinds of parameterization using types: Polymorphism and Overloading (Typeclasses)
- *Overloading*: The use of the same name (identifier) for *multiple* functions and operators that have different argument and/or result types. In an actual function call, the compiler uses the types of the arguments/results to determine which of the actual multiple functions is intended; this compiler activity is called *overloading resolution*.
  - Example: “+” on integers, floating point numbers, complex numbers, vectors, ...
  - The programmer explicitly declares each possibility
- FYI: overloading in other languages:
  - C, Verilog, SystemVerilog: limited overloading (built-in only, not user-definable), for common operators like +, -, \*, /, ==, !=, ...
  - C++: Fully extensible. User can extend overloading of built-in operators like +, -, ... and can also overload user-defined functions and methods
- BSV: overloading that is as powerful as C++
  - BSV’s overloading is type-checked even during separate compilation (not so in C++)
  - Overloading is used extensively within *synthesizable* designs
  - BSV’s overloading system is taken from the Haskell functional programming language

# Overloading

Overloading in BSV involves the following concepts:

- *Typeclasses*: defining a set of overloaded names (identifiers) and their types
- *Instances*: defining actual functions for overloaded names
- *Provisos*: constraints specifying that a particular function or module parameter must belong to a certain typeclass, i.e., must be overloaded in a certain way
- “deriving” clauses on type definitions: automatic creation of certain instances

In this lecture we focus on the most common uses of overloading:

- Attaching “deriving” to type definitions, most often “deriving (Bits, Eq)”.
- Attaching “provisos” to functions and modules
- Defining new instances of existing typeclasses

# Separating the logical view of types from representations

Consider the following type declarations (from a CPU example):

```
typedef Bit#(4) RegName;

typedef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq);

typedef struct {
    Opcode    op;
    RegName   dest;
    RegName   src1;
    RegName   src2;
} Instr
deriving (Bits);
```

There are many choices in how we implement these in bits:

- How many bits for Opcode (3 are enough, but should we use more, e.g., for future expansion)?
- What encodings of bits for Noop, Add, Bz, Ld and St?
- How many bits for Instr, and what layout for the fields op, dest, src1 and src2?

Further, the choices may change (as we add more opcodes, re-architect the CPU, ...).

In BSV, using overloading, we cleanly separate out the logical view (in the red box above) from implementation choices, so that:

- we only have to change two overloaded functions to change our implementation choices
- the rest of our code (typically > 99%) only uses the logical view, and never has to be changed

# Separating the logical view of types from representations

We use two overloaded functions, pack() and unpack() to “convert” between the logical view and bits:

Why use overloaded functions? Because we'd like to use the same function names, “pack” and “unpack” for many types (like “Opcode” and “Instr”) and not have to invent new names for each case.

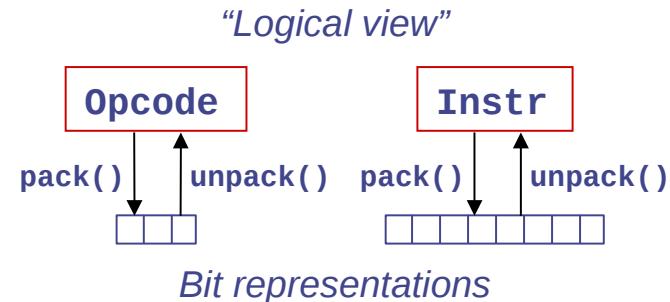
Note: this is a common “experienced programmer” trick in C++. Instead of exposing the representation members of a class as “public” members, the programmer labels them “private”, and only accesses them from outside via public “setter” and “getter” methods. This allows future changes in representation without having to change all the code that uses objects of that class.

When you use a BSV primitive like Reg to hold a type like Instr:

```
Reg #(Instr) rg_instr <- mkReg( Instr{op:Noop,  
... dest:0,  
... src1:0,  
... src2:0});  
...  
rule rl_exec ( rg_instr.op == Add);  
...  
endrule
```

When you initialize the reg, or later use \_write, we only use the logical view; the module internally uses “pack()” to convert to bits for storing in a hardware register.

When you use \_read, we only see the logical view; the module internally uses “unpack()” to convert the bits in the hardware register into the logical view.



# A typeclass is a group of related overloaded identifiers

pack() and unpack() are related:

- we typically define them together for each new logical type
- we typically expect them to be inverses
  - pack (unpack (x)) == x      unpack (pack (y)) == y

Similarly, when we define arithmetic operations on various types (int, float, vector, ...)

- we typically define the operations together: +, -, \*, /, ...
- we typically expect them to have an “algebra”:
  - $x + (y + z) == (x + y) + z$        $x * (y + z) = x * y + x * z$  ... and so on

In BSV, this idea of a related group of overloaded identifiers is formalized into the concept of a “typeclass”. A typeclass declares which overloaded identifiers belong to the group, and what are their most general types. This is independent of any actual definitions of these overloaded identifiers (which is called an “instance” of the typeclass).

For example, the BSV library pre-defines the following typeclasses:

```
typeclass Bits #(type t, numeric type n);
  function Bit #(n) pack (t x);
  function t          unpack (Bit #(n) bs);
endtypeclass
```

Note: using a standard Verilog trick to  
use “\” for non-alphanumeric identifiers

```
typeclass Arith #(type t);
  function t \+ (t x, t y);
  function t \- (t x, t y);
  function t \* (t x, t y);
  function t \/ (t x, t y);
  ...
endtypeclass
```

# The actual overloaded functions form an “instance”

To specify a custom representation for a new type, we use an “instance” declaration:

```
instance Bits #(Opcode, 4);
  function Bit #(4) pack (Opcode op);
    case (op)
      Noop : return 4'b0010;
      Add  : return 4'b0100;
      Bz   : return 4'b0001;
      ... etc. ...
    endcase
  endfunction

  function Opcode unpack (Bit #(4) b4);
    case (b4)
      4'b0010: return Noop;
      4'b0100: return Add;
      ... etc. ...
    endcase
  endfunction
endinstance
```

Here, we have chosen a 4-bit representation

This is arbitrary user-written code, so we can choose any representation that is appropriate for our application

What about cases of 4-bit values that do not represent any of the five opcodes? As usual, it's your design choice about what should happen. E.g.,

- Leave it unspecified. In fact, if you don't use 'pack' and 'unpack' explicitly (only used inside BSV primitives), BSV's strong type-checking guarantees that b4 cannot be an illegal value.
- Map illegal values to Noop.
- Test for illegal values and raise an error, before calling pack only on legal values
- etc.

# The “deriving” shortcut (instead of “instance”)

In practice, the “obvious” representation is almost always perfectly fine.

In these cases, we use the “deriving (Bits)” shortcut instead of declaring an instance.

```
typedef Bit#(4) RegName;  
  
typedef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq);  
  
typedef struct {  
    Opcode    op;  
    RegName   dest;  
    RegName   src1;  
    RegName   src2;  
} Instr  
deriving (Bits);
```

*bsc will implicitly declare an instance for Opcode, using 3 bits, and encoding the labels as 001, 010, 011, 100 and 101, respectively*

*bsc will implicitly declare an instance for Instr, using 15 bits, and representing the struct by the bit-concatenation:*

*{ pack(op), pack(dest), pack(src1), pack(src2) }*

*Note that these are recursive uses of pack() on the types Opcode and Bit#(4)*

For every type you use in the hardware, you must either use “deriving (Bits)” or provide an explicit “instance” declaration. (You don’t need to do this for types only used during static elaboration.)

# Other typeclasses pre-defined in the BSV library

There are many pre-defined typeclasses, described in detail in Reference Manual, sec B.1.  
You can use the “deriving” shortcut for Bits, Eq, and Bounded.

Typeclass “Eq” declares the overloaded operators: == !=

Typeclass “Bounded” declares : minBound maxBound  
(the smallest and largest values of a type)

Typeclass “Arith” declares: + - \* /  
(and many more arithmetic operators and functions)

Typeclass “Ord” declares: < <= > >= compare min max

Typeclass “Bitwise” declares : & | ^ ~^ ^~ invert << >> msb lsb

Typeclass “BitExtend” declares: extend zeroExtend signExtend truncate

For any user-defined type, you can define your own instance of these typeclasses.  
E.g., if you define Arith#(Matrix) then you can use +, -, \*, ... on Matrix values

Users can also define their own new typeclasses (and populate them with instances).

# Motivating “provisos”

Suppose we define a function to sort vectors:

```
function Vector #(n, Int#(32)) sort (Vector #(n, Int#(32)) v);  
    ...  
    if (v [j] < v [k]) ...  
    ...  
endfunction
```

Inside the function, we are comparing two of the vector elements with the “<” operator, which is an overloaded operator in the “Ord” typeclass. This is ok, since the elements have type “Int #(32)”, which has been pre-declared in BSV to be in the “Ord” typeclass, and so it is clear what “<” means here.

But what if we want “sort” to be polymorphic, i.e., to sort vectors of elements of any type t, not just “int”, provided “<” is meaningful on t?

# “Provisos” are assertions on polymorphic types

The *polymorphic* function to sort vectors adds a “provisos” assertion:

```
function Vector #(n, t) sort (Vector #(n, t) v)
    provisos (Ord #(t));
    ...
    if (v [j] < v [k]) ...
    ...
endfunction
```

The “provisos” clause asserts that any “t” on which this function is used must be in the “Ord” typeclass. This helps *bsc*’s typechecker in two ways:

- *bsc* knows that “<” here is a meaningful operation
  - If you omitted the “provisos” clause, the type-checker will complain
- *bsc* can check, at each place where “sort” is called, that the actual argument is ok, namely a vector of elements of a type which is in the “Ord” typeclass

## Note for C++ programmers

You can write a similar “sort” function in C++, where “t” is a C++ template type.

So, why does C++ not use/need provisos?

*It’s because in C++, templates are explained (and implemented) in terms of in-line expansion, so type-checking of such a function is postponed until each actual call. Eventually, after enough in-line expansion, the code is no longer polymorphic, and so it can be type-checked. This is not only inefficient (repeated type-checking of the function body at each call), but leads to some of C++’s famously complex error messages.*

# “Provisos”: more examples

```
module mkFoo #(... parameters ...) (InterfaceType#(t))
  provisos (Bits #(t,tsize), Eq #(t), Arith #(t));
  ...
endmodule
```

- The “Bits” proviso is an assertion that “t” has the “pack” and “unpack” functions to convert to bits and back.
  - This proviso is typically necessary because inside the module you are probably storing values of type “t” in registers, or other state elements, where they are, of course, ultimately stored as bits.
- The “Eq” proviso is an assertion that “t” has the “==” and “!=“ operators defined.
  - This proviso is typically necessary because somewhere inside the module you are probably comparing values of type “t” for equality or inequality.
- The “Arith” proviso is an assertion that “t” has the arithmetic operators “+”, “-”, “\*” etc. defined
  - This proviso is typically necessary because somewhere inside the module you are probably performing arithmetic operations on values of type “t”

If you perform overloaded operations on a polymorphic type inside the module, and you forgot to add a corresponding “provisos” clause in the module header, the type-checker will complain about it.

# The “Literal” typeclass for integer conversion

Suppose we have an assignment statement like this:

```
rg_x <= 2012;
```

How should we interpret the literal “2012”? How many bits should it have? Is it signed or unsigned? If rg\_x contains a user-defined type, is this assignment even meaningful? Can we extend the language so that it is meaningful for user-defined types?

Most languages have *ad hoc* answers to these questions.

In BSV, there is a systematic answer, based on overloading.

BSV has a built-in type called “Integer” which stands for mathematical (unbounded) integers.\* All literals, like “2012” have this type.

BSV has a pre-defined typeclass containing an overloaded function “fromInteger” to convert from Integer to some destination type t:

```
typeclass Literal #(type t);
    function t fromInteger (Integer t);
endtypeclass
```

Thus, the above assignment is interpreted as:

```
rg_x <= fromInteger (the Integer with value 2012);
```

This gives a precise semantics for integer literals, and is extensible to user-defined types (by just defining new instances for the Literal typeclass):

\* Of course, in reality an “unbounded” Integer will be limited by the size of your computer’s memory, but that is a pretty good practical abstraction of infinity!

# Motivating the Fmt type and the FShow typeclass

When debugging code, it is useful to “pretty-print” user-defined types.

```
typedef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq);  
  
typedef struct {  
    Opcode    op;  
    RegName   dest;  
    RegName   src1;  
    RegName   src2;  
} Instr  
deriving (Bits);
```

E.g., when printing an Opcode value with \$display:

```
$display ("The current opcode is: ", rg_instr.op);
```

we should see an output with the symbolic value, not the bit representation value:

```
The current opcode is: Add
```

E.g., we would like to print a full Instr value with \$display (not just its scalar fields):

```
$display ("The current instr is: ", rg_instr);
```

should produce an output like this:

```
The current instr is: Instr { op: Add, dest:3, src1:5, src2: 6 };
```

# The Fmt type

BSV provides a pseudo-function, \$format, whose arguments are just like the arguments to \$display. However, instead of printing anything, it returns an object of type Fmt. E.g.,

```
function Fmt showOpcode (Opcode op);
  case (op)
    Noop: return $format ("Noop");
    Add:  return $format ("Add");
    ... and so on ...
  endcase
endfunction
```

In BSV, \$display also accepts Fmt arguments, in addition to the usual types of arguments.

Now, we can do what we wished for in the last slide:

```
$display ("The current opcode is: ", showOpcode (rg_instr.op));
```

and we see an output with the symbolic value, not the bit representation value:

```
The current opcode is: Add
```

Similarly, we could define a showInstr () for Instr values, producing a Fmt value.

# The FShow typeclass

Instead of inventing new and different names for the “show” functions for each user-defined type, BSV pre-defines a typeclass FShow with the overloaded function name “fshow”

```
typeclass FShow #(t);
    function Fmt fshow (t);
endtypeclass
```

Instead of defining “showOpCode” (prev. slide), we define “fshow” for type “Opcode”:

```
instance FShow #(Opcode);
    function Fmt fshow(Opcode op);
        case (op)
            Noop: return $format ("Noop");
            Add:  return $format ("Add");
            ... and so on ...
        endcase
    endfunction
endinstance
```

and we use “fshow” in \$displays:

```
$display ("The current opcode is: ", fshow (rg_instr.op));
```

# The “deriving” shortcut for “FShow”

For most user-defined types—enums, structs, tagged unions and vectors—there is a “natural” textual representation for printing them

- like the examples in the previous slides for the enum Opcode and the struct Instr

So, instead of having to write an instance of FShow explicitly for each user-defined type, we can let bsc do it automatically using the “deriving” shortcut:

```
typedef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq, FShow);  
  
typedef struct {  
    Opcode    op;  
    RegName   dest;  
    RegName   src1;  
    RegName   src2;  
} Instr  
deriving (Bits, FShow);
```

(Note: this is a new feature in October 2012. If your installation predates this, you may not have it yet. You will have it when you next upgrade your installation.)

# Numeric typeclasses and provisos

In many codes, the sizes of various entities have some defined numeric relationship. BSV provides a pre-defined (and not user-extensible) set of numeric typeclasses. These can be used in provisos to assert numeric relationships.

E.g., suppose we are defining a new FIFO type that is parameterized by  $n$ , its capacity. Inside the FIFO, we will have a vector of depth  $n$  to hold the data. But we also need registers of width  $\log(n)$  bits to keep track of the head and tail of the FIFO.

```
interface MyFifo #(n, t);
    ... methods ...
endinterface

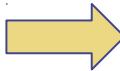
module mkMyFifo (MyFifo #(n, t))
    provisos (Log #(n, m));
        Vector #(n, Reg #(t)) vr_data <- replicateM (mkRegU);
        Reg #(Bit #(m)) rg_head <- mkReg (0);
        Reg #(Bit #(m)) rg_tail <- mkReg (0);
    ...
    ... rest of module ...
endmodule
```

This proviso asserts that  $\log(n) = m$ . Since  $n$  is known, it constrains  $m$  to this value. (BSV numeric provisos can be read as true equations, and constraints can flow in both directions). Subsequently, we use  $m$  to declare the width of our FIFO head and tail registers.

# Numeric typeclasses and provisos

BSV provides a collection of numeric typeclasses,  
described in Sec. B.3.1 in the BSV Reference Guide:

Add #(n1, n2, n3)		$n1 + n2 = n3$
Mul #(n1, n2, n3)	corresponding	$n1 * n2 = n3$
Div #(n1, n2, n3)	assertions	$\text{ceiling}(n1/n2) = n3$
Max #(n1, n2, n3)		$\max(n1, n2) = n3$
Min #(n1, n2, n3)		$\min(n1, n2) = n3$
Log #(n1, n2)		$\text{ceiling}(\log_2(n1)) = n2$



# Summary on typeclasses

BSV typeclasses are very powerful (the concept is borrowed with almost no change from the advanced functional programming language Haskell, which has had it since 1991 and with which there is widespread experience).

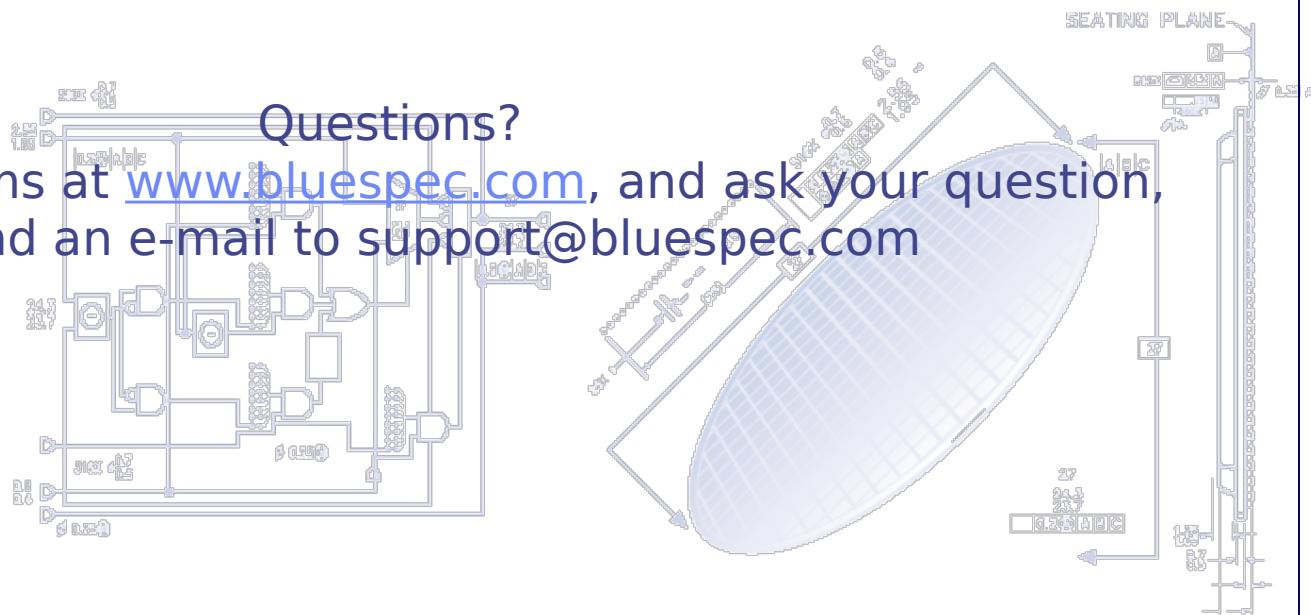
Typeclasses clean up certain things that are typically done in an ad hoc (and therefore not scalable) manner in many existing SW programming languages and HW design languages:

- Encapsulated, orthogonal, changeable bit representations for user-defined types
- Systematic overloading, with separate compilation
- Conversion of integer literals to built-in and user-defined types
- Composable, recursive pretty-printing of user-defined types
- Numeric type relationships of sizes of hardware components



# End

```
import PFR#(type#(Bit#(8)) default);
module end;
  method Action check();
    Integer file;
    function Bit#(8) determine_gpa(Bit#(8) val);
      return val;
    endfunction
    function Bit#(8) calculate();
      return determine_gpa(val);
    endfunction
    action
      file = $fopen("gpa.txt");
      $fputs(file, calculate());
      $fclose(file);
    endaction
  endaction
endmodule
```



Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

# BSV Training

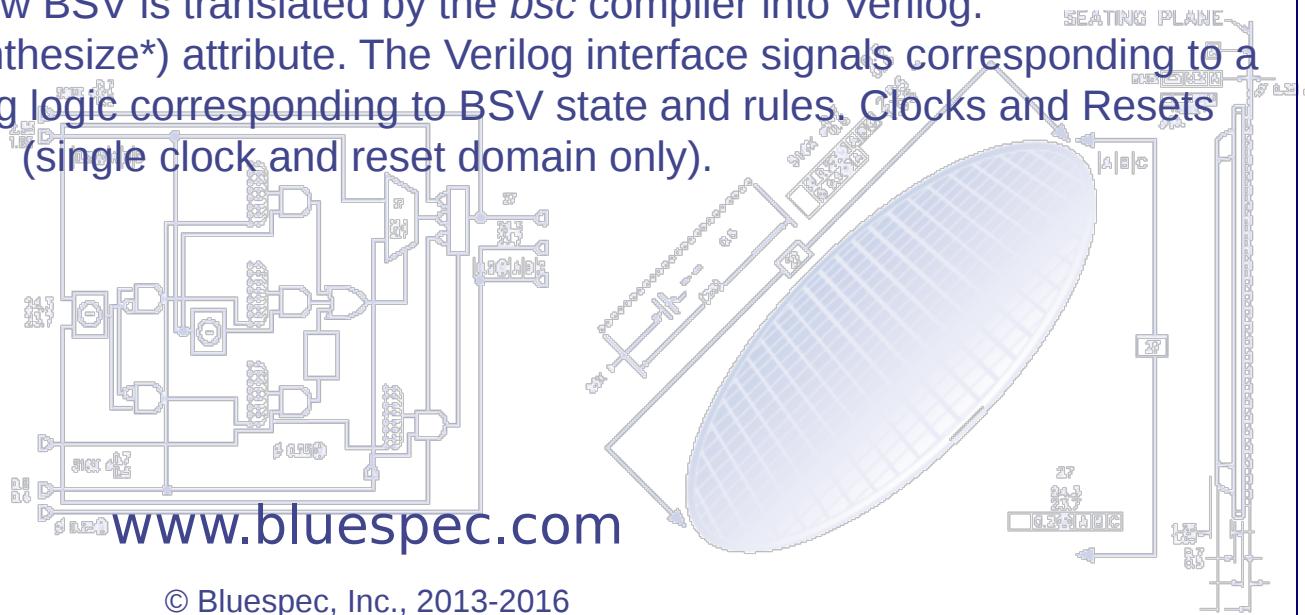
## Lec\_BSV\_to\_Verilog

Describes how BSV is translated by the *bsc* compiler into Verilog.

**Module hierarchy.** The (\*synthesize\*) attribute. The Verilog interface signals corresponding to a BSV interface. The Verilog logic corresponding to BSV state and rules. Clocks and Resets (single clock and reset domain only).

```

type Action;
  module Action write();
    method Action writeData(Bit#(T) data);
      action
        FIFO#(Bit#(T)) dataFIFO = FIFO#(Bit#(T))();
        dataFIFO.enq(data);
        if (dataFIFO.size() == 1)
          dataFIFO.deq();
      endaction
    endaction
  endinterface
endinterface
  
```



# Introduction

This lecture is optional, and is intended for those who have prior experience with Verilog and digital hardware, and are curious about the Verilog produced by the *bsc* tool from BSV source code. It is also useful for those who need to interface BSV-generated Verilog to other existing Verilog/VHDL.

For everybody else, this lecture can safely be skipped.

Analogy: One can learn a programming language (e.g., C, C++, Java) and use it productively without knowing anything about the machine code produced by compilers for the language. Similarly, BSV is self-contained and has a clear semantics independent of any particular implementation, and so can be used and debugged by just understanding the source semantics. Verilog is just the “assembly language” for BSV.

# Contrasting Verilog and BSV module structure

Verilog params are typically scalar numbers.  
Verilog interfaces are signal port lists.

```
module m #(params) (ports)
```

```
  input ...
  output ...
  wire ...
```

wire decls  
The only type is 'bits'

```
  reg x;
  reg y;
```

'reg' is not a module.  
'reg' may not even be a register.  
'reg' just holds bits.

```
  module m1 #(params) p (port connections);
  module m1 #(params) q (port connections);
  module m2 #(params) r (port connections);
```

```
  assign w = 10 + wire from instance q;
  assign ...
```

```
  always @(posedge clk) ...
  always @(posedge clk) ...
```

```
endmodule
```

BSV params can be of arbitrary type  
(incl. functions, interfaces, modules, ...).  
BSV interfaces are interface types (with methods)

```
module m #(params) (interface type);
```

Registers are instantiated  
just like any other module,  
and are strongly typed.

```
Reg #(t1) x <- mkReg (0);
Reg #(t2) y <- mkReg (12);
```

```
Ifc_m1 p <- mkM1a (params);
Ifc_m1 q <- mkM1b (params);
Ifc_m2 r <- mkM2 (params);
```

```
int w = 10 + q.method();
```

Typed var decls

Rules

Methods

```
endmodule
```

Module instantiation

"Behavior"

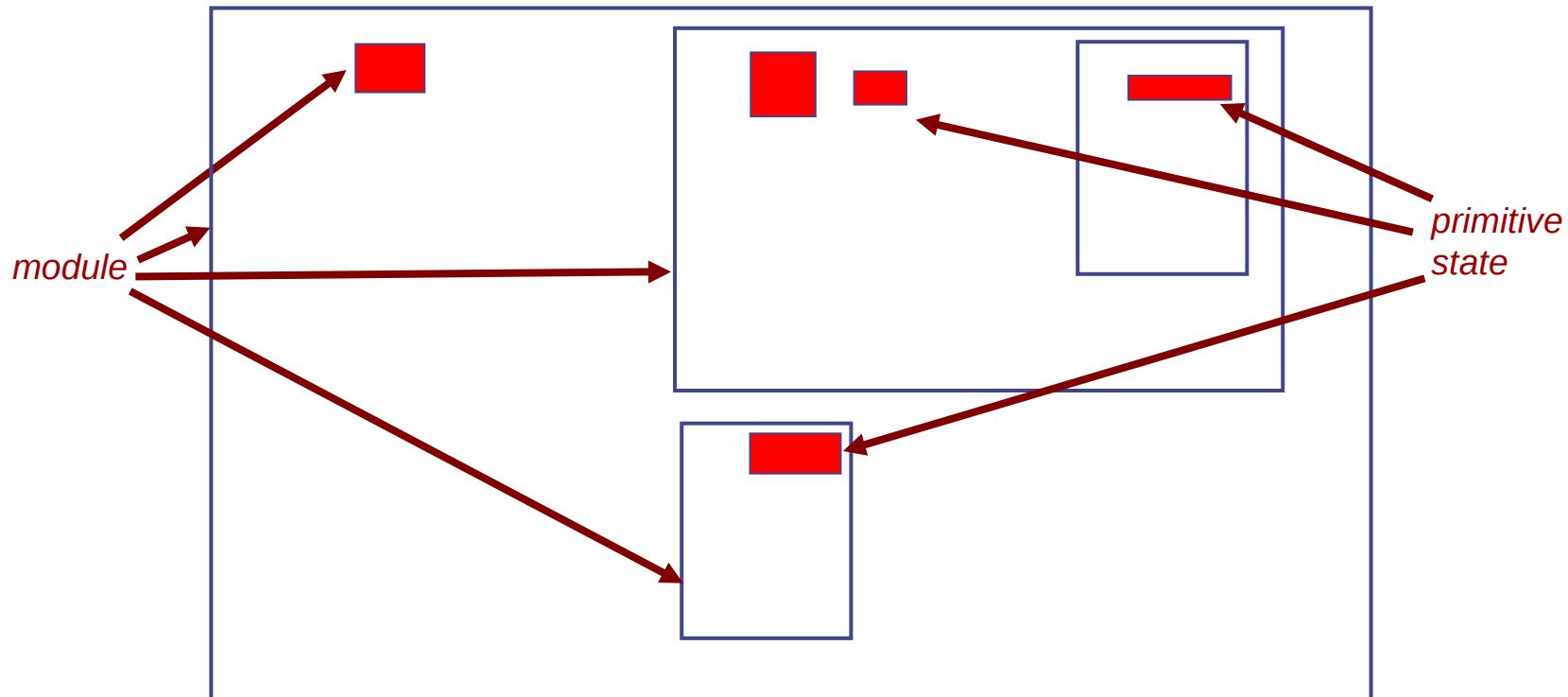
```
endmodule
```

# Module hierarchy and state

A BSV design consists of a *module hierarchy* (just like in Verilog, SystemVerilog and SystemC)

The leaves of the hierarchy are “primitive” state elements, including registers, FIFOs, etc.

Even registers are (semantically) modules (unlike in Verilog, SystemVerilog, ...).



All “primitives” in BSV are in fact implemented in Verilog and “imported” using BSV’s standard import mechanism. Hence, you can easily create new primitives or import existing Verilog IP.

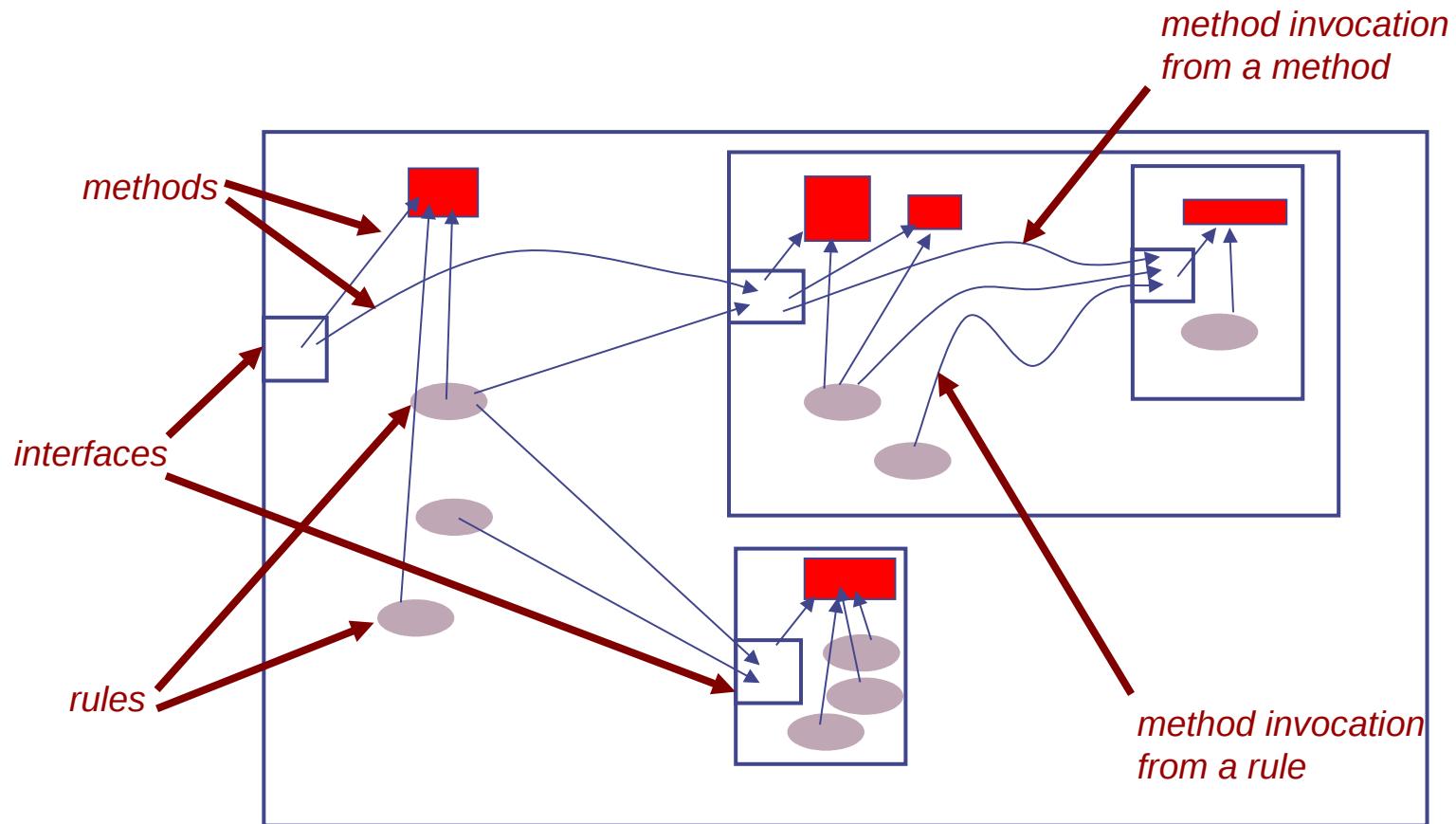
# Rules and interface methods

Modules provide interfaces, which contain *interface methods*.

Modules contain rules, which use methods in other modules.

All inter-module communication is via methods (object-oriented)

A method can itself use methods of other modules.



# From BSV module hierarchy to Verilog module hierarchy

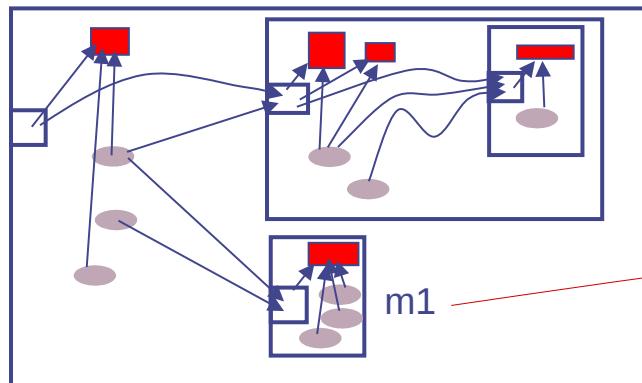
Broadly speaking, the BSV module hierarchy can be preserved in the generated Verilog:

- Each BSV module mkM becomes a corresponding Verilog module mkM (in file mkM.v)
- If a BSV module mkM1 instantiates a BSV module mkM2,  
the corresponding Verilog mkM1 instantiates the corresponding Verilog mkM2

However, a BSV module may be “inlined” wherever it is instantiated.

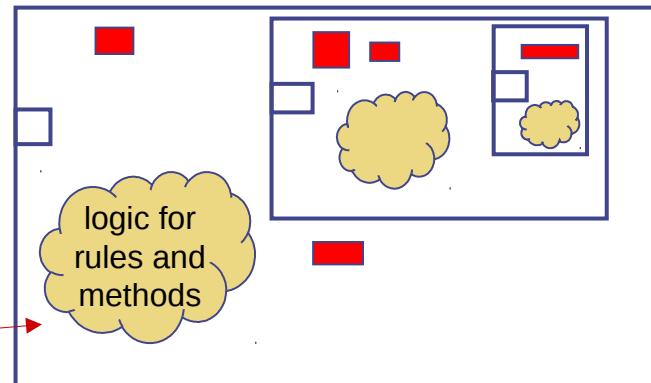
In this case, there will be no corresponding separate Verilog module.

Module hierarchy in BSV source code



Note: m1 has  
been “inlined” into  
its parent module

Module hierarchy in generated Verilog



This inlining is controlled by the optional (\* synthesize \*) attribute (see next slide)

# Example of the optional (\* synthesize \*) attribute

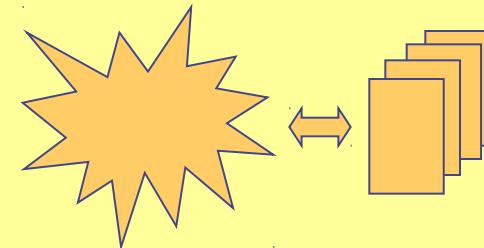
BSV source code

```
(* synthesize *)
module mkTest (Empty);
  Mult_ifc m <- mkMult;
  ...
endmodule: mkTest

module mkMult (Mult_ifc);
  ...
endmodule: mkMult
```

Generated Verilog

mkTest.v



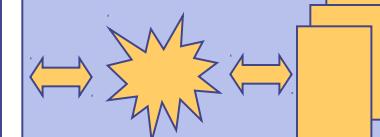
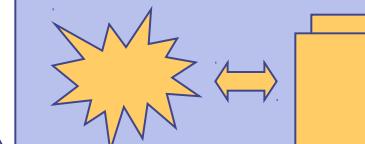
```
(* synthesize *)
module mkTest (Empty);
  Mult_ifc m <- mkMult;
  ...
endmodule: mkTest

(* synthesize *)
module mkMult (Mult_ifc);
  ...
endmodule: mkMult
```

mkTest.v

mkMult.v

instantiates  
mkMult.v



Note: saying “-g mkTest” on the bsc command line is equivalent to adding the (\*synthesize\*) attribute

# Limitation on the “(\*synthesize\*)” attribute

The “(\* synthesize \*)” is written just before a “module mkFoo (...);” header.

It can only be written before certain modules (*bsc* will complain if it is disallowed). This is because Verilog is less expressive than BSV:

- In BSV, module parameters and interfaces can contain arbitrary types, including functions, other interfaces, modules, etc.
- In Verilog, module parameters and ports can only carry bits, scalars and bit-vectors

Thus, the “(\* synthesize \*)” attribute can only be placed on those BSV modules whose parameters and interfaces can be mapped to Verilog parameters and ports.

*Important note:* when a BSV module cannot have a “(\*synthesize\*)” attribute, that does not mean that it cannot be used in synthesizable designs; it simply means that it cannot be *separately* synthesized. It can still be instantiated in other BSV modules which, in turn, can be synthesized. In this sense, *all* of BSV can be used in synthesizable code; there is no limitation of a “synthesizable subset” which is common in other High-Level Synthesis tools based on C and C++.

# Mapping BSV interfaces to Verilog ports

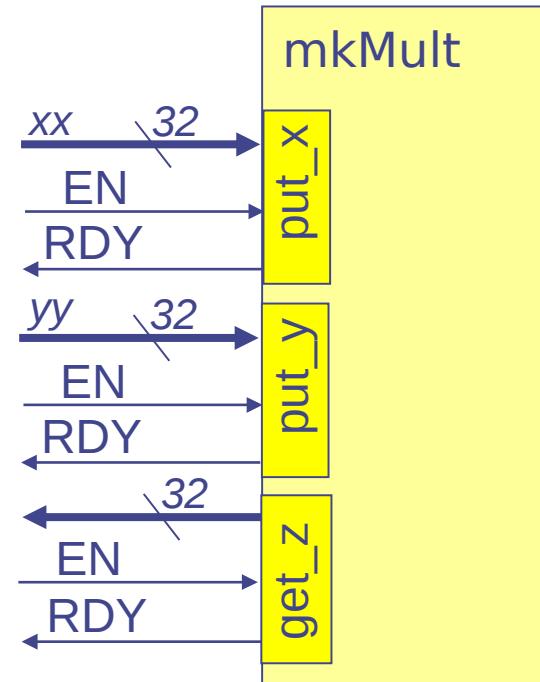
Interface methods are mapped directly to Verilog ports, based on their types:

- BSV method arguments become Verilog module input ports
- BSV method results become Verilog module output ports
- A BSV method condition becomes a Verilog “ready” (RDY) output signal
- BSV Action and ActionValue methods also have a Verilog “enable” (EN) input signal

# Mapping interfaces to HW: example

```
interface Mult_ifc;  
    method Action put_x (int xx);  
    method Action put_y (int yy);  
    method ActionValue #(int) get_z ();  
endinterface: Mult_ifc
```

- RDY = the method condition
- EN = signal asserted by external rule when it invokes an Action or ActionValue method (causes the internal Actions to happen)



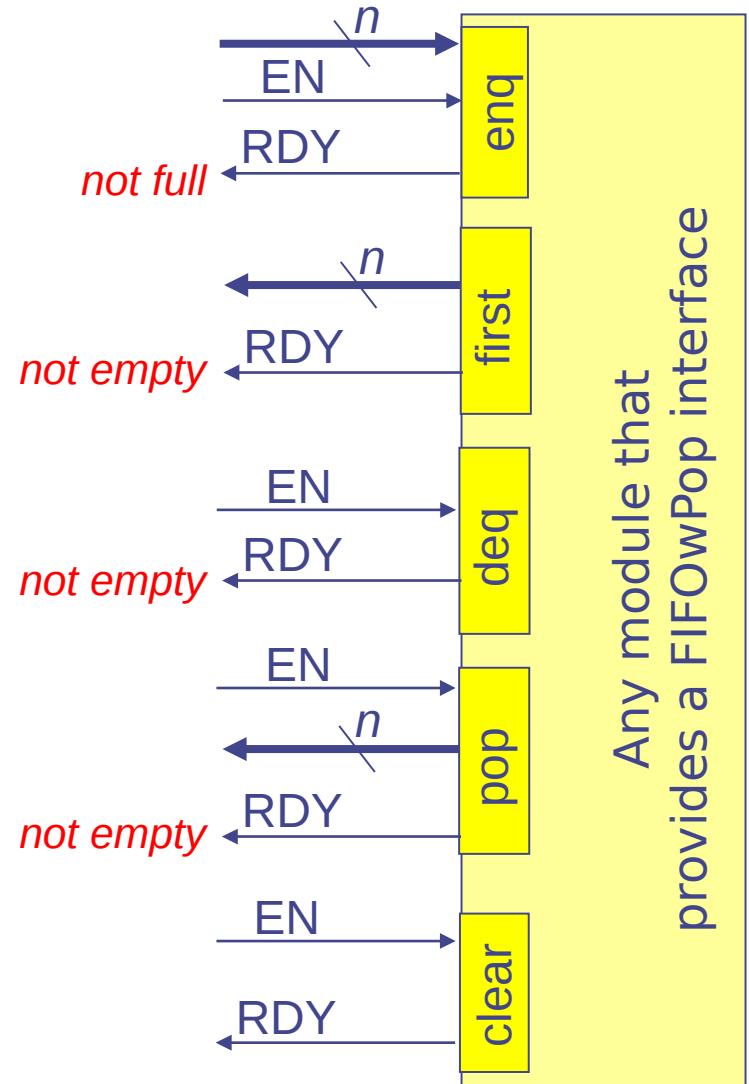
In a separate lecture on interfacing to RTL we'll see that it is possible:

- to eliminate the RDY signal when the method is always ready, and
- to eliminate the EN signal when external rule invokes the method on every clock.

Thus, Verilog ports are just a special case of BSV methods.

# Mapping interfaces to HW: another example

```
interface FIFOwPop #(type t);
    method Action enq (t x);
    method t first;
    method Action deq;
    method ActionValue#(t) pop;
    method Action clear;
endinterface
```

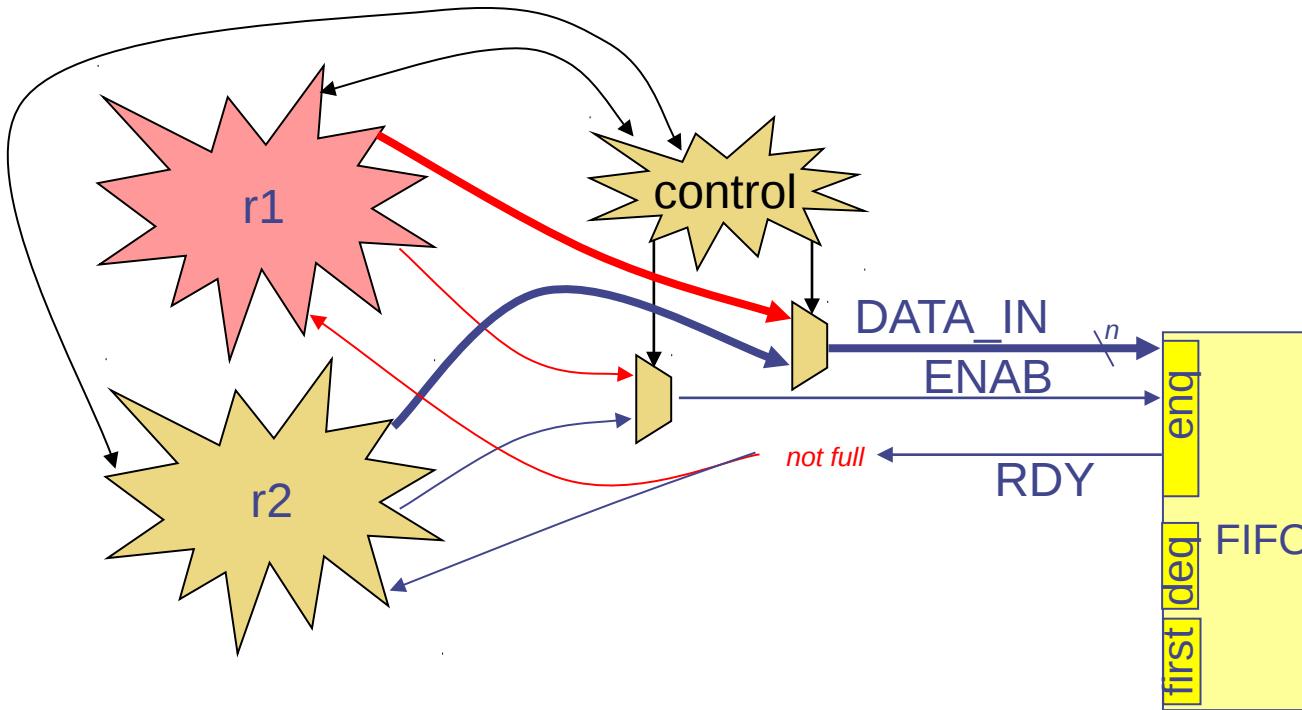


# Sharing: two rules invoking a common method

```
module mkTest (...);  
  ...  
  FIFO#(int) f <- mkFIFO;  
  ...  
  rule r1 (... cond1 ...);  
    ...  
    f.enq (... expr1 ...);  
  ...  
  endrule  
  
  rule r2 (... cond2 ...);  
    ...  
    f.enq (... expr2 ...);  
  ...  
  endrule  
endmodule: mkTest
```

```
interface FIFO#(type t);  
  Action enq (t n);  
  ...  
endinterface  
  
module mkFIFO (...);  
  ...  
  method enq(x) if (...notFull...);  
    ...  
  endmethod  
  ...  
endmodule: mkFIFO
```

# HW for a shared method



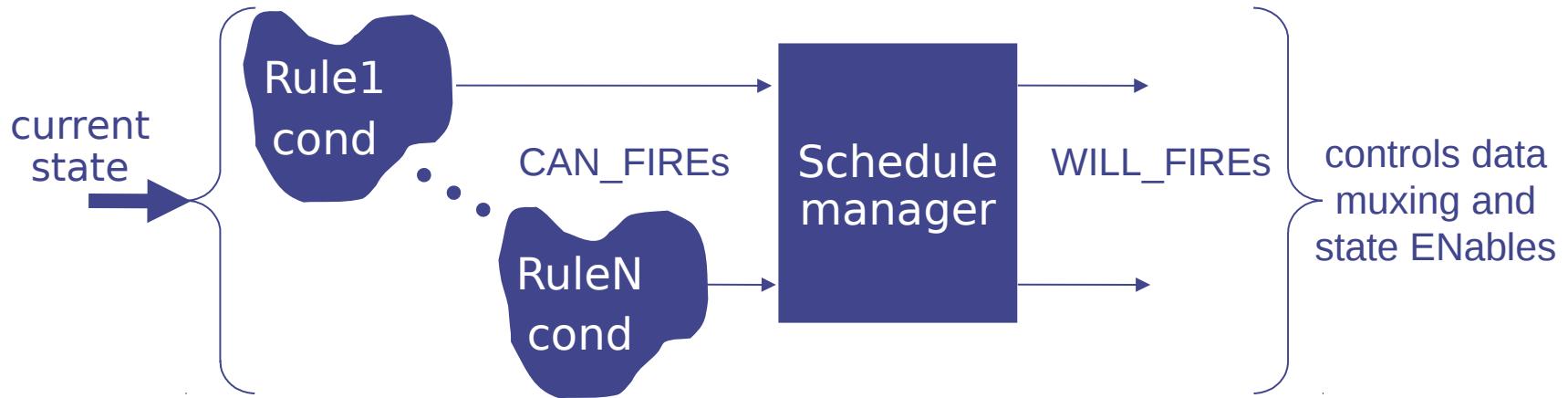
Note:

- Any input wire is potentially a rule resource conflict
  - (only one rule can drive it in each clock)
- Examples:
  - Argument of any method (whether value, Action or ActionValue)
  - EN signal of any Action or ActionValue method

Corollary:

- Only 0-argument value methods don't have resource conflicts (no input wires)

# CAN\_FIRE and WILL\_FIRE signals in synthesized HW

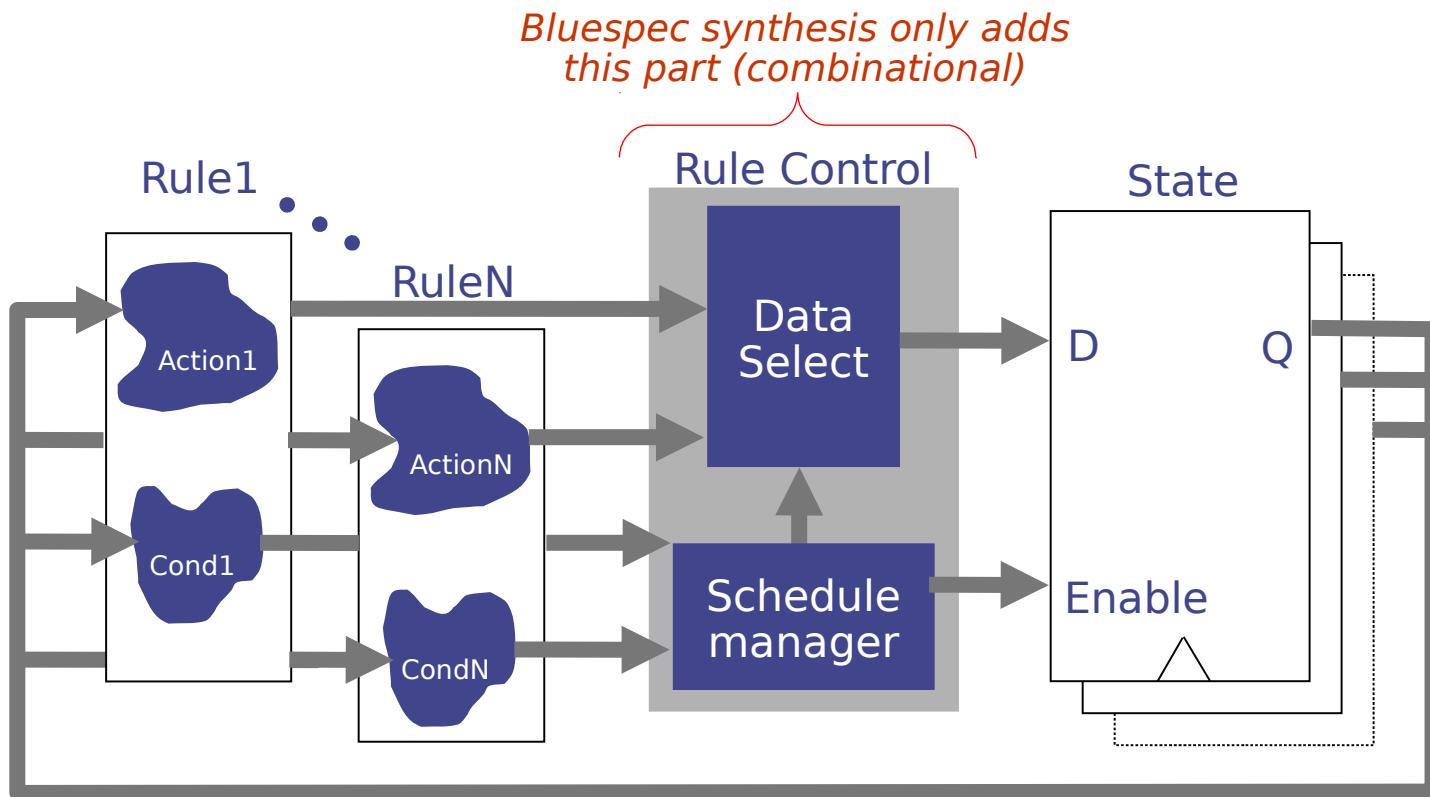


- The compiler performs conflict analysis of all rules in a BSV program, and generates a corresponding HW schedule manager (purely combinational)
- The CAN\_FIRE signal of a rule = its rule condition along with the conditions of methods that it invokes, directly or indirectly
- The WILL\_FIRE signal of a rule =  
 $\text{CAN\_FIRE} \&\& (\text{! WILL\_FIRE of any earlier rule that conflicts with this rule})$

Note:

- Conflict analysis requires sophisticated analysis of rule conditions and resources, potentially across module boundaries
- In practice, the “schedule manager” is not a monolithic circuit; it is distributed across modules and is built incrementally

# Overall schematic of synthesized HW



- The schedule manager ensures consistency with Rule logical semantics
  - Represents control logic that is normally hand-written in RTL in ad hoc ways, and usually the most error-prone part of RTL
  - Here, correct-by-construction, because of rule semantics
- Bluespec patented technology

# A small example to build HW intuitions about rules

Can you guess what these rules compute? (Hint: “Euclid”)

```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr

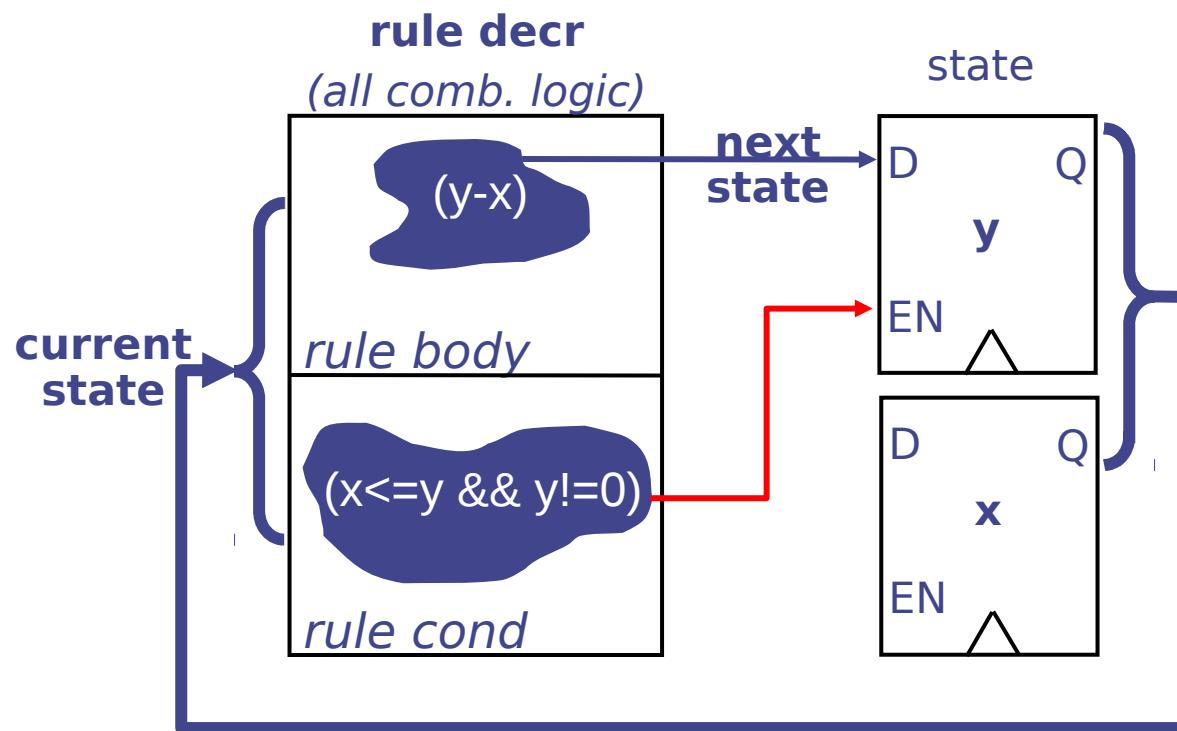
rule swap (x > y && y != 0);
    x <= y; y <= x;
endrule: swap
```

Answer:

*Euclid's algorithm for computing GCD (Greatest Common Divisor)  
of initial values in x and y registers; result is in x*

# HW for one of the rules

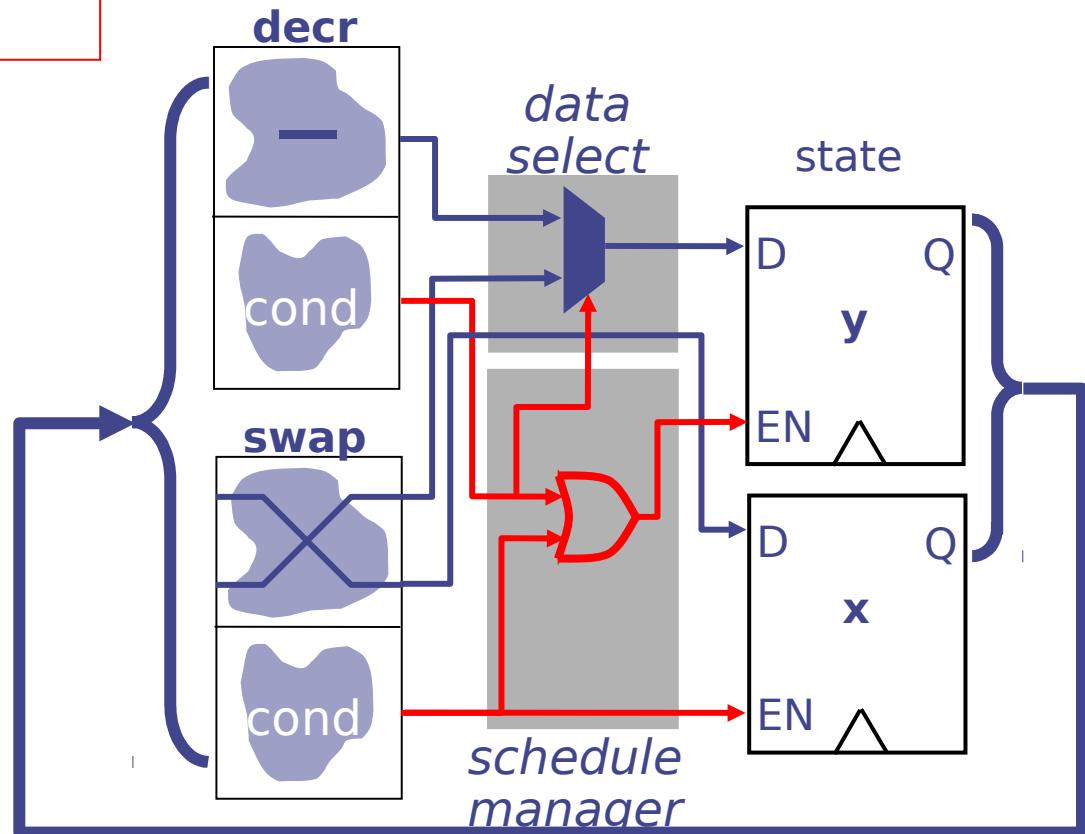
```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr
```



# HW for two mutually exclusive rules

```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr
```

```
rule swap (x > y && y != 0);
    x <= y; y <= x;
endrule: swap
```



# A brief word about clocks and resets

For BSV designs with a single clock and reset domain:

- There is no mention of clocks or resets in the BSV source code
- In the generated Verilog, every module has an additional CLK and RST input port, and these are connected to all sub-modules and state elements (including registers and memories)

*[ There is a separate lecture going into more detail on clocks and resets, including multiple clock and reset domains, clock domain-crossing synchronizers, positive and negative resets, synchronous and asynchronous resets, etc. ]*

# Hands-on

For any of the example codes in this training course, or in the BSV-by-Example book,

- Use *bsc* to compile it to Verilog using the “-verilog” flag (instead of the “-sim” flag which is used to create a Bluesim executable)
- Examine the generated Verilog files in light of the descriptions in this lecture
- Run the Verilog code in a Verilog simulator (3<sup>rd</sup> party, not supplied by Bluespec) and use the simulator’s facilities to examine the Verilog code, single-step it, generate and view waveforms, etc.
  - You can use commercial simulators like ModelSim (Mentor Graphics), VCS (Synopsys), NCSim/Incisive (Cadence), Riviera Pro (Aldec), ISim (Xilinx), etc.
  - You can use the free tools available for Linux: Verilog simulator iverilog and waveform viewer gtkwave



# End

```
import PFR#( );
typedef Bit#(8) Bmuf;
module end;
endinterface

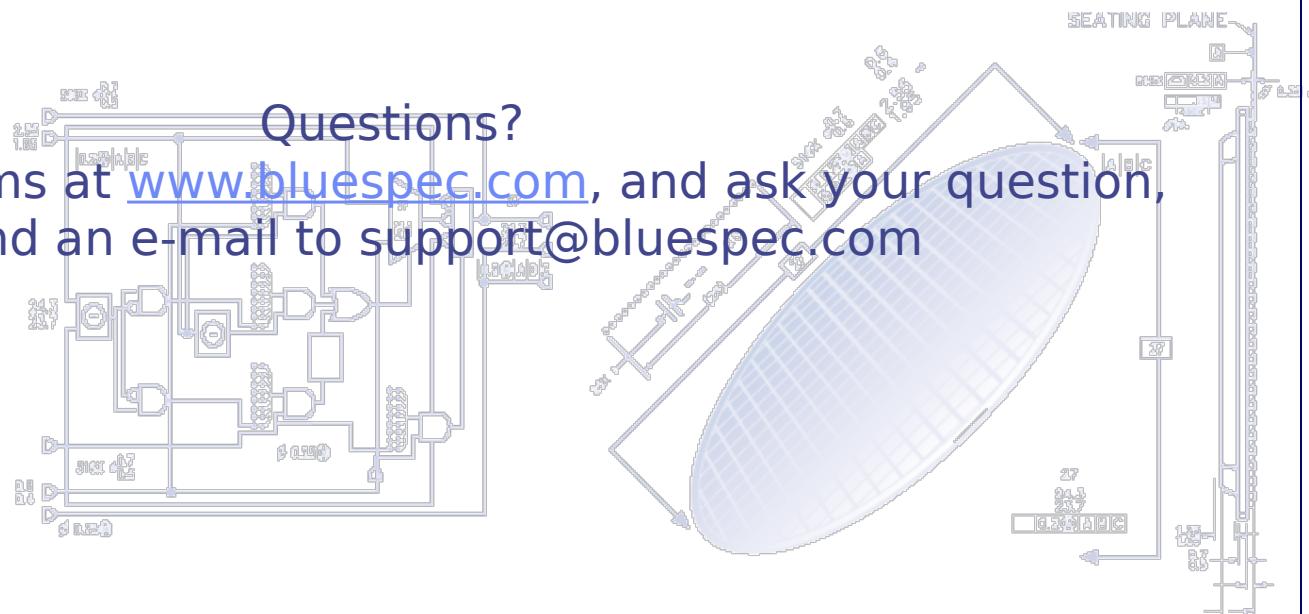
Integer FIFO#(Bit#(8)) determine_pmu(Bit#(8));
function Bit#(8) determine_pmu(Bit#(8));
    return {1{1'b0}, 7{1'b1}};
endfunction

PFR#(Bit#(8)) bmmu#(Bit#(8)) the_bmmu();
PFR#(Bit#(8)) emmu#(Bit#(8)) the_emmu();
PFR#(Bit#(8)) cmmu#(Bit#(8)) the_cmmu();
PFR#(Bit#(8)) ammu#(Bit#(8)) the_ammu();
PFR#(Bit#(8)) smmu#(Bit#(8)) the_smmu();

rule end (true);
    Bmuf b_muf = bmmu.first;
    PFR#(Bit#(8)) em_muf = emmu.first;
    if(em_muf.pmu_id == 0) emmu.enq(em_muf);
    else if(em_muf.pmu_id == 1) bmmu.enq(em_muf);
    else if(em_muf.pmu_id == 2) cmmu.enq(em_muf);
    else if(em_muf.pmu_id == 3) ammu.enq(em_muf);
    else if(em_muf.pmu_id == 4) smmu.enq(em_muf);
endrule
endinterface : end_end_ifc
```

Questions?

Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

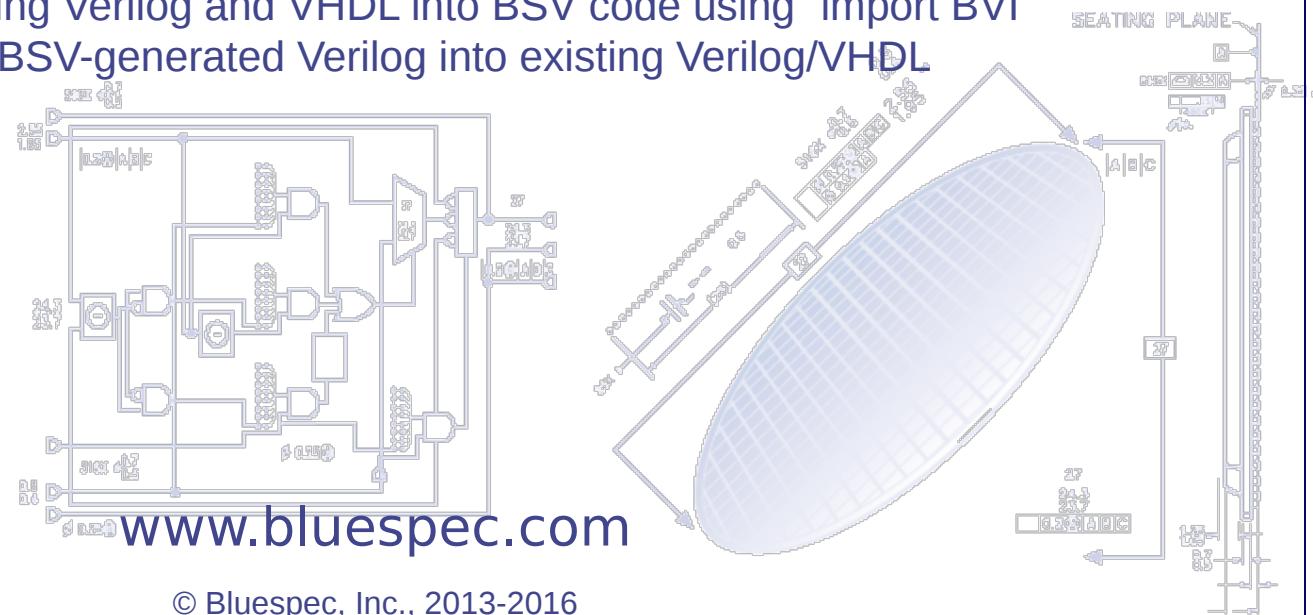




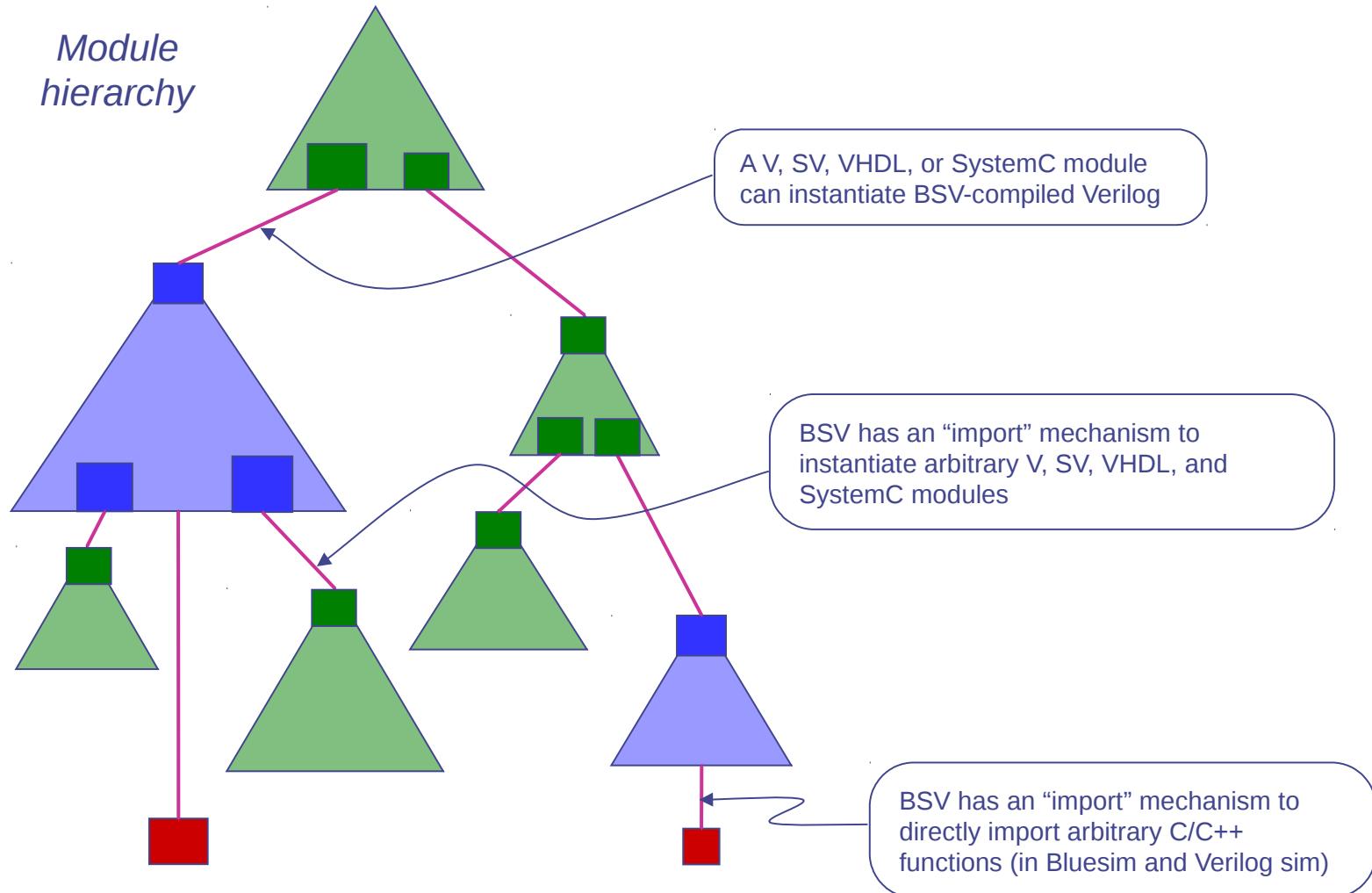
# BSV Training

## Lec\_Interop\_RTL

## Importing existing Verilog and VHDL into BSV code using “import BVI” Plugging BSV-generated Verilog into existing Verilog/VHDL



# BSV interoperates with V, SV, VHDL, SystemC, and C/C++



## Legend

- [Green square] V (Verilog), SV (SystemVerilog), VHDL, or SystemC (event-driven)
- [Blue trapezoid] BSV
- [Red square] C, C++

# Plugging RTL (or RTL-like SystemC) into BSV

# When does BSV interface to RTL (V, SV, VHDL)?

## *Plugging RTL into BSV:*

- Many BSV designs are used in projects where there is already some existing verified RTL IP, which we simply wish to re-use
- Users may wish to add a new “primitive” to BSV that is important for an application domain
  - In fact, all BSV “primitives” are imported this way—knowledge about primitives is not built into the *bsc* compiler

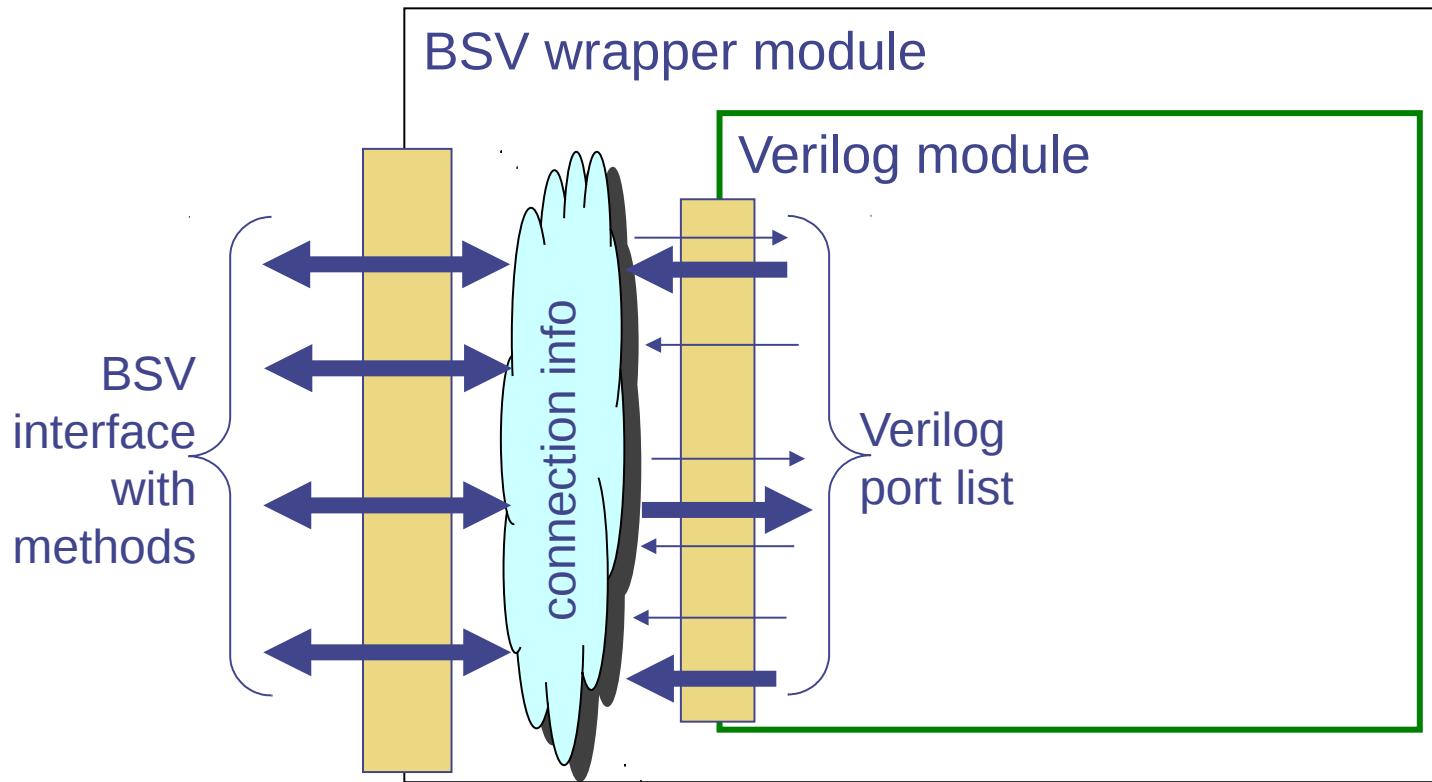
*This is done using BSV’s “import BVI” capability*

## *Plugging BSV into RTL:*

- A BSV design may fit into a larger RTL design
- A BSV design may be verified in an existing verification environment that is based on SystemVerilog (e.g., VMM), ‘e’, etc.

*This is done using by scheduling and naming control on the top-level interface, to be compatible with the RTL environment into which it fits*

# Basic structure of importing a Verilog component



- Define the BSV interface type that the module should provide to its environment
- Use the “import BVI” mechanism to define a wrapper module, which will:
  - Describe how the interface method arguments, results, enables, clocks, resets, etc. connect to the Verilog module ports
  - Describe the BSV *scheduling constraints* on the methods

# Example: importing a MAC (multiply-accumulate) module

The Verilog module we wish to import:

```
module mymac(EN, a, b, clear_value, clear, out, clk, rst_b);  
  
    input a, b, EN, clear, clear_value, clk, rst_b;  
    output out;  
  
    reg [15:0] out;  
    wire [15:0] a, b, clear_value;  
  
    always@(posedge clk or negedge rst_b)  
        if (!rst_b)  
            out <= 0;  
        else  
            out <= clear ? clear_value : (EN ? out+a*b: out);  
endmodule
```

- When  $\text{clear} == 1$ ,  $\text{out} <= \text{clear\_value}$
- When  $\text{EN} == 1$ ,  $\text{out} <= \text{out} + a * b$
- The value of the “out” register is continuously available as an output

# Define the desired BSV interface

```
interface Mac_IFC ;  
    method Action acc (Int#(16) aa, Int#(16) bb);  
    method Action reset_acc (Int#(16) value);  
    method Int#(16) read_y;  
endinterface
```

- When clear == 1, out <= clear\_value
  - *The related Verilog signals “clear” and “clear\_value” will be part of the BSV “reset\_acc” method*
- When EN == 1, out <= out + a \* b
  - *The related Verilog signals “EN”, “a” and “b” will be part of the BSV “acc” method*
- The value of the “out” register is continuously available as an output
  - *This will be done with the “read\_y” method*

# Define the module wrapper

Verilog module name

BSV module name

```
import "BVI" mymac =
```

```
module mkMac(Mac_IFC);
```

```
default_clock dclk (clk);
```

```
default_reset dreset (rst_b);
```

```
method acc(a, b)
```

```
enable(EN);
```

```
method reset_acc(clear_value) enable(clear);
```

```
method out read_y();
```

```
schedule (read_y) SB (reset_acc, acc);
```

```
schedule (acc) C (reset_acc);
```

```
endmodule
```

} *Clock and reset*

} *Methods*

} *Scheduling constraints*

- The “default\_clock” line indicates that the BSV default clock will be connected to Verilog “clk”, and that it will be known within mkMac as “dclk” (although in this example it is not used elsewhere; typically it’s used in a “clocked\_by” clause).
- In the “method” lines we can see how method signals (arguments, results, ENABLEs) connect to various Verilog ports (which are shown in green). Since the optional “ready()” clauses are missing, these methods will be always ready.

# Define the module wrapper

Verilog module name

BSV module name

```
import "BVI" mymac =  
  module mkMac(Mac_IFC);
```

```
    default_clock dclk (clk);  
    default_reset dreset (rst_b);
```

```
    method acc(a, b) enable(EN);  
    method reset_acc(clear_value) enable(clear);  
    method out read_y();
```

```
    schedule (read_y) SB (reset_acc, acc);  
    schedule (acc) C (reset_acc);
```

```
endmodule
```

} *Clock and reset*

} *Methods*

} *Scheduling constraints*

The “schedule” lines inform the *bsc* compiler about scheduling constraints on the methods.

- The first line indicates that `read_y < reset_acc` and `read_y < acc` (“SB” = “sequenced before” = “<”). This captures the Verilog semantics that “out” carries the current value in the register, and that any change due to `reset_acc` or `acc` will only be visible on the next clock.
- The second line indicates that `acc` and `reset_acc` conflict (“C”), so they can never fire concurrently (in the same clock). This design choice is stricter than the Verilog, which *does* allow both to happen in the same clock, giving priority to `reset_acc`. To match those semantics we could have instead specified: `read_y SB acc` and `acc SB reset_acc`

# Notes on the import Verilog mechanism

The previous slides showed only a simple example. The Reference Guide Sec. 15 goes into a lot more detail:

- Connecting clocks and resets
- Connecting method ENABLE and RDY signals
- The method-to-RTL connections need not be 1-to-1, and can involve other logic
- The available scheduling annotations
- Caveat: the schedule is just an assertion by you taken at face value by the compiler when using this module in your BSV program; the compiler makes no attempt to ‘verify’ that the scheduling assertions are sensible

The actual Verilog code is not touched by the *bsc* compiler. The *bsc* compiler simply generates a Verilog instantiation of the module, and your RTL simulator finds and instantiates the Verilog module

These import mechanisms can also be used for importing VHDL and SystemVerilog, and RTL-style SystemC

- Most RTL simulators allow free intermixing of Verilog, VHDL, SystemVerilog and SystemC

# What about Bluesim, when you import RTL?

Bluesim does not currently support “co-simulation” of Bluesim with a Verilog simulator

- If you just import a Verilog module, your only simulation option is Verilog simulation, i.e.,
  - Use *bsc* to generate Verilog from the BSV code
  - Use a Verilog simulator to simulate all the Verilog code (BSV-generated, and imported)

How do the primitives in the BSV library work in Bluesim?

- Every primitive in the BSV library is implemented both in Verilog and in C
  - The C code is imported using the “import BDPI” mechanism (described later), into a module with exactly the same BSV interface
- For every primitive, we have a wrapper module that instantiates one or the other, like this:

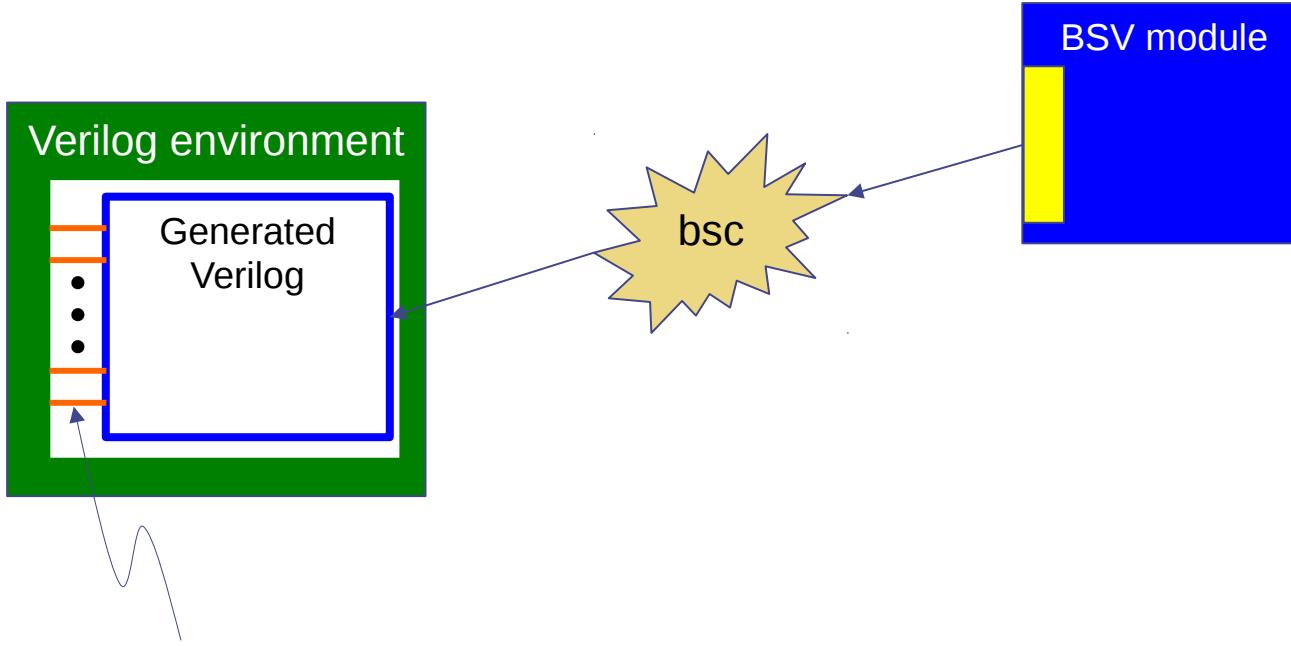
```
IfcType m <- if (genVerilog) mkVerilogWrapper;  
else mkCWrapper;
```

- “genVerilog” is a built-in boolean which is true when compiling for Verilog and False when compiling for Bluesim (Reference Guide Sec. B.6). Thus, at compile time, *bsc* chooses the appropriate instantiation (i.e., this is a “static elaboration” choice).

# Plugging BSV into RTL

# Plugging BSV into RTL: key issues

When plugging BSV into RTL (more accurately: BSV-generated RTL into RTL), the key issues are:



We need:

- Exactly the right set of input and output signal ports, with exactly the right port names, as expected by the RTL environment
- Exactly the right signaling protocols on these ports, as expected by the RTL environment (may be different from standard BSV method protocol of RDY and EN signal)

Example: the “Verilog environment” may be an AMBA AXI bus port

# Exposing method conditions

- A method's condition can always be exposed as an explicit Bool method
- The examples below are from the BSV library
  - Note: in the lib FIFOF, enq, first and deq are still guarded by their implicit conditions; the notFull and notEmpty methods are just additional methods

```
interface FIFO#(type t);
  // enq has "notFull" condition
  method Action      enq(t x);
  // first/deq have "notEmpty" condition
  method t          first;
  method Action      deq;
  method Action      clear;
endinterface: FIFO
```

```
interface FIFO#(type t);
  // enq has "notFull" condition
  method Action      enq(t x);
  // first/deq have "notEmpty" condition
  method t          first;
  method Action      deq;
  method Action      clear;
  method Bool      notEmpty;
  method Bool      notFull;
endinterface: FIFO
```

# Exposing method conditions

- Implementing exposed conditions is trivial: just replicate the method condition
  - E.g., below, enq()'s condition canEnque is returned explicitly as a Boolean in notFull()
  - Don't worry: the generated code will share a single instance of the condition logic
    - Or, share it explicitly by writing let x = canEnque and using x twice

```
module mkFIFO (FIFO#(type t));  
  ...  
  method Action enq(t x) if (canEnque); ... endmethod  
  method t first if (canDequeue); ... endmethod  
  method Action deq if (canDequeue); ... endmethod  
  
  method Bool notFull; return canEnque; endmethod  
  method Bool notEmpty; return canDequeue; endmethod  
endmodule: mkFIFO
```

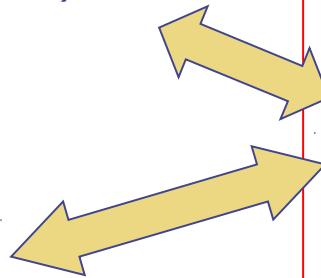
# Removing RDY signals

- The “always\_ready” attribute can be applied to a method to indicate
  - that it is always ready (the compiler will check this!)
  - that the RDY signal must be removed from the generated Verilog

```
(* always_ready = "add, result" *)
module mkAdder (Adder);
    Reg#(int) acc <- mkRegA(0);

    method Action add(a, b);
        acc <= a + b;
    endmethod

    method int result;
        return acc;
    endmethod
endmodule
```



```
interface Adder;
    method Action add(int a, int b);
    method int      result;
endinterface
```

*The compiler will check that add and result are always ready to be invoked, i.e. their implicit and explicit conditions are always True*

*The compiler will not generate any RDY signal for add and result*

# Removing ENABLE signals

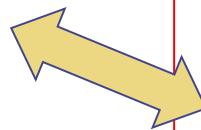
- The “always\_enabled” attribute can be attached to a method (Action or ActionValue) to indicate
  - that it is assumed True, i.e., the data input signals (args) are driven by the environment on every cycle
  - Of course, this implies “always ready”, i.e., the method’s condition must be always True (otherwise it would be an error to drive the EN signal)
  - that the EN signal must be removed from the generated Verilog

```
(* always_enabled = "add" *)
module mkAdder (Adder);
  Reg#(int) acc <- mkRegA(0);

  method Action add(a, b);
    acc <= a + b;
  endmethod

  method int result;
    return acc;
  endmethod
endmodule
```

```
interface Adder;
  method Action add(int a, int b);
  method int      result;
endinterface
```



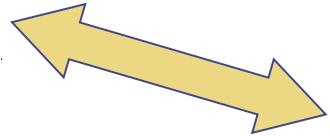
*The compiler will check that add can be always invoked, i.e. its method condition is always True*

*The compiler will not generate any EN signal for add*

# Unguarded methods

- Suppose a module must read datum on EVERY CYCLE

```
(* always_enabled = "accept" *)
module mkFoo (IfcFoo);
  FIFO#(int) fifo <- mkFIFO;
  ...
  method Action accept (int);
    fifo.enq(bus);
  endmethod
endmodule
```



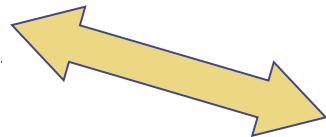
The bsc compiler will signal an error, since the enq() method may not be ready (the fifo may be full).

*It cannot verify that accept() is always ready, so it cannot allow the always\_enabled spec*

# Unguarded methods

- Answer: use “unguarded” methods, i.e., methods with no conditions, relying on you to guard them explicitly
  - The BSV library provides FIFOs with unguarded methods
  - Warning: these are dangerous! Use with care, only in these “impedance matching” situations

```
(* always_enabled = "accept" *)
module mkFoo (IfcFoo);
    FIFO#(int) fifo <- mkUGFIFO();
    ...
method Action accept (int);
    fifo.enq(bus);
endmethod
endmodule
```



*This will work.*

*Warning: you have to use some other means  
to avoid buffer overflow!*

# Summary: plugging BSV into RTL

- A BSV Action method with one argument that is always ready and always enabled becomes, exactly, a Verilog input port. E.g.,

```
(* always_ready, always_enabled *)
method Action accept (int);
```

- A BSV value method that is always ready becomes, exactly, a Verilog output port. E.g.,

```
(* always_ready *)
method int yield_data;
```

- Sec. 13.2.1 shows attributes by which you can control the exact names of these generated Verilog ports

*With these capabilities, and unguarded methods, you can produce a Verilog interface with any desired ports, with any desired names, and with any desired behavior (protocol)*

- *The BSV AXI library has examples of interfaces to the AMBA AXI bus (masters, slaves, etc.)*

# Hands-on

- BSV-by-Example book: Examples in Chapter 15



End

# Questions?

Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

```

import FIFOF#(bit);
typedef FIFOF#(bit) bitF;
module ex_1_bit_level;
    Integer fifodepth;
    function bitF#(bit) determine_pump(bitT val);
        return bitF{val};
    endfunction

    FIFOF#(bit) bitqueue;
    bitT last;
    bitT the_bitread();
    bitT the_bitread(bitT b);
    bitT the_bitread(bitT b, bitT c);
    bitT the_bitread(bitT b, bitT c, bitT d);
    bitT the_bitread(bitT b, bitT c, bitT d, bitT e);

    rule read();
        bitT b_bit = bitqueue.pop();
        if (b_bit == last)
            $display("The bit read is %b", b_bit);
        else
            $display("The bit read is %b", b_bit);
        last = b_bit;
    endrule
endmodule

```



# BSV Training

Lec\_Interop\_C

Importing C code into BSV (for simulation only) using “import BDPI”.  
Exporting a BSV subsystem into a SystemC program

```

import FIFO::*;

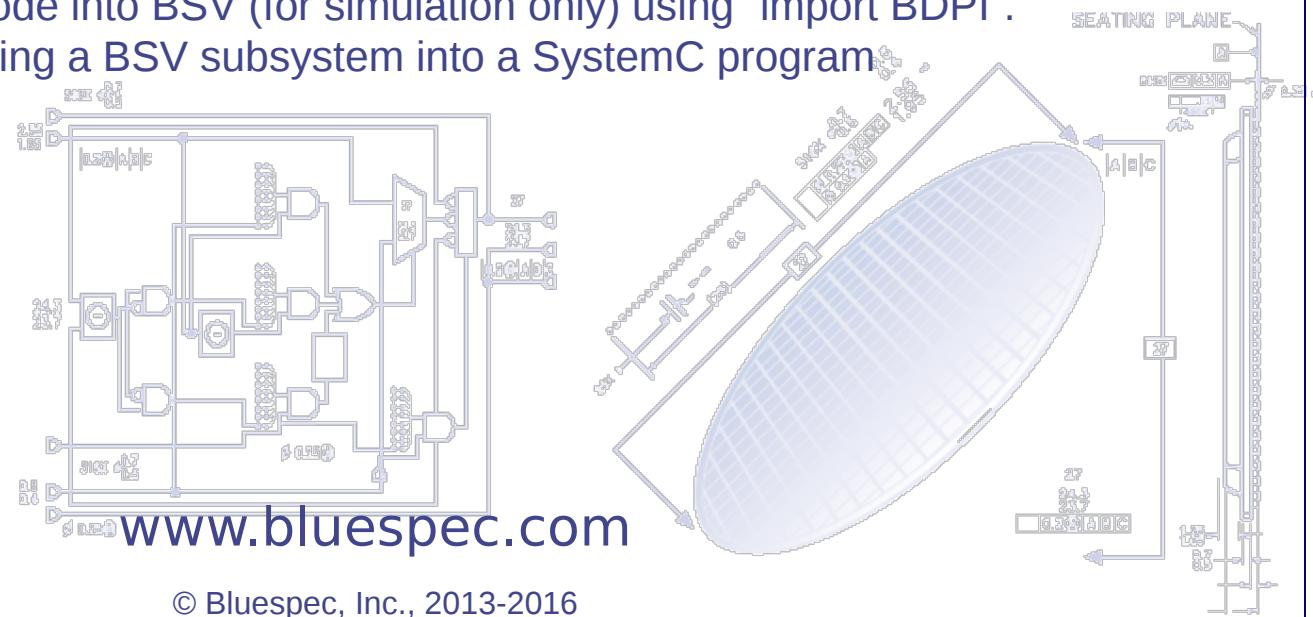
typedef FIFO#(Bit#(8)) bitmap;
module CtrlUnit#(Bit#(8));
    Integer fifoDepth = 16;

    function Bit#(8) determine_pump(Bit#(8) val);
        return {val[7], val[5:0]};
    endfunction

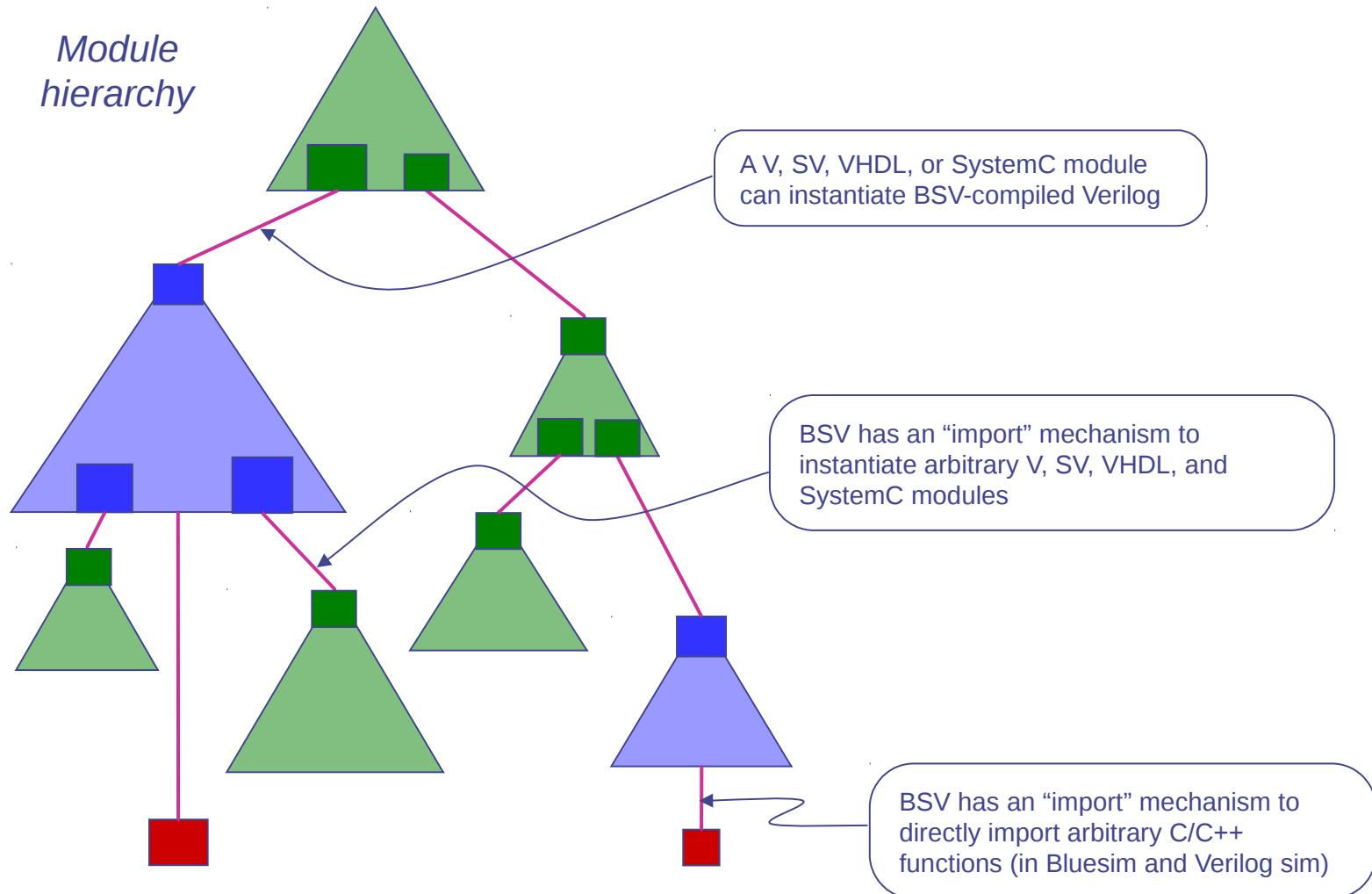
    FIFO#(Data) libmem;
    FIFO#(Bit#(8)) the_bitmap(bitmap);
    FIFO#(Bit#(8)) the_circshift(circshift);
    FIFO#(Bit#(8)) the_circread(circread);
    FIFO#(Data) the_circwrite(circwrite);
    FIFO#(Bit#(8)) the_stitch(stitch);

    rule end (true);
        if((!the_bitmap.empty() && !the_circshift.empty()) ||
           (!the_circread.empty() && !the_circwrite.empty()) ||
           (!the_stitch.empty()))
            begin
                Data tempData = determine_pump(the_bitmap.first());
                if((tempData == 0) ? circread : circshift);
                else if((tempData == 1) ? circwrite : stitch);
                the_bitmap.pop();
                the_circshift.pop();
                the_circread.pop();
                the_circwrite.pop();
                the_stitch.pop();
            end
    endrule
endmodule

```



# BSV interoperates with V, SV, VHDL, SystemC, and C/C++



## Legend

- [Green square] V (Verilog), SV (SystemVerilog), VHDL, or SystemC (event-driven)
- [Blue square] BSV
- [Red square] C, C++

# Plugging C/C++ into BSV

# Motivations for importing C/C++

- Many applications begin life as C/C++ models:
  - When you are creating a HW accelerator for an existing SW program
  - When you are trying to choose the best algorithm before going to HW implementation
- As you incrementally develop your BSV implementation, you may wish to reuse C/C++ components temporarily (or even permanently, for the testbench)
- When you add a new “primitive” to BSV by importing Verilog, you may also want to import a corresponding C model for running in Bluesim
  - In fact, all BSV “primitives” are imported this way—the knowledge is not built into the compiler

# Importing C

- You declare a BSV function prototype (i.e., just the types) which is then implemented in C
- Example:

```
// BSV code
```

```
import "BDPI" rand32 =  
    function ActionValue #(Bit#(32)) bsv_rand32();  
  
module test(Empty);  
    FIFO#(Bit#(32)) myFIFO <- mkSizedFIFO(9);  
  
    rule fill;  
        let x <- bsv_rand32;  
        myFIFO.enq(x);  
    endrule  
  
    rule empty;  
        myFIFO.deq;  
        $display("Number %d", myFIFO.first);  
    endrule  
endmodule
```

```
// C code
```

```
#include <stdio.h>  
#include <stdlib.h>  
  
unsigned int rand32()  
{  
    return (unsigned int) rand();  
}
```

# Importing C: arguments and results

- There is a 1-to-1 correspondence between the arguments and result in the BSV prototype and in the C function.
- Allowed arguments types
  - any type that is in the Bits#() typeclass
  - String
  - polymorphic types
- Allowed result types:
  - any type that is in the Bits#() typeclass
  - Action
  - ActionValue#(t) where t is in the Bits#() typeclass
  - polymorphic types (and polymorphic ActionValue #(t) types)
- For “small” types ( $\leq 64b$ ), the corresponding C argument or result is the nearest C scalar integer type adequate to contain it (char, short, long, long long)
- For “larger” types and polymorphic types, the C function receives a “void \*” pointer to the storage for that value
  - This storage contains the “raw bits” of the BSV representation
  - For return types that are passed using “void \*” like this, the “void \*” pointer is passed as the first argument of the C function, not returned as the C function value
- Details and examples in the Reference Guide, Sec. 16

# Recommendation: arguments and results

- Although the ‘import C’ arguments and results can be of arbitrary type, it’s usually too much effort, and too error-prone, and too non-portable to exploit this capability
  - The representation of C types varies across C compilers and across target architectures (insertion of padding for word alignment, little- and big-endianness, etc.)
- **Recommendation:**
  - a) Stick to just a few standard, highly portable types: 32b and 64b scalars, and arrays/vectors thereof.
    - In the C code, use standard types declared in <stdint.h>: `uint32_t`, `int32_t`, `uint64_t`, `int64_t`, and arrays of those types
    - In the BSV code, correspondingly, use `UInt#(32)`, `Int#(32)`, `UInt#(64)`, `Int#(64)`, and `Vector#(n,...)` of those types
  - b) In both the C code and the BSV code, write functions to convert from “native” types (enums, structs, unions) to standard types (a) and back.
    - Being purely in C and BSV, such conversion functions are easy to write and very predictable and reliable
    - E.g., convert a struct to and array of `uint32_t` or `uint64_t`
    - E.g., convert a C pointer to a `uint64_t` (pointers may not fit in `uint32_t`)
  - c) In both the C code and the BSV code, use these functions on arguments and results so that only these standard types (a) are passed between C and BSV

# Linking in the imported C function(s)

- Compilation of an “import BDPI” produces intermediate “.ba” files

```
# bsc -u -sim DUT.bsv
checking package dependencies
compiling DUT.bsv
Foreign import file created: compute_vector.ba
code generation for mkDUT starts
Elaborated Bluesim module file created: mkDUT.ba
code generation for mkTB starts
Elaborated Bluesim module file created: mkTB.ba
```

- In the link stage (for Bluesim or Verilog sim, you supply these .ba files and the C file

```
# bsc -sim -e mkTB -o bsim mkTB.ba mkDUT.ba compute_vector.ba vectors.c
Bluesim object created: mkTB.{h,o}
Bluesim object created: mkDUT.{h,o}
Bluesim object created: schedule.{h,o}
User object created: vectors.o
Bluesim binary file created: bsim
```

When you compile for Verilog sim, bsc generates the appropriate “VPI”-like linkage files that allows your Verilog simulator to import the C code (for most popular simulators like VCS, NCSim, Modelsim, iVerilog, CVC, etc.)

# Importing C++ (as opposed to C)

The BSV “import BDPI” mechanism can only directly invoke C functions (it assumes function linkage conventions for C, and C++ typically has different linkage conventions).

C++ has an ‘extern “C”’ construct that tells the C++ compiler to compile a function with C linkage instead of C++ linkage. That function, in turn, can freely call C++ functions.

Thus, the following example illustrates the idiom for calling C++ from BSV:

- We use “import BDPI” to invoke a function `myMainC_function()` which has C linkage
- This, in turn, calls the desired C++ function `myMainCPP_function()`:

```
// File foo.cpp  (C++)

// This is the C++ function we would like to invoke from BSV
int myMainCPP_function(int a, int b) {
    ... C++ code ...
}

extern "C" { // This is the function imported into BSV
    void myMainC_function (int a, int b)
    {
        return myMainCPP_function (a, b);
    }
}
```

# Plugging BSV into SystemC

# Plugging BSV into SystemC

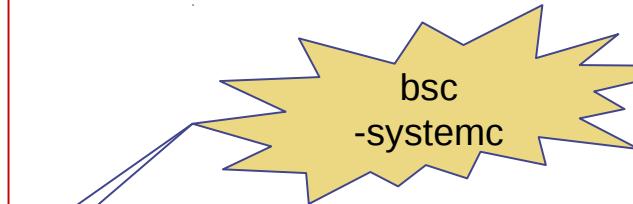
The `bsc` compiler directly supports, via a command-line flag, the creation of a “plug-in” into a SystemC program (User Guide Sec. 4.3.2)

```
// File mkFoo_systemc.h (SystemC)

SC_MODULE (mkFoo)
{
    public:
        sc_in<...> ...;
        sc_out<...> ...;
        ...
    public:
        SC_CTOR (mkFoo) ...
        ~mkFoo() ...
}
```

```
// File mkFoo_systemc.cxx
// C++ code (not SystemC)
// implementing the declarations
// in the .h file
...
...
```

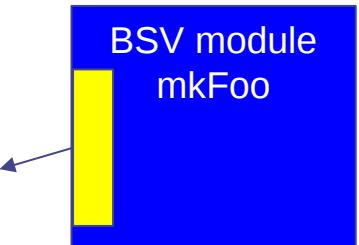
(Bluespec tests this facility with the standard OSCI SystemC simulator, but expects it to work with any SystemC simulator)



*The .h file is standard SystemC. It declares `mkFoo` as a SystemC `SC_MODULE`. The `sc_in` and `sc_out` ports are SystemC signal declarations, corresponding exactly to what you would have got if you had compiled the BSV code to Verilog.*

*You should “#include” this .h file in your SystemC program, which can then invoke the constructor `mkFoo` to instantiate the module, and use the `sc_in` and `sc_out` ports to communicate with it, in the usual SystemC way.*

*The .cxx file is compiled with your C++ compiler as usual, and linked in with the rest of your SystemC program’s object files, to create the SystemC executable.*



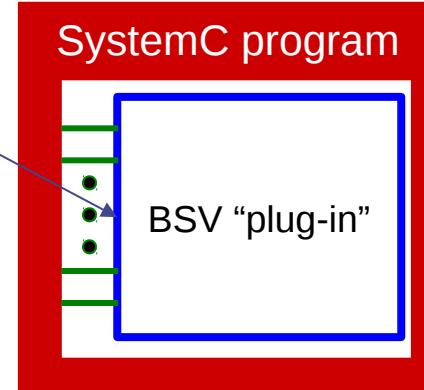
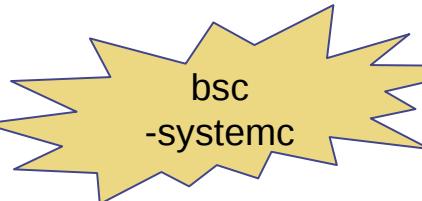
# Plugging BSV into SystemC

Note: Only the .h file is intended for human readability.

Although the .cxx file is C++ source code, it is a complex translation of the cycle-level behavior of the source BSV module and does not try to mirror the BSV source structure in any recognizable way.

```
// File mkFoo_systemc.cxx  
// C++ code (not SystemC)  
// implementing the declarations  
// in the .h file  
  
...
```

*The .cxx file is compiled with your C++ compiler as usual, and linked in with the rest of your SystemC program's object files, to create the SystemC executable.*





End

# Questions?

Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

```
import PFPDF;
typecast! ElstPFPDF! docPDF;
modules ex_11_Letters_Letterhead;
Integer fileNumber;
function ElstPFPDF! determine_header(Bool bHeader);
    return {<img alt="Logo" style="vertical-align: middle; margin-right: 10px;"> <span>ABC Company</span>};
```

# Join online for or

```
function ElstPFPDF! determine_header(Bool bHeader);
    return {<img alt="Logo" style="vertical-align: middle; margin-right: 10px;"> <span>ABC Company</span>};
```

## File and Print

```
    Bool b_file = bHeader ? true : false;
    ElstPFPDF! printPDF() out_file = null;
    determine_header(out_file) == 0 ? out_file : null;
    ElstPFPDF! printPDF() out_file;
    out_file : printPDF();
endcode;
```

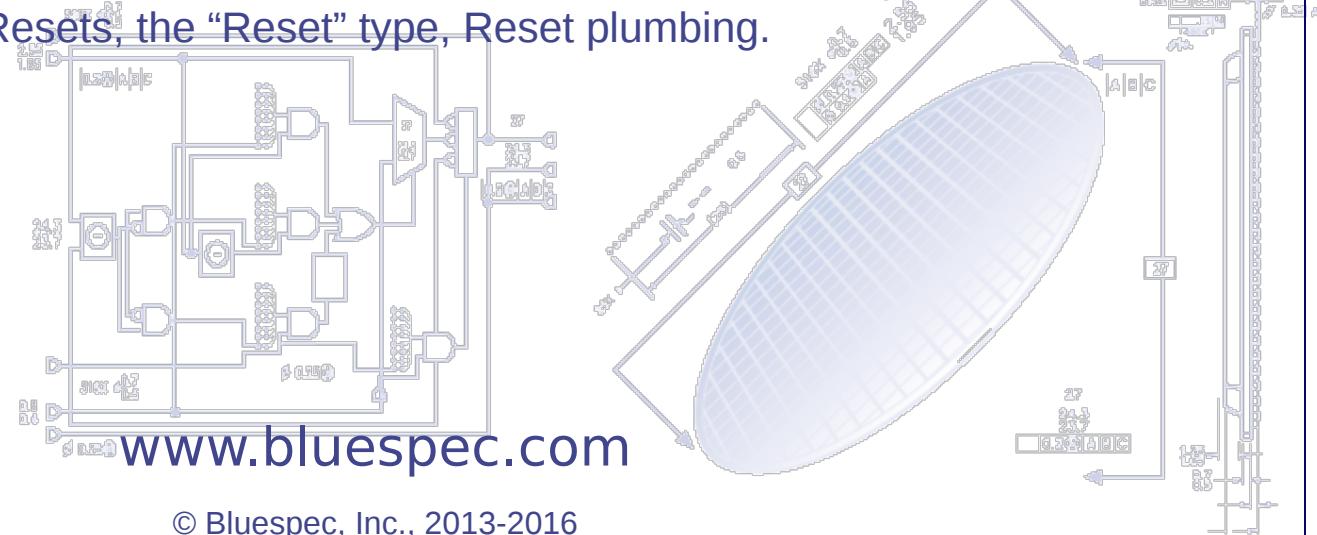
```
endcode; // ex_11_Letters_Letterhead
```

# BSV Training

## Lec\_Multiple\_Clock\_Domains (MCD)

Clocks, the “Clock” type, clock plumbing (passing clocks down the module hierarchy to modules and primitives), clock domains and clock discipline, synchronizers for clock domain-crossing.

Resets, the “Reset” type, Reset plumbing.



# Introduction

- Most hardware systems have components driven by multiple clocks, for various engineering reasons (performance, power, standards, silicon technology, legacy, etc.).
- In modern systems, clock signals are special. Clock signals are not used as ordinary signals, and vice versa.
- In modern systems, clocks are often *gated*, so that they can be switched off under certain conditions to avoid power consumption by components.
- The set of components sharing a common clock is known as a *clock domain*. Clock domains are traditionally disjoint.
- The organization of a system into clock domains may not be the same as its organization into modules, i.e., a module may involve multiple clocks.
- Communications between clock domains must usually go through specially engineered circuits called *synchronizers*, to avoid “metastability” problems.

In this lecture, we describe:

- The “Clock” type (different from all other types), clock gating, and clock “families” (that differ only in gating condition)
- Clock “plumbing”: how clocks are passed down the module hierarchy and to primitive modules (including, for example, registers)
- Clock domains in BSV
- Synchronizers (communicating between clock domains)

(we will also briefly discuss Resets, at the end)

## The Clock type, gated clocks, and clock families

# The Clock type

- There is a primitive type “Clock” in BSV
  - Because of BSV’s strong type checking, entities of type Clock can never be accidentally used as ordinary values, or vice versa
  - The full power of BSV’s static elaboration is available to manipulate clocks:
    - Can write expressions with clocks
    - Can write functions on clocks
    - Can create data structures (e.g., arrays) of clocks
    - etc.
- These are all statically resolved (no dynamic clock selection)

```
Clock c1, c2;  
  
Clock c = (b ? c1 : c2); // b must be known at compile time
```

# BSV supports gated clocks

- Gated clocks are common in modern designs. By gating a clock “off”, it stops switching activity in a circuit, and therefore reduces power consumption.
- In BSV, a gated clock consists of two signals
  - an oscillator
  - a gating signalimplemented (in the generated Verilog) with two wires
- If the gate is True, oscillator is running
  - If the gate is False, oscillator may or may not be running, depends on implementation library—tool doesn’t care
- Clock gate conditions integrate smoothly into rule semantics, i.e., clock gate signals contribute to method and rule conditions (more about this later)

# Clock gating is optional

- By default, a Clock input to a module is ungated (and has a single wire)
- By using certain attributes at module synthesis boundaries, one can override this default and request either all input clocks to be gated, or selected input clocks to be gated (gated clocks have two wires)

All these modules have two input clocks

```
(* synthesize *)
module mkMod1#(Clock clk2) (Ifc);
  ...
endmodule
```

Default clock: *ungated*      clk2: *ungated*

```
(* synthesize, gate_all_clocks *)
module mkMod2#(Clock clk2) (Ifc);
  ...
endmodule
```

Default clock: *gated*      clk2: *gated*

```
(* synthesize, gate_input_clock = "default_clock"
*)
module mkMod3#(Clock clk2) (Ifc);
  ...
endmodule
```

Default clock: *gated*      clk2: *ungated*

```
(* synthesize, gate_input_clock = "clk2" *)
module mkMod3#(Clock clk2) (Ifc);
  ...
endmodule
```

Default clock: *ungated*      clk2: *gated*

# Clock families

- Clocks that share the same oscillator, and differ only in gating, are termed a “clock family”.
  - The *bsc* tool keeps track of clock families, to avoid unnecessary synchronization (clocks in the same family are in the same clock domain, since their edges will never be so close as to cause metastability problems)
- If  $c_2$  is a gated version of  $c_1$ , we say  $c_1$  is an “ancestor” of  $c_2$ 
  - If some clock is running, then so are all its ancestors, i.e., gating only increases down the ancestor relationship
- The functions `isAncestor(c1,c2)` and `sameFamily(c1,c2)` are provided to test these relationships
  - Can be used to control static elaboration (e.g., to optionally insert or omit a synchronizer)

# Making gated clocks with mkGatedClock

A gated clock can be created from an existing clock (gated or ungated) using the following module:

```
module mkGatedClock #(Bool v) (Clock clk_in, GatedClockIfc ifc);
```

The boolean parameter is the state on reset (True = gate on, False = gate off). The clk\_in parameter is the clock to be gated. It provides the following interface:

```
interface GatedClockIfc;
    method Action setGateCond(Bool gate);
    method Bool getGateCond();
    interface Clock new_clk;
endinterface
```

The new, gated clock is new\_clk. Its gating condition can be changed using setGateCond. The current gating condition can be read with getGateCond.

(These facilities are described in the Reference Guide, Sec C.9.1)

## Example: Making gated clocks with mkGatedClock

```
Clock clk <- exposeCurrentClock;  
GatedClockIfc gc1 <- mkGatedClock(True, clocked_by clk);  
Clock clk1 = gc1.new_clk;  
GatedClockIfc gc2 <- mkGatedClock(True, clocked_by clk1);  
Clock clk2 = gc2.new_clk;  
  
rule gate_clocks;  
    Bool gateClk1 = ...; Bool gateClk2 = ...;  
    gc1.setGateCond(gateClk1);  
    gc2.setGateCond(gateClk2);  
endrule
```

- clk1 is a version of clk, gated by gateClk1.
- clk2 is a version of clk1, gated by gateClk2.
  - i.e. it is the current clock gated by (gateClk1 && gateClk2)
- clk, clk1 and clk2 are from the same family
- clk1 is an ancestor of clk2; clk and clk1 are ancestors of clk2

Clock “plumbing”:  
feeding clocks to modules in the module hierarchy, all the  
way down to primitive modules such as registers, FIFOs  
and memories

# Default clocks and default plumbing

- Every BSV module has an implicit default input clock (being implicit, it is not mentioned in the interface or in the module header)
- When a module m1 instantiates a module m2, by default the m2 instance inherits m1's default clock as its own default clock
  - This structure is followed recursively down the module hierarchy, all the way down to primitive modules (such as registers, FIFOs and memories)

This explains why, in a single-clock-domain BSV design, one never sees any mention of a clock at all—there is a single, default clock that covers everything in the design

# Explicit clocks and explicit plumbing

- Extra (non-default) clock inputs to a module can be declared as parameters of type “Clock”
- When such a module is instantiated, the extra clocks are passed as parameters just like any other parameter
- During module instantiation, the default clock for the new instance can be specified explicitly using the optional “clocked\_by” attribute (otherwise the instance’s default clock is the same as the parent’s default clock)
- Inside a module instance, its default clock can be named using the primitive module “exposeCurrentClock”

```
module mkMod1#(Clock c1b, Clock c1c, ...) (Ifc1);
  ...
  Clock c1a <- exposeCurrentClock;
  Ifc2 m2 <- mkMod2 (c1a, True, clocked_by c1b);
  ...
endmodule

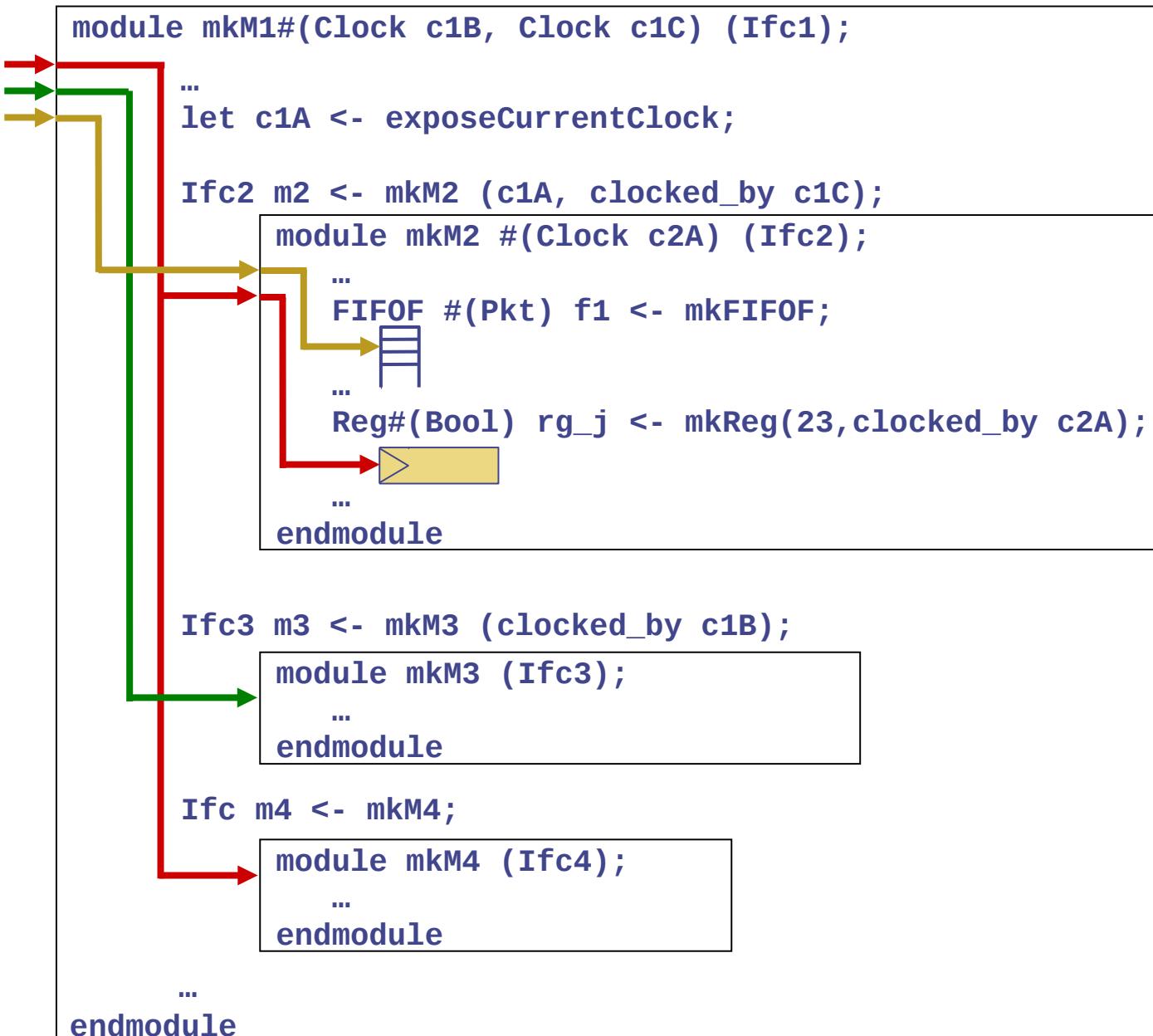
module mkMod2 #(Clock c2c, Bool b) (Ifc2);
  ...
endmodule
```

mkMod1 has 3 input clocks: default, c1b, c1c. c1a is a local name for the default clock.

The m2 instance of mkMod2 uses c1b as its default clock, and also has access to c1a as an explicit parameter.

Inside m2, c1a is known as c2c. The boolean b is just an ordinary (non-clock) parameter.

# Example: explicit clocks and explicit plumbing



(instance of) mkM1:  
→ default clock, c1A  
→ c1B  
→ c1C

Instance m2:  
→ default clock  
→ c2A

f1 clocked by default clock  
rg\_j clocked by c2A

Instance m3:  
→ default clock

Everything inside m3 will have the same clock

Instance m4:  
→ default clock

Everything inside m4 will have the same clock

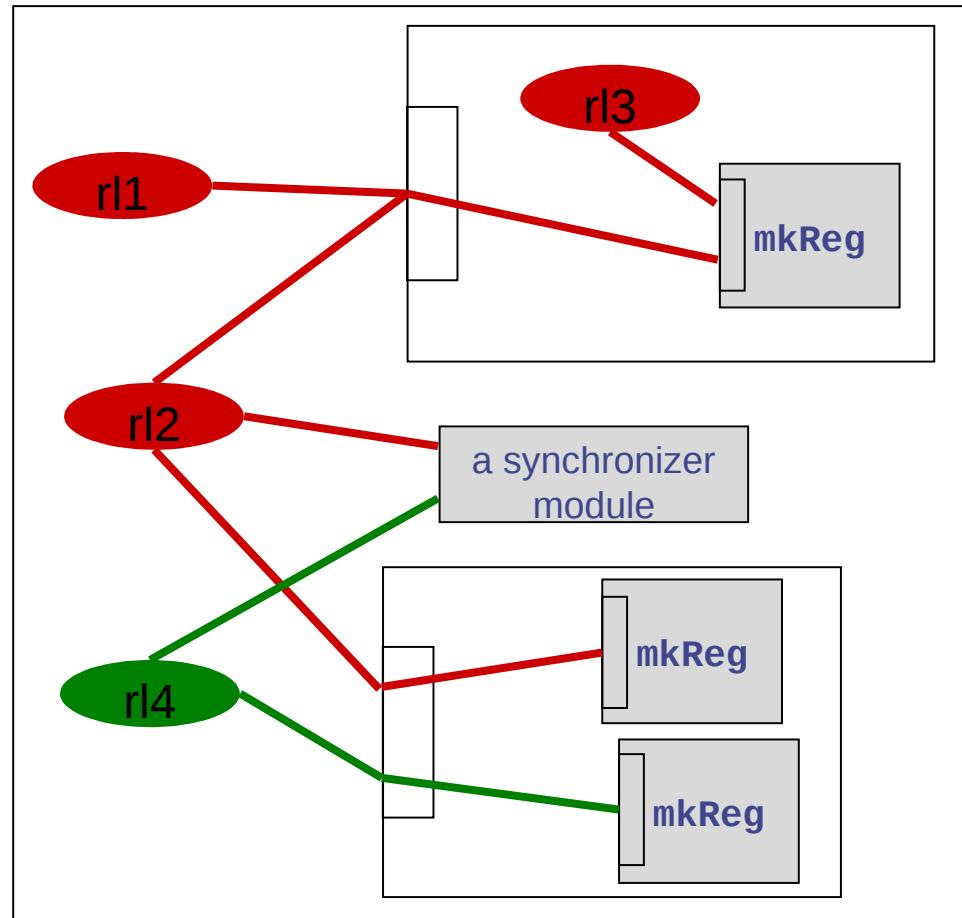
## Clock domains in BSV

# Clock domains in BSV

- A rule, and every method it invokes, directly or indirectly, must be on the same clock.
- All the rules that are on the same clock constitute a clock domain.
- Clock gating does not play any role in defining clock domains

The diagram illustrates two clock domains, colored red and green, respectively. Note:

- rl1 and rl2 must be on the same clock since they call a common method
- A module can contain rules/methods in different clock domains, i.e., clock domains may not follow the module hierarchy
- Clock domains are completely disjoint, and cover the entire design



The only place where different clock domains “touch” is at a synchronizer module. These are primitive modules with methods on different clocks (many examples to be discussed shortly).

# The clockOf() function

- From the previous slide, it should be clear that every combinational circuit (every data expression in BSV) must be in some clock domain, since every data expression is part of some rule or method
- The ‘clockOf()’ function may be applied to any BSV expression and returns a value of type Clock—the clock for that expression
  - If the expression is a constant (does not involve any method calls), the result is the special value noClock

Example:

```
Reg #(UInt #(17)) x <- mkReg (0, clocked_by c);
let y = x + 2;
Clock c1 = clockOf (x + 3);
Clock c2 = clockOf (y - 10);
```

c1 and c2 are the same clock

# Clocks and rule/method scheduling

- Each clock domain is scheduled independently of all other clock domains
- This is because the methods of primitive synchronizer modules that are on different clocks are always defined to be “conflict-free”. Since this is the only place where two clock domains “meet”, this implies that one clock domain never imposes any scheduling constraint on another.
- Concretely, this means that the rules in a clock domain have their own linear schedule, which has no relationship to the schedule of rules in another clock domain.

# Gated Clocks and rule/method readiness

- The clock domain of a rule or a method may contain zero or more gated clocks (different methods they invoke may have different gatings).
- These gating conditions are AND'ed together and contribute to rule and method conditions (CAN\_FIRE for a rule, READY for a method). This ensures that we do not accidentally try to invoke a method whose clock gate is currently OFF.
- Value methods are not disabled by clock gating conditions
  - They remain ready if they were ready when the clock was gated off
- Example:

```
FIFO #(Int#(3)) myFifo <- mkFIFO (clocked_by clk1);
```

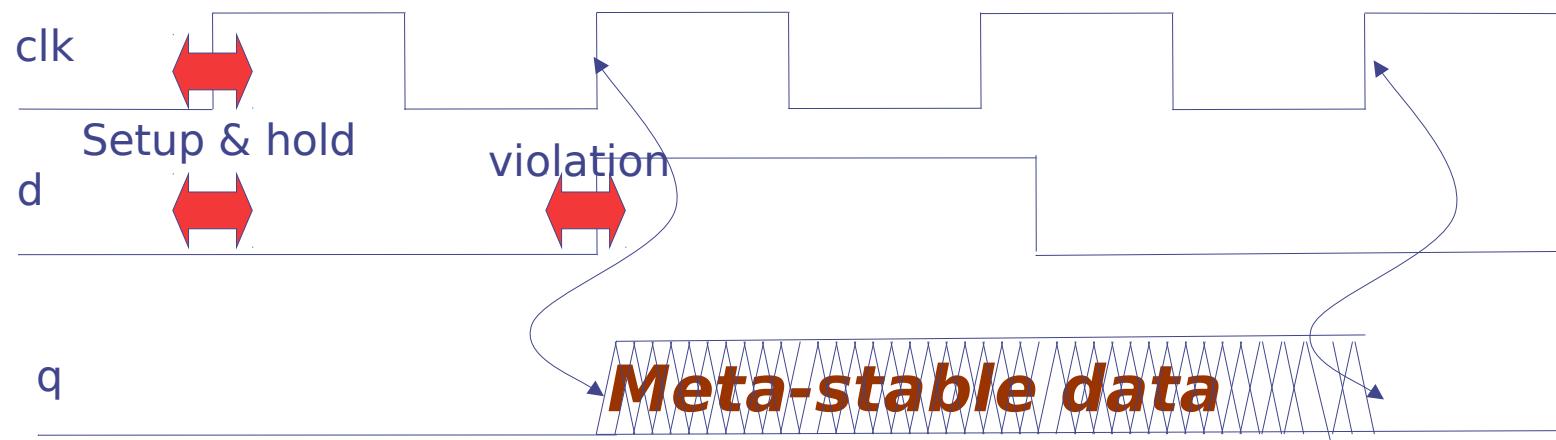
If clk1 is gated off:

- myFifo.enq, myFifo.deq and myFifo.clear are not READY
  - (and any rule that invokes these methods cannot fire)
- myFifo.first remains READY if the FIFO was non-empty when the clock was gated off

## Communicating between clock domains using synchronizers

# Moving Data Across Clock Domains

- Data moved across clock domains appears asynchronous to the receiving (destination) domain, because clock edges in the source and destination domains have no specific timing relationship
- Asynchronous data can cause meta-stability, i.e., circuits can violate digital discipline by sitting for an arbitrarily long time at an analog voltage that cannot be detected unambiguously either as a '0' or a '1'



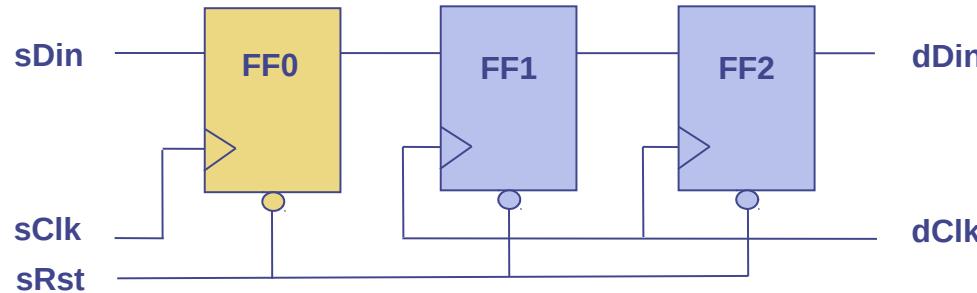
- The only safe way to move data between clock domains is through certain carefully engineered primitives called *synchronizers*

# Synchronizers

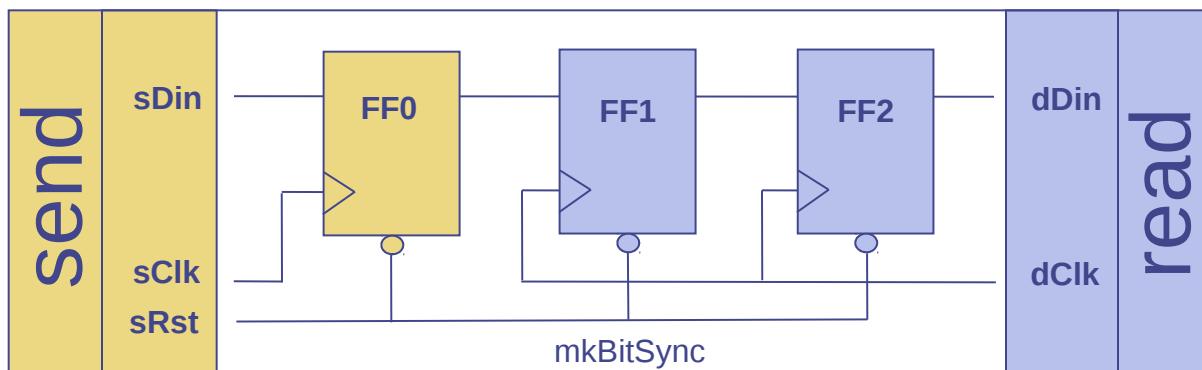
- Good synchronizer design and usage reduces the probability of observing meta-stable data
- Good design discipline insists on synchronizers for all domain crossings
  - BSV enforces this through static (compile-time) checking
  - (With RTL, clock domain discipline is usually checked with post-facto “lint”-like tools which can be inaccurate, quite fragile, and susceptible to minor variations in design style)
- The BSV library provides a wide range of conservative (speed independent) synchronizers (examples on the following pages)
- However, many organizations have their own libraries of internally-developed synchronizers. These can be used instead of, or in addition to, BSV’s libraries, by utilizing BSV’s mechanism to “import” RTL (see Reference Guide Sec.15)

# 1-bit synchronizer

- This is the most common type of synchronizer, for carrying a 1-bit signal across a clock domain boundary
- The picture below shows a typical implementation. Even though FF1 can go meta-stable, FF2 does not look at the data until a clock period later, giving FF1 time to stabilize.
- Limitations:
  - When moving from fast to slow clocks data may be overrun
  - Cannot use  $n$  of these in parallel to synchronize an  $n$ -bit word, since the bits may not arrive at the destination on the same clock edge



# 1-bit synchronizer: BSV encapsulation



The classical 1-bit synchronizer circuit is encapsulated into BSV as a module that is clocked by the source and destination clocks, and has “send” and “read” methods on those clocks, respectively:

```
interface SyncBitIfc ;
    method Action send (Bit#(1) bitData);
    method Bit#(1) read;
endinterface
```

```
module mkBitSync #(Clock sClk, Reset sRst, Clock dClk)
    (SyncBitIfc);
```

# Example: 1-bit synchronizer

- A counter that is enabled by a signal from another clock domain
- Registers:

```
Reg#(Bit#(32)) cntr <- mkReg(0);           // Default clock  
Reg#(Bit#(1)) enable_count <- mkReg(0, clocked_by clk2);
```

- The Rule (attempt 1):

```
rule countup (enable_count == 1);  
    cntr <= cntr + 1;  
endrule
```

Illegal Clock  
Domain Crossing

- It's illegal because the method `enable_count._read()` is on a different clock from the methods `cntr._read()` and `cntr._write()`
- This error will be detected and reported by the *bsc* tool

## Example: 1-bit synchronizer (contd.)



```
module mkTopLevel #(Clock clk2, Reset rst2) (Empty);
    Reg#(Bit#(32)) cntr <- mkReg(0); // Default clock
    Reg#(Bit#(1)) enable_count <- mkReg(0, clocked_by clk2);

    Clock currentClk <- exposeCurrentClock; // Default clock
    SyncBitIfc#(Bit#(1)) sync <- mkSyncBit(clk2, rst2, currentClk);

    rule cross;
        sync.send(enable_count);
    endrule

    rule countup ( sync.read == 1 );
        cntr <= cntr + 1;
    endrule
endmodule
```

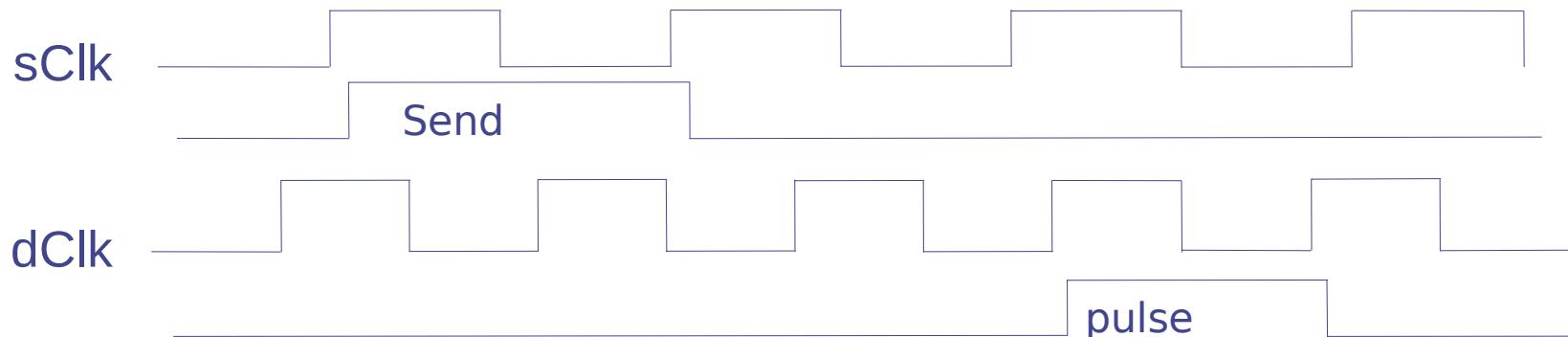
All methods in rule countup are now in the same clock domain

# Pulse Synchronizer

- This is also 1-bit wide, but instead of delivering a level (0/1) across clock domains, it sends single clock width pulses
- Caveat: two “send”s may not be seen if they occur too close together (faster than  $2 \cdot \text{dstClk}$ ), i.e., pulses can be dropped

```
interface SyncPulseIfc ;
  method Action send;
  method Bool  pulse;
endinterface
```

```
module mkSyncPulse #(Clock sClkIn, Reset sRstIn, Clock dClkIn)
  (SyncPulseIfc);
```



# Handshake Pulse Synchronizer

- A Pulse Synchronizer with a handshake protocol, so that pulses are not dropped
- After the send method is invoked, its method condition goes False (it is disabled) until the pulse has reached the other domain
- Latency:
  - send to read is ( $2 * \text{dstClk}$ )
  - send to next send ( $2 * \text{dstClks} + 2 * \text{srcClk}$ )

Same interface, and similar module header, as previous example:

```
interface SyncPulseIfc ;  
    method Action send;  
    method Bool    pulse;  
endinterface
```

```
module mkSyncHandshake #(Clock sClkIn, Reset sRstIn, Clock dClkIn)  
    (SyncPulseIfc);
```

# Register Synchronizer

- Uses same Reg#(a) interface as ordinary registers, but `_read` and `_write` are in different clock domains
- The `_write` method has an (implicit) ready condition, to allow time for data to be available for `_read`.
- No guarantee that destination reads the data, only that it arrives

```
interface Reg#(type t);
    method Action  _write(t a);
    method t       _read;
endinterface
```

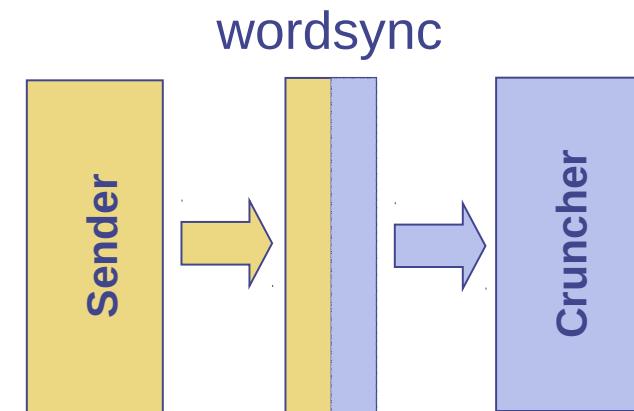
Standard Reg interface

```
module mkSyncReg #(t initialValue, Clock sClkIn,
                    Reset sRstIn,
                    Clock dClkIn)
    (Reg#(t));
```

# Example: Register Synchronizer

```
interface Sender;
  method Bit#(32) wordOut;
  method Bool      valid;
endinterface
```

```
interface Cruncher;
  method Action crunch(Bit#(32) dataRead);
endinterface
```



```
module top#(Clock clk2, Reset rst2)(Top);
  Sender   sender   <- mkSender(clocked_by clk2, reset_by rst2);
  Cruncher cruncher <- mkCruncher;
  ...
  Clock     clk          <- exposeCurrentClock;
  Reg#(Bit#(32)) wordSync <- mkSyncReg(0, clk2, rst2, clk);
  ...
  rule r1(sender.valid); // in domain: clk2, rst2
    wordSync <= sender.wordOut;
  endrule
  ...
  rule r2;
    cruncher.crunch( wordsync ); // in domain: clk
  endrule
endmodule
```

# FIFO Synchronizer

- Good for buffered, flow-controlled transmission across clock domains
- Like a standard FIFO, except that items are enqueued in one clock domain, and read/dequeued in another clock domain

```
interface SyncFIFOIfc#(type t);
    method Bool notFull;
    method Action enq (t a);

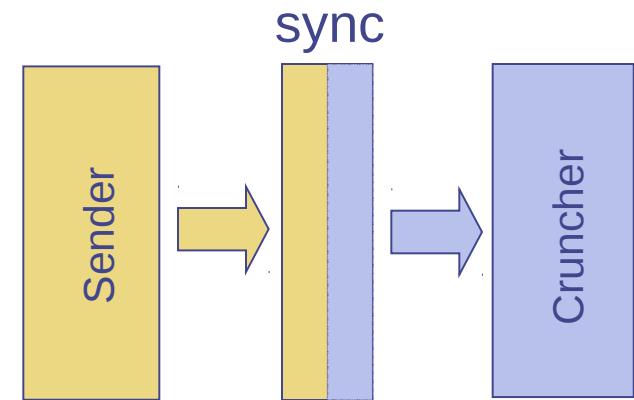
    method Bool notEmpty;
    method Action deq;
    method t first;
endinterface
```

```
module mkSyncFIFO #(Integer depth, Clock sClkIn,
                    Reset sRstIn,
                    Clock dClkIn)
    (SyncFIFOIfc #(t));
```

# Example: FIFO Synchronizer

```
interface Sender;
    method Bit#(32) wordOut;
    method Bool      valid;
endinterface

interface Cruncher;
    method Action crunch(Bit#(32) dataRead);
endinterface
```



```
module mkTop#(Clock clk2, Reset rst2)(Top);
    Sender    sender    <- mkSender(clocked_by clk2, reset_by rst2);
    Cruncher  cruncher  <- mkCruncher;
    ...
    Clock      clk       <- exposeCurrentClock;
    SyncFIFOIfc#(Bit#(32)) fifoSync <- mkSyncFIFO(4,clk2,rst2,clk);
    ...
    rule r1 ( sender.valid );           // in domain: clk2, rst2
        fifoSync.enq( sender.wordOut );
    endrule
    ...
    rule r2;
        cruncher.crunch(fifoSync.first); // in domain: clk
    endrule
endmodule
```

# Null Synchronizer

- There are some situations where the user may have external knowledge that synchronization logic is not necessary (no danger of meta-stability). For example, suppose a design specifies two input clocks, but is used in some contexts where both clocks are fed by the same external clock.
- In such situations, you can use a Null synchronizer, which appeases *bsc*'s strict clock discipline checker, but which contains no synchronization logic

```
interface ReadOnly#(type t);
    method t _read;
endinterface
```

This is a standard BSV interface (not specifically for synchronizers). It is the `_read` half of a `Reg` interface.

This module contains nothing but a wire

```
module mkNullCrossingWire #(Clock dclkIn, t dataIn)
    (ReadOnly #(t));
```

(for  $n$ -bit null crossings, you can use `mkNullCrossingReg`)

# Example: Null Synchronizer

```
module mkTopLevel #(Clock clk2, Reset rst2 ) (Empty);
  Reg#(Bit#(32)) cntr <- mkReg (0) ;           // Default clock
  Reg#(Bit#(1))  enable_count <- mkReg(0, clocked_by clk2);

  ReadOnly#(Bit#(1)) nullSync <- mkNullCrossingWire(clk2, enable_count);

  rule countup ( nullSync == 1 );
    cntr <= cntr + 1;
  endrule
endmodule
```

Null synchronizers are often used conditionally, i.e., depending on whether the context is safe or not for use of a null synchronizer; see example on next page.

## Example: Null Synchronizer (contd.)

```
module mkTopLevel #(Bool safe_context, Clock clk2, Reset rst2 ) (Empty);
  Reg#(Bit#(32)) cntr <- mkReg (0) ;           // Default Clock
  Reg#(Bit#(1)) enable_count <- mkReg(0, clocked_by clk2);

  Bit#(1) b;
  if (safe_context) begin
    ReadOnly#(Bit#(1)) nullSync <- mkNullCrossingWire(clk2, enable_count);
    b = nullSync;
  end
  else begin
    Clock currentClk <- exposeCurrentClock ;      // Default clock
    SyncBitIfc#(Bit#(1)) sync <- mkSyncBit(clk2, rst2, currentClk);
    rule cross;
      sync.send( enable_count );
    endrule
    b = sync.read;
  end

  rule countup ( b == 1 );
    cntr <= cntr + 1;
  endrule
endmodule
```

- Here, `safe_context` is an external boolean indicating synchronization is unnecessary
  - It must be static, i.e., resolved at compile time by *bsc*
- If True, `enable_count` is taken through a null synchronizer before use in rule `countup`
- If False, `enable_count` is taken through `mkSyncBit`

# Resets

# Resets

Much of our description of Clocks can be repeated for Resets

- There is a special type called “Reset”, and strong type-checking ensures that it can never be confused with ordinary signals or clocks
- Reset are “plumbed” through the module hierarchy just like clocks
  - Every module instance has a default Reset, and can take additional resets as parameters
  - By default, a module instance inherits its parents’ default Reset, but this can be overridden using a “reset\_by” clause in the instantiation
  - When the whole design uses a single default reset throughout, the source code never mentions resets
- “Reset discipline” is not as strict as clock domain discipline:
  - Every method is associated with a reset (just like it is with a clock)
  - However, a rule or method can invoke methods with different resets. The *bsc* tool will warn about this (it is not an error). A rule that executes while some of its methods are still in reset will have unpredictable behavior (and so it is not good practice to have mixed resets in a rule or method).

## Creating clocks and resets for simulation

# Creating clocks and reset for simulation

In actual hardware, clocks and resets are usually “external” inputs to a design, and are generated, shaped and conditioned by specially engineered circuits before being fed to the design. As such, one does not use BSV to design clock- and reset-generation circuits.

However, in simulation, for testing, debugging and analysis, one may want to “generate” clocks and resets in the testbench, to be fed to the DUT (Design Under Test). For this purpose, BSV provides a number of modules for creation and derivation of clocks and resets.

Please see the Reference Guide Secs. C.9.1 and C.9.10 for details about these facilities

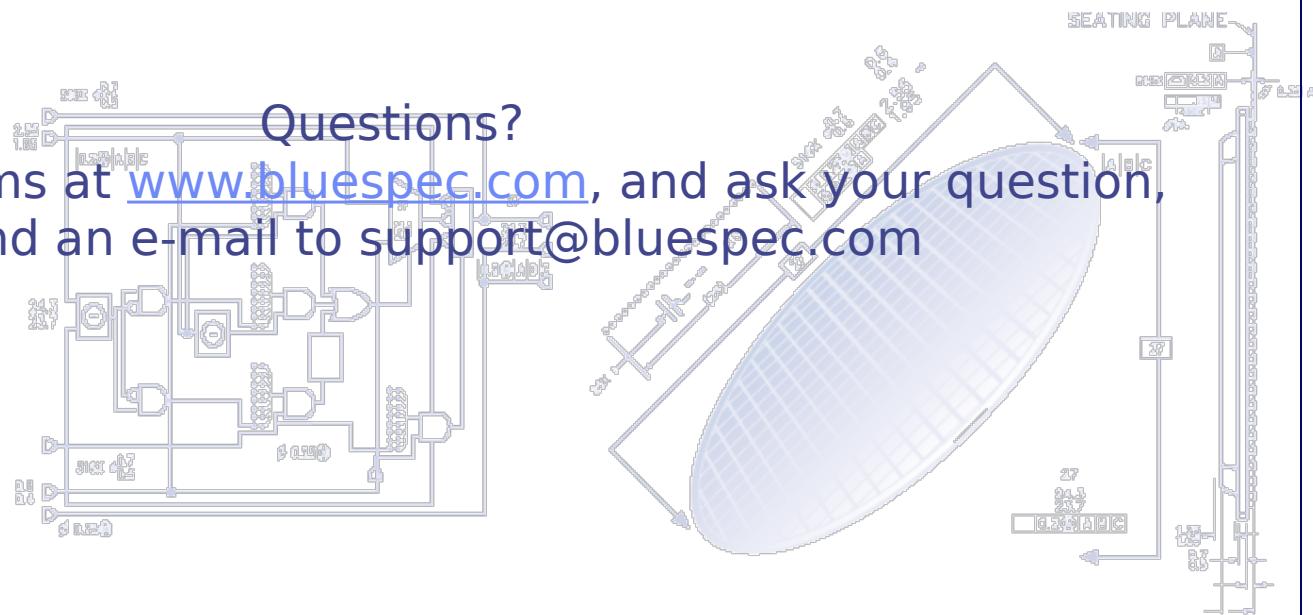
# Summary

- The Clock type, and strong type checking ensures that all circuits are clocked by actual clocks
- BSV provides ways to create, derive and manipulate clocks, safely
- BSV clocks are gated, and gating fits into Rule-enabling semantics
  
- BSV provides a full set of speed-independent data synchronizers, already tested and verified
- The user can define new synchronizers
- BSV precludes unsynchronized domain crossings



# End

```
import PFR#(type#(Bit#(8)) default);
module end;
  method Action check();
    Integer file;
    function Bit#(8) determine_gpa(Bit#(8) val);
      return val;
    endfunction
    function Bit#(8) calculate();
      return determine_gpa(val);
    endfunction
    action
      file = $fopen("gpa.txt");
      $fputs(file, calculate());
      $fclose(file);
    endaction
  endaction
endmodule
```



Questions?

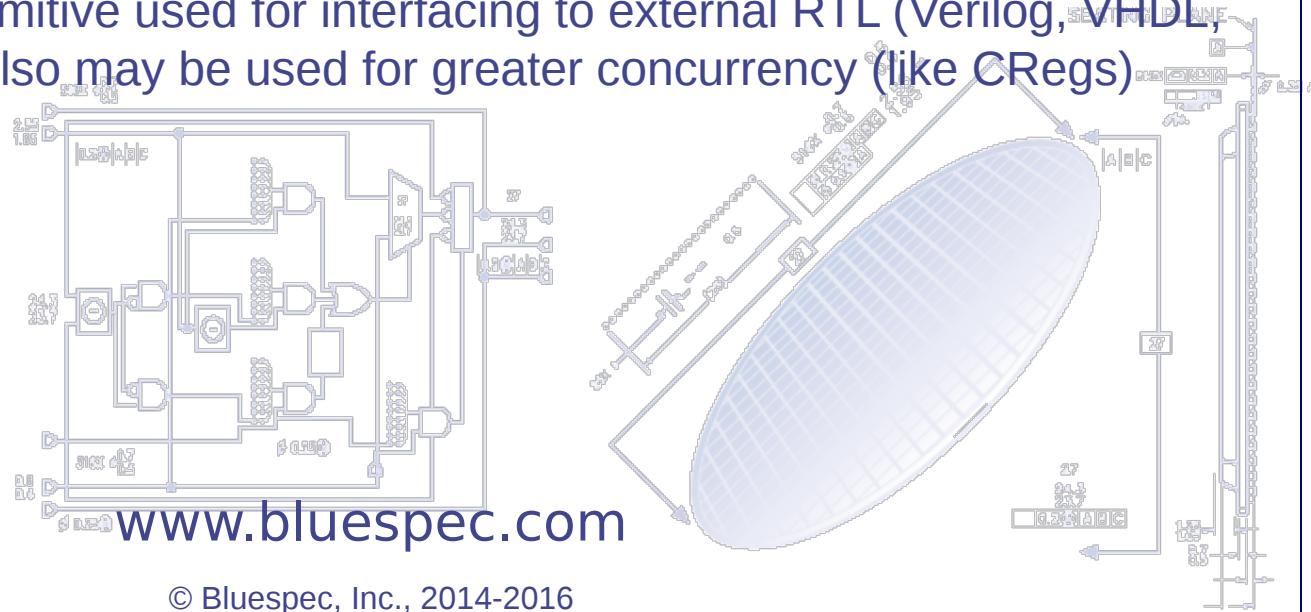
Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com

# BSV Training

## Lec\_RWires

RWires: a low-level primitive used for interfacing to external RTL (Verilog, VHDL, SystemVerilog). Also may be used for greater concurrency (like CRegs)

```
import PFR#(T);
typedef Bit#(2) Count;
module Action RWire#(T)(PFR#(T));
    integer fifo_depth = 15;
    function Bit#(1) determine_pwr(Bit#(1) val);
        return (val == 1);
    endfunction
    action
        FIFO#(Bit#(1)) fifo;
        if(!fifo.first)
            fifo.enq({1'b0, 1'b1});
        else
            if(determine_pwr(fifo.first))
                fifo.enq({1'b0, 1'b1});
            else
                fifo.enq({1'b1, 1'b0});
        end
        if(fifo.length > fifo_depth)
            fifo.pop();
    endaction
    endaction
endinterface
```



# Introduction to RWires

RWires are low-level primitives in the BSV library

- (they can in fact be used to implement CRegs)

An RWire is an abstraction of ordinary wires in terms of BSV concepts of modules, methods, and method orderings.

RWires are frequently used at the boundary of a BSV design to interface with existing Verilog or VHDL for which the exact signal and protocol specifications are already given.

RWires must be used with great care:

- With all other primitives (including CRegs), functional correctness is typically preserved across arbitrary schedules (even one rule per clock). Improved scheduling is primarily concerned with tuning performance without affecting functional correctness.
- This is typically not true with RWires, which are only meaningful for intra-clock communication (and therefore assume certain minimum concurrency in schedules).

The basic primitive is called the “RWire”.

Special cases include PulseWires, Wires, DWires and BypassWires.

# RWires

The most general form of “wire” family is the RWire interface and mkRWire primitive module:

```
interface RWire #(type t);
    method Action wset (t datain);
    method Maybe#(t) wget;
endinterface

module mkRWire (RWire#(t)); // primitive
```

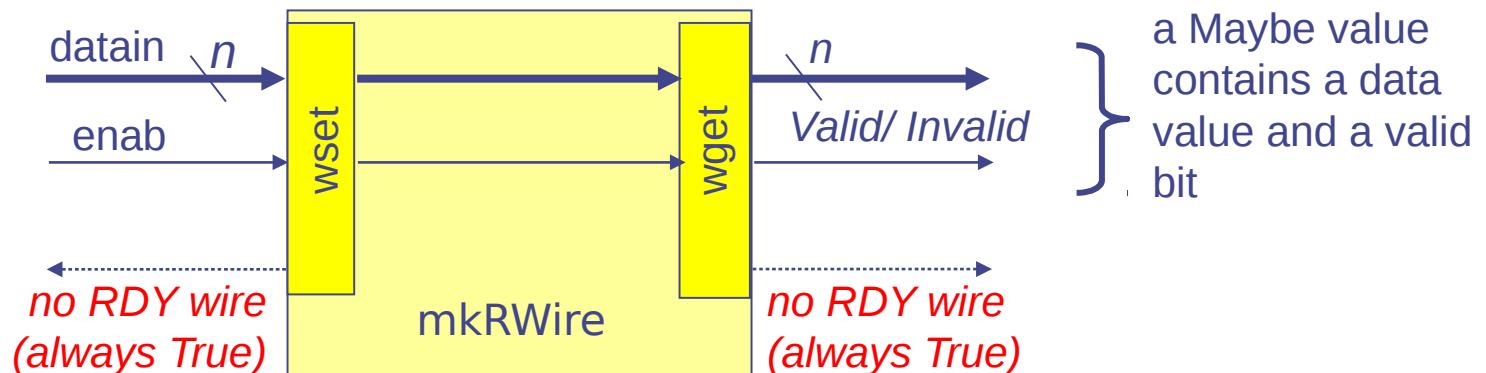
Ordering constraint:  
wset < wget

Suppose rule rA invokes rw.wset (x)

Then, in rule rB (logically later in the schedule):

- if (rw.wget matches tagged Valid .x) then rB knows that rA is firing in this clock and communicating the value x
- if (rw.wget matches tagged Invalid) then rB knows that rA is not firing in this clock

Implementation:



# PulseWires

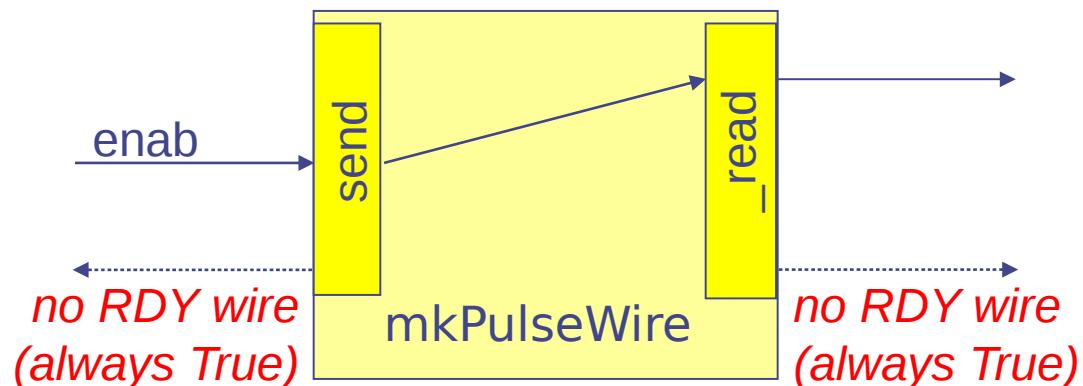
The PulseWire interface and mkPulseWire module is a special case of RWires where there is no value to be communicated

```
interface PulseWire;  
    method Action send;  
    method Bool _read;  
endinterface  
  
module mkPulseWire (PulseWire); // primitive
```

Ordering constraint:  
send < \_read

The \_read method returns True if the send method is being invoked, else returns False

Implementation:



# The Wire interface and mkDWire module

The Wire interface is just a synonym for the Reg interface.

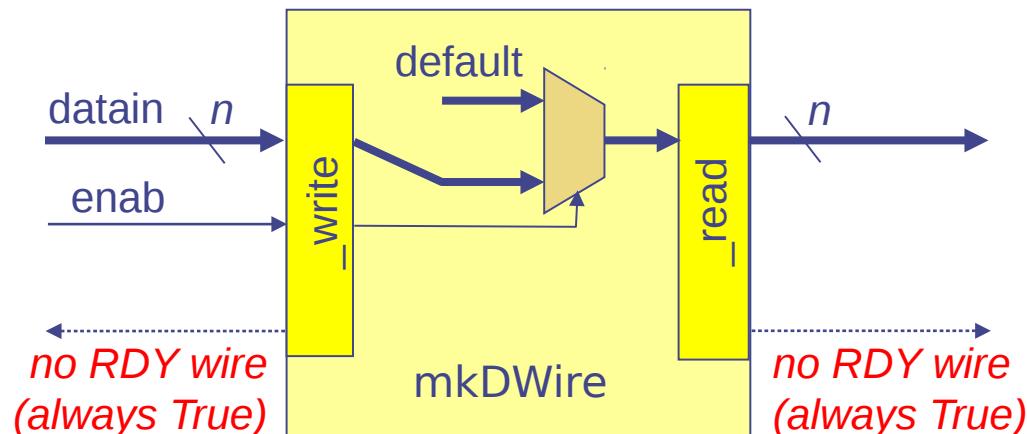
The mkDWire module is a “primitive” (D for default)

```
typedef Reg#(t) Wire#(type t);  
  
module mkDWire #(t default) (Wire#(t));  
  ... primitive imported Verilog ...  
endmodule
```

Ordering constraint:  
`_write < _read`

Note! this is the opposite schedule of mkReg!

Implementation:



# Hands-on

- BSV-by-Example book: Examples in Chapter 8



End

# Questions?

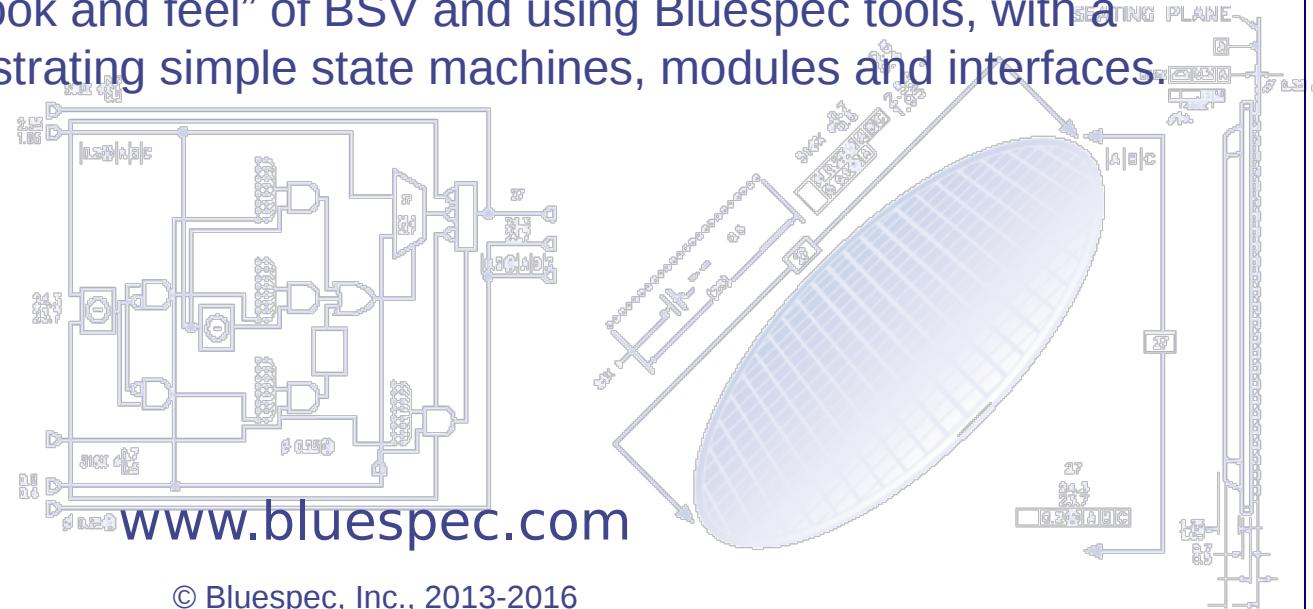
Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to support@bluespec.com



# BSV Training

## Eg02: Warmup exercise; simple state machines

## Introduction to “look and feel” of BSV and using Bluespec tools, with a



# Eg02a: A “Hello World” example

Ever since the classic book “The C Programming Language” by Kernighan and Ritchie in 1978, it has become customary to start with a very simple example to introduce the student to the basic “look and feel” of a language, and to get familiar with basic logistics: how programs texts are organized into files, how to compile them, execute them, and observe results. We shall do the same with BSV.

Our first BSV program just prints “Hello World!” (and a little more!) and halts.

The source code is in Example\_Programs/Eg02a\_HelloWorld/src\_BSV/Testbench.bsv

BSV programs are organized into *modules*.

This program has one module only (“mkTestbench”).

All BSV modules have *interfaces*.

(This interface is the “Empty” *interface type*, which has no “methods” to interact with the environment.)

```
module mkTestbench (Empty);  
  
    rule rl_print_answer;  
        $display ("Deep Thought says: Hello, World! The answer is 42.");  
        $finish;  
    endrule  
endmodule
```

All behavior in BSV is expressed using rules. This program has one rule (“rl\_print\_answer”). A rule is a potentially infinite process, i.e., it may “fire” (execute) repeatedly, forever.

\$display is like “printf” in C/C++ (but it also always prints a final newline after the given output).

\$finish is like “exit()” in C/C++—it causes the whole program to halt immediately. Thus, in this example, the rule only fires once.

# Compiling and running: Bluesim

Compiling, linking, and running is similar to compiling, linking and running a C/C++ program. Here, we show this using the “Bluesim” simulator.

The code can be found in: Example\_Programs/Eg02a\_HelloWorld/src\_BSV/Testbench.bsv

You can type the commands as shown below. Or, for your convenience, there is also a Makefile in the Build/ directory, and you can invoke the commands by typing ‘make compile’, ‘make link’ and ‘make simulate’, respectively.

```
$ bsc -sim -g mkTestbench Testbench.bsv
checking package dependencies
compiling Testbench.bsv
code generation for mkTestbench starts
Elaborated module file created: mkTestbench.ba
All packages are up to date.
```

Or: \$ make compile

```
$ bsc -sim -e mkTestbench -o ./mkTestbench_bsim
Bluesim object reused: mkTestbench.{h,o}
Bluesim object created: model_mkTestbench.{h,o}
Simulation shared library created:
mkTestbench_bsim.so
Simulation executable created: ./mkTestbench_bsim
```

Or: \$ make link

```
$ ./mkTestbench_bsim
Deep Thought says: Hello, World! The answer is 42.
```

Or: \$ make simulate

“bsc” is the Bluespec BSV compiler.  
-sim: compile for Bluesim simulator  
-g mkTestBench: top-level module  
Testbench.bsv: source file

“bsc” is also the Bluespec BSV linker  
-sim: link for Bluesim simulator  
-e mkTestBench: top-level module  
-o ./mkTestbench\_bsim: name of output executable file

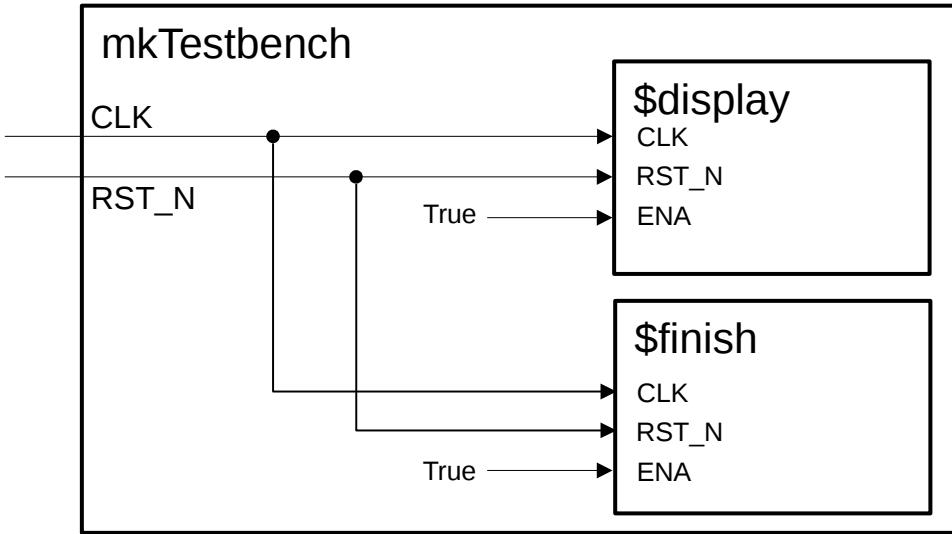
./mkTestbench\_bsim: run the Bluesim executable like any executable.

“Deep ...”: \$display output is displayed on your screen.



# A “hardware” view of our example

There is not much visible hardware in this example, since all the magic is in the \$display and \$finish primitives.



- The hardware generated by bsc is a *hierarchy of module instances* (module instances nested inside module instances).
- Every module has (at least) a CLK (clock) and RST\_N (reset) input, and may have other inputs.
  - The environment asserts low (0, False) on RST\_N for a short period after power is initially applied to the design
  - The environment oscillates CLK between low (0, False) and high (1, True) forever
- \$display and \$finish can be thought of as primitive modules that perform their action when the ENA signal is asserted. In this example, the ENA inputs are driven with the constant True.
  - Although \$display and \$finish are just used in simulation, one could actually build hardware modules that exhibit the same behavior

# Compiling and running: Verilog simulation

Here, we show this using a Verilog simulator.

```
$ bsc -verilog -g mkTestbench Testbench.bsv  
Verilog file created: mkTestbench.v
```

Or: \$ make verilog

-verilog: generate verilog file (“mkTestbench.v”)  
-g mkTestBench: top-level module  
Testbench.bsv: source file

```
$ bsc -verilog -e mkTestbench -o mkTestbench_vsim -vsim iverilog mkTestbench.v  
Verilog binary file created: mkTestbench_vsim
```

Or: \$ make v\_link

-verilog: link for Verilog simulator  
-e mkTestbench: top-level module  
-o mkTestbench\_vsim: name of output executable file  
-vsim iverilog: use “iVerilog” Verilog simulator.  
Alternatives: “modelsim” (Mentor), “ncverilog” (Cadence), “vcs” and “vcsi”  
(Synopsys), “cver” and “cvc” (Tachyon), “veriwell”, “isim” (Xilinx)

```
$ ./mkTestbench_vsim  
Deep Thought says: Hello, World! The answer is 42.
```

Or: \$ make v\_simulate

./mkTestbench\_vsim: run the Verilog simulation  
executable like any executable.

“Deep …”: \$display output is displayed on your screen.

# Synthesizing for FPGA or ASIC

(We won't actually do this, for this very simple example.)

The first step is the same as for Verilog simulation: generate Verilog files:

```
% bsc -verilog -g mkTestbench Testbench.bsv  
Verilog file created: mkTestbench.v
```

Or: \$ make verilog

These Verilog files are then synthesized just like any other Verilog files using the synthesis tool of the target technology's vendor:

- ASIC: Design Compiler (Synopsys) or other vendor's RTL synthesis tool
- FPGA: Xilinx Vivado, Altera Quartus, or other vendor's RTL synthesis tool

Please consult the vendor's tools and training for details on how to do this.

# Example variations

The supplied code includes three variations:

Eg02a_HelloWorld/	First version (previous slides)
Eg02b_HelloWorld/	Splits the first version into two separately compiled modules: "Testbench.bsv" and "DeepThought.bsv"
Eg02c_HelloWorld/	Adds some "state machine" functionality so that DeepThought "thinks for 7.5 million years" before yielding its answer <sup>1</sup> , while the testbench waits. This will give a first view of rule conditions and method conditions.

We will now go through Eg02b and Eg02c.

<sup>1</sup>You may have recognized that we are alluding to the book *The Hitchhiker's Guide to the Galaxy* by Douglas Adams (1979). In the book, a supercomputer named Deep Thought is asked to calculate the Answer to the Ultimate Question of Life, the Universe, and Everything. After 7.5 million years, it answers: "42".

# Eg02b: Splitting into separately compiled modules

The code can be found in: Example\_Programs/Eg02b\_HelloWorld/

Eg02a_HelloWorld/	First version (previous slides)
Eg02b_HelloWorld/	Splits the first version into two separately compiled modules: “Testbench.bsv” and “DeepThought.bsv”
Eg02c_HelloWorld/	Adds some “state machine” functionality so that DeepThought “thinks for 7.5 million years” before yielding its answer <sup>1</sup> , while the testbench waits. This will give a first view of rule conditions and method conditions.

# Eg02b: Splitting into separately compiled modules

We split our previous program into two modules, a top-level ``testbench'' module that instantiates a ``design'' module. We define an interface for the design module, containing one ``method''. The testbench module invokes this method in the interface to interact with the design.

Each file is a separate package.

The filename must be ``packagename.bsv''

Here, package Testbench imports everything defined in package DeepThought.

```
package Testbench;
import DeepThought :: *;
(* synthesize *)
module mkTestbench (Empty);
    DeepThought_IFC deepThought <- mkDeepThought;
    rule rl_print_answer;
        let x <- deepThought.getAnswer;
        $display ("Deep Thought says: Hello, World! The answer is %0d.", x);
        $finish;
    endrule
endmodule
```

```
package DeepThought;
// Interface declaration

interface DeepThought_IFC;
    method ActionValue #(int) getAnswer;
endinterface

// Module definition

(* synthesize *)
module mkDeepThought (DeepThought_IFC);
    method ActionValue #(int) getAnswer;
        return 42;
    endmethod
endmodule

endpackage
```

Top-level module creates an instance of subordinate module

Invoking an ActionValue method

``synthesize'' tells bsc to preserve this module boundary when generating Verilog (else it would in-line it).

# Compiling and running Eg02b: Bluesim

The code can be found in: Example\_Programs/Eg02b\_HelloWorld/

The source code is in src\_BSV/Testbench.bsv and src\_BSV/DeepThought.bsv

Build it just like you did Eg02a.

```
$ bsc -u -sim -simdir build_bsim -bdir build_bsim -info-dir build_bsim  
-keep-fires -aggressive-conditions -p ../../src_BSV:/%Prelude:/%Libraries  
-g mkTestbench src_BSV/Testbench.bsv  
checking package dependencies  
compiling ./src_BSV/DeepThought.bsv  
code generation for mkDeepThought starts  
Elaborated module file created: build_bsim/mkDeepThought.ba }  
compiling src_BSV/Testbench.bsv  
code generation for mkTestbench starts  
Elaborated module file created: build_bsim/mkTestbench.ba }  
All packages are up to date.
```

Only the top-level file and module need be mentioned; bsc will follow the ``import'' links and recompile whatever is needed

Or: \$ make compile

```
$ bsc -e mkTestbench -sim -o ./mkTestbench_bsim -simdir build_bsim -bdir  
build_bsim -info-dir build_bsim -p ../../src_BSV:/%Prelude:/%Libraries  
Bluesim object created: build_bsim/mkTestbench.{h,o}  
Bluesim object created: build_bsim/mkDeepThought.{h,o}  
Bluesim object created: build_bsim/model_mkTestbench.{h,o}  
Simulation shared library created: mkTestbench_bsim.so  
Simulation executable created: ./mkTestbench_bsim
```

Code generation and linking of all the modules for Bluesim.

Or: \$ make link

```
% ./mkTestbench_bsim  
Deep Thought says: Hello, World! The answer is 42.
```

Or: \$ make simulate



# Compiling Eg02b into Verilog

Here, we show this using a Verilog simulator.

```
bsc -u -verilog -vdir verilog -bdir build_v -info-dir build_v -elab  
-keep-fires -aggressive-conditions -no-warn-action-shadowing -p  
.:/src_BSV:%/Prelude:%/Libraries -g mkTestbench  
src_BSV/Testbench.bsv  
checking package dependencies  
compiling ./src_BSV/DeepThought.bsv  
code generation for mkDeepThought starts  
Verilog file created: verilog/mkDeepThought.v  
Elaborated module file created: build_v/mkDeepThought.ba  
compiling src_BSV/Testbench.bsv  
code generation for mkTestbench starts  
Verilog file created: verilog/mkTestbench.v  
Elaborated module file created: build_v/mkTestbench.ba  
All packages are up to date.  
Compiling for Verilog finished
```

Creates separate Verilog modules (each in its own ".v" file, for each BSV module that had the ``synthesize'' attribute.

Or: \$ make verilog

You can of course link and simulate this in a Verilog simulator, as shown earlier for Eg02a.

In practice we mostly use Bluesim simulation, because it is much faster (10x-50x) and it has exactly the same cycle behavior as the corresponding Verilog simulation.

We typically generate Verilog only when we are ready to take it through post-RTL synthesis for ASIC or FPGA.

## Eg02c: Adding some “state machine” functionality

The code can be found in: Example\_Programs/Eg02c\_HelloWorld/

Eg02a_HelloWorld/	First version (previous slides)
Eg02b_HelloWorld/	Splits the first version into two separately compiled modules: “Testbench.bsv” and “DeepThought.bsv”
Eg02c_HelloWorld/	Adds some “state machine” functionality so that DeepThought “thinks for 7.5 million years” before yielding its answer <sup>1</sup> , while the testbench waits. This will give a first view of rule conditions and method conditions.

# Eg02c: Adding some “state machine” functionality

Adds some “state machine” functionality so that DeepThought “thinks for 7.5 million years” before yielding its answer<sup>1</sup>, while the testbench waits. This will give a first view of rule conditions and method conditions.

```
module mkTestbench (Empty);

    DeepThought_IFC deepThought <- mkDeepThought;

    rule rl_ask;
        $display ("Asking the Ultimate Question of Life, The Universe and Everything");
        deepThought.whatIsTheAnswer;
    endrule

    rule rl_print_answer;
        let x <- deepThought.getAnswer;
        $display ("Deep Thought says: Hello, World! the answer is %0d.", x);
        $finish;
    endrule
endmodule
```

Rule “rl\_ask” invokes the method “whatIsTheAnswer” to start a computation in the mkDeepThought module instance.

Some time later, rule “rl\_print\_answer” is able to invoke the method “getAnswer” and print the result.

```
// Interface definition

interface DeepThought_IFC;
    method Action whatIsTheAnswer;
    method ActionValue #(int) getAnswer;
endinterface

// Module definition

(* synthesize *)
module mkDeepThought (DeepThought_IFC);

    ... to be shown on next slides ...

endmodule
```

# Eg02c: Adding some “state machine” functionality

```
typedef enum { IDLE, THINKING, ANSWER_READY } State_DT
deriving (Eq, Bits, FShow);

module mkDeepThought (DeepThought_IFC);

    Reg #(State_DT) rg_state_dt <- mkReg (IDLE);           Define a type State_DT. The module will start
                                                               in the IDLE state, move to THINKING, then to
                                                               ANSWER_READY, and finally back to IDLE.

    Reg #(Bit #(4)) rg_half_millenia <- mkReg (0);        Instantiate a register (variable) to hold the module state,
                                                               initialized to IDLE

    let millenia = rg_half_millenia [3:1];                  Instantiate register to count half-millenia
    let half_millenium = rg_half_millenia [0];               Define some useful values

    rule rl_think (rg_state_dt == THINKING);                 Rule can fire whenever in THINKING state
        $write ("      DeepThought: ... thinking ... (%0d", millenia);
        if (half_millenium == 1) $write (".5");
        $display (" million years");                          Print the passing of the millenia

        if (rg_half_millenia == 15)                         If seven and a half millenia, move to ANSWER_READY state
            rg_state_dt <= ANSWER_READY;
        else
            rg_half_millenia <= rg_half_millenia + 1;       else increment half millenia
    endrule

    method Action whatIsTheAnswer if (rg_state_dt == IDLE);
        rg_state_dt <= THINKING;
    endmethod

    method ActionValue#(int) getAnswer if (rg_state_dt == ANSWER_READY);
        rg_state_dt <= IDLE;
        rg_half_millenia <= 0;
        return 42;
    endmethod
endmodule
```

Define a type State\_DT. The module will start in the IDLE state, move to THINKING, then to ANSWER\_READY, and finally back to IDLE.

Instantiate a register (variable) to hold the module state, initialized to IDLE

Instantiate register to count half-millenia

Define some useful values

Rule can fire whenever in THINKING state

Print the passing of the millenia

If seven and a half millenia, move to ANSWER\_READY state

else increment half millenia

This method can be invoked when IDLE; then, move to THINKING state

This method can be invoked when ANSWER\_READY; then, return 42 and move to IDLE state

# Compiling and running Eg02c: Bluesim

The code can be found in: Example\_Programs/Eg02c\_HelloWorld/

```
$ make compile link
Compiling for Bluesim ...
bsc -u ...
    as before

Compiling for Bluesim finished
Linking for Bluesim ...
bsc -e ...
    as before

Linking for Bluesim finished
```

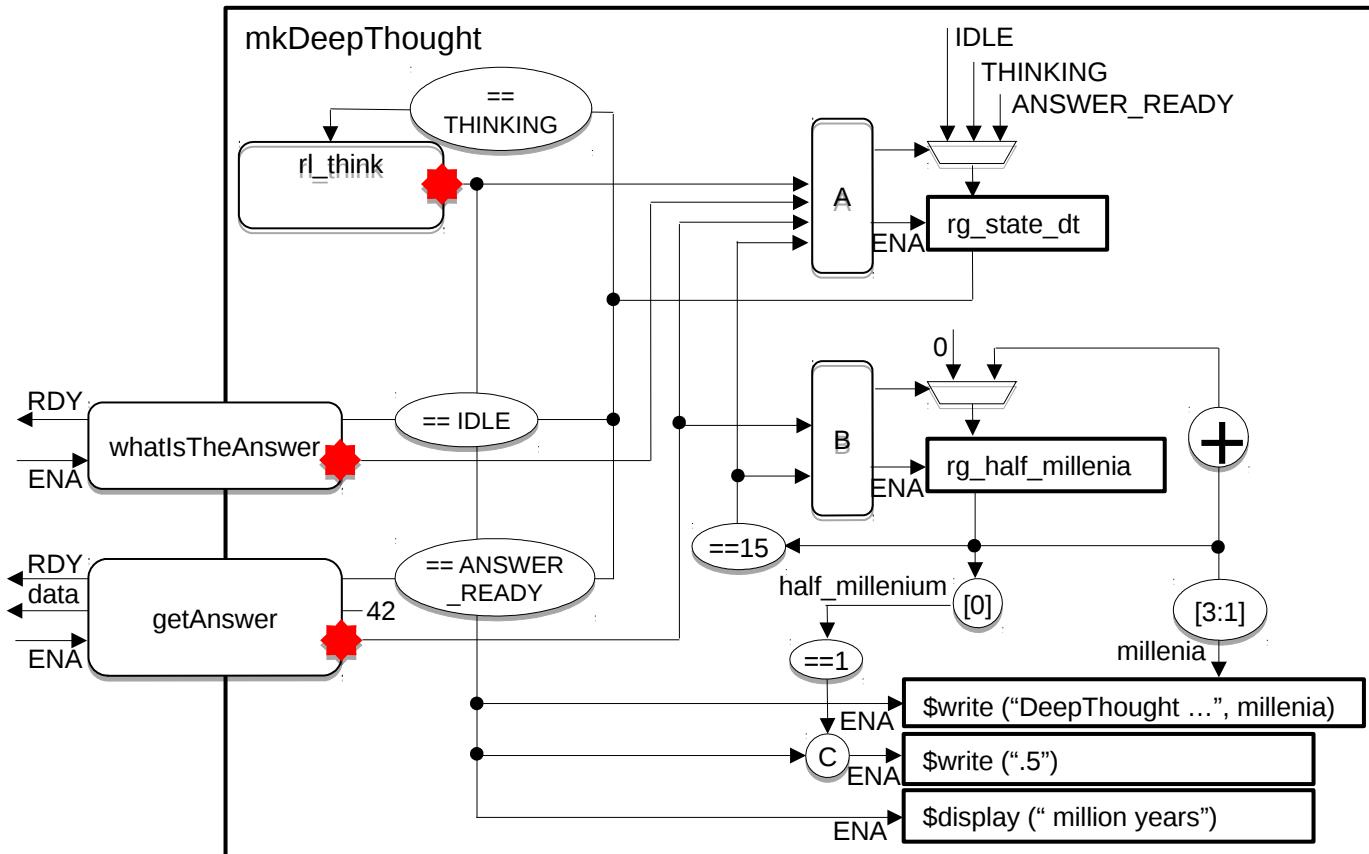
```
$ ./mkTestbench_bsim
Asking the Ultimate Question of Life, The Universe and Everything
  DeepThought: ... thinking ... (0 million years)
  DeepThought: ... thinking ... (0.5 million years)
  DeepThought: ... thinking ... (1 million years)
  DeepThought: ... thinking ... (1.5 million years)
  DeepThought: ... thinking ... (2 million years)
  DeepThought: ... thinking ... (2.5 million years)
  DeepThought: ... thinking ... (3 million years)
  DeepThought: ... thinking ... (3.5 million years)
  DeepThought: ... thinking ... (4 million years)
  DeepThought: ... thinking ... (4.5 million years)
  DeepThought: ... thinking ... (5 million years)
  DeepThought: ... thinking ... (5.5 million years)
  DeepThought: ... thinking ... (6 million years)
  DeepThought: ... thinking ... (6.5 million years)
  DeepThought: ... thinking ... (7 million years)
  DeepThought: ... thinking ... (7.5 million years)
Deep Thought says: Hello, World! The answer is 42.
```

From rule mkTestbench/rl\_ask

From repeated firings of rule  
mkDeepThought/rl\_think

From rule mkTestbench/rl\_print\_answer

# Hardware for Eg02c mkDeepThought



= "WILL\_FIRE" signal of a rule/method (for a method, same as ENA)

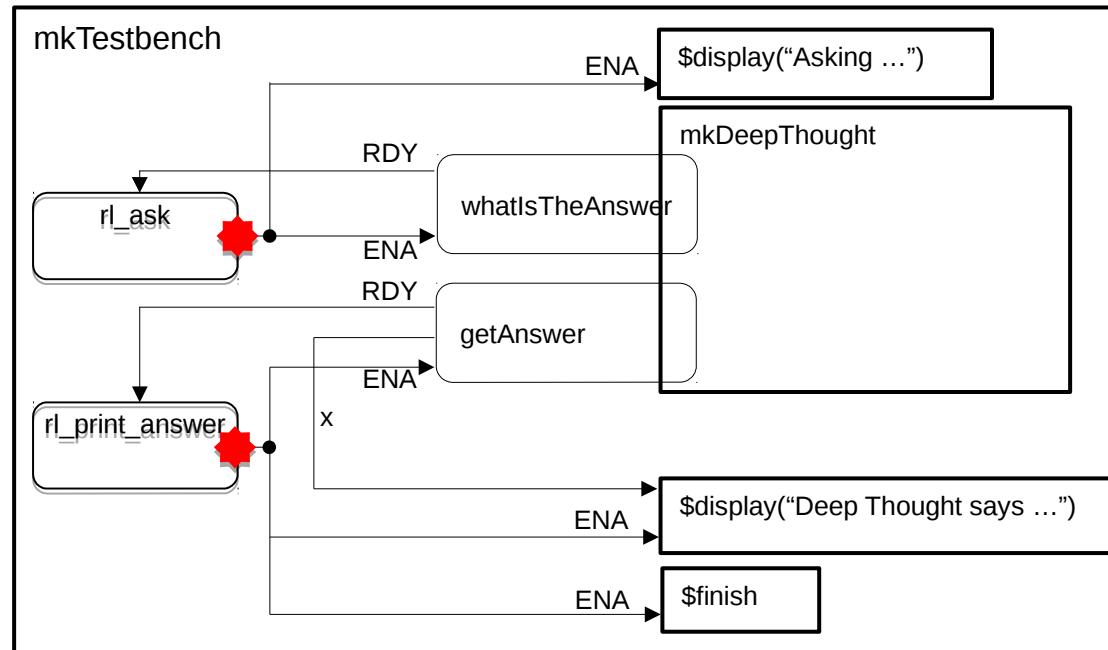
A: controls rg\_state\_dt: selects input data (mux) and whether it is updated (ENA)

B: controls rg\_half\_millenia: selects input data (mux) and whether it is updated (ENA)

C: controls \$write (ENA)

In each case its output is a simple boolean combination of its inputs

# Hardware for Eg02c mkTestbench



★ = "WILL\_FIRE" signal of a rule

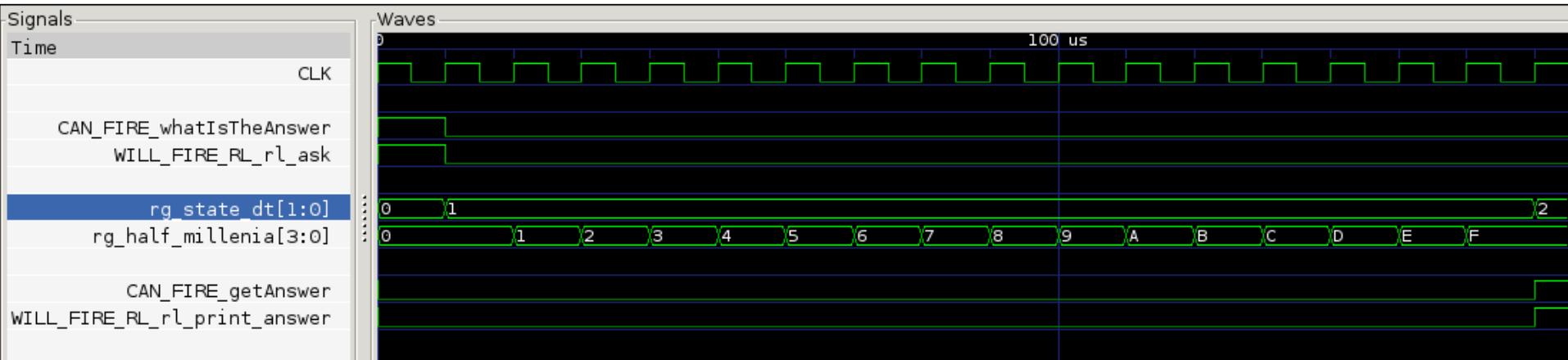
# Waveforms from the circuit

```
% ./mkTestbench_bsim -V  
Deep Thought says: Hello, World! The answer is 42.
```

-V: tells Bluesim simulation to dump waveforms from the circuit into “dump.vcd” file. Verilog simulators also have commands to capture VCDs. Note: you’ll get the same waveform whether from Bluesim or from Verilog sim.

```
% gtkwave dump.vcd
```

Displays the waves using “gtkwave” (you can use any convenient waveform viewer).



- The first wave shows the clock signal for the circuit (CLK)
- rg\_state can be seen transitioning from 0 (IDLE) to 1 (THINKING) to 2 (ANSWER\_READY)
- CAN\_FIRE\_whatIsTheAnswer shows that the method is enabled on the first clock, and WILL\_FIRE\_RL\_rl\_ask shows that the rule fires, invoking the method
- When rg\_state is THINKING, rg\_millenia and rg\_half\_millenia can be seen counting up. When they reach 7 and 1, respectively, rg\_state transitions to ANSWER\_READY (last clock)
- Then, CAN\_FIRE\_getAnswer is enabled, and WILL\_FIRE\_RL\_rl\_print\_answer shows that the rule fires, invoking the method

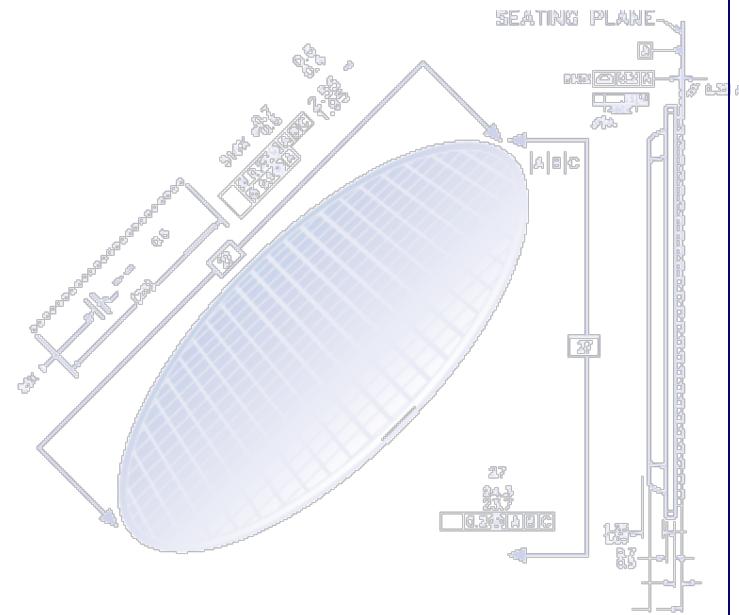
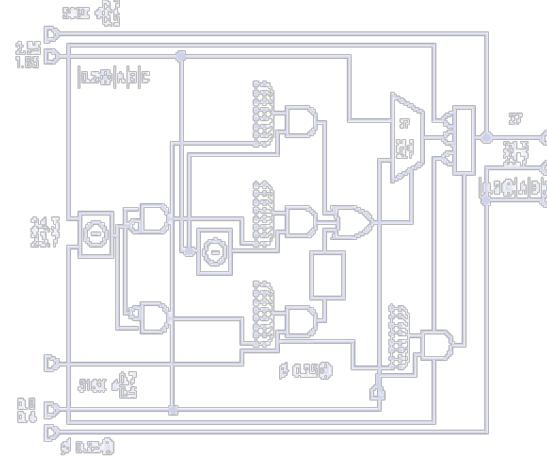
# Suggested exercises

*In this and future examples, we suggest extra exercises to deepen your understanding of BSV*

- In Eg02a, in rule rl\_print\_answer, exchange the two actions (\$display and \$finish). Is there any difference in behavior?
- In Eg02c, use the Makefile and build and run a Verilog simulation. Notice the “+bscvcd” flag in the v\_simulate action. This causes a “dump.vcd” file to be created, just like when you gave the “-V” flag to Bluesim. View this in a waveform viewer and check that it has the same cycle behavior as Bluesim.
- Examine the generated Verilog files mkTestbench.v and mkDeepThought.v (in the “verilog/” directory).
  - Look at the input and output ports, and understand how they correspond to the BSV interface DeepThought\_IFC and its methods.
  - Skim the interior of the Verilog module, and notice correspondences with the BSV source module (registers, rules, rule and method conditions, ...).
- In Eg02c, in module mkDeepThought, change the initial value of rg\_state\_dt from IDLE to THINKING and re-run the program. Change the initial value to ANSWER\_READY and re-run. Discuss the behaviors.
- In the waveforms we saw that IDLE, THINKING and ANSWER\_READY were encoded as 0, 1 and 2 respectively. Change the initial value of rg\_state\_dt from IDLE to 4, and try re-compiling. Discuss.



End





# BSV Training

## Eg03: Concurrent Bubblesort

## A simple concurrent Bubblesort example.

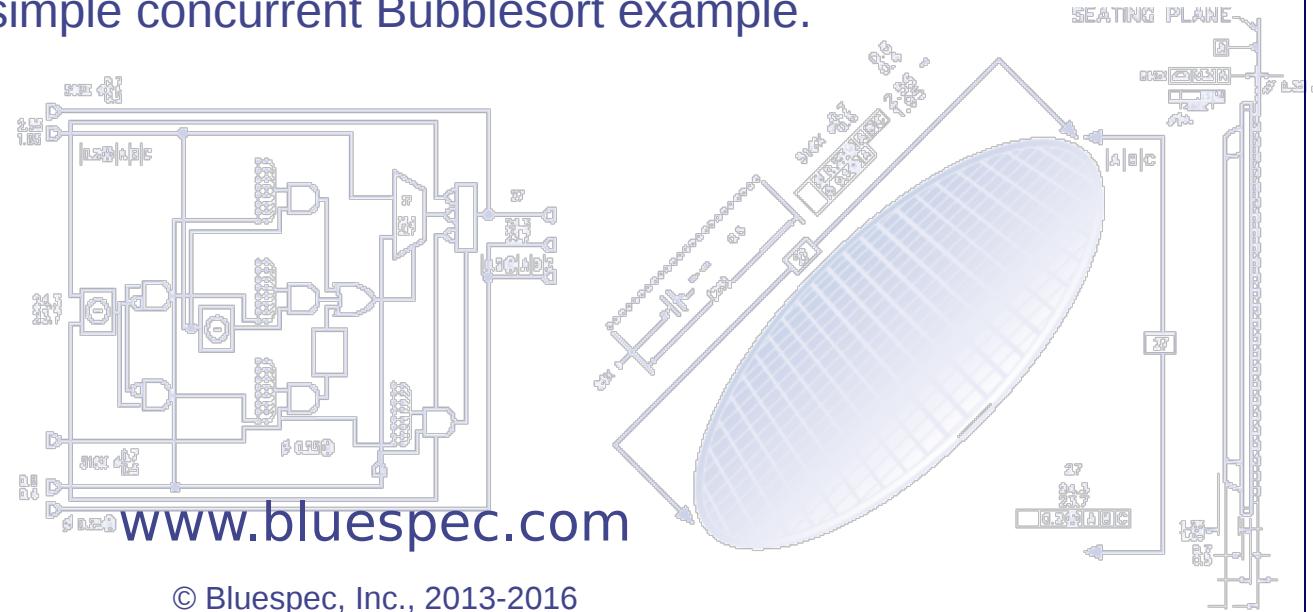
```

import POFN.*;

typedef Char(32) CharT;
module ex_Utf8Encoder;
Integer fileLength = 16;
function Int32() determineLength(DataT);
  return {32}();
endfunction

POFN(DataT) libfuncData;
POFN(CharT) libfuncChar;
POFN(DataT) encodeData();
POFN(CharT) encodeChar();
POFN(DataT) encodeData();
POFN(CharT) encodeChar();
endmodule

```



# Generalization via 5 versions

The accompanying code demonstrates several versions:

Eg03a_Bubblesort/	Sorts 5 items, each of type `Int#(32)'. Completely sequential, to show correspondence with conventional software implementation.
Eg03b_Bubblesort/	Parallel version. Uses 'maxBound' (largest Int#(32)) as a "special" value different from any of the sorted values (like "+infinity").
Eg03c_Bubblesort/	Generalizes '5' items to 'n' items.
Eg03d_Bubblesort/	Generalizes items of type 'Int#(32)' to items of arbitrary type 't' (i.e., makes the program polymorphic).
Eg03e_Bubblesort/	Uses the 'Maybe#(t)' type to eliminate the need for a 'maxBound' special value.

Note: this is a pedagogical introductory example focusing on concurrency and modularity, and is not intended as an example of efficient sorting!  
(See 'Eg06\_Mergesort' for more efficient and scalable sorting.)

# 1<sup>st</sup> version: directory Eg03a\_Bubblesort/

Examine the two source files: src\_BSV/Testbench.bsv src\_BSV/Bubblesort.bsv

We suggest that you take multiple passes through the files, incrementally increasing your understanding. Initially, just try to understand the syntactic structure, making frequent reference to the lecture slides in “Lec\_Basic\_Syntax” in the “Reference” directory.

The following excerpts show an outline of the syntactic structure of the two source files.

```
package Testbench;
...
import ...
import Bubblesort :: *;
...
Int#(32) n = 5;
...
module mkTestbench (Empty);
    Reg #(Int#(32)) rg_j1 <- mkReg (0);
    ...
    Sort_IFC sorter <- mkBubblesort;
    ...
rules
```

```
package Bubblesort;
...
import ...
...
interface Sort_IFC;
    ...
module mkBubblesort (Sort_IFC);
    Reg #(UInt #(3)) rg_j <- mkReg (0);
    ...
rules
...
method definitions
```

Examine the two source files: src\_BSV/Testbench.bsv src\_BSV/Bubblesort.bsv

```
package Testbench;
...
import ...
import Bubblesort :: *;
...
Int#(32) n = 5;
...
module mkTestbench (Empty);
    Reg #(Int#(32)) rg_j1 <- mkReg (0);
    ...
    Sort_IFC sorter <- mkBubblesort;
    ...
rules
```

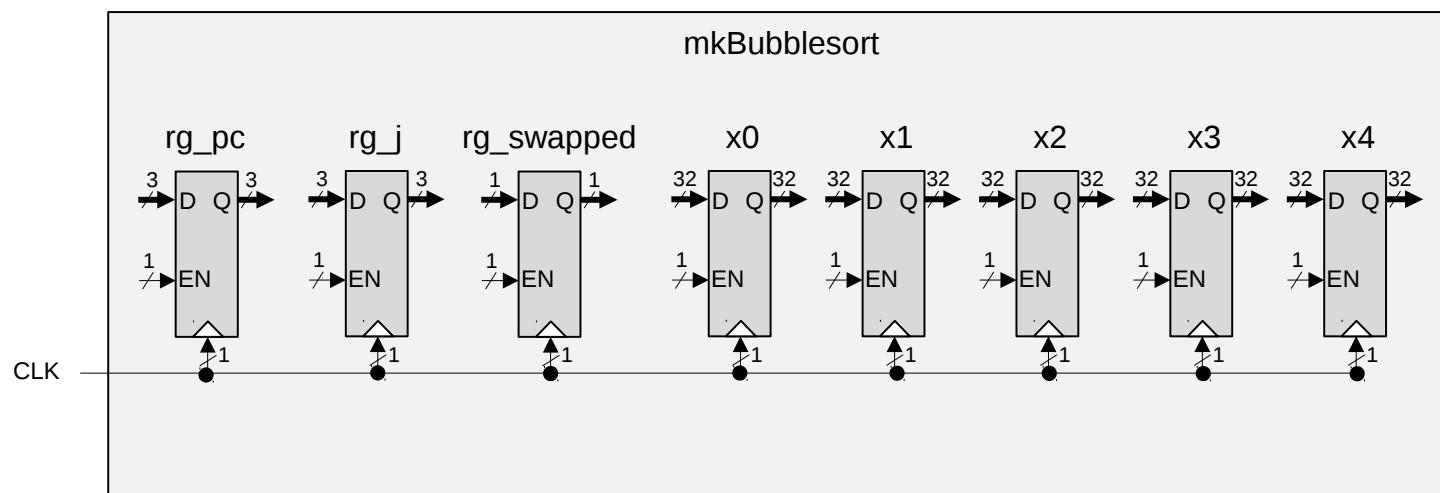
```
package Bubblesort;
...
import ...
...
interface Sort_IFC;
    ...
module mkBubblesort (Sort_IFC);
    Reg #(UInt #(3)) rg_j <- mkReg (0);
    ...
rules
...
method definitions
```

Notes on the syntactic structure:

- The top-level module is mkTestbench; it has an Empty interface (an interface with no methods).
- It instantiates several sub-modules: a few registers, a pseudo-random number generator (LFSR, for Linear Feedback Shift Register), and the mkBubblesort module.
- The mkBubblesort module, in turn, instantiates some registers as its sub-modules.
- The rules in the testbench invoke the put and get methods in the bubblesort module's interface.
- All the rules and methods have Boolean conditions indicating when they are “ready”.

In module mkBubblesort:

- The register rg\_pc is used to sequence the sorting actions. The name is suggestive of “Program Counter”. 3 bits are enough to distinguish all the states.
- The register rg\_j is used to count 5 incoming values. Thus, it is typed UInt#(3), i.e., an unsigned integer of 3 bits, capable of holding values 0..7. It’s reset value (initial value) is 0.
- The register rg\_swapped remembers whether any swap occurred in the current pass.
- The registers x0..x4 hold the 5 values to be sorted. Each value is of type Int#(32), i.e., a signed integer of 32 bits. Each register’s reset value is maxBound, a symbolic name representing the largest Int#(32), i.e.,  $+2^{31-1}$ . We assume all input values to be sorted will be strictly  $<$  maxBound



The module mkBubblesort, showing the registers in isolation

The rules in module mkBubblesort essentially encode a sequential algorithm similar to that shown below in a pseudo-C notation:

```
swapped = False;
while (True)
    for (pc = 1; pc < 5; pc++) {
        if (x[pc-1] > x[pc] {
            swap them;
            swapped = True;
        }
        if (! swapped) break;
    }
```

Each rule has:

- A condition that says when it can fire, and
- A body that says what happens if it fires.

# Building and running the codes

Each variation is built and run in the same way:

- In the Build/ directory you can use the ‘Makefile’ for building and running Bluesim or Verilog sim:

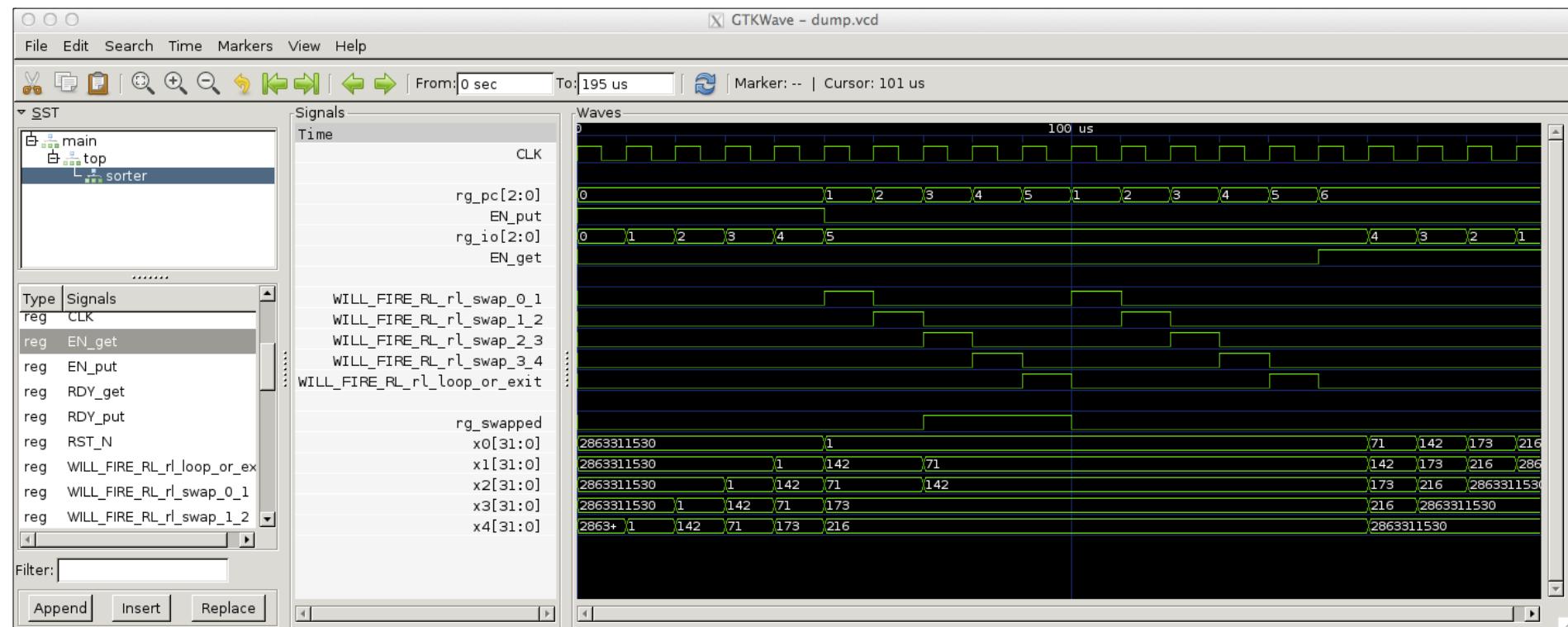
```
% make compile link simulate      // for Bluesim  
% make verilog v_link v_simulate // for Verilog sim
```

Note: When building for Bluesim, compiler-temporary files are created in the build\_bsim/ directory.  
When building for Verilog, temporaries are in build\_v/ and Verilog files are in verilog\_dir/.  
'make clean' and 'make full\_clean' will clean up these directories.

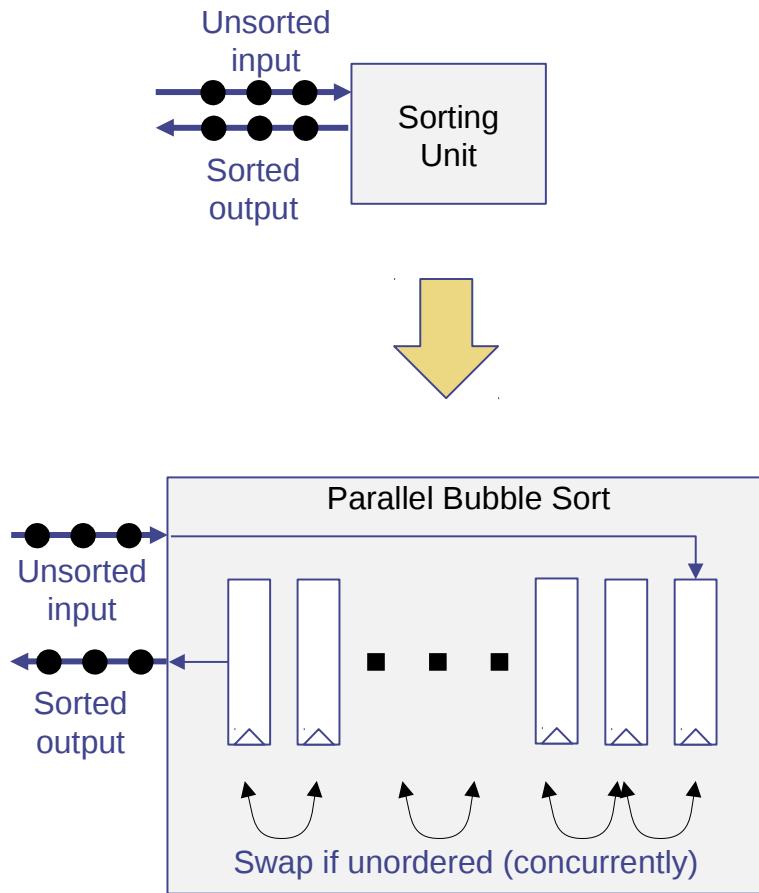
# Build and run the 1<sup>st</sup> version

- In the Build/ directory, practice building and running the program in both ways:
  - Using ‘make’ commands, for Bluesim.
  - Using ‘make’ commands, for Verilog sim.
- Observe the inputs and outputs and verify that they are reasonable (there are 5 inputs and outputs, and the outputs are a sorted version of the inputs).
- When you run your simulation, a file “dump.vcd” is created, containing waveforms, which you can view in your favorite waveform viewer.
  - (It is created because of the –V flag for Bluesim or the +bscvc flag for Verilog sim.)
  - The best way to view waveforms is from BDW (Bluespec Development Workstation), because it can customize the view to show data using the more expressive BSV source code types, instead of the less expressive Verilog bits.
  - The picture “Waves\_screenshot.tiff” is a screenshot of such a view.
  - Study the waveform and make sure you understand how it reflects the behavior of the BSV code.

# Waveforms from the 1<sup>st</sup> version



# Eg03: Basic concurrency and modularity



## Algorithm/Architecture for remaining versions:

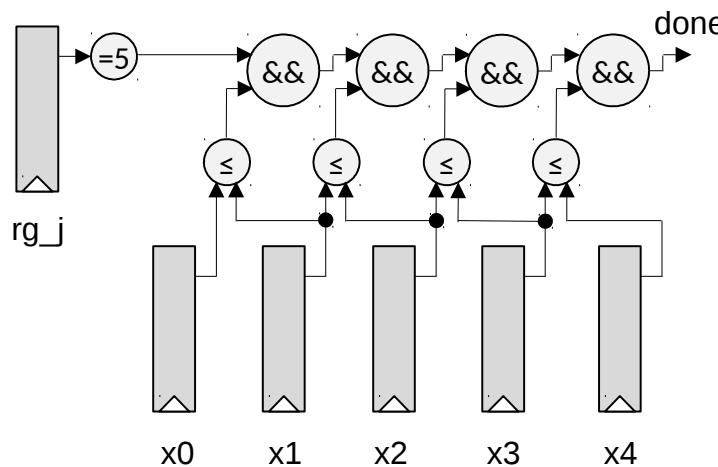
- The module accepts a stream of 'n' input items (unsorted).
- These are shifted in to 'n' registers.
- Concurrently (and even while inputs are arriving), whenever two adjacent registers contain values in the wrong order, we swap their contents.
- When 'n' inputs have been received, and all of them are sorted, the module yields a stream of 'n' sorted outputs by shifting them out.
- When 'n' outputs have been streamed out, the module is ready for its next set of 'n' inputs.

In module mkBubblesort:

- The Bool function “done” defines the condition that all values have been received and are sorted:

```
// Test if array is sorted
function Bool done ();
    return ((rg_j == 5) && (x0 <= x1) && (x1 <= x2)
            && (x2 <= x3) && (x3 <= x4));
endfunction
```

- Note: in BSV, a zero-argument function like this is identical to a constant
- In BSV, value-expressions like this just represent combinational circuits:

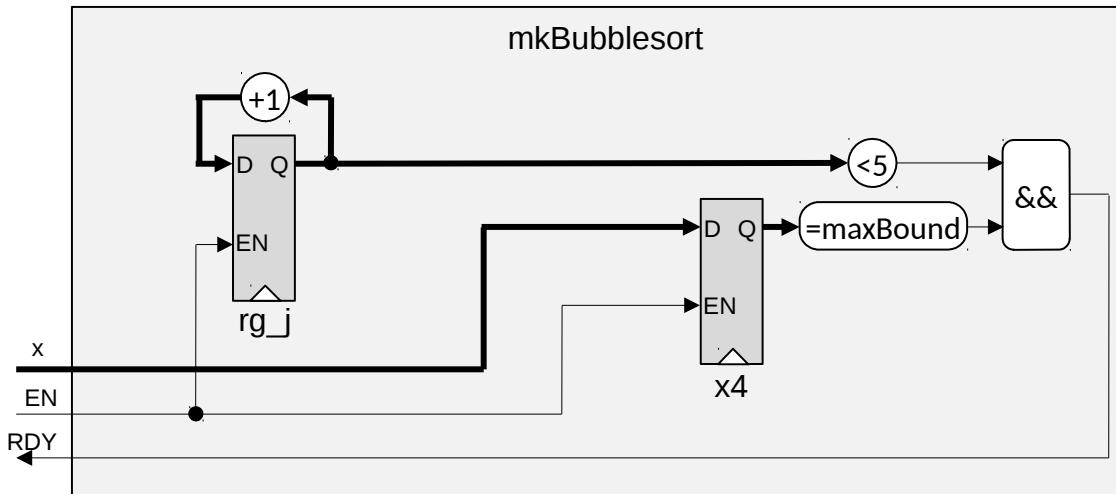


```
// Inputs: feed input values into x4
method Action put (Int#(32) x) if ((rg_j < 5) && (x4 == maxBound));
    x4 <= x;
    rg_j <= rg_j + 1;
endmethod
```

In module mkBubblesort:

- The “put” method is of type Action, i.e., its body is an expression of type Action.
  - The body has two sub-Actions: one places a new input value x into register x4, and the other increments the input count rg\_j.
  - The method condition ensures two things:
    - $(rg_j < 5)$  ensures that we stop after receiving 5 inputs
    - $(x4 == maxBound)$  ensures that we don’t over-write a meaningful value already in x4, i.e., it can place a new value in x4 only after the swap rules have moved the previous value out and replaced it with maxBound.

Hardware for put method, in isolation



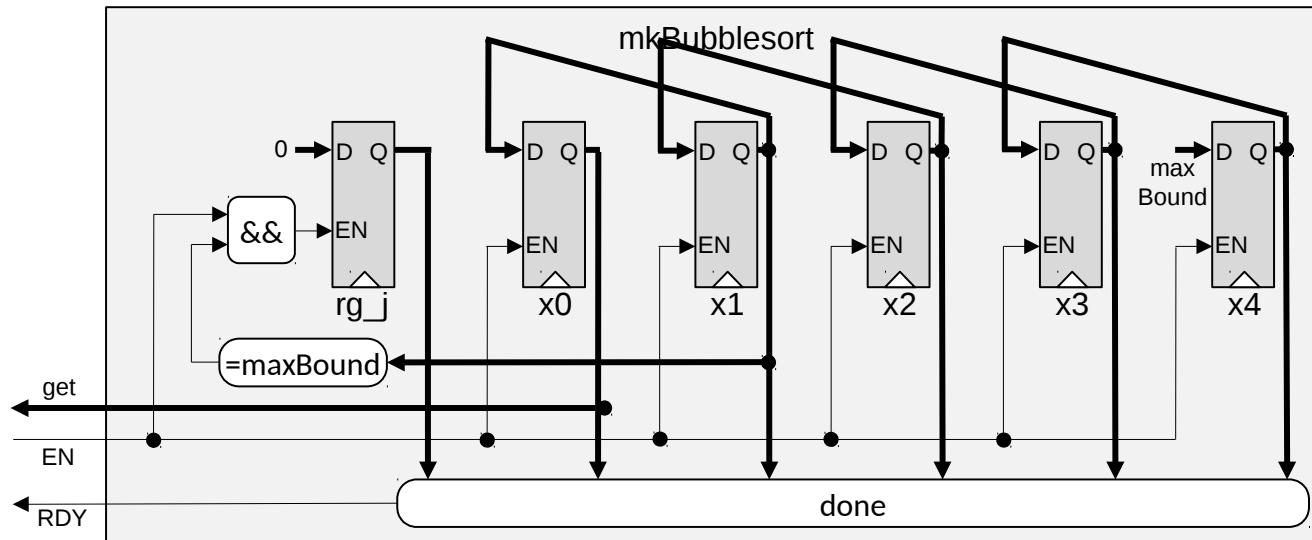
```
// Outputs: drain by shifting them out of x0
method ActionValue#(Int#(32)) get () if (done);
    x0 <= x1;
    x1 <= x2;
    x2 <= x3;
    x3 <= x4;
    x4 <= maxBound;
    if (x1 == maxBound) rg_j <= 0;
    return x0;
endmethod
```

In module mkBubblesort:

- The “get” method is of type ActionValue #(Int#(32)), i.e., it’s body is an Action, and it also returns a value of type Int#(32).
  - The body returns x0, shifts all remaining values in x1..x4 down into x0..x3, respectively, and shifts the value maxBound into x4.
  - The method condition of the “get” method:
    - Is only enabled when “done” is true, i.e., all inputs have been received and all values are sorted
    - Is only enabled until we’ve returned the 5<sup>th</sup> value, after which x0 contains maxBound
  - When we return return the 5<sup>th</sup> value (from x0), x1 will be maxBound; at this time we can reset rg\_j to 0
  - Note that after returning 5 values, the module is again in its original state—rg\_j contains 0, and x0..x4 contain maxBound—and so it is ready to receive the next 5 values to be sorted

```
// Outputs: drain by shifting them out of x0
method ActionValue#(Int#(32)) get () if (done);
    x0 <= x1;
    x1 <= x2;
    x2 <= x3;
    x3 <= x4;
    x4 <= maxBound;
    if (x1 == maxBound) rg_j <= 0;
    return x0;
endmethod
```

Hardware for “get” method, in isolation



```

rule rl_swap_0_1 (x0 > x1);
  x0 <= x1;
  x1 <= x0;
endrule

rule rl_swap_1_2 (x1 > x2);
  x1 <= x2;
  x2 <= x1;
endrule

```

```

rule rl_swap_2_3 (x2 > x3);
  x2 <= x3;
  x3 <= x2;
endrule

(* descending_urgency =
"rl_swap_3_4, rl_swap_2_3,
rl_swap_1_2, rl_swap_0_1" *)
rule rl_swap_3_4 (x3 > x4);
  x3 <= x4;
  x4 <= x3;
endrule

```

In module mkBubblesort:

- In each of the four rules:
  - The rule condition (an expression of type Bool), e.g., “(x0>x1)” tests if two adjacent values are in the wrong order. This is a prerequisite for the rule to “fire”, i.e., to execute its body.
  - The rule body (an expression of type Action) is composed of two sub-Actions whose effect is to swap the contents of the corresponding two registers. Note:
    - Expressions/methods/functions of type “Action” or “ActionValue#(t)” (potentially) change the state of the system (e.g., write to a register, write to memory, enqueue onto a FIFO, etc.). We also say that such expressions have “side-effects”.
    - Expressions that are not of type Action or ActionValue (e.g., Bool) *never* change the state of the system (this is guaranteed by BSV’s type-checking rules). We also say that these are “pure” or “value” expressions.
    - All Actions in a rule are considered instantaneous and simultaneous. The two assignments are not read sequentially (as is typical in software programs), and this explains why there is no need for a “temporary” intermediate variable (as is typical in software programs)

```

rule rl_swap_0_1 (x0 > x1);
  x0 <= x1;
  x1 <= x0;
endrule

rule rl_swap_1_2 (x1 > x2);
  x1 <= x2;
  x2 <= x1;
endrule

```

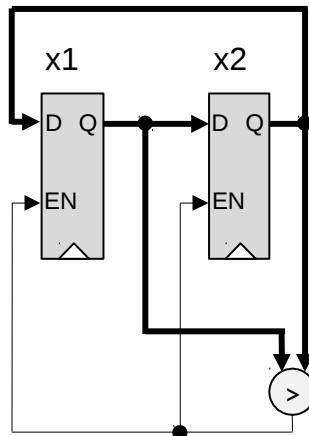
```

rule rl_swap_2_3 (x2 > x3);
  x2 <= x3;
  x3 <= x2;
endrule

(* descending_urgency =
"rl_swap_3_4, rl_swap_2_3,
rl_swap_1_2, rl_swap_0_1" *)
rule rl_swap_3_4 (x3 > x4);
  x3 <= x4;
  x4 <= x3;
endrule

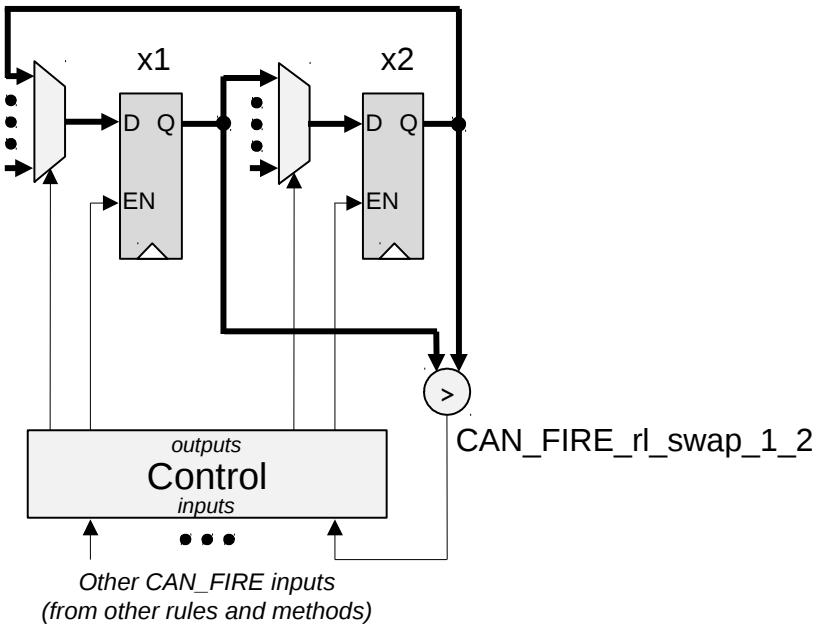
```

Hardware for rl\_swap\_1\_2, in isolation



Other data inputs (from other rules and methods)

... and in more detail



In module mkTestbench:

- rg\_j1 and rg\_j2 are used to count 5 inputs and 5 outputs respectively. They are somewhat arbitrarily typed at Int#(32) (since they count 0..4, even Int#(3) or UInt#(3) would have been ok, but we usually don't worry so much about minimizing state bits in a testbench).
- mkLFSR\_8 instantiates, from the BSV library, a pseudo-random number generator of 8-bit values (see Sec.C.8.1 in the Reference Guide).
- In rule “rl\_feed\_inputs”:
  - The rule condition ensures that it only enabled to generate 5 inputs
  - lfsr.value() is a “value method” that returns a Bit#(8); zeroExtend() extends this to Bit#(32); unpack() converts this to an Int#(32). lfsr.next() is an Action method that tells the LFSR to advance to its next random number
  - “sorter.put(x)” sends this new input into the bubblesort module
    - Recall that the “put” method in mkBubblesort has a method condition that prevents overwriting a previous value. Thus, rl\_feed\_inputs can only fire when that condition is true.
- In rule “rl\_drain\_outputs”
  - The rule condition ensures that it only retrieves 5 outputs
  - “sorter.get()” yields the next output, whenever the method condition on the “get” method allows, and this value is displayed.

## Summary of behavior:

- Rule “rl\_feed\_inputs” in the testbench module feeds 5 random values into the sorter module using the “put” method.
- The sorter module receives the 5 inputs via the “put” method. Sorting begins as soon as the first input arrives (the first step sorts the first input against its +infinity neighbor), and continues as more inputs arrive. Sorting may continue for some time after all inputs arrive, until all 5 registers are sorted.
- When all 5 registers are sorted, the “get” method is enabled, and the testbench drains out the 5 values in order. The sorter module yields these 5 values in order by shifting them through the registers.
- After yielding 5 outputs, the sorter module is back in its initial state, and ready for another 5 inputs (although our testbench quits after the first set).

*This behavior can be seen in the waveforms for this design created during simulation. We will shortly describe simulation and waveform generation but, for your convenience, the directory includes a screen shot (“waves\_screenshot.tiff”) which you can view directly.*

# Build and run the 2<sup>nd</sup> version

- In the Build/ directory, practice building and running the program in both ways:
  - Using ‘make’ commands, for Bluesim.
  - Using ‘make’ commands, for Verilog sim.
- Observe the inputs and outputs and verify that they are reasonable (there are 5 inputs and outputs, and the outputs are a sorted version of the inputs).
- Note that an input is supplied only on *every other clock*. Why?
  - (Hint: see previous remarks on method condition of the “put” method.)
- When you run your simulation, a file “dump.vcd” is created, containing waveforms, which you can view in your favorite waveform viewer.
  - (It is created because of the –V flag for Bluesim or the +bscvc flag for Verilog sim.)
  - The best way to view waveforms is from BDW (Bluespec Development Workstation), because it can customize the view to show data using the more expressive BSV source code types, instead of the less expressive Verilog bits.
  - The picture “waves\_screenshot.tiff” is a screenshot of such a view. Note that it shows BSV source code types Int#(32), Bool, ..., and not raw Verilog bits.
  - Study the waveform and make sure you understand how it reflects the behavior of the BSV code.

# Suggested exercises

*In this and future examples, we suggest extra exercises to deepen your understanding of BSV, which you can pursue on your own (may not be covered during classroom training)*

- Examine the generated Verilog file mkBubblesort.v (in the “verilog/” directory).
  - Look at the input and output ports, and understand how they correspond to the BSV interface Sort\_IFC and its methods.
  - Skim the interior of the Verilog module, and notice correspondences with the BSV source module (registers, rules, rule and method conditions, ...).
- Analyze and explain in detail what would happen if the testbench supplied ‘maxBound’ as one (or more) of the input values. If you wish, modify the testbench to test this.
- Modify the testbench to sort two (or three, or more) rounds of 5-input sequences, instead of quitting after the first round.
- Modify the testbench and the bubblesort modules to sort sequences of some other length (say, 10) instead of 5.

# Time-out to reinforce some concepts

Before moving on with the examples, please study the lecture: [Lec\\_Rule\\_Semantics](#) to understand the concepts behind Rules:

- Semantics of individual rules (and the methods they call)
  - “simultaneous/parallel actions”, “instantaneous”
- Semantics of concurrency of a set of rules in each clock
  - “concurrency”
  - “ordering constraints” between methods in different rules
  - “rule schedules”

### 3<sup>rd</sup> version: directory Eg03c\_Bubblesort/

The 3<sup>rd</sup> version generalizes the 2<sup>nd</sup> version from sorting 5 numbers to sorting  $n$  numbers

Examine the two source files: src\_BSV/Testbench.bsv src\_BSV/Bubblesort.bsv  
It is useful to compare these side-by-side with the corresponding two files in Eg03b\_Bubblesort/src\_BSV/, to see how they have changed.

Notice that the interface type is now parameterized by  $n$ :

```
interface Sort_IFC #(numeric type n_t);
```

In BSV, certain types and type parameters can be *numeric types*.

Note that *numeric types* (which are only meaningful at compile time) are distinct from *numeric values* (which can of course occur at run time).

BSV is very strict in type-checking—there are no automatic conversions  
The code uses these explicit conversions:



The 2<sup>nd</sup> version's explicit registers x0..x4 have here been replaced by a vector of registers:

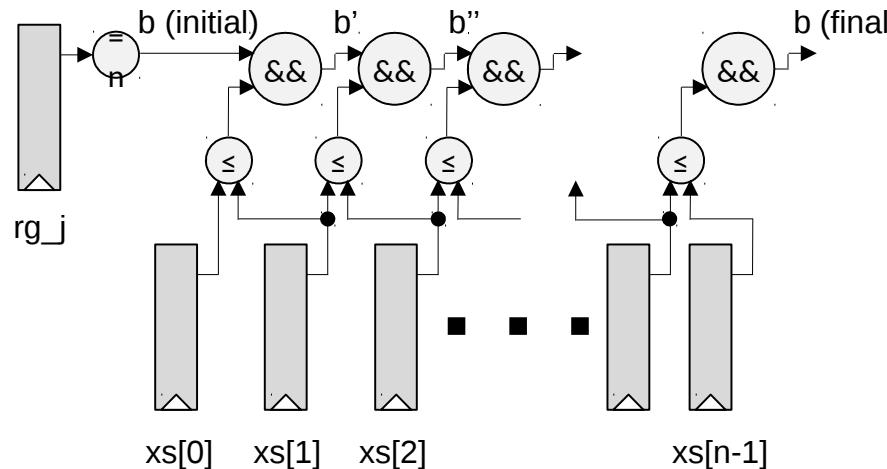
```
Vector #(n_t, Reg #(Int #(32))) xs <- replicateM (mkReg (maxBound));
```

“replicateM” repeatedly applies “mkReg(maxBound)” to create them.

The 2<sup>nd</sup> version's explicit “done” test is here computed in a for-loop:

```
function Bool done ();
    Bool b = (rg_j == fromInteger (n));
    for (Integer i = 0; i < n-1; i = i + 1)
        b = b && (xs[i] <= xs[i+1]);
    return b;
endfunction
```

Note: this for-loop is also “statically elaborated” (i.e., unfolded by the compiler) into a circuit of combinational logic



Observe that, unlike traditional languages, there is no storage location corresponding to variables like `i` and `b`. The former has disappeared completely, and the latter exists in multiple versions (`b`, `b'`, `b''`, ...) each of which just names a wire.

The 2<sup>nd</sup> version's separately-written rules are here replaced by a for-loop generating them:

```
for (Integer i = 0; i < n-1; i = i+1)
  rule rl_swap_i (xs [i] > xs [i+1]);
    xs [i]   <= xs [i+1];
    xs [i+1] <= xs [i];
  endrule
```

Note: this for-loop is “statically elaborated” (i.e., unfolded by the compiler), and is equivalent to writing out  $n-1$  rules explicitly.

In the “put” method, the following line:

```
writeVReg (xs, shiftInAtN (readVReg (xs), maxBound));
```

uses a number of BSV library functions on vectors:

- readVReg() returns a vector of values corresponding to the contents of a vector of registers
- shiftInAtN() takes a vector of values x<sub>1</sub>,...,x<sub>N</sub> and a new value y, and returns a vector of values x<sub>2</sub>,...,x<sub>N</sub>,y (discards x<sub>1</sub>)
- writeVReg() writes a vector of values into a vector of registers

The above line could also have been written like this:

```
for (Integer i = 0; i < n-1; i = i+1)
    xs[i] <= xs[i+1];
xs[n-1] <= maxBound;
```

Note: this for-loop is also “statically elaborated” (i.e., unfolded by the compiler)

# Time-out to reinforce some concepts

Before moving on with the examples, please study the lecture: [Lec\\_Types](#) to understand the concepts behind types, polymorphism, and numeric types.

Please also *skim* through Section C.3 (“Vectors”) in the Reference Guide.

# Synthesis hierarchy

- In the 1<sup>st</sup> and 2<sup>nd</sup> versions, in front of ‘module mkBubblesort’ line, there is a (\*synthesize\*) attribute:

```
(* synthesize *)
module mkBubblesort(Sort_IFC);
...
endmodule
```

- When generating Verilog, this creates a ‘mkBubblesort.v’ file with a ‘mkBubblesort’ Verilog module
- In the 3<sup>rd</sup> version, this (\*synthesize\*) attribute is removed.
- This is because BSV cannot separately synthesize *polymorphic* modules; they can only be inlined into a parent module
- Instead, in Testbench.bsv, we have created a specific instance of the module (for  $n=20$ ); since this is no longer polymorphic, it can be separately synthesized. This is the module actually instantiated in the module mkTestbench:

```
typedef 20 N_t;
...
(* synthesize *)
module mkBubblesort_nt (Sort_IFC #(N_t));
    Sort_IFC #(N_t) m <- mkBubblesort;
    return m;
endmodule
```

- The above is a common idiom in BSV code, for creating a separately synthesized instance of a polymorphic module

## Build and run the 3<sup>rd</sup> version

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier
- Observe the inputs and outputs and verify that they are reasonable (that there are 20 inputs and outputs, and the outputs are a sorted version of the inputs)
- In Testbench.bsv, change the 1 line that defines the size of the problem:

```
typedef 20 N_t;
```

Rebuild and re-run to test it.

## 4<sup>th</sup> version: directory Eg03d\_Bubblesort/

The 4<sup>th</sup> version generalizes the 3<sup>rd</sup> version from sorting values of type Int#(32) to sorting values of any type “t”

Examine the two source files: src\_BSV/Testbench.bsv src\_BSV/Bubblesort.bsv

It is useful to compare these side-by-side with the corresponding two files in the previous version, to see how they have changed.

Notice that the interface type is now further parameterized by *t*:

```
interface Sort_IFC #(numeric type n_t, type t);
```

The “module mkBubblesort” line is now extended with *provisos*:

```
provisos (Bits #(t, wt),
          Ord #(t),
          Eq #(t),
          Bounded #(t));
```

These are assertions, respectively, that:

- values of type *t* have a bit representation (with bit-width *wt*), since we need to store them in registers
- values of type *t* can be compared with the `<=` operator (ordering)
- values of type *t* can be compared with the `==` operator (equality)
- there exists a ‘maxBound’ value of type *t*

Without these assertions, the compiler would emit a type-checking error, since all these properties are used inside the module.

# Time-out to reinforce some concepts

Please study the lecture: Lec\_Typeclasses

to understand the concepts behind Bits, Ord, Eq, Bounded.

## Build and run the 4<sup>th</sup> version

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier
- Observe the inputs and outputs and verify that they are reasonable (that there are 20 inputs and outputs, and the outputs are a sorted version of the inputs)
- In Testbench.bsv, change the two lines that define the size of the problem and the type of values being sorted:

```
typedef 20 N_t;
typedef UInt #(24) MyT;
```

Rebuild and re-run to test it.

## 5<sup>th</sup> version: directory Eg03e\_Bubblesort/

The 5<sup>th</sup> version eliminates the dependency on the existence of a separate ‘maxBound’ value that is separate from legitimate input values.

Before proceeding, please review the lecture: Lec\_Types and, specifically, the section on “Maybe” types.

Examine the two source files: src\_BSV/Testbench.bsv src\_BSV/Bubblesort.bsv

It is useful to compare these side-by-side with the corresponding two files in the previous version, to see how they have changed.

Notice that the “Bounded#(t)” proviso has been removed from module mkBubblesort.

The vector of registers now contain values of type “Maybe#(t)” instead of “t”:

```
Vector #(n_t, Reg #(Maybe #(t))) xs <- replicateM (mkReg (tagged Invalid));
```

These are initialized/reset to the value “tagged Invalid”.

This plays the role previously played by “maxBound”.

In the “put” method, instead of inserting x into the register, we insert:

```
xs[jMax] <= tagged Valid x;
```

After the module mkBubblesort, there is some new code

```
instance Ord #(Maybe #(t))
  provisos (Ord #(t));
  ...

```

A value of type Maybe#(t) has two forms:

- tagged Invalid
- tagged Valid x

Conceptually, if values of type t are represented in n bits, then a value of type Maybe#(t) is represented in n+1 bits. The extra bit is called a tag. When the tag is 0 (Invalid), the remaining n bits don't matter. When the tag is 1 (Valid), the remaining n bits represent a legitimate value of type t.

The “instance” declaration defines how to compare two values of type Maybe#(t):

- tagged Invalid <= tagged Invalid
- tagged Valid x < tagged Invalid      for any x
- tagged Invalid > tagged Valid y      for any y
- tagged Valid x <= tagged Valid y      if ( $x \leq y$ )

Please see the lecture: Lec\_Typeclasses  
for a more detailed explanation of these concepts.

# Build and run the 5<sup>th</sup> version

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier
- Observe the inputs and outputs and verify that they are reasonable (that there are 20 inputs and outputs, and the outputs are a sorted version of the inputs)
- In Testbench.bsv, change the two lines that define the size of the problem and the type of values being sorted:

```
typedef 20 N_t;
typedef UInt #(24) MyT;
```

Rebuild and re-run to test it.

# Time-out to reinforce some concepts

Please study the lecture: Lec\_Types  
to understand the concepts behind the Maybe type.

# Suggested exercises

- Change the program to sort in *descending* order instead of ascending order.
- Instead of sorting scalar values, change the testbench to sort struct values:
  - Define a struct type with at least two fields.
  - Generate structs with random values for these fields.
  - Define the “<” operator on this struct type so as to compare only one field (you will have to declare an “Ord” instance for this struct type).
  - Test your program.
- Study Section C.3.10 (“Fold functions”) in the Reference Guide. In the sorter module, redefine the “done” function to use foldl(), foldr() or fold() in place of the explicit for-loop. What is the circuit structure using the for-loop, foldl, foldr and fold? What is the advantage of the circuit structure using fold?
- The “done” function is a combinational circuit (using any of the above implementations). For large  $n$ , this can limit the clock speed at which the circuit can be synthesized. Split the circuit into multiple smaller stages, by adding registers at suitable points and introducing rules to propagate values.
  - This means that detecting “done” will be delayed by a few cycles. Does this matter?
  - How should you reset these registers during final output in the “get” method?

# Summary

These examples have provided a basic familiarity with many of the concepts in the BSV language:

- File and package structure
- Module structure, module instantiation, interfaces and methods
- Rules and methods, and their semantics
- Types and typeclasses, numeric types
- Static elaboration
- Parameterization on sizes
- Parameterization on types (polymorphism)

It has also provided practice in using various Bluespec tools:

- Building and executing in Bluesim
- Generating Verilog, and executing in Verilog sim
- Using Makefiles
- Generating waveforms and viewing them
- BDW workstation



End

```

import POFSet;

typedef POFSet<T> SetT;
typedef SetT::iterator SetIter;
typedef SetT::const_iterator ConstSetIter;

Integer fileDepth = 15;

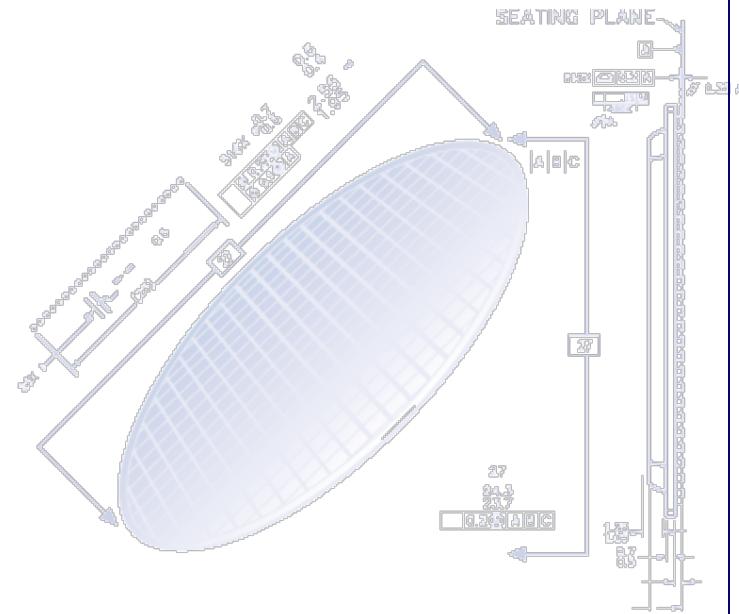
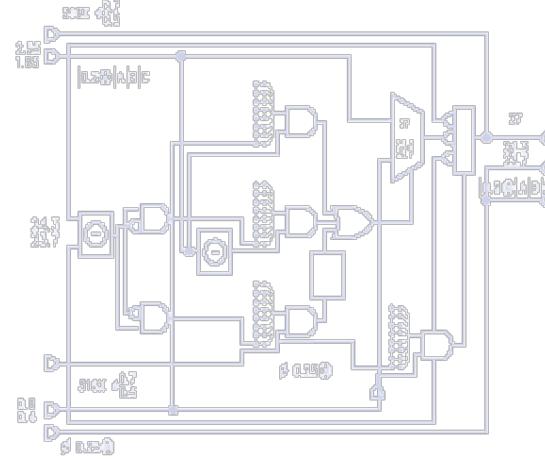
function UInt32 determine_group(DataT&); // returns <0,0>; exit when
exit when;
exit when;

POFSet<DataT> knownS;
DataT& POFSet<DataT>::operator[](the_index);
POFSet<DataT> containedS;
DataT& POFSet<DataT>::operator[](the_index);
POFSet<DataT> containedS;
DataT& POFSet<DataT>::operator[](the_index);

rule end (true);
    DataT b_file = bknownS[file];
    POFSet<DataT> cut_group =
        determine_group(b_file) == 0 ? containedS : bknownS;
    cut_group[the_index] = b_file;
    exit when;
    exit when;
endrule

endpackage : ex_1st_cnf_ba

```



# BSV Training

## Eg04: Microarchitectures: FSMs and Pipelines

Using a “dynamic shifter” as an example, we illustrate how BSV rules can be used to express common microarchitecture structures: iterative, rigid-pipelined and elastic-pipelined.

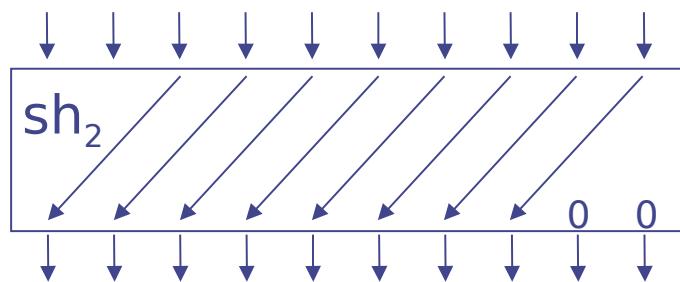
```
import PFR#(type);
type Bit#(T) default;
module end
  localparam T shift;
  integer fifo_depth = 16;
  function Bit#(T) determine_pwr(Bit#(T) val);
    return (val > 0) ? val : 0;
  endfunction
  PFR#(Bit#(T) inword) the_inword#(inword);
  PFR#(Bit#(T) outword) the_outword#(outword);
  PFR#(Bit#(T) inaddr) the_inaddr#(inaddr);
  PFR#(Bit#(T) outaddr) the_outaddr#(outaddr);
  PFR#(Bit#(T) inctrl) the_inctrl#(inctrl);
  PFR#(Bit#(T) outctrl) the_outctrl#(outctrl);
  rule end (true);
    Bit#(T) in_ctrl = inctrl.first;
    PFR#(Bit#(T)) out_ctrl = inctrl;
    inctrl.pop();
    if (in_ctrl > 0) begin
      inword.enq(inword.first);
      inctrl.pop();
    end
    else begin
      inctrl.pop();
    end
  endrule
endmodule
```



# Dynamic shifts

- Goal: circuit to implement a left-shift by a dynamic amount:  $z = \text{shift}(x, y)$   
i.e.,  $z = x$  left-shifted by  $y$  positions, where  $y$  is dynamic (run-time value)
- Algorithm: a dynamic shift can be achieved as a composition of static shifts corresponding to each bit of  $y$ .
- Example: Suppose  $y$  has type Bit #(3)
  - $\text{shift}(x, y) =$ 

shift $x$ by 1 ( $= 2^0$ )	if $y[0] == 1$
and by 2 ( $= 2^1$ )	if $y[1] == 1$
and by 4 ( $= 2^2$ )	if $y[2] == 1$
- Note: shifting by constant  $2^j$  is trivial: just a “lane change” using only wires, no gates:



# Example variations

The accompanying code demonstrates six variations:

- The following are in Eg04a\_MicroArchs/src\_BSV/. Each shifts an 8-bit value x by a 3-bit value y.

Shifter_iterative.bsv	Sequential, iterative
Shifter_pipe_rigid.bsv	Pipelined. Rigid (“synchronous”, assumes no gaps in data stream)
Shifter_pipe_elastic.bsv	Pipelined. Elastic (“asynchronous”, accommodates gaps in input stream)

- The following are in Eg04b\_MicroArchs/src\_BSV/. They are generalizations of the previous three, such that each shifts an  $n$ -bit value x by a  $\log(n)$ -bit value y. The testbench demonstrates instances where  $n = 16$

Shifter_iterative.bsv	... ditto ...
Shifter_pipe_rigid.bsv	... ditto ...
Shifter_pipe_elastic.bsv	... ditto ...

# Building and running the codes

Each variation is built and run in the same way:

- In the “src\_BSV” directory, create a symbolic link from “Shifter.bsv” to the variation of interest. E.g.,

```
% ln -s -f Shifter_iterative.bsv Shifter.bsv
```

- In the Build directory you can use the ‘Makefile’ for building and running Bluesim or Verilog sim:

```
% make compile link simulate      // for Bluesim  
% make verilog v_link v_simulate // for Verilog sim
```

# Interface for the shifter(s)

All three variations of the shifter have the same interface (see file Shifter\_IFC.bsv):

```
typedef Server #(Tuple2 #(Bit #(8), Bit #(3)),  
                  Bit #(8))  
    Shifter_IFC;
```

This is an example of a common BSV practice—to re-use “standard” interfaces already provided in the BSV library, rather than defining new, *ad hoc* interfaces for each new module:

```
interface Server #(t1, t2);  
    interface Put #(t1) request;  
    interface Get #(t2) response;  
endinterface
```

(from the ClientServer library)

```
interface Put #(t1);  
    method Action put (t1 x);  
endinterface
```

```
interface Get #(t2);  
    method ActionValue #(t2) get ();  
endinterface
```

(from the GetPut library)

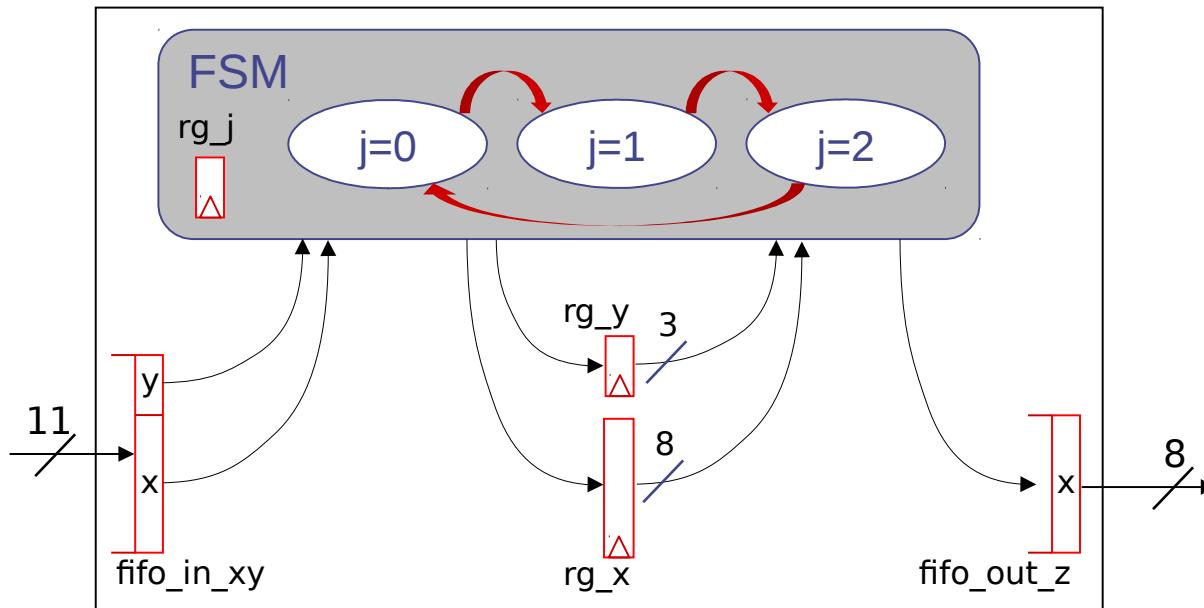
*Note: these are similar to interfaces in the SystemC TLM 2.0 library*

Our interface **Shifter\_IFC** will therefore have

- a **request.put** method by which the environment can send a 2-tuple input
  - The 2-tuple is a pair of values, 8 bits (for x) and 3 bits (for y)
- a **response.get** method by which the environment can receive an 8 bit output

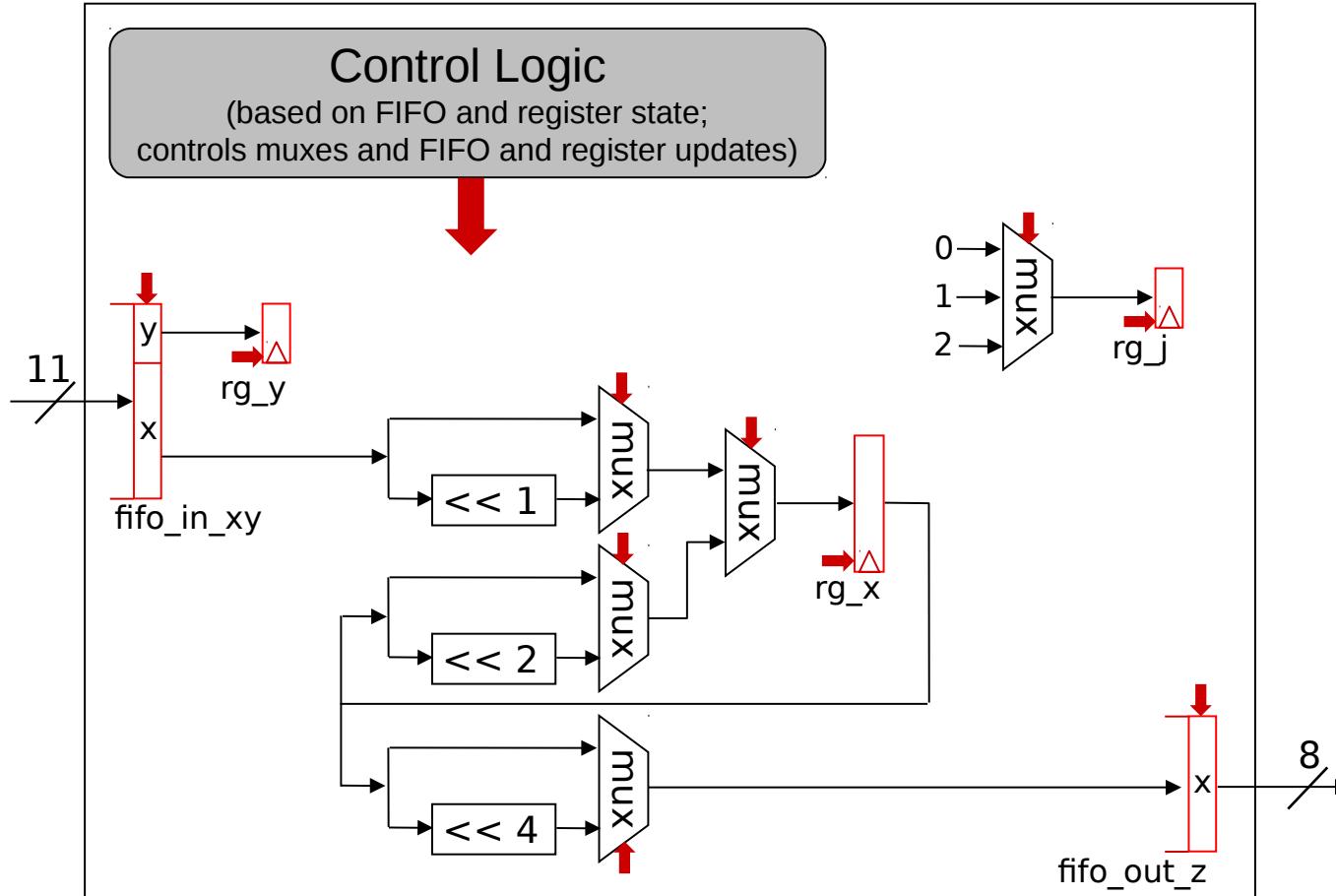
# Sequential, iterative shifter

File: Eg04a\_MicroArchs/src\_BSV/Shifter\_iterative.bsv



The FSM (and its actions) is implemented using 3 BSV rules

# Sequential, iterative shifter: more detail



*The Control Logic is automatically compiled by bsc from the BSV rules*

# A testbench to drive the shifter module

File: Eg04a\_MicroArchs/src\_BSV/Testbench.bsv

```
module mkTestbench (Empty);
    Shifter_Ifc  shifter <- mkShifter;

    Reg #(Bit #(4)) rg_y <- mkReg (0);

    rule rl_gen (rg_y < 8);
        shifter.request.put (tuple2 (8'h01, truncate (rg_y))); // or rg_y[2:0]
        rg_y <= rg_y + 1;
    endrule

    rule rl_drain;
        let z <- shifter.get_z.get ();
        $display ("Output = %8b", z);
        if (z == 8'h80) $finish ();           // 8'b10000000
    endrule
endmodule: mkTestbench
```

*rl\_gen* sends in the following inputs:

00000001	0
00000001	1
00000001	2
...	
00000001	7

*rl\_drain* should show the following outputs:

00000001
00000010
00000100
...
10000000

*(The same testbench will be used for all three versions of the shifter)*

# Build and run, using the iterative shifter

- In the “src\_BSV” directory, create a symbolic link from “Shifter.bsv” to the variation of interest:

```
% ln -s -f Shifter_iterative.bsv Shifter.bsv
```

- In the upper directory (Eg04a\_MicroArchs or Eg04b\_MicroArchs), build and run either using BDW or the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier
- Verify that the program produces the expected output
- The \$displays on the input and output also print the clock cycle on which each input and output is done
  - Observe that after a start-up transient, input and output occur every 3 cycles. Why?

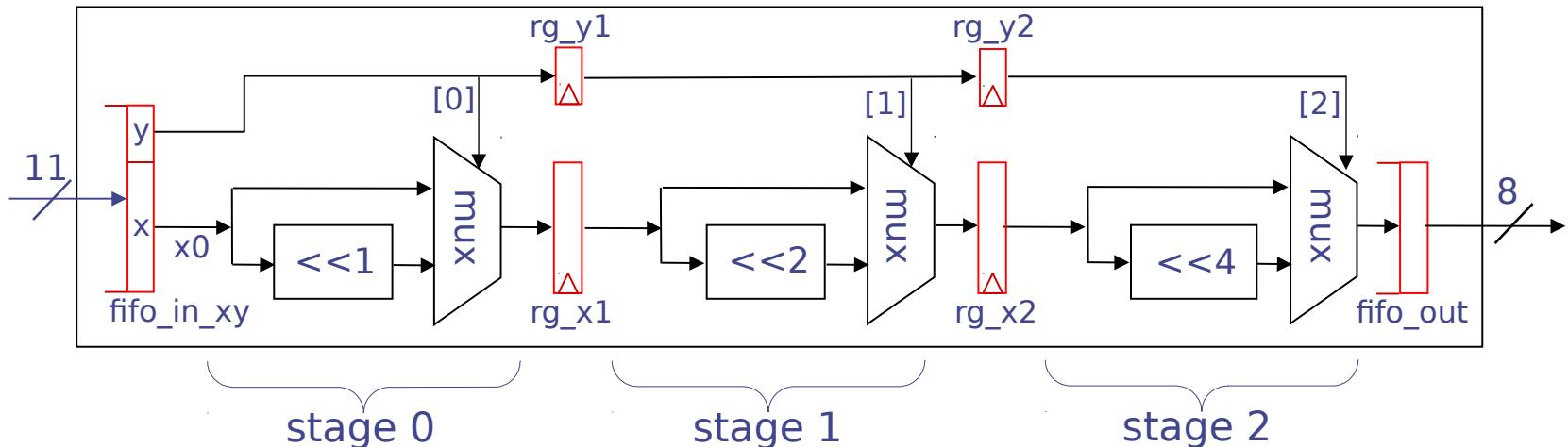
# Time-out to reinforce some concepts

Please study the lecture: Lec\_Interfaces\_TLM  
to understand the concepts Get, Put, Client and Server interfaces, and the mkConnection module.

Please also look at Section 10, “Pattern Matching” in the Reference Guide for more information on the “match” construct.

# “Rigid” pipelined shifter

File: Eg04a\_MicroArchs/src\_BSV/Shifter\_pipe\_rigid.bsv



```
rule rl_all_together;
    // Stage 0
    match { .x0, .y0 } = fifo_in_xy.first; fifo_in_xy.deq;
    rg_x1 <= ((y0[0] == 0) ? x0 : (x0 << 1));
    rg_y1 <= y0;

    // Stage 1
    rg_x2 <= ((rg_y1[1] == 0) ? rg_x1 : (rg_x1 << 2));
    rg_y2 <= rg_y1;

    // Stage 2
    fifo_out_z.enq (((rg_y2[2] == 0) ? rg_x2 : (rg_x2 << 4)));
endrule
```

# Build and run, using the rigid pipelined shifter

- In the “src\_BSV” directory, create a symbolic link from “Shifter.bsv” to the variation of interest:

```
% ln -s -f Shifter_pipe_rigid.bsv Shifter.bsv
```

- In the upper directory (Eg04a\_MicroArchs or Eg04b\_MicroArchs), build and run either using BDW or the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier
- Verify that it is pipelined, i.e., that input and output happen on every clock
- However, the output does not seem to be fully correct:

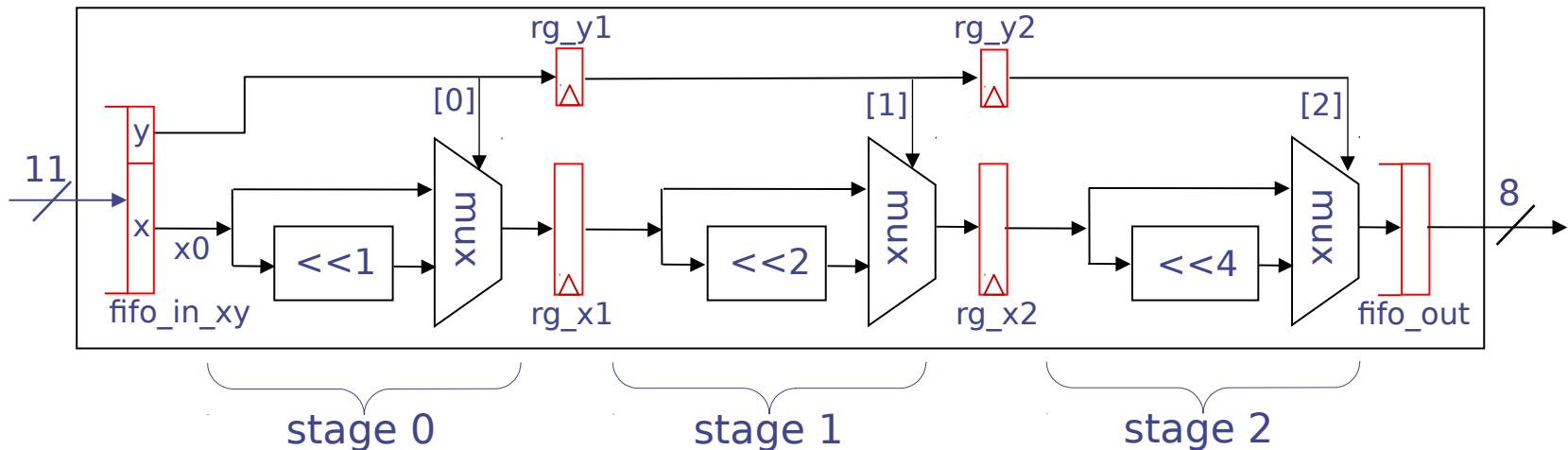
01010101  
10101000  
00000001  
00000010  
00000100  
00001000  
00010000  
00100000



Why?

*... and then the program hangs*

# “Stranding” in the rigid shifter



Actually output is:

01010101  
10101000  
00000001  
00000010  
...  
00010000  
00100000

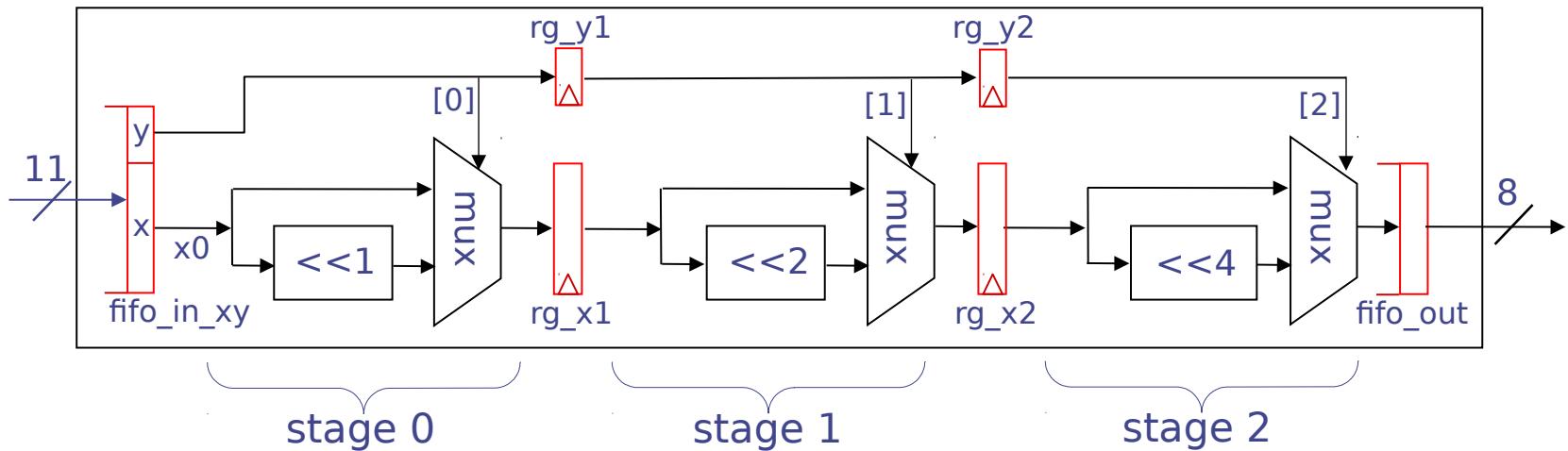
*... and then the program hangs*

The first two outputs are just based on the initial unspecified values in rg\_x1, rg\_y1, rg\_x2 and rg\_y2, as they are pushed through the pipeline.  
bsc usually uses ‘hA...A (10101010...1010) for initial values of unspecified state.

The remaining outputs are the correct outputs, but:

- when rl\_gen stops feeding the input fifos,
- rl\_all\_together can no longer fire (since it invokes fifo\_in\_x.first whose method condition will be false)
- rl\_drain can no longer fire if fifo\_out is empty
- and so the last two values are “stranded” in rg\_x1 and rg\_x2

# Observations about the rigid shifter

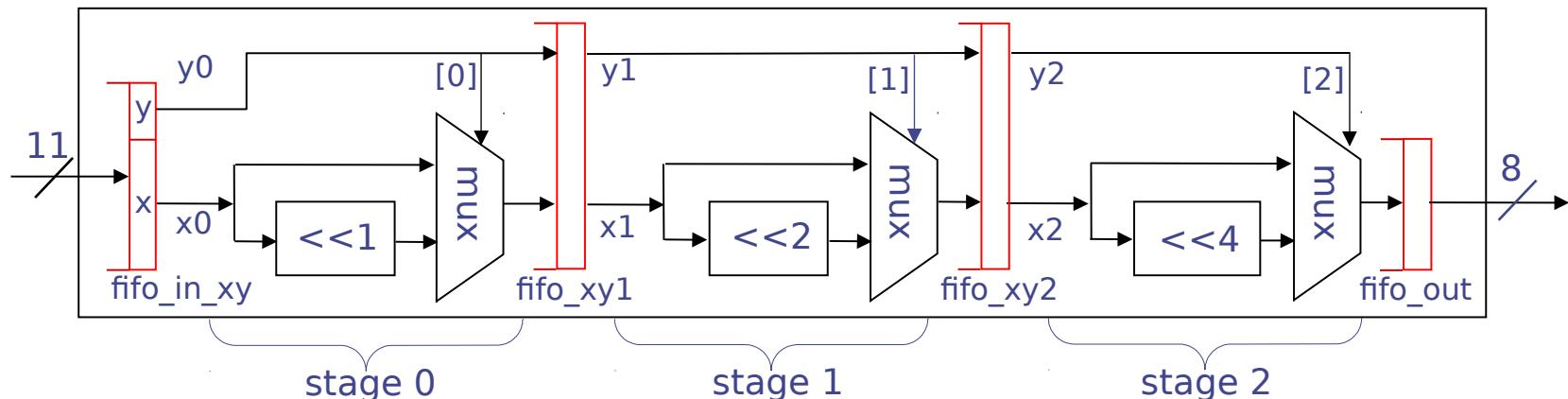


The “rigid” shifter is an example of a simple pipeline implementation that is ok if we have a non-stop, continuous stream of data with no gaps/bubbles. We also call such pipelines “synchronous”, or “lock-step”.

The key observation about the program structure is that all the actions in the pipe are expected to be simultaneous, and therefore placed in a single rule.

# Elastic, pipelined shifter

File: Eg04a\_MicroArchs/src\_BSV/Shifter\_pipe\_elastic.bsv



```
module mkShifter (Shifter_Ifc);
  ...
  FIFOF #(Tuple2 #(Bit #(8), Bit #(3))) fifo_xy1 <- mkFIFO();
  FIFOF #(Tuple2 #(Bit #(8), Bit #(3))) fifo_xy2 <- mkFIFO();

  rule rl_stage0;
    match { .x0, .y0 } = fifo_in_xy.first;  fifo_in_xy.deq;
    fifo_xy1.enq (tuple2 (((y0[0] == 0) ? x0 : (x0 << 1)), y0));
  endrule

  rule rl_stage1;
    match { .x1, .y1 } = fifo_xy1.first;  fifo_xy1.deq;
    fifo_xy2.enq (tuple2 (((y1[1] == 0) ? x1 : (x1 << 2)), y1));
  endrule

  rule rl_stage2;
    match { .x2, .y2 } = fifo_xy2.first;  fifo_xy2.deq;
    fifo_out_z.enq ((y2[2] == 0) ? x2 : (x2 << 4));
  endrule
  ...
endmodule
```

We now have a separate rule for each stage.

Each rule can independently fire if its condition is true.

# Build and run, using the elastic pipelined shifter

- In the “src\_BSV” directory, create a symbolic link from “Shifter.bsv” to the variation of interest:

```
% ln -s -f Shifter_pipe_elastic.bsv Shifter.bsv
```

- In the upper directory (Eg04a\_MicroArchs or Eg04b\_MicroArchs), build and run either using BDW or the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier
- Verify that the program produces the expected output
- Verify that it is pipelined, i.e., that input and output happen on every clock

# Summarizing what we've seen so far

- Iterative, rigid-pipelined, and elastic-pipelined structures are three examples of micro-architectural choices for the user.
  - They have different characteristics: area, clock speed, throughput, energy consumption, ...
  - Which one is “best” depends on your design objectives.
- BSV does not (and should not!) make these choices for the user.
  - In creating software, algorithm design is best done by humans (not by programming languages).
  - Similarly, in creating hardware, architectural design is best done humans (not hardware design languages).
  - In both cases, languages can only facilitate quick and reliable expression of the choice made by the human designer, together with efficient implementation.

## Generalizing the dynamic shifters for arbitrary bit-width

Please change directories  
from Eg04a\_MicroArchs/  
to Eg04b\_MicroArchs/

# Time-out to reinforce some concepts

Before moving on with the examples, please study the lecture: [Lec\\_Types](#) to understand the concepts behind types, polymorphism, and numeric types.

Please also *skim* through Section C.3 (“Vectors”) in the Reference Guide.

# Generalized interface for the shifters

All three variations of the shifter have the same interface (see file Shifter\_IFC.bsv):

```
typedef Server #(Tuple2 #(Bit #(n), Bit #(TLog #(n))),  
                  Bit #(n))  
    Shifter_IFC #(numeric type n);
```

The interface is now parameterized by ‘n’, the bit-width of the x input.

The bit-width of the y input is constrained to  $\log(n)$ , in order to express any shift amount from 0 to n.

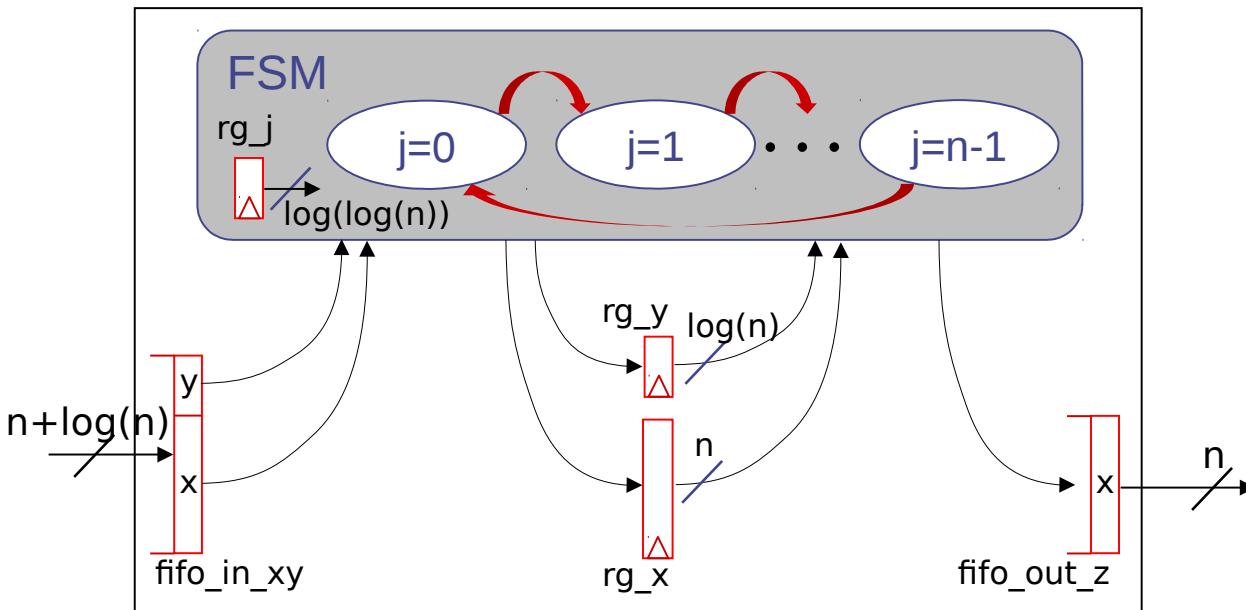
In BSV, certain types and type parameters can be *numeric types*.

Note that *numeric types* (which are only meaningful at compile time) are distinct from *numeric values* (which can of course occur at run time). This is why we use the special notation “TLog#(n)” to express a computation in numeric types.

Strictly speaking, TLog#(n) represents the ceiling of the  $\log(n)$ , i.e., an integer number of bits adequate to hold the binary representation of n.

# Sequential, iterative n-bit shifter

File: Eg04b\_MicroArchs/src\_BSV/Shifter\_iterative.bsv

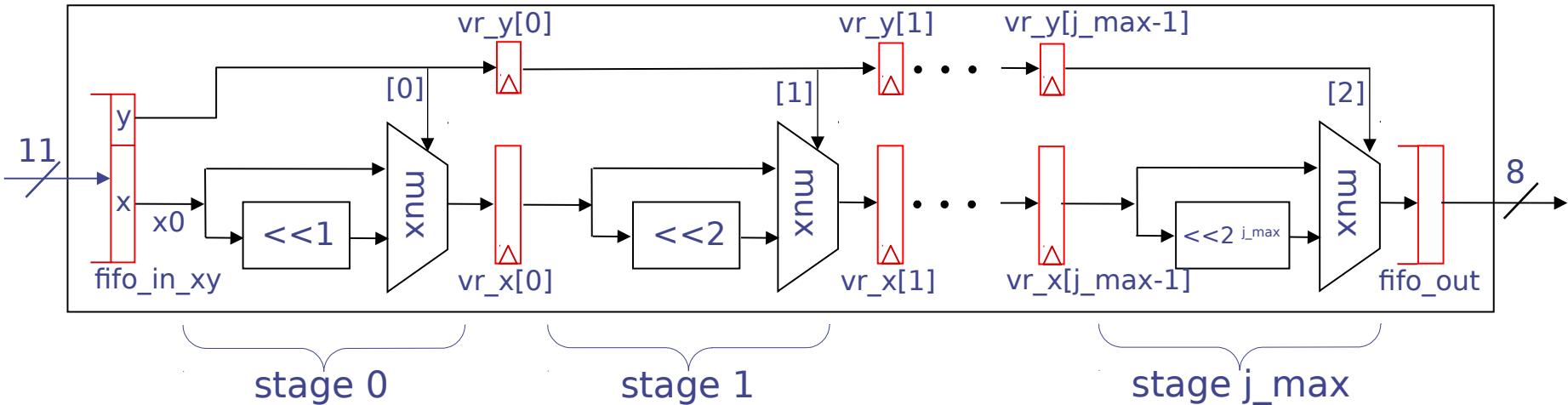


- The FSM (and its actions) is implemented using  $n$  BSV rules
- The rules corresponding to  $j=1..(n-2)$  are generated in a for-loop
- $rg_j$  is  $\log(\log(n))$  bits wide, since it selects bits  $0..\log(n)-1$  in  $y$
- Note: `rg_x << (2**j)` is not a dynamic shift:  $2^{**j}$  is compile-time constant
- BSV is very strict in type-checking—there are no automatic conversions  
The code uses these explicit conversions:



# “Rigid” pipelined n-bit shifter

File: Eg04b\_MicroArchs/src\_BSV/Shifter\_pipe\_rigid.bsv



- The  $x$  and  $y$  registers are generalized to vectors of  $\log(n)-1$  registers:

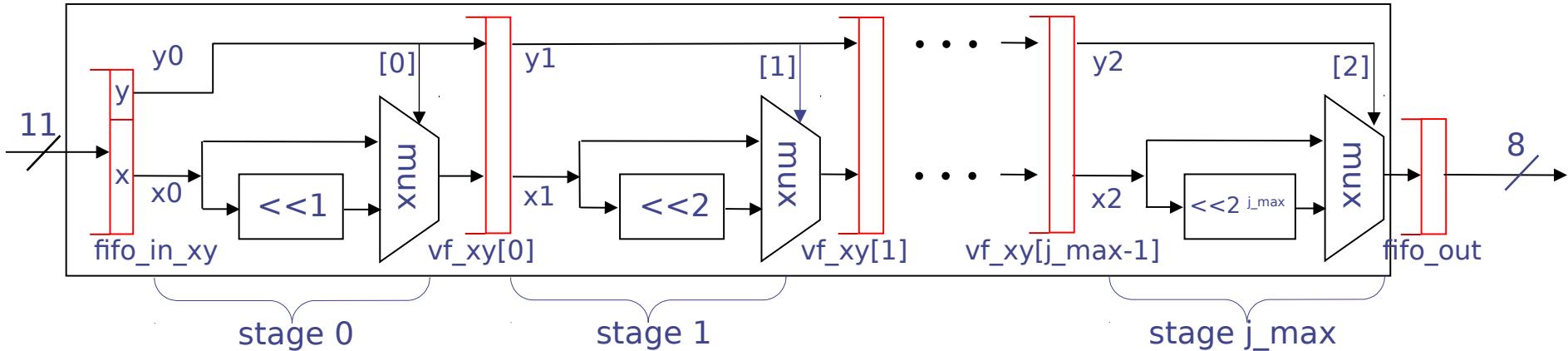
```
Vector #(TSub #(TLog #(n), 1), Reg #(Bit #(n)))           vr_x <- replicateM (mkRegU);  
Vector #(TSub #(TLog #(n), 1), Reg #(Bit #(TLog #(n))))  vr_y <- replicateM (mkRegU);
```

- The Actions of stages 1.. $j_{\max}-1$  are generated using a for-loop:

```
rule rl_all_together;  
    // Stage 0  
    ...  
    // Stage j: 1..j_max-1  
    vr_x[j] <= ((vr_y[j-1][j] == 0) ? vr_x[j-1]: (vr_x[j-1] << (2**j)));  
    vr_y[j] <= vr_y[j-1];  
  
    // Stage j_max  
    ...  
endrule
```

# Elastic, pipelined n-bit shifter

File: Eg04b\_MicroArchs/src\_BSV/Shifter\_pipe\_elastic.bsv



- The  $xy$  FIFOs are generalized to *vectors* of  $\log(n)-1$  FIFOs:

```
Vector #(TSub #(TLog #(n), 1),
           FIFOF #(Tuple2 #(Bit #(n),
                           Bit #(TLog #(n)))))) vf_xy <- replicateM (mkFIFOF);
```

- The Rules for stages  $1..j_{\max}-1$  are generated using a for-loop:

```
for (Integer j = 1; j < j_max; j = j + 1)
  rule r1_j;
    match { .x1, .y1 } = vf_xy[j-1].first; vf_xy[j-1].deq;
    vf_xy[j].enq (tuple2 ((y1[j] == 0) ? x1 : (x1 << (2**j))), y1));
  endrule
```

# Synthesis hierarchy

- In the original 8-bit shifters (in Eg04a\_MicroArchs/ directory), in front of each ‘module’ line, there is a (\*synthesize\*) attribute:

```
(* synthesize *)
module mkShifter (Shifter_IFC);
...
endmodule
```

- When generating Verilog, this creates a ‘mkShifter.v’ file with a ‘mkShifter’ Verilog module
- In the generalized n-bit shifters (in Eg04b\_MicroArchs/ directory), these (\*synthesize\*) attribute are removed.
- This is because BSV cannot separately synthesize *polymorphic* modules; they can only be inlined into a parent module
- Instead, in Testbench.bsv, we have created a specific instance of the module (for shifting 16-bit values); since this is no longer polymorphic, it can be separately synthesized. This is the module actually instantiated in the module mkTestbench:

```
(* synthesize *)
module mkShifter_16_4 (Shifter_IFC #(16));
    let m <- mkShifter;
    return m;
endmodule
```

- The above is a common idiom in BSV code, for creating a separately synthesized instance of a polymorphic module

# Build and run the generalized shifters

- Build and run the generalized shifters (in Eg04b\_MicroArchs/ directory) in a similar manner to how you ran the 8-bit shifters (in Eg04a\_MicroArchs/ directory)
- Verify that the behaviors are as expected (including the throughputs)

# Suggested exercises

- Change the program to *rotate* x by n bits, instead of shifting, i.e., instead of losing bits at the MSB end and shifting in zeroes at the LSB end, the MSB bits should be shifted in at the LSB end.
- Change any one of the mkShifter modules so that it takes a *static* boolean parameter such that:
  - If True, we get a circuit that performs left-shifting.
  - If False, we get a circuit that performs left-rotation.
  - (Note: this is a fixed-function circuit, either left-shifting or left-rotation, chosen at compile time.)
- Change any one of the *pipelined* mkShifter modules so that it can perform left- and right-shifting and left- and right-rotation, selected dynamically.
  - Change the interface so that the input is now a 3-tuple, where the new, third component is an “opcode” specifying left/right shift/rotate. Define an enum type for this opcode. Note: successive 3-tuple inputs may carry different opcodes, i.e., different pipeline stages may be performing different operations at the same time.
  - (Note: this circuit is a piece of a full-blown “ALU” for a CPU.)

# Summary

- These variations on a “dynamic shifter” illustrate how BSV can be used to express a range of micro-architectures, from FSMs to rigid/synchronous pipelines to elastic/asynchronous pipelines
- They also show how architectures can be parameterized flexibly



End

```

import POFSet;

typedef POFSet<T> SetT;
typedef SetT::iterator SetIter;
typedef SetT::const_iterator ConstSetIter;

Integer fileDepth = 15;

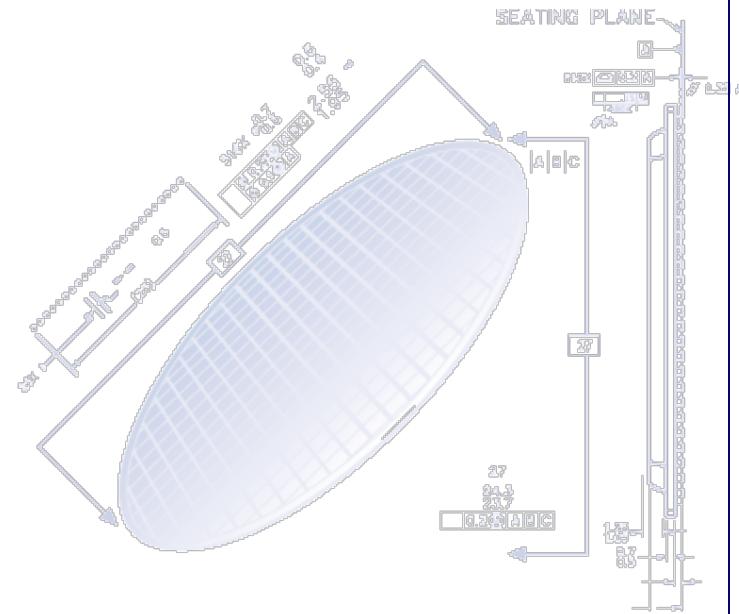
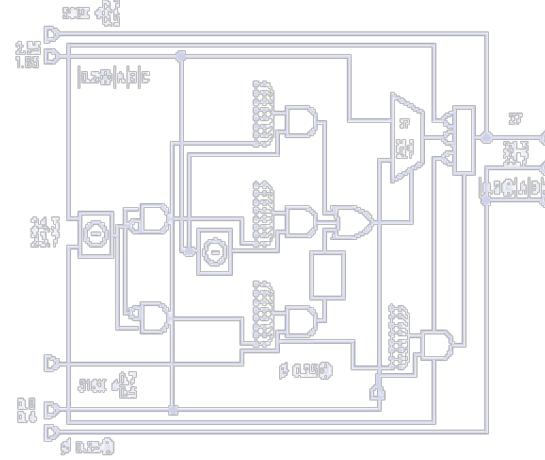
function UInt32 determine_group(DataT&); // returns <0,0>; exit when
exit when

POFSet<DataT> knownS;
DataT POFSet<DataT> the_Lbound();
POFSet<DataT> the_Rbound();
POFSet<DataT> the_Cbound();
POFSet<DataT> the_Ubound();
POFSet<DataT> the_Dbound();

rule end (true);
    DataT b_file = bknownS.first;
    POFSet<DataT> cut_group = 
        determine_group(b_file) == 0 ? the_Lbound() : 
        determine_group(b_file) == 1 ? the_Rbound() : 
        determine_group(b_file) == 2 ? the_Cbound() : 
        determine_group(b_file) == 3 ? the_Ubound() : 
        determine_group(b_file) == 4 ? the_Dbound() : 
        exit();
endrule

endpackage : ex_1st_cnf_1a

```





# BSV Training

## Eg05: Concurrent Registers (CRegs) and Greater Concurrency

Using two examples (a 2-port counter and a FIFO implementation), we see that CRegs permit greater concurrency than traditional registers.

```

import PDFKit;
see that CR
typedoc! Etwi(35) (x) {
  module ex_Url(36) ex_Prop(37);
}

Integer fileLength = 16;

function UInt32() determineLength(Data16);
  return [x[0]];
  extension

PDFOM(Data16) libmain();
PDFOM(Data16) e_main();
PDFOM(Data16) d_main();
PDFOM(Data16) c_main();
PDFOM(Data16) b_main();
PDFOM(Data16) a_main();

FileGrid(17) {
  Data16 file;
  libmain() libfile = libsys();
  PDFOM(Data16) out_main();
  libmain(), queueInFile() == 0 ? out_main() : null;
  libmain(), queueInFile() != 0;
  libmain();
  out_main();
}

endcode: end

```



# Prerequisites and related material

Before you study the examples here in `Eg05_Greater_Concurrency`

you should understand the concept of “concurrency of rules” in BSV as described in:

- Lecture: `Lec_Rule_Semantics`
- Example: `Eg03_Bubble_Sort`

The following lectures:

- Lecture: `Lec_CRegs`
- Lecture: `Lec_RWires`

describe CRegs in greater detail (along with the related topic of RWires)

## Eg05a: A counter with 2 concurrent ports

Each port is an ``increment method'', i.e., allows the counter to incremented.

By ``concurrent ports'' we mean that both methods can be invoked in the same clock.

# The interface for the counter

File src\_BSV/Counter2\_IFC.bsv

```
interface Counter2_IFC;
    method ActionValue #(Int #(32)) count1 (Int #(32) delta);
    method ActionValue #(Int #(32)) count2 (Int #(32) delta);
endinterface
```

- The interface has two methods count1 and count2, with identical types.
- A module implementing this interface should have an internal register representing the current count.
- When either method is invoked, the argument is used to increment the internal counter, and it returns the old value of the counter.

# 1<sup>st</sup> attempt, using ordinary registers

File src\_BSV/Counter2\_Reg.bsv

```
(* synthesize *)
module mkCounter2 (Counter2_IFC);

    Reg #(Int #(32)) rg <- mkReg (0);

    method ActionValue #(Int #(32)) count1 (Int #(32) delta);
        rg <= rg + delta;
        return rg;
    endmethod

    method ActionValue #(Int #(32)) count2 (Int #(32) delta);
        rg <= rg + delta;
        return rg;
    endmethod

endmodule: mkCounter2
```

But:

- count1 and count2 both read and write the register rg
- Because of the “\_read < \_write” ordering constraint on register methods, both methods could not be invoked in the same clock

==> they could never be concurrent

# A testbench for the counter

```
module mkTestbench (Empty);

    Counter2_IFC ctr <- mkCounter2;

    Reg #(int) step <- mkReg (0);

    rule rl_step;
        step <= step + 1;
        if (step == 10) $finish;
    endrule

    rule rl_1 (step <= 6);
        let delta_1 = step + 10;
        let old_v_1 <- ctr.count1 (delta_1);
        $display ("%0d:          rl_1: delta_1 %0d      v_1 %0d", step, delta_1, old_v_1);
    endrule

    rule rl_2 (step >= 4);
        let delta_2 = 5 - step;
        let old_v_2 <- ctr.count2 (delta_2);
        $display ("%0d:          rl_2: delta_2 %0d      v_2 %0d", step, delta_2, old_v_2);
    endrule

endmodule: mkTestbench
```

File src\_BSV/Testbench.bsv

- The testbench runs for 10 steps (see rl\_step).
- rl\_1 attempts to invoke the count1 method on steps 0-6
- rl\_2 attempts to invoke the count2 method on steps 4-10
- Question: what happens on steps 4-6, when both attempt to fire?

# Build and run, using Counter2\_Reg.bsv

- In the “src\_BSV” directory, create a symbolic link from “Counter2.bsv” to Counter2\_Reg.bsv:

```
% ln -s -f Counter2_Reg.bsv Counter2.bsv
```

- In the Build directory, build and run using the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier.
- During compilation, note that bsc produces this warning message concerning the conflict between count1 and count2:

```
Warning: "src_BSV/Testbench.bsv", line 18, column 8: (G0010)
  Rule "rl_1" was treated as more urgent than "rl_2". Conflicts:
    "rl_1" cannot fire before "rl_2": calls to ctr.count1 vs. ctr.count2
    "rl_2" cannot fire before "rl_1": calls to ctr.count2 vs. ctr.count1
```

# Build and run, using Counter2\_Reg.bsv

Simulation output:

```
0: rl_1: delta_1 10 old_v_1 0
1: rl_1: delta_1 11 old_v_1 10
2: rl_1: delta_1 12 old_v_1 21
3: rl_1: delta_1 13 old_v_1 33
4: rl_1: delta_1 14 old_v_1 46
5: rl_1: delta_1 15 old_v_1 60
6: rl_1: delta_1 16 old_v_1 75
7:                               rl_2: delta_2 -2    old_v_2 91
8:                               rl_2: delta_2 -3    old_v_2 89
9:                               rl_2: delta_2 -4    old_v_2 86
10:                             rl_2: delta_2 -5   old_v_2 82
```

Each line shows the step, the rule that fired, the argument to the method call (delta), and the returned previous value of the counter (v).

Observe that in cycles 4-6, only rl\_1 fires.

## 2<sup>nd</sup> attempt: Concurrent Registers

File src\_BSV/Counter2\_CReg.bsv

```
import Counter2_IFC :: *;

(* synthesize *)
module mkCounter2(Counter2_IFC);

    Reg #(Int #(32)) crg [2] <- mkCReg (2, 0);

    method ActionValue #(Int #(32)) count1 (Int #(32) delta);
        crg[0] <= crg[0] + delta;
        return crg[0];
    endmethod

    method ActionValue #(Int #(32)) count2 (Int #(32) delta);
        crg[1] <= crg[1] + delta;
        return crg[1];
    endmethod

endmodule: mkCounter2
```

- The internal register has been replaced by a CReg
- count1 uses port [0] of the CReg
- count2 uses port [1] of the CReg
- The can be invoked concurrently. If invoked concurrently:
  - the counter will be incremented by both delta arguments
  - the ``old value'' returned by count2 will be the value incremented by count1

# Build and run, using Counter2\_CReg.bsv

- In the “src\_BSV” directory, create a symbolic link from “Counter2.bsv” to the Counter2\_CReg.bsv:

```
% ln -s -f Counter2_CReg.bsv Counter2.bsv
```

- In the Build directory, build and run using the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier.

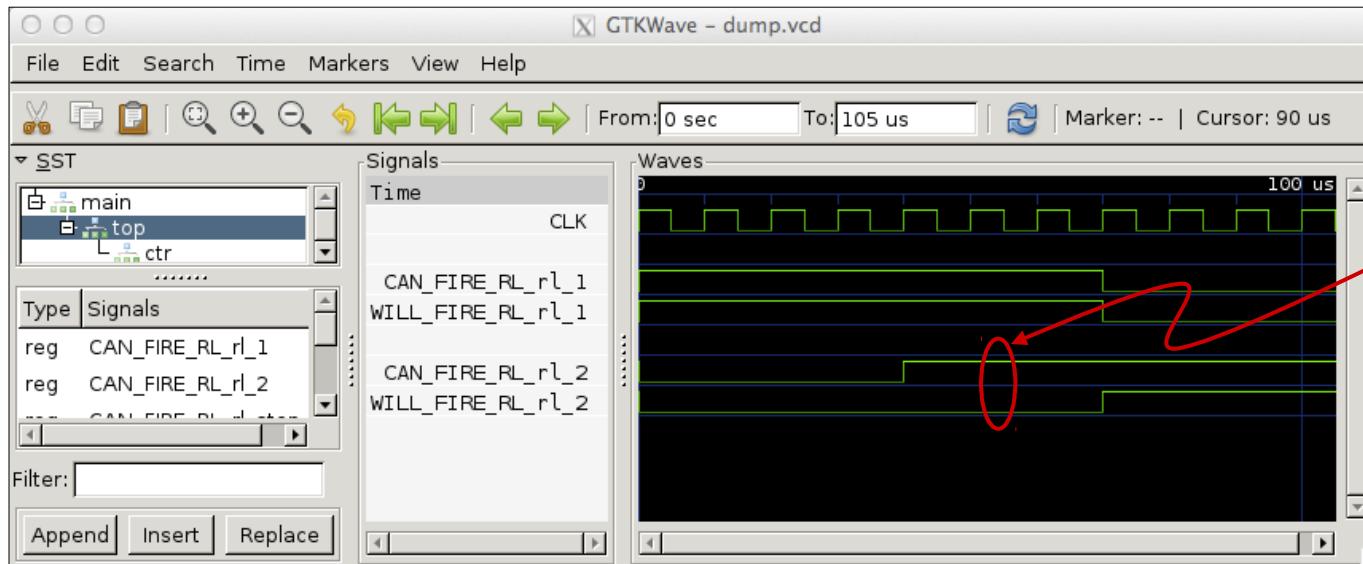
Simulation output:

```
0: rl_1: delta_1 10 old_v_1 0
1: rl_1: delta_1 11 old_v_1 10
2: rl_1: delta_1 12 old_v_1 21
3: rl_1: delta_1 13 old_v_1 33
4: rl_1: delta_1 14 old_v_1 46
4:                               rl_2: delta_2  1   old_v_2 60
5: rl_1: delta_1 15 old_v_1 61
5:                               rl_2: delta_2  0   old_v_2 76
6: rl_1: delta_1 16 old_v_1 76
6:                               rl_2: delta_2 -1  old_v_2 92
7:                               rl_2: delta_2 -2  old_v_2 91
8:                               rl_2: delta_2 -3  old_v_2 89
9:                               rl_2: delta_2 -4  old_v_2 86
10:                              rl_2: delta_2 -5 old_v_2 82
```

Both rules fire in cycles 4-6

# Waveforms

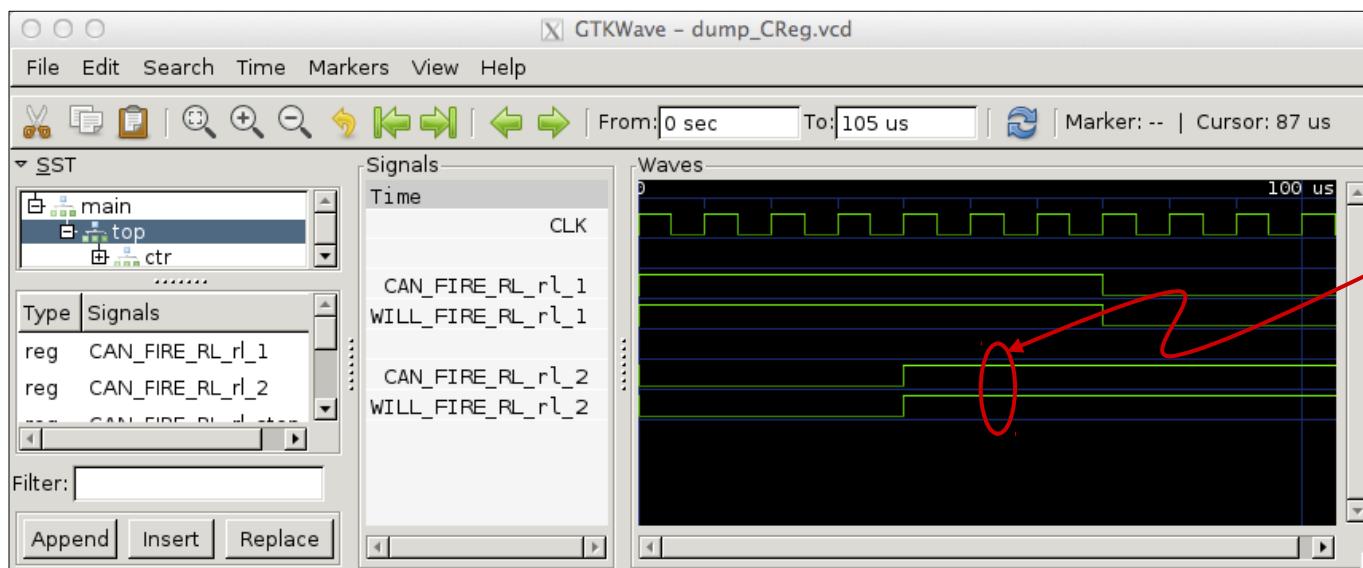
Picture file: Waves\_Reg.tiff



Reg version:

CAN\_FIRE\_rl\_2 is true, but  
WILL\_FIRE\_rl\_2 is false as  
long as WILL\_FIRE\_rl\_1 is  
true, because of the conflict on  
methods count1 and count2

Picture file: Waves\_CReg.tiff



CReg version:

CAN\_FIRE\_rl\_2 is true, and  
WILL\_FIRE\_rl\_2 is also true  
even though WILL\_FIRE\_rl\_1  
is true, because of there the  
methods count1 and count2 do  
not conflict.

# Suggested exercises

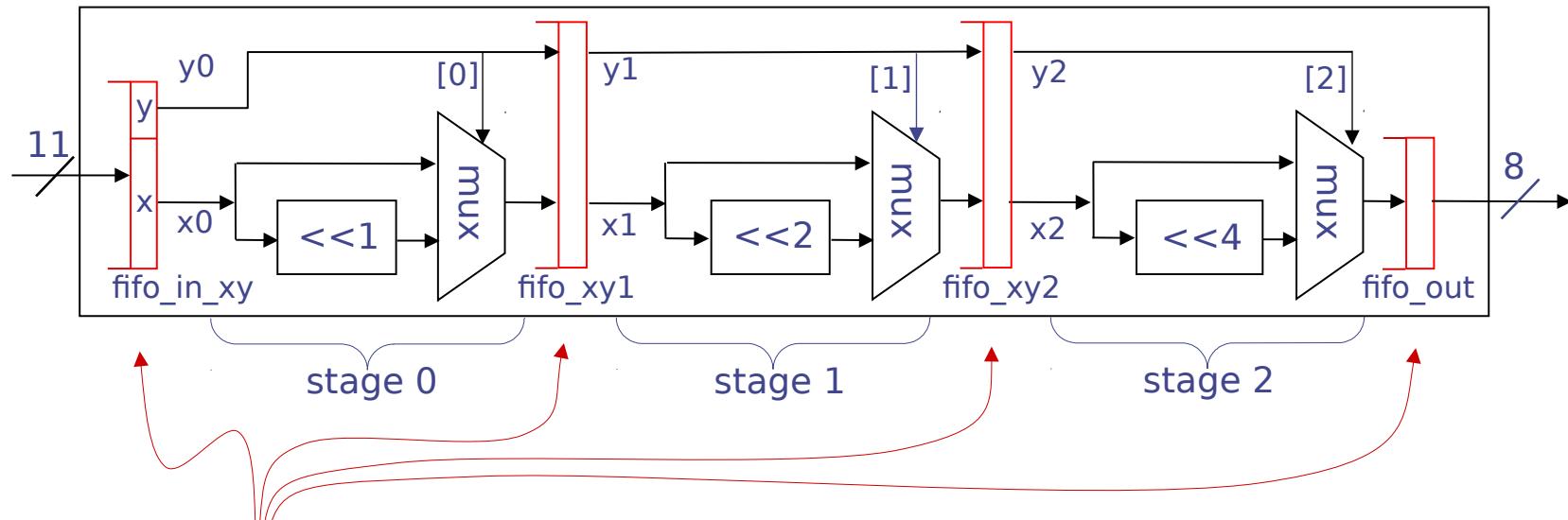
- What is the scheduling order of the two methods count1 and count2?
- Change the CReg port usage to get the opposite schedule for count1 and count2.
- Add a third concurrent count port to the counter.
- Add another method to the counter that is just a value method that returns its net post-increment value, taking into account that any or all of the count methods may be invoked in any clock.

## Example 2: Concurrent FIFOs

By a ``Concurrent FIFO'' we mean a FIFO where the ``enq'' method can be invoked concurrently (in the same clock) as the ``first/deq'' methods .

# Introduction and summary

- We use a copy of `Eg04a_MicroArchs/src_BSV/Shifter_pipe_elastic.bsv` and `Testbench.bsv` as our starting point:



- For the FIFOs in the design, we replace the BSV library `mkFIFO` module with our own `mkMyFIFO` module (implementing a FIFO of depth 1)
- Our first attempt (`MyFIFO_reg.bsv`) will just use traditional registers, and we'll see that it does not have enough concurrency to pipeline properly
- Our second attempt (`MyFIFO_creg.bsv`) will use CRegs instead of traditional registers; these will have enough concurrency to enable proper pipelining
- Takeaway lesson: with registers and CRegs one can implement microarchitectures with any desired degree of concurrency (and therefore performance)

# 1<sup>st</sup> attempt, using ordinary registers

```
module mkMyFIFOF (FIFO #(t))
  provisos (Bits #(t, tsz));
```

File src\_BSV/MyFIFOF\_Reg.bsv

```
  Reg #(t)           rg         <- mkRegU;      // data storage
  Reg #(Bit #(1))   rg_count <- mkReg (0);  // # of items in FIFO (0 or 1)

  method Bool notEmpty = (rg_count == 1);
  method Bool notFull  = (rg_count == 0);

  method Action enq (t x) if (rg_count == 0); // can enq if not full
    rg <= x;
    rg_count <= 1;
  endmethod

  method t first () if (rg_count == 1); // can see first if not empty
    return rg;
  endmethod

  method Action deq () if (rg_count == 1); // can deq if not empty
    rg_count <= 0;
  endmethod

  method Action clear;
    rg_count <= 0;
  endmethod
endmodule
```

But: enq and {first, deq} could never be concurrent, with mutually exclusive conditions:  
rg\_count == 0 and rg\_count == 1

==> enq could never execute in the same clock as {first,deq} (it isn't really a pipeline!)

# A testbench for the pipeline

```
module mkTestbench (Empty);
    Shifter_IFC shifter <- mkShifter;
    Reg #(Bit #(4)) rg_y <- mkReg (0);

    rule rl_gen (rg_y < 8);
        shifter.request.put (tuple2 (8'h01, truncate (rg_y))); // or rg_y[2:0]
        rg_y <= rg_y + 1;
        $display ("%0d: Input 0x0000_0001 %0d", cur_cycle, rg_y);
    endrule

    rule rl_drain;
        let z <- shifter.response.get ();
        $display ("%0d: Output %8b", cur_cycle, z);
        if (z == 8'h80) $finish ();
    endrule
endmodule: mkTestbench
```

File src\_BSV/Testbench.bsv

- This is the same testbench as before (Eg04b).
- Rule rl\_gen continuously attempts to feed the pipeline with the value 8'h01 and increasing shift amounts 0,1,2,...
- Rule rl\_drain continuously attempts to drain the output and displays it.
- If the shifter behaves like a proper pipeline, we should be able to feed and drain it on every cycle.

# Build and run, using MyFIFO\_Reg.bsv

- In the “src\_BSV” directory, create a symbolic link from “MyFIFOF.bsv” to MyFIFO\_Reg.bsv:

```
% ln -s -f MyFIFO_Reg.bsv MyFIFOF.bsv
```

- In the Build directory, build and run using the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier.

## Simulation output:

```
1: Input 0x0000_0001 0
3: Input 0x0000_0001 1
5: Input 0x0000_0001 2
7: Input 0x0000_0001 3
9: Input 0x0000_0001 4
11: Input 0x0000_0001 5
13: Input 0x0000_0001 6
15: Input 0x0000_0001 7
                                5: Output 00000001
                                7: Output 00000010
                                9: Output 00000100
                               11: Output 00001000
                               13: Output 00010000
                               15: Output 00100000
                               17: Output 01000000
                               19: Output 10000000
```

Observe that we can feed inputs into the pipeline and drain outputs from the pipeline only on every other cycle.

## 2<sup>nd</sup> attempt, using CRegs (MyFIFO\_CReg.bsv)

```
module mkMyFIFO (FIFO #(t))
  provisos (Bits #(t, tsz));
```

File src\_BSV/MyFIFO\_CReg.bsv

```
  Reg #(t)          crg [3]      <- mkCRegU (3);      // data storage
  Reg #(Bit #(1))  crg_count [3] <- mkCReg (3, 0);    // # of items in FIFO

  method Bool notEmpty = (crg_count [0] == 1);
  method Bool notFull = (crg_count [1] == 0);

  method Action enq (t x) if (crg_count [1] == 0);
    crg [1] <= x;
    crg_count [1] <= 1;
  endmethod

  method t first () if (crg_count [0] == 1);
    return crg [0];
  endmethod

  method Action deq () if (crg_count [0] == 1);
    crg_count [0] <= 0;
  endmethod

  method Action clear;
    crg_count [2] <= 0;
  endmethod
endmodule
```

Note: {first, deq} access CReg port [0], and enq accesses port [1].

These can run concurrently, in that order, thereby allowing pipelining.

# Build and run, using MyFIFO\_CReg.bsv

- In the “src\_BSV” directory, create a symbolic link from “MyFIFOF.bsv” to MyFIFO\_CReg.bsv:

```
% ln -s -f MyFIFO_CReg.bsv MyFIFOF.bsv
```

- In the Build directory, build and run using the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier.

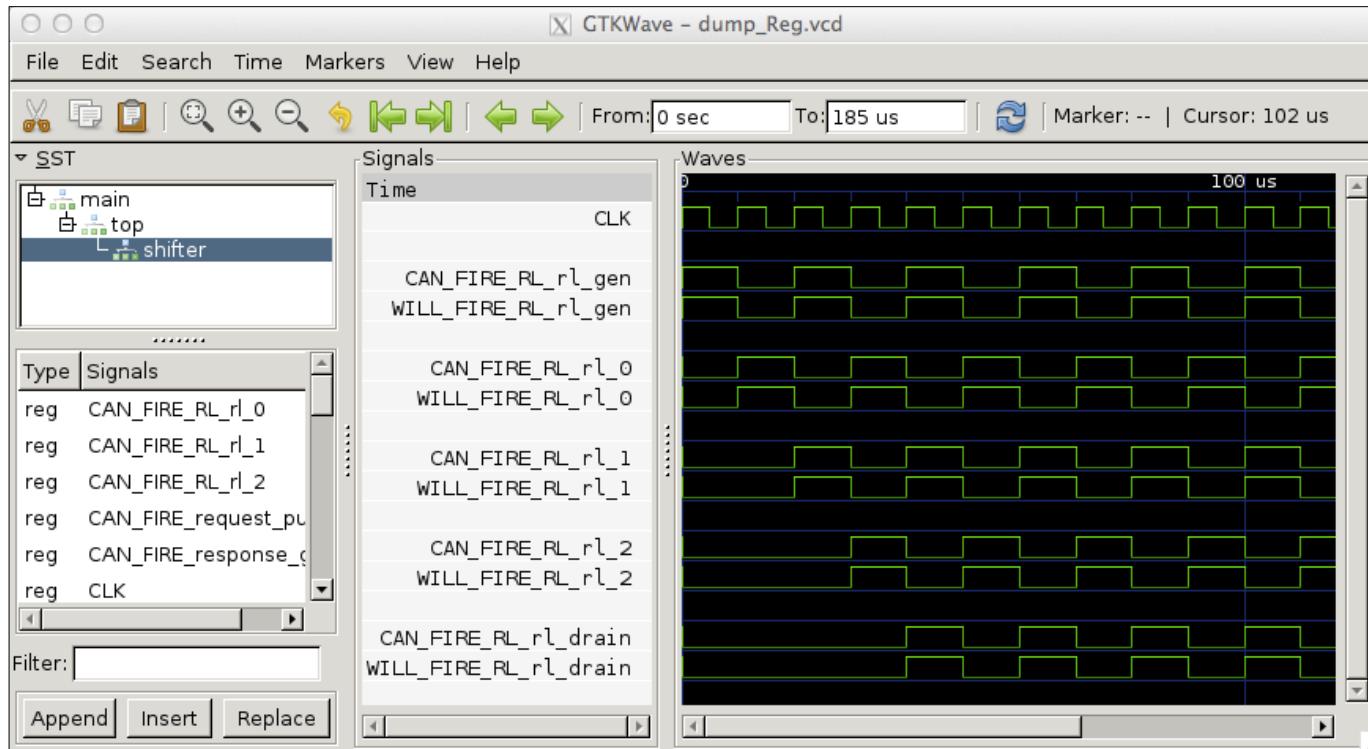
## Simulation output:

```
1: Input 0x0000_0001 0
2: Input 0x0000_0001 1
3: Input 0x0000_0001 2
4: Input 0x0000_0001 3
5: Input 0x0000_0001 4
6: Input 0x0000_0001 5
7: Input 0x0000_0001 6
8: Input 0x0000_0001 7
9: Output 00000001
10: Output 00000010
11: Output 00000100
12: Output 00001000
13: Output 00010000
14: Output 00100000
15: Output 01000000
16: Output 10000000
```

Observe that we can feed inputs into the pipeline and drain outputs from the pipeline on every cycle.

# Waveforms: using MyFIFO\_Reg.bsv

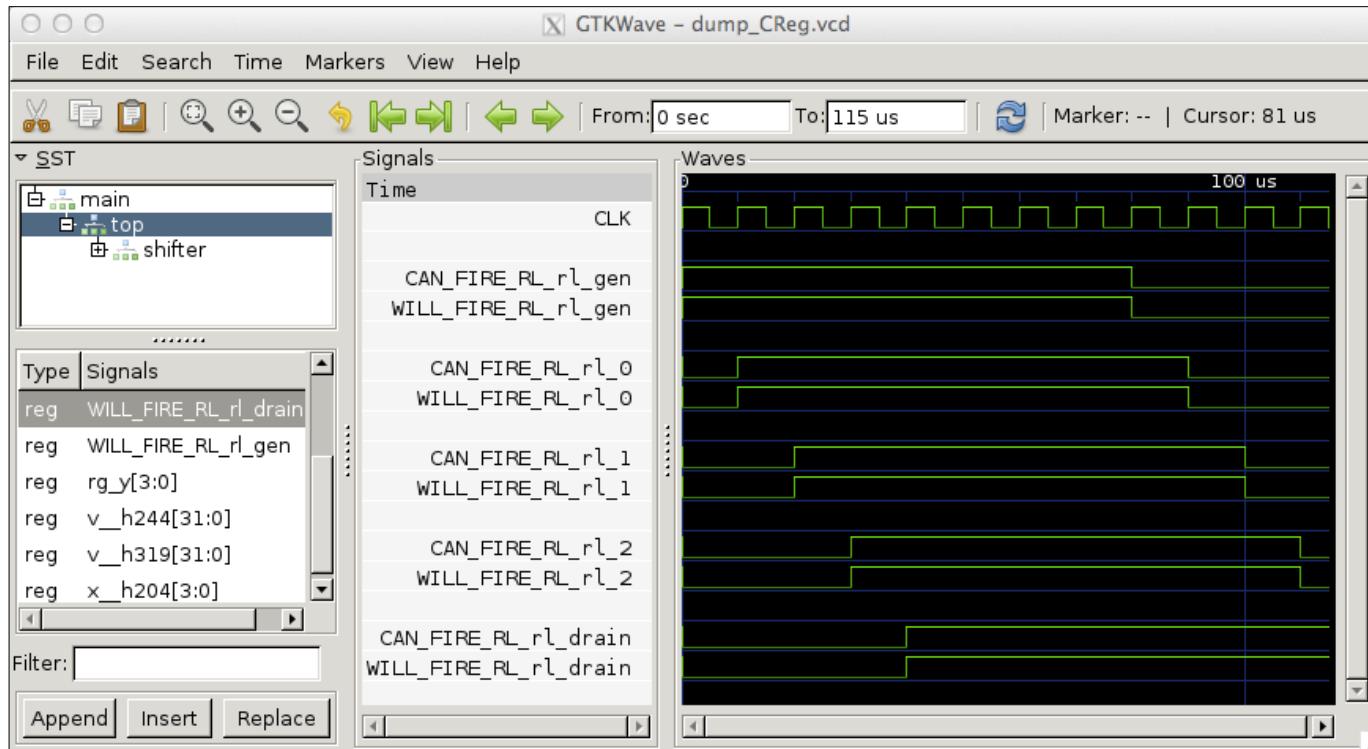
Picture file: Waves\_Reg.tiff



rl\_gen (feeding the pipeline),  
rl\_0, rl\_1 and rl\_2 (stages in the pipeline), and  
rl\_drain (draining the pipeline)  
all can fire and will fire only every other cycle

# Waveforms: using MyFIFO\_CReg.bsv

Picture file: Waves\_CReg.tiff



rl\_gen (feeding the pipeline),  
rl\_0, rl\_1 and rl\_2 (stages in the pipeline), and  
rl\_drain (draining the pipeline)  
all can fire and will fire on other cycle

# Suggested exercises

- Note: the mkMyFIFO module already exists in the BSV library, and is called mkPipelineFIFO (see Lec\_Regs\_and\_RWires, and also Section C.2.2 in the Reference Guide).
- When you use MyFIFO\_reg.bsv:
  - What are the scheduling constraints between the “put” and “get” methods of the shifter?
  - What is the longest combinational path in the circuit?
- When you use MyFIFO\_creg.bsv:
  - What are the scheduling constraints between the “put” and “get” methods of the shifter?
  - What is the longest combinational path in the circuit?
- Lec\_CRegs\_and\_RWires describes another concurrent FIFO: mkBypassFIFO.
  - What is the behavior of the program if you use this FIFO in the Shifter instead?
  - What are the scheduling constraints between the “put” and “get” methods of the shifter?
  - What is the longest combinational path in the circuit?

# Summary

- CRegs enable a tighter scheduling of rules into clocks, i.e., greater concurrency.
- By replacing Regs with CRegs you can fine-tune any micro-architecture to have any desired concurrency
  - With experience, one often pro-actively uses CRegs, anticipating the need for a certain level of concurrency.
- CRegs are semantically “clean” in that they will work with any schedule (if different ports are not used concurrently, it behaves like an ordinary register).



End

```

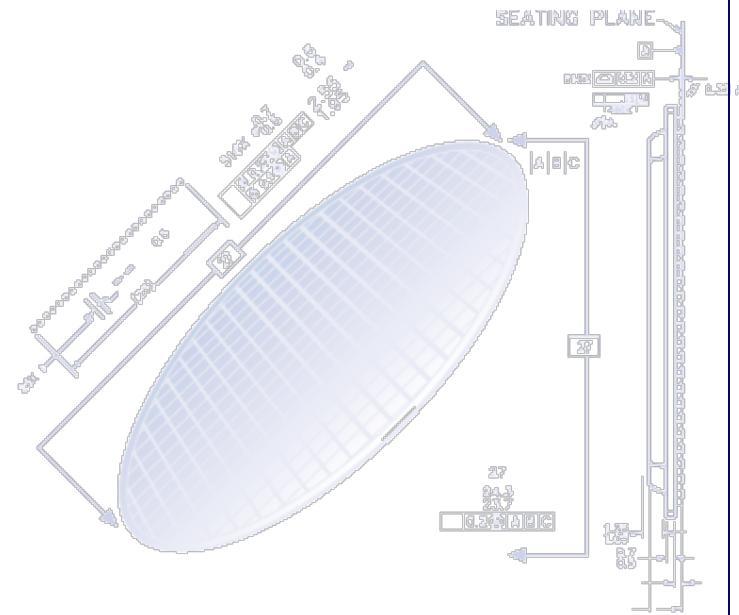
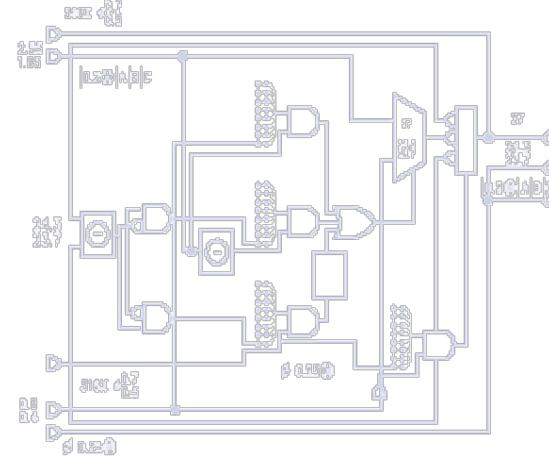
import FIFO#(T);
typedef FIFO#(T) FifoT;
module ext_ifc#(T)(FifoT(fifodepth));
    Integer fifodepth = 16;

    function Bit#(T) determine_psize(Bit#(T)val);
        return {>val[31],<}extension;
    endfunction

    FIFO#(T) mbevnt;
    FIFO#(T) mbevent(fifodepth) the_mbevent(mbevnt);
    FIFO#(T) mbread(fifodepth) the_mbread(mbread);
    FIFO#(T) mbwrite(fifodepth) the_mbwrite(mbwrite);
    FIFO#(T) mbstatus(fifodepth) the_mbstatus(mbstatus);

    action
        if(mbstatus.read() >= 1)
            $display("MB write error");
    endaction
endmodule

```



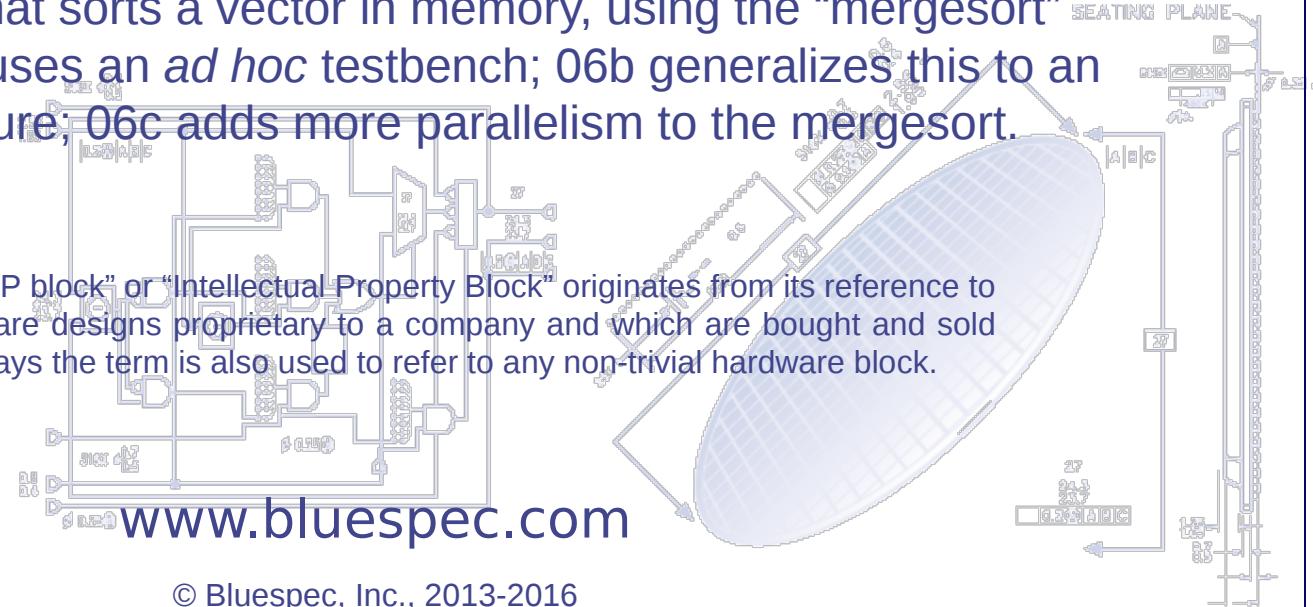


# BSV Training

## Eg06: Mergesort

An IP block\* that sorts a vector in memory, using the “mergesort” algorithm. 06a uses an *ad hoc* testbench; 06b generalizes this to an “SoC” structure; 06c adds more parallelism to the mergesort.

\*Note: the term “IP block” or “Intellectual Property Block” originates from its reference to non-trivial hardware designs proprietary to a company and which are bought and sold as units. Nowadays the term is also used to refer to any non-trivial hardware block.



# Mergesort algorithm

Binary mergesort is a standard sorting algorithm, described in many textbooks and courses on algorithms. The basic idea is illustrated below.

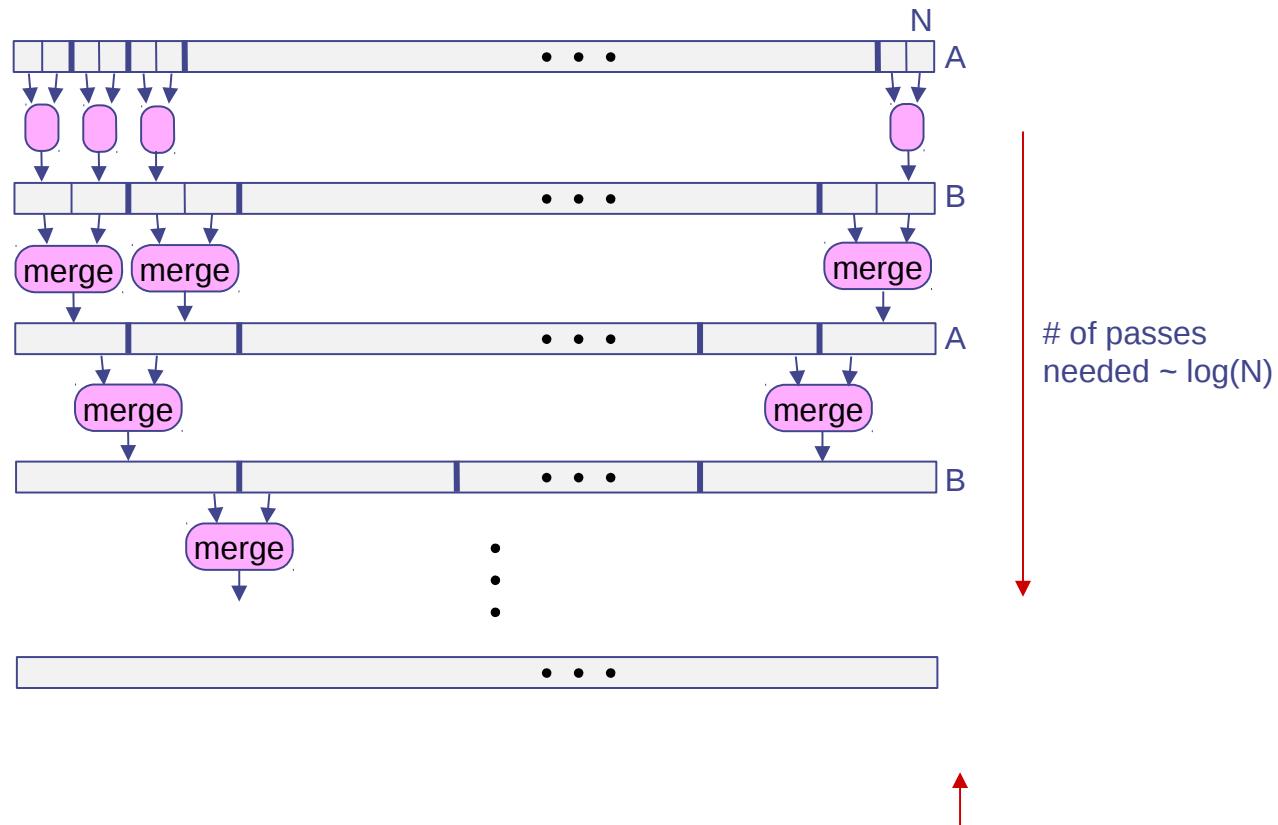
1<sup>st</sup> pass: merge segments of span 1 into segments of span 2

2<sup>nd</sup> pass: merge segments of span 2 into segments of span 4

3<sup>rd</sup> pass: merge segments of span 4 into segments of span 8

•  
•  
•

... and so on, until segment span  $\geq N$ , the length of the array



Note: every segment that is input to “merge” is already sorted

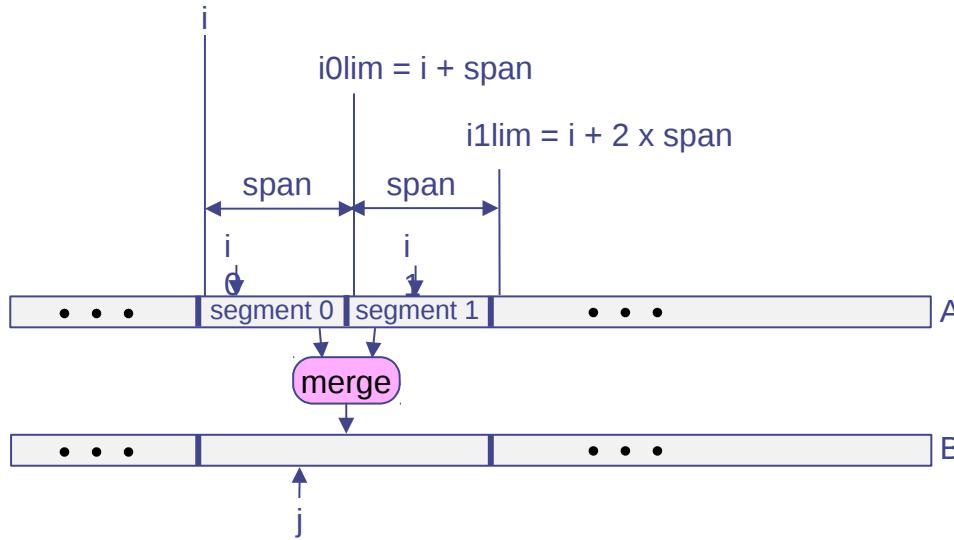
Some edge conditions we need to take care of:

- $N$  is usually not a power of 2, so last two spans may have unequal length
- Depending on  $N$ , the final sorted array may be in B, and so may have to be copied back into the original array A

If we complete one pass before starting the next, we can alternate between two arrays A and B.

# Mergesort algorithm (contd.)

The “merge” step sorts two already-sorted segments of length ‘span’ into a sorted segment of length ‘ $2 \times \text{span}$ ’



```
merge (A,B,i,span) {
    i0lim = i + span; i1lim = i + 2*span;
    j = i0 = i; i1 = i0lim;

    while (j < i1lim) {
        if (i0 >= i0lim)          y = A[i1++]; // segment 0 exhausted; take from segment 1
        else if (i1 >= i1lim)      y = A[i0++]; // segment 1 exhausted; take from segment 0
        else if (A[i0] <= A[i1])  y = A[i0++]; // take from segment 0
        else                      y = A[i1++]; // take from segment 1
        B[j++] = y;
    }
}
```

# Example variations

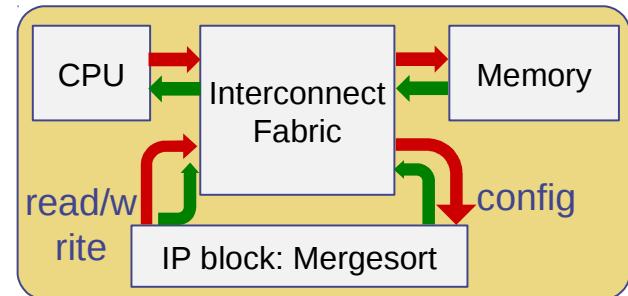
The accompanying code demonstrates three variations:

Eg06a_Mergesort/	Mergesort using a single “merge engine”. Testbench connects this to a single-port memory model.
Eg06b_Mergesort/	Generalizes system structure to an “SoC” containing an “CPU” (here, just an FSM), the mergesort IP module as a programmable “accelerator”, the memory model, and an interconnect fabric.
Eg06c_Mergesort/	Generalizes mergesort model to n parallel merge engines using n memory ports. Generalizes memory model to n memory ports. Handles memory ordering issue.

# 1<sup>st</sup> version: directory Eg06a\_Mergesort/

Rather than create an *ad hoc* interface for our mergesort module, let us prepare it to be ready for plugging into an “SoC” (System on a Chip) as an “accelerator” module, illustrated to the right.

An SoC typically consists of CPUs, memories, an interconnect, and custom IP blocks (“Intellectual Property Blocks”) that perform particular functions for reasons of greater speed (acceleration) and/or less power consumption (compared to executing the same function in software on a CPU).



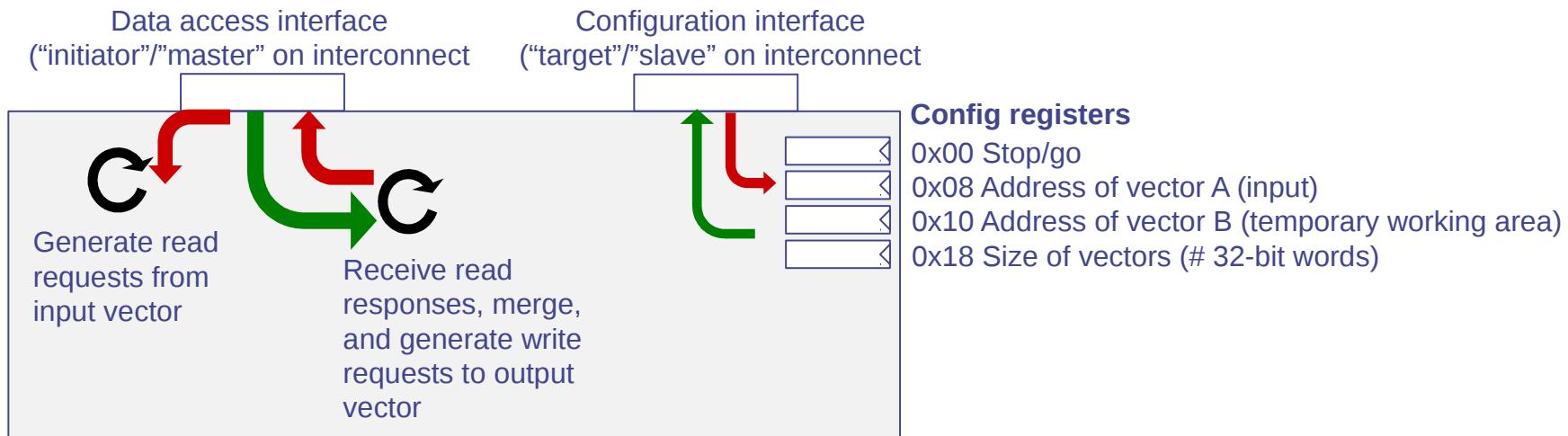
The interconnect fabric carries memory requests (red paths in the figure) and responses (green paths).

- *Initiator* ports (like the CPU port and the IP block read/write port) send requests and receive responses
- *Target* ports (like the Memory port and the IP block config port) receive requests and send responses

Memory requests are routed to the memory or to the IP block config port based on the address contained in the request. I.e., the (usually small number of) configuration registers in the IP block appear, to the CPU, just like memory locations at a particular base address (these addresses are disjoint from addresses serviced by the Memory block). We also say that the config registers are “memory-mapped”.

To operate the IP block, the CPU writes information needed for the operation to the config registers in the IP block, after which the IP block can perform its function (by reading and writing to memory). When the function is completed, the IP block may write a particular value to one of its config registers. The CPU can detect when the IP block has completed its function by “polling” (repeatedly reading) this register.

[In practice, IP blocks can also “interrupt” the CPU on completion; our examples here do not do this.]



To perform the mergesort:

- The external environment must write the addresses and size of the vectors A and B to the config registers at offset 0x08, 0x10 and 0x18, and finally write a “1” (meaning: “start running”) to the config register at offset 0
- The mergesort module then does its work, reading and writing through its data access port; when completed, it writes “0” to the config reg at offset 0
- The external environment can “poll” (repeated read) the config reg at offset 0, to detect completion

# Time-out to reinforce some concepts

Please study the lectures:

- Lec\_Types to review types, which are used to define memory requests and responses
- Lec\_Interfaces\_TLM to review the concepts behind interfaces like Get, Put, Client and Server, which are used for most of the interfaces in this example.
- Lec\_Interfaces\_TLM and Lec\_Typeclasses to review the concepts behind the mkConnection abstraction, which is used in the testbench to connect all components together.
- Lec\_StmtFSM for the concepts behind structured rule-based processes, which are used both in the mergesort module and in the testbench.
- Lec\_Interop\_C for the concepts behind importing C code into BSV, which is used in the memory model in this example.

# Memory requests and responses: Common/Req\_Rsp.bsv

```
typedef enum { READ, WRITE, UNKNOWN } TLMCommand  
deriving (Bits, Eq);  
  
typedef enum { BITS8, ... BITS32, BITS64, ...} TLMBSize  
deriving (Bits, Eq);  
  
typedef struct {  
    TLMCommand      command;  
    Bit #(addr_sz)  addr;  
    Bit #(data_sz)  data;    // Only for write requests  
    TLMBSize        b_size;  
    Bit #(tid_sz)   tid;  
} Req #(type tid_sz, type addr_sz, type data_sz)  
deriving (Bits);
```

Memory requests contain a command (READ/WRITE), an address, data (for WRITE commands), a spec of the size of data being transferred, and a “transaction id” (tid).

```
typedef enum {OKAY, ..., SLVERR, DECERR} TLMStatus  
deriving (Eq, Bits);  
  
typedef struct {  
    TLMCommand      command;  
    Bit #(data_sz)  data;  
    TLMStatus       status;  
    Bit #(tid_sz)   tid;  
} Rsp #(type tid_sz, type data_sz)  
deriving (Bits);
```

Memory responses contain the original command (READ/WRITE), data (for READ commands), a status, and the original “transaction id” (tid).

Transaction Ids (tids) are common on memory requests/responses in modern SoCs, because:

- There may be multiple initiators, and the tid can serve as a “return address” identifying where a response should go
- Responses may be in a different order from the original requests, and the tid can identify the original order

## Specific choices for memory requests and responses in our Mergesort example

Common/Req\_Rsp.bsv contains generic definitions for the types of memory requests and responses.

In particular, they are parameterized by the bit-widths of addresses (addr\_sz), data (data\_sz) and transaction ids (tid\_sz), so that they can be used in various SoCs with various requirements.

In Eg06a\_Mergesort/src\_BSV/Sys\_Configs.bsv, we make particular choices for these parameter for our mergesort example.

```
typedef 64 ASZ;
typedef 64 DSZ;

typedef Bit #(ASZ) Addr;
typedef Bit #(DSZ) Data;

typedef 1 TID_SZ_I;
typedef 1 TID_SZ_T;

typedef Bit #(TID_SZ_I) TID_I;
typedef Bit #(TID_SZ_T) TID_T;

typedef Req #(TID_SZ_I, ASZ, DSZ) Req_I;
typedef Rsp #(TID_SZ_I, DSZ) Rsp_I;

typedef Req #(TID_SZ_T, ASZ, DSZ) Req_T;
typedef Rsp #(TID_SZ_T, DSZ) Rsp_T;
```

Addresses are 64-bits wide

Data are 64-bits wide

Tids are 1-bit wide, both at initiators and targets

Req\_I, Rsp\_I, Req\_T and Rsp\_T are shorthand synonyms for requests and responses at initiators and targets with the specified sizes

## Specific choices for memory-mapping in our Mergesort example

In Eg06a\_Mergesort/src\_BSV/Sys\_Configs.bsv, we also make particular choices for the number of memory ports and the addresses serviced by memory and the config registers

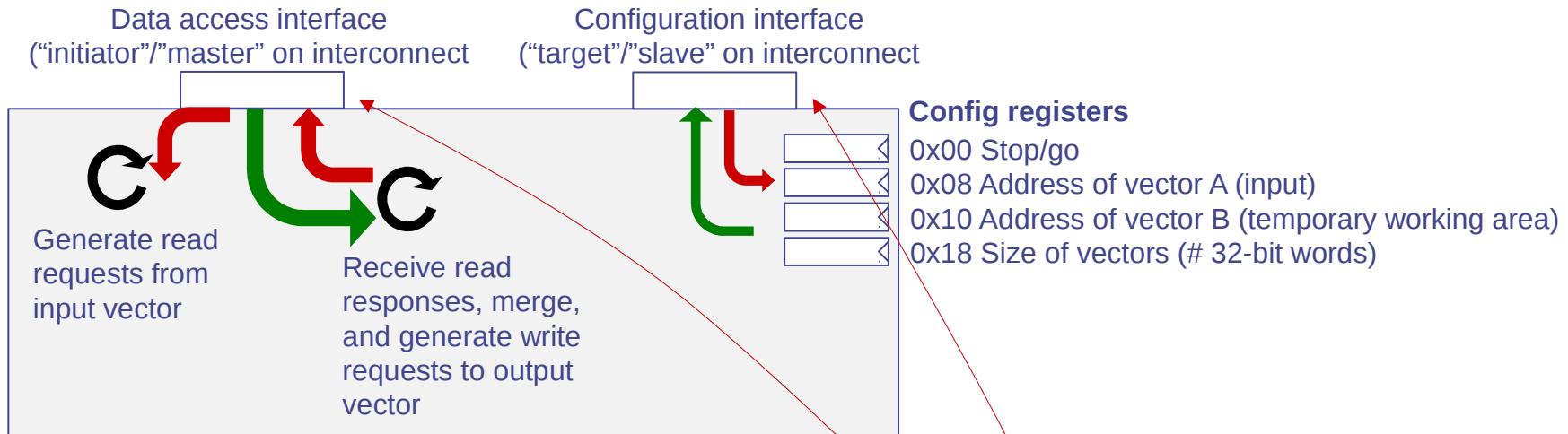
```
// Memory
Addr mem_base_addr = 0;
Addr mem_size      = 'h10_0000;
Addr mem_max_addr  = mem_base_addr + mem_size - 1;

// Accelerator
Addr accel_base_addr = 'h80_0000;
typedef 4 N_Accel_Config_Regs;    // # of config registers
Addr accel_size       = fromInteger (valueOf (N_Accel_Config_Regs) * 8);
Addr accel_max_addr  = accel_base_addr + accel_size - 1;
```

Memory services addresses 0..0x10\_0000-1

The accelerator (mergesort) has 4 config registers, each 8 bytes wide (64b), at base address 0x80\_0000

## Interface for our mergesort module



In file Mergesort.bsv:

```
interface Mergesort_IFC;
    method Action reset (Addr base_addr);
    interface Server #(Req_T, Rsp_T) config_bus_ifc;
    interface Client #(Req_I, Rsp_I) mem_bus_ifc;
endinterface
```

## In Mergesort.bsv: mkMergeSort module structure

```

module mkMergeSort (Mergesort_IFC);

// -----
// Section: Configuration
...
Vector #(N_Config_Regs, Reg #(Data)) vrg_configs;
...
rule rl_handle_configReq;
  ...
endrule

// -----
// Section: Merge sort behavior
...
MergeEngine_IFC mergeEngine <- mkMergeEngine;
...
mkAutoFSM (
  seq
  ...
  endseq);
...

// -----
// INTERFACE
...
interface mem_bus_ifc = mergeEngine.mem_bus_ifc;
endmodule

```

The mergeEngine's memory interface is directly used as the memory interface

Instantiate the configuration registers

This rule receives incoming config requests, reads/writes config regs, sends responses

Instantiate module for the “merge” step

This FSM implements the following pseudo-code:

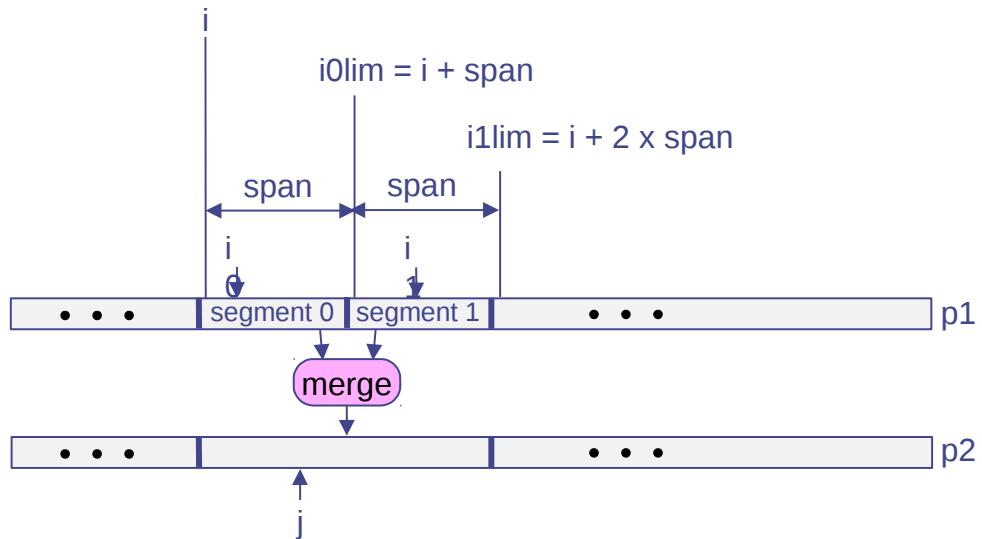
```

while (True)
  wait for 'run' command, init span=1, p1=A, p2=B
  while (span < n) // do another pass:
    i=0;
    while (i < n)
      merge (i, span, p1, p2);
      i += 2*span;
    swap p1,p2; span = 2x span
    if final array is B, copy it back to A
    config reg [0] = 0 (announce completion)

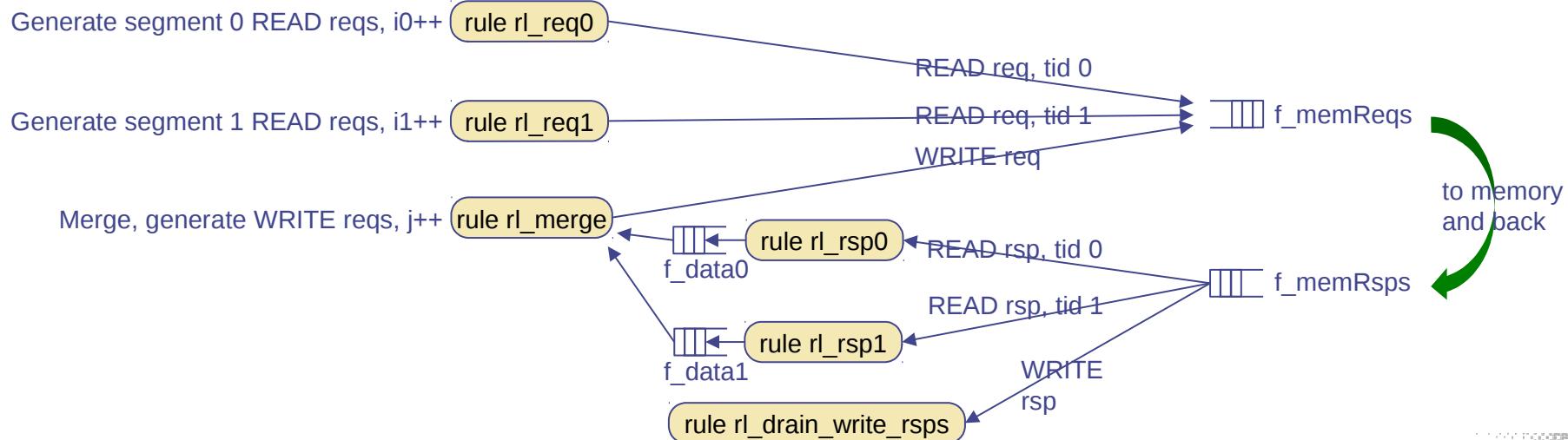
```

In Mergesort.bsv: mkMergeEngine

This module implements the “merge” step which we saw earlier:



### mkMergeEngine module data flow

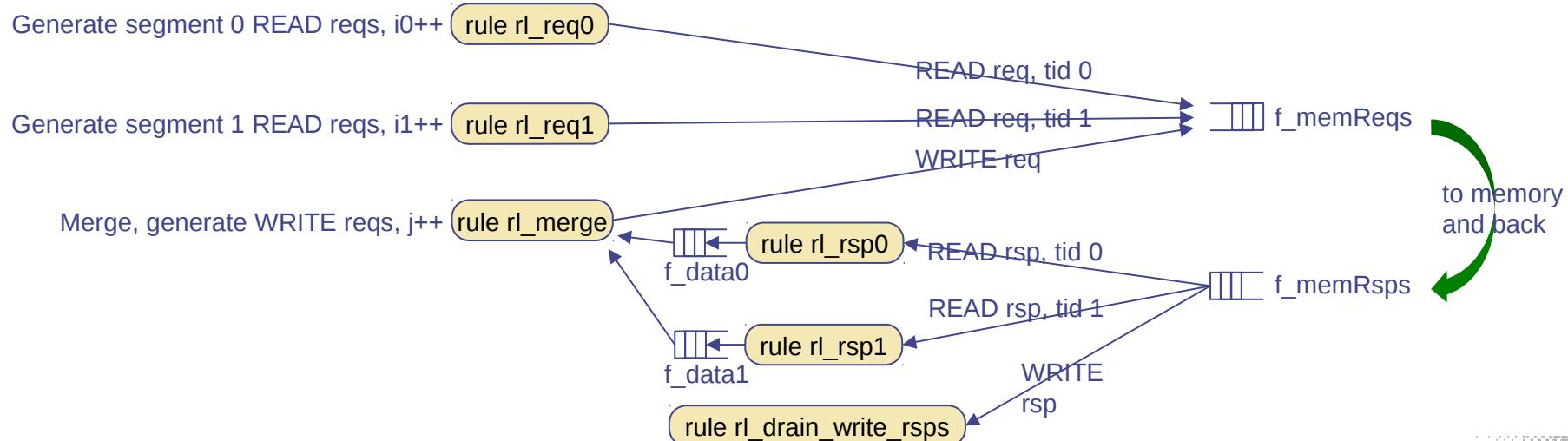


mkMergeEngine is highly concurrent:

- rl\_req0, rl\_req1 and rl\_merge continuously stream requests to memory
- rl\_rsp0, rl\_rsp1 and rl\_drain... continuously handle the stream of responses

This is typical of high-performance accelerators which try to maximize utilization of available memory bandwidth. A software implementation on a CPU may not be able to generate such concurrent, pipelined memory accesses.

### mkMergeEngine module data flow



*There is a danger of deadlock. Example:*

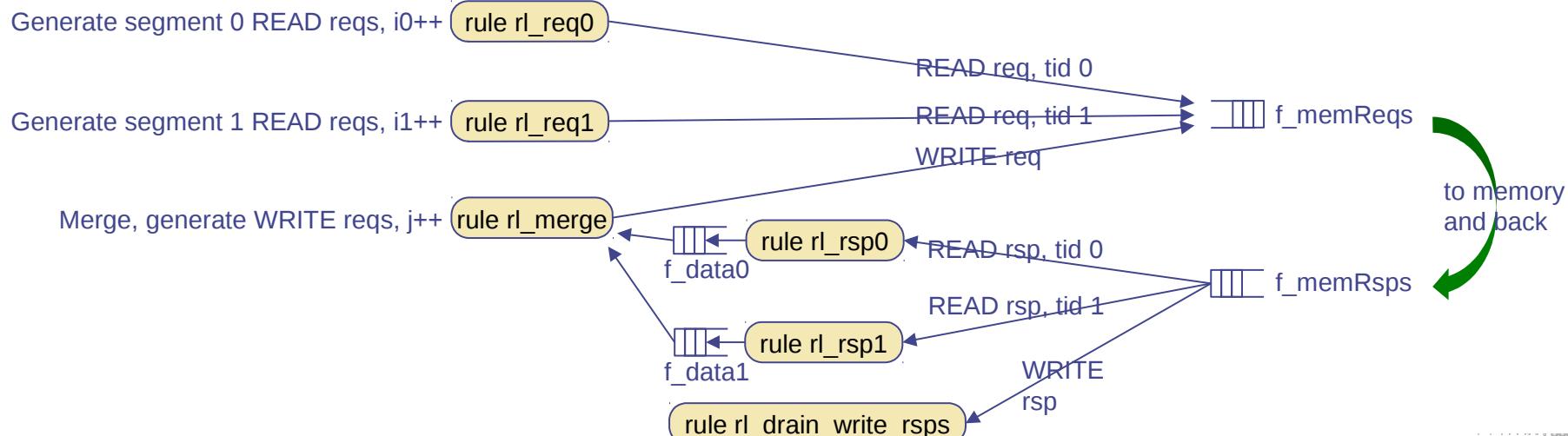
- Suppose rl\_merge does not consume f\_data1, for example because the next segment 1 item is > many segment 0 items
- Then, f\_data1 may become full, and if the first item in f\_memRsp is from segment 1, then we get stuck (the segment 0 items we need may be behind it). This kind of deadlock is called “head-of-line blocking”

*Solution:*

- The code has a parameter: max\_n\_reqs\_in\_flight = 8
- f\_data0 and f\_data1 are sized to accommodate 8 responses.
- rl\_req0 and rl\_rsp0 decrement and increment ehr\_credits0, respectively, to never allow more than 8 requests in flight. rl\_req1 and rl\_rsp1 similarly maintain ehr\_credits1.

This prevents the above deadlock situation.

## mkMergeEngine module data flow



## In Common/Memory\_Model.bsv: a memory model

To test our mergesort block, we need to provide it a memory containing the vector A to be sorted and the vector B for its scratch working area.

Large memories (particularly those implemented in DRAM) are typically not expressed in a hardware design language. Hence we merely use a *model* of memory for testing our IP block in simulation.

This is provided in Common/Memory\_Model.bsv, which is excerpted below:

```
interface Memory_IFC;
    interface Vector #(N_Mem_Ports, Server #(Req_T, Rsp_T)) bus_ifc;
    ...
endinterface

module mkMemory_Model (Memory_IFC);
    ...
endmodule
```

The interface provides N\_Mem\_Ports servers for memory requests. In Eg06a we will only use 1 port, but in Eg06c we will increase this, to model a memory with higher bandwidth.

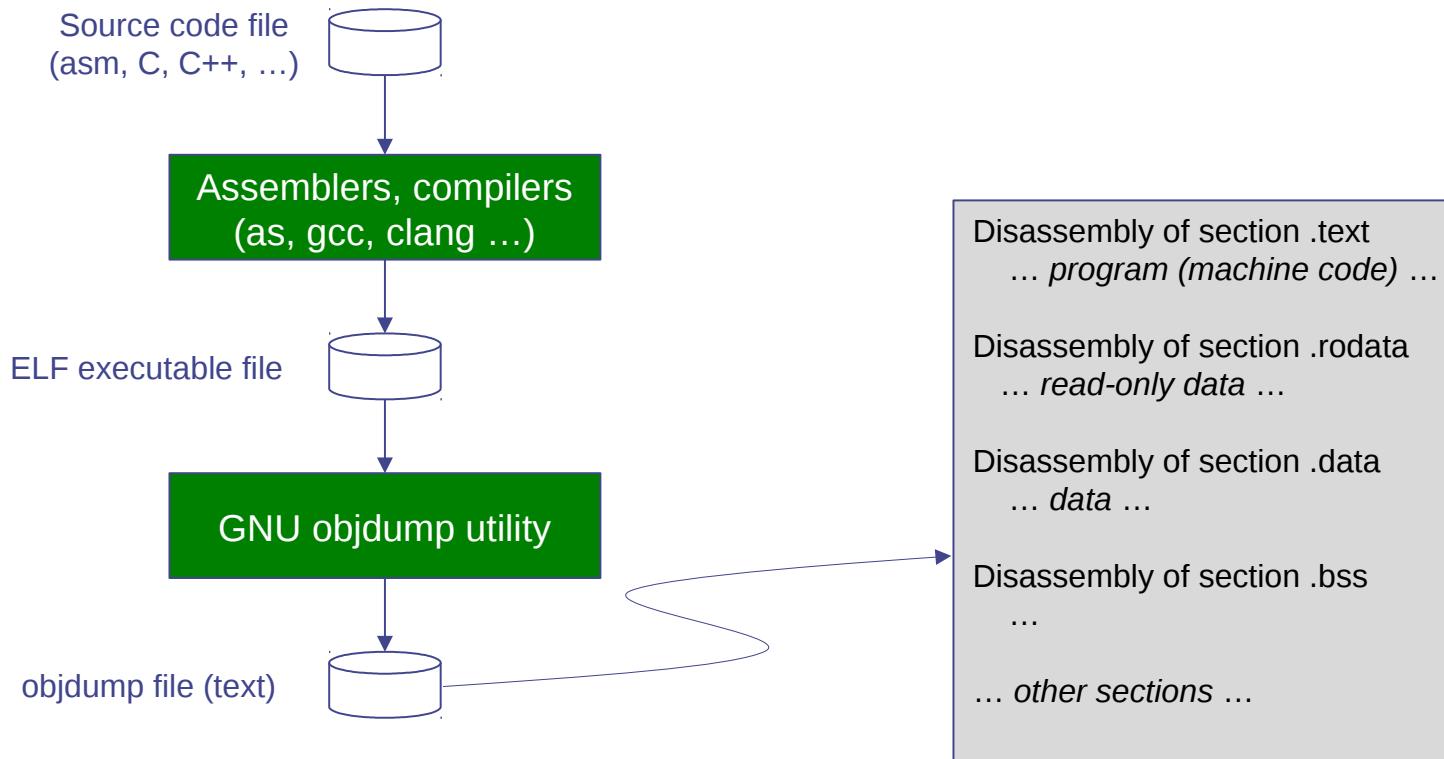
The body of the module mkMemory\_Model is fairly straightforward. It illustrates how we can import a C function to do some work (in simulation only). The associated files are also in Common/:

- C\_imports.{h,c} C functions to ‘malloc’ an array representing memory, and to read/write the array
- C\_import\_decls.bsv BSV declarations to connect BSV to the C functions

# A word about “objdump” files

Our memory model initializes memory by reading in a file called “objdump”.

Objdump files are a standard format on several flavors of Unix (including Linux and OS X)



*Note: We have not included any tools to create objdump files. The standard way is to use GNU tools. The pre-built objdump files distributed with this training were created using some ad hoc tools at Bluespec.*

## In Testbench.bsv: a testbench

Our testbench is excerpted below:

```
module mkTestbench (Empty) ;  
  
    Memory_IFC      mem           <- mkMemory_Model;  
    Mergesort_IFC   mergesort     <- mkMergesort;  
  
    mkConnection (mergesort.mem_bus_ifc, mem.bus_ifc [0]);  
    ...  
    mkAutoFSM (  
        seq  
            mem.initialize (mem_base_addr, mem_size, init_from_file);  
            mergesort.reset (accel_base_addr);  
            dump_mem_range;  
            ... write mergesort's config regs ...  
            ... loop, polling mergesort's config reg for completion ...  
            dump_mem_range;  
        endseq);  
    endmodule
```

The testbench only performs a very small test so that you can easily inspect the outputs:

- Addr of the data array: 0x1000; scratch array: 0x1800; number of elements: 13

The 'dump\_mem\_range' calls (above) show memory contents before and after the sort.

(You are free to edit the program to try larger examples.)

# Build and run the 1<sup>st</sup> version

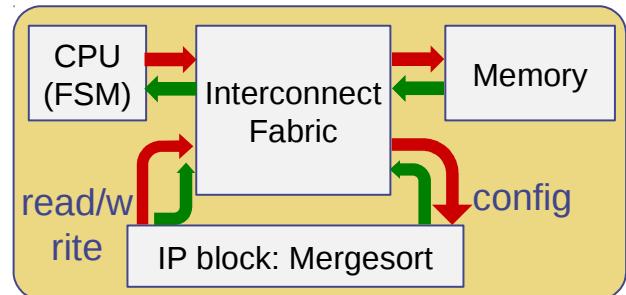
- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier
- Observe the inputs and outputs and verify that they are reasonable (final memory contents are a sorted version of initial memory contents)

## 2<sup>nd</sup> version: directory Eg06b\_Mergesort/

In this version we use exactly the same Mergesort.bsv as in Eg06a.

We only generalize the environment around it into a “SoC” model.

Please study: src\_BSV/Sys\_Configs.bsv  
in this directory, to see the changes to describe this new SoC.



Note that we do some “type-level” arithmetic to derive the number of fabric targets based on the number of memory ports; to derive “initiator numbers” (INums) based on the number of initiators, etc.:

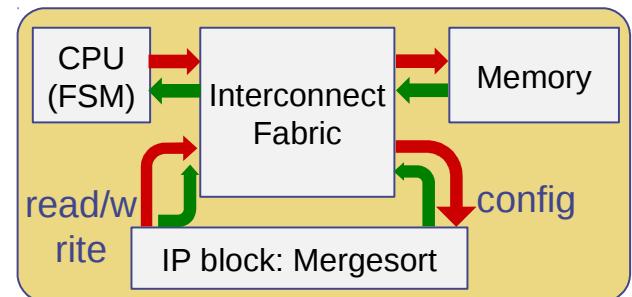
```
typedef 1 N_Mem_Ports;
typedef 2 Max_Initiators;      // CPU Data access, Accelerator data port

typedef TAdd #(N_Mem_Ports, 1) Max_Targets;

typedef TLog #(Max_Initiators) INum_Sz;

typedef Bit #(TLog #(Max_Initiators)) INum;
typedef Bit #(TLog #(Max_Targets)) TNum;
```

## 2<sup>nd</sup> version: directory Eg06b\_Mergesort/



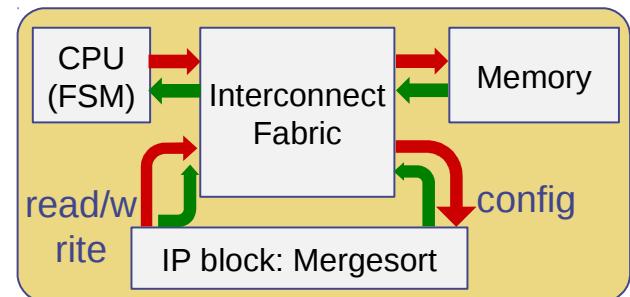
Please study: src\_BSV/Sys\_Configs.bsv  
in this directory, to see the changes to describe this new SoC.

Since we have more than one initiator on the fabric, transaction ids for targets have  $\log(N)$  more bits than transaction ids for initiators, where  $N$  is the number of initiators, because the fabric must tack on  $\log(N)$  bits to remember which initiator must get the corresponding response:

```
typedef 1 TID_SZ_I;  
  
// Transaction ids at targets  
typedef TAdd #(TLog #(Max_Initiators), TID_SZ_I) TID_SZ_T;  
  
typedef Bit #(TID_SZ_I) TID_I;  
typedef Bit #(TID_SZ_T) TID_T;
```

Finally, the end of the file now contains an “address decoder” module, which will be used by the Fabric to route memory requests either to the Memory or to the IP block, depending on the address.

## 2<sup>nd</sup> version: directory Eg06b\_Mergesort/



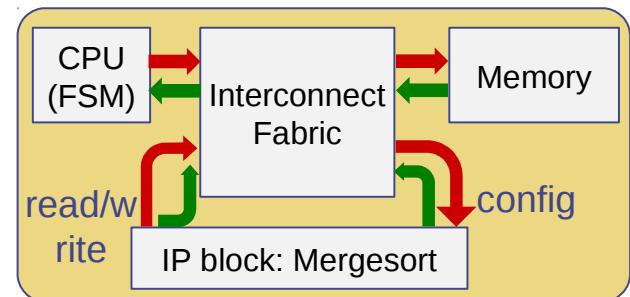
Please study: src\_BSV/CPU.bsv

You will see that it is not really a CPU, but just the testbench FSM from Eg06a, wrapped in a module that pretends to be a CPU. The interface is a step towards a real CPU interface, containing a “data-cache” interface, and other methods that will enable the GDB debugger to control the CPU:

```
interface CPU_IFC;
  // Interface to Data Memory
  interface Client #(Req_I, Rsp_I) dcache_ifc;

  // GDB handling
  method Action run_continue ();
  method CPU_Stop_Reason stop_reason;
  method Action req_read_memW (Addr addr);
  method ActionValue #(Data) rsp_read_memW ();
  method Action write_memW (Addr addr, Data d);
endinterface
```

## 2<sup>nd</sup> version: directory Eg06b\_Mergesort/



Please study: Common/Fabric.bsv

Its interface is just a vector of Servers facing the initiators, and Clients facing the targets:

```
interface Fabric_IFC;
    interface Vector #(Max_Initiators, Server #(Req_I, Rsp_I)) v_servers;
    interface Vector #(Max_Targets, Client #(Req_T, Rsp_T)) v_clients;
endinterface
```

Recall that Req\_I is different from Req\_T, and Rsp\_I is different from Rsp\_T:

- For requests, the fabric will tack on extra transaction id (tid) bits to remember the “return address” where responses should go
- For responses, the fabric will use those extra tid bits to route the responses, and will also strip them off to restore the original tid.

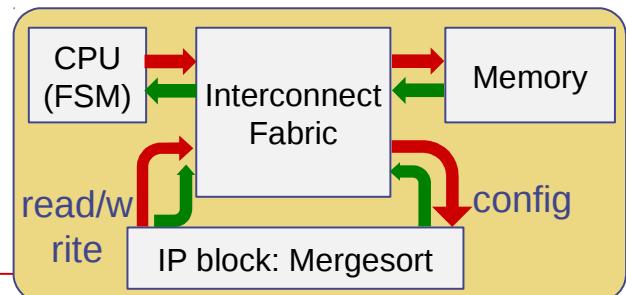
The module mkFabric is quite straightforward. It represents a “full crossbar” switch, i.e., there is a separate datapath from each input port to each output port, in both directions. These are represented by the rules that are generated in for-loops. The only interesting thing in each rule is the tid manipulation as described above.

## 2<sup>nd</sup> version: directory Eg06b\_Mergesort/

Please study: src\_BSV/Testbench.bsv

The system is instantiated with this excerpt:

```
module mkTestbench (Empty) ;  
  
    CPU_IFC          cpu      <- mkCPU_Model;  
    Memory_IFC       mem      <- mkMemory_Model;  
    Fabric_IFC       fabric   <- mkFabric;  
    Mergesort_IFC    mergesort <- mkMergesort;  
  
    mkConnection (cpu.dcache_ifc, fabric.v_servers [cpu_d_iNum]);  
    mkConnection (mergesort.mem_bus_ifc, fabric.v_servers [accel_iNum]);  
    mkConnection (fabric.v_clients [mem_tNum], mem.bus_ifc [0]);  
    mkConnection (fabric.v_clients [accel_tNum], mergesort.config_bus_ifc);
```



The behavior of the testbench (using mkAutoFSM) is a step towards a GDB-like interactive console to control the CPU. It uses an imported C command “c\_console\_get\_command” to prompt the user for a GDB-like command and to return the command entered by the user, and then it executes the command.

In this first version, it recognizes just three commands: “continue” (to execute the program), “quit”, and “memory dump” to show a region of memory.

# Build and run the 2<sup>nd</sup> version

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier
- When you simulate
  - You will initially see some INFO messages from the memory model, loading the objdump file
  - You will see the output of the first ‘dumpmem’, showing memory contents before the sort
  - Then, you will get a GDB-like prompt:

```
Command? [type 'h' for help]:
```

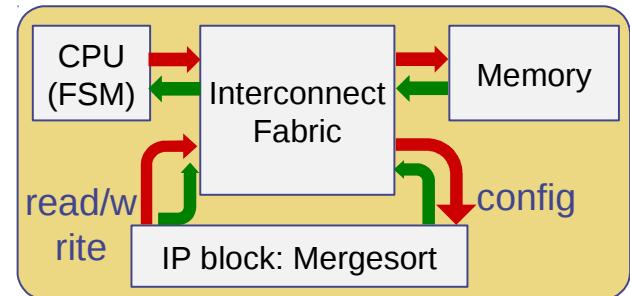
- Go ahead and type ‘h’ for a list of commands. It will list several GDB-like commands, but In this first version, it recognizes just three commands: “continue” (to execute the program), “quit”, and “memory dump” to show a region of memory.
- Type ‘c’ for ‘continue’, and it will perform the mergesort and do the final ‘dumpmem’ showing the memory contents after sorting. You can also use the ‘m’ command to display this memory region.
- Type ‘q’ to quit back to the terminal prompt.

### 3<sup>rd</sup> version: directory Eg06c\_Mergesort/

In this version we use the same source codes for the SoC environment (although we will increase the number of initiators and targets on the Fabric).

We generalize the Mergesort module to have multiple merge engines (instead of just one) that can operate in parallel. Each merge engine will have its own initiator port on the interconnect fabric. The source code is parameterized to have N\_Mergers; in the example code we instantiate this to 2.

Correspondingly, we also increase the number of target ports for the memory. The source code is parameterized to have N\_Mem\_Ports; in the example code we set that to 2.



Please study: src\_BSV/Sys\_Configs.bsv

in this directory, to see the changes to describe this new SoC.

The overall structure is the same; just the details have changed to describe the extra target and initiator ports.

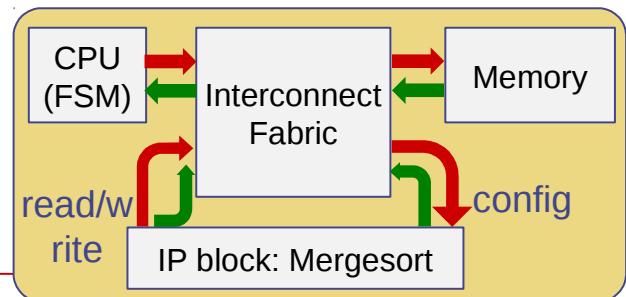
The address-decode function now interleaves memory addresses across the memory ports in 8-byte steps (addr 0..7 in first port, 8..15 in second port, ... and so on).

The file: src\_BSV/CPU.bsv  
is unchanged.

Please study: src\_BSV/Mergesort.bsv

It's memory interface has now become a vector of Clients:

```
interface Mergesort_IFC;
    method Action reset (Addr base_addr);
    interface Server #(Req_T, Rsp_T) config_bus_ifc;
    interface Vector #(N_Mergers, Client #(Req_I, Rsp_I)) mem_bus_ifc;
endinterface
```



In the module mkMergesort, we now instantiate a vector of merge engines, instead of just one:

```
Vector #(N_Mergers, MergeEngine_IFC) mergeEngines <- replicateM (mkMergeEngine);
```

In the module mkMergesort's behavior, where we used to start the single merge engine:

```
mergeEngine.start (0, 0, vrg_configs [n], rg_p1, rg_p2, vrg_configs [n]);
```

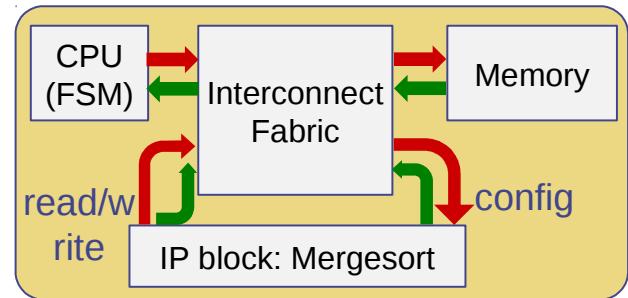
instead, we now just enqueue another “task” on to a queue:

```
f_tasks.enq (tuple4 (0, vrg_configs [n], rg_p1, rg_p2));
```

and, concurrently, we feed these tasks to any available merge engine:

```
for (Integer j = 0; j < valueOf (N_Mergers); j = j + 1)
    rule rl_exec_task;
        match { .i, .span, .p1, .p2 } = f_tasks.first; f_tasks.deq;
        mergeEngines[j].start (fromInteger (j), i, span, p1, p2, vrg_configs [n]);
    endrule
```

## Memory ordering problem



When we introduce multiple memory ports with interleaved addresses in an SoC, we introduce a memory ordering problem:

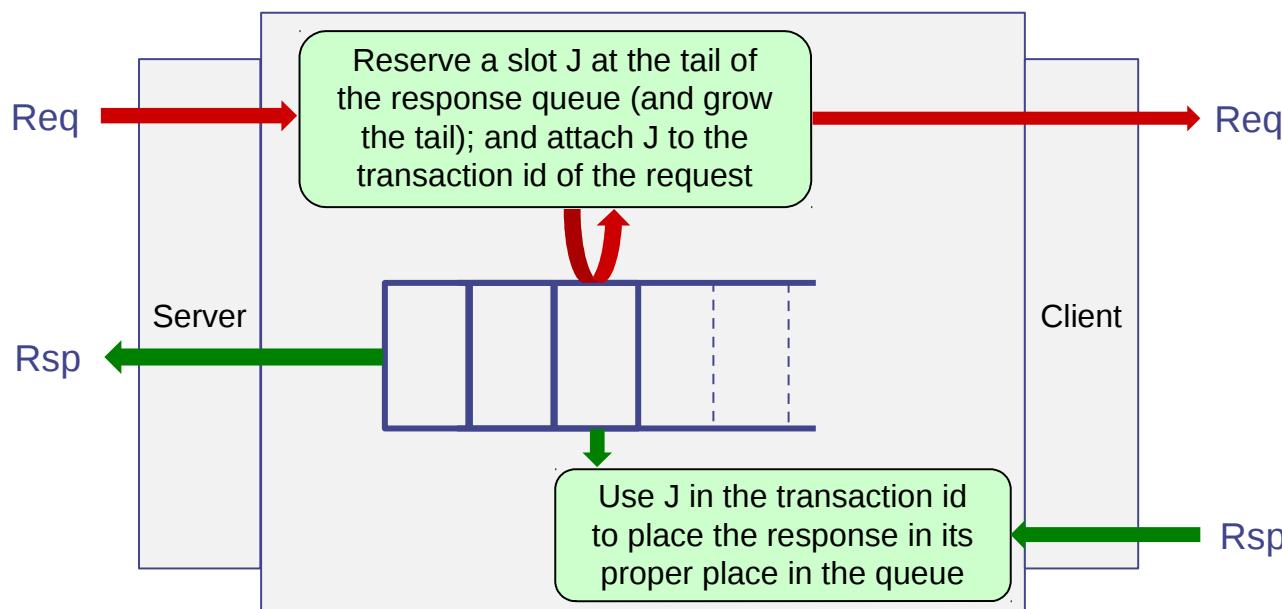
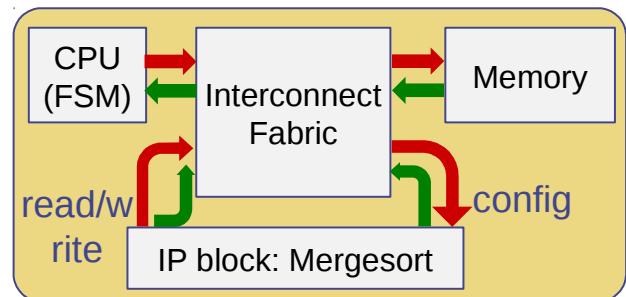
- Suppose the IP block sends two requests, one after the other, to memory at two addresses Addr1 and Addr2.
- These requests may go to two different memory ports, depending on which port services which address
- Since those two ports may be access different regions of memory and face different delays and contention, the responses may come back in a different order
- If the IP block assumes that responses come back in the same order as requests, it will produce wrong results!

Our mkMergeSort module does assume that responses will come back in order!  
(Exercise: study the code and convince yourself that it makes such an assumption.)

Solution: we place a “reorder buffer” between the IP block and the Fabric, to restore the order of responses delivered from the Fabric to the IP block.

Please study: Common/Reorder\_Buffer.bsv

- An incoming request reserves a slot J at the current tail of the response queue (and grows the tail). The position J is carried along with the request in its transaction id.
- A response is inserted into its correct position in the response queue by looking at J in the transaction id



Please study: src\_BSV/Testbench.bsv

The system is instantiated with this excerpt:

```
module mkTestbench (Empty) ;
    CPU_IFC      cpu      <- mkCPU_Model;
    Memory_IFC   mem      <- mkMemory_Model;
    Fabric_IFC   fabric   <- mkFabric;
    Mergesort_IFC mergesort <- mkMergesort;

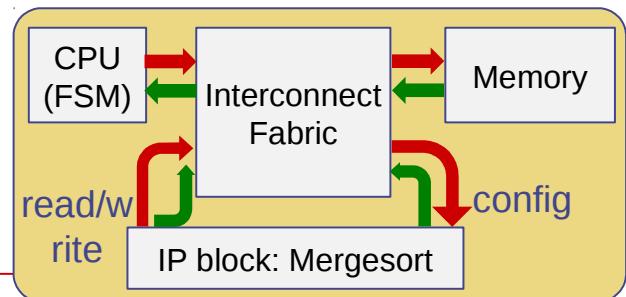
    mkConnection (cpu.dcache_ifc, fabric.v_servers [cpu_d_iNum]);

    for (Integer j = 0; j < valueOf (N_Accel_Clients); j = j + 1) begin
        Reorder_Buffer_IFC reorder_buffer <- mkReorder_Buffer;
        mkConnection (mergesort.mem_bus_ifc [j], reorder_buffer.server);
        mkConnection (reorder_buffer.client, fabric.v_servers [accel_iNums[j]]);
    end

    for (Integer j = 0; j < valueOf (N_Mem_Ports); j = j + 1)
        mkConnection (fabric.v_clients [mem_tNums [j]], mem.bus_ifc [j]);

    mkConnection (fabric.v_clients [accel_tNum], mergesort.config_bus_ifc);

```



i.e., the for-loops connect the vector of mergesort initiators and memory targets to the corresponding ports of the interconnect fabric. We also place reorder buffers on the mergesort initiator ports.

## Build and run the 3<sup>rd</sup> version

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier
- When you simulate you will see the same GDB-like command prompt as in Eg06b. Type ‘c’ (continue) to run the mergesort, and verify that the output looks reasonable.

# Suggested exercises

- All three versions of the example sort 32-bit (4-byte) words of memory.
  - Modify the design to have a *static* parameter such that it will compile to a circuit that sorts memory in units of 1, 2, 4 or 8 bytes (static the size is fixed at compile time). Note that Common/Req\_Rsp.bsv already defines an enum type TLMBSIZE to specify byte size.
    - The mergesort engine should issue memory requests with the selected unit size.
  - Modify the design so that the byte-size selection is done *dynamically*:
    - Add another config register in which the CPU can specify the size.
    - The mergesort engine should issue memory requests with the selected unit size.
  - Modify the last design so that memory requests are always for 8 bytes, even if the sort is on smaller units. E.g., if the sort is on 1-byte units:
    - Only 1 memory read is needed to fetch 8 units.
    - Only 1 memory write is needed to store 8 units.
- All the examples perform a *binary* (radix 2) merge sort, i.e., the basic merge step merges *two* spans.
  - Modify the program to perform a radix 4 merge sort, i.e., the basic merge step should merge *four* spans. Question: when sorting an array of length  $n$ , how many memory references does this perform, compared to the binary merge sort?
  - Parameterize the module for a radix  $k$  mergesort, where  $k$  is a static parameter that may take some chosen range of values (2, 3, 4, ...).

# Summary

This example has shown you key features of an IP block built for high-performance in an SoC context:

- Useful functionality (sorting, which is useful in *many* applications)
- Implementation using an efficient mathematical algorithm (mergesort)
- Key concepts of SoC structure: Fabrics, initiators, targets, memory mapping, ...
- Key concepts of high-performance: pipelining, task queue parallelism, memory bandwidth, managing out-of-order communication, ...
- Generality through parameterization on many dimensions (and hence capable of much re-use in other contexts)



End

```

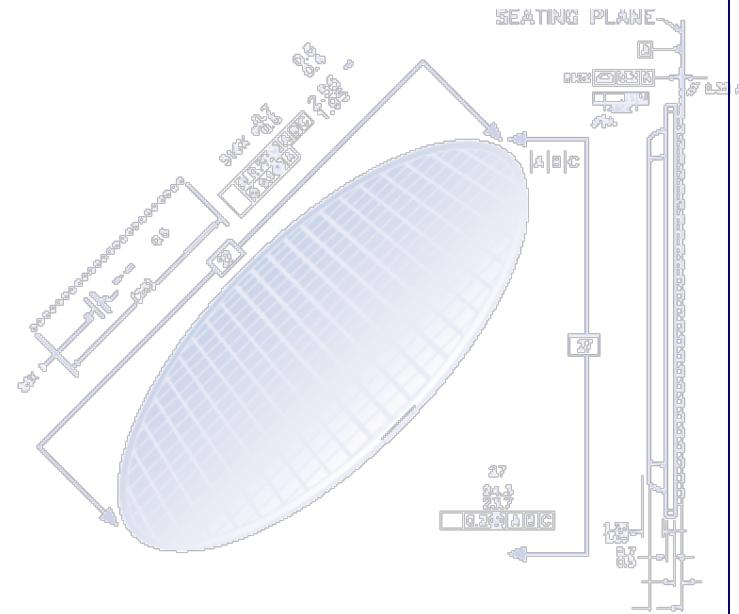
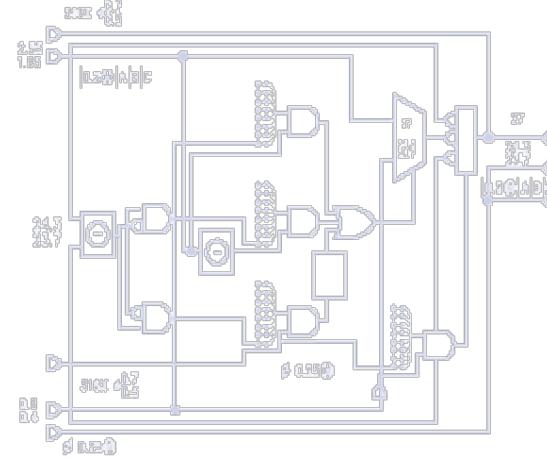
import PDFKit;
typed! Data!(JS) DataT;
module ex_jsd_jsapi;
Integer fileDepth = 15;
function UInt32 determine_gauge(DataT);
    return 0x00000000;
endfunction

PDFNID(DataT) knownAs;
PDFNID(PDFNID(UInt32)) the_header((iszero));
PDFNID(DataT) unknown();
PDFNID(UInt32)(fileDepth) the_culture((iszero));
PDFNID(DataT) culture();
typedef PDFNID(UInt32) the_culture((iszero));

Rule end (DataT);
    DataT h_file = knownAs(file);
    PDFNID(UInt32) cut, max, =
        determine_gauge(file) == 0 ? culture() : culture(
            file,(iszero));
    if (culture != cut)
        culture = cut;
    culture;
endrule

endpackage

```



# BSV Training

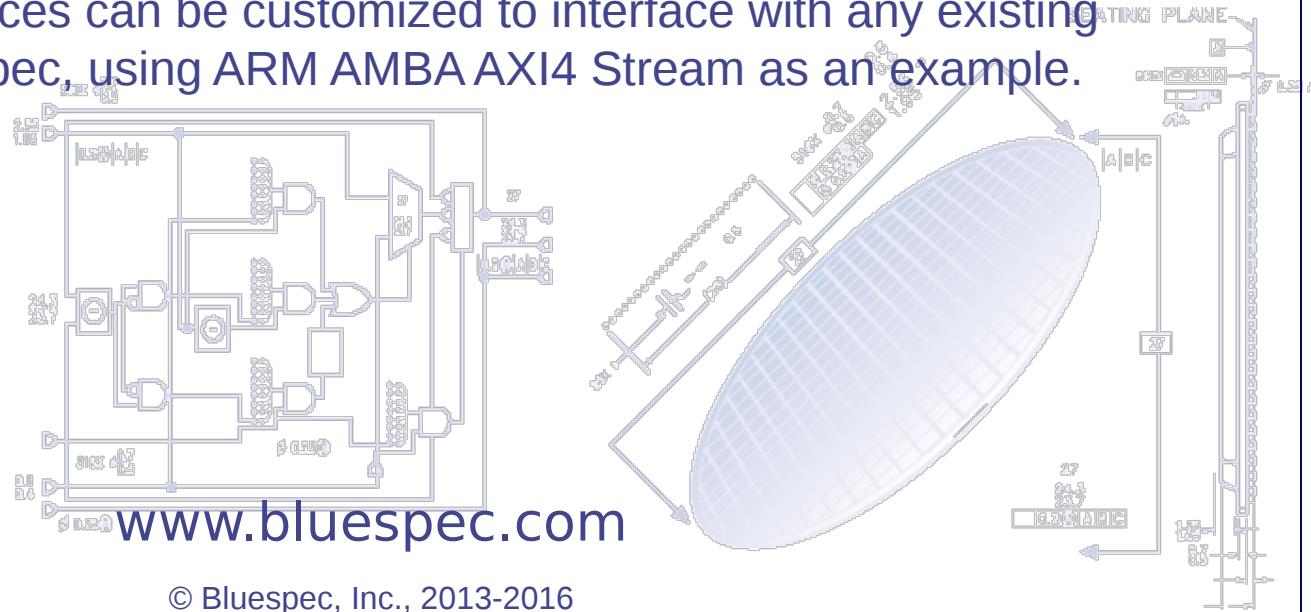
## Eg09: Interfacing to AXI4 Stream

How BSV interfaces can be customized to interface with any existing RTL interface spec, using ARM AMBA AXI4 Stream as an example.

```

import FIFO#(T);
typedef FIFO#(T) Fifo;
module AXI4Stream#(T)(input AXI4Stream#(T) stream_in,
                      output AXI4Stream#(T) stream_out);
    Integer fifo_depth = 15;
    function Bit#(T) determine_psize(Bit#(T) size);
        return (size <= 32) ? 4 : 8;
    endfunction
    FIFO#(T) fifo;
    method Action write(T data);
        if(fifo.length < fifo_depth)
            fifo.enq(data);
        else
            stream_out.write(data);
    endaction
    method T read();
        T data;
        if(fifo.length == 0)
            stream_in.read(data);
        else
            data = fifo.first();
            fifo.deq();
        return data;
    endmethod
endmodule

```

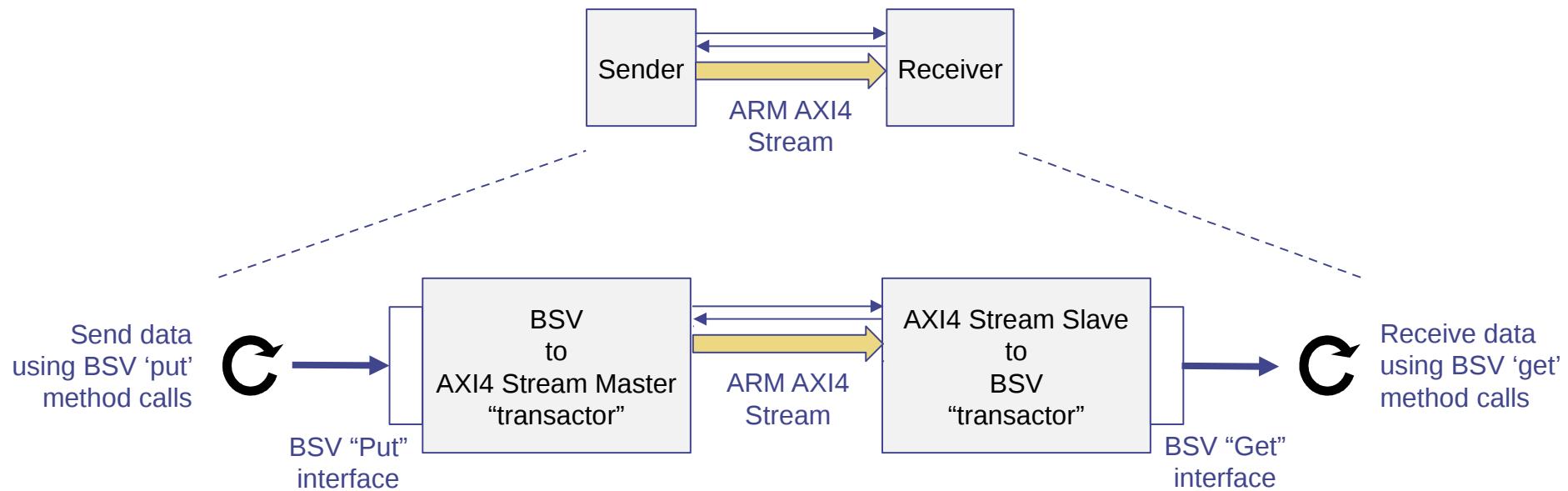


## A note on training logistics

- This example concerns interfacing BSV to existing RTL blocks and protocols. As such, it may not be of interest when an entire design (including testbench) is done in BSV.
- This example can therefore be deferred/ postponed/ omitted if you are not currently trying to interface BSV to existing RTL, and studied later only if/ when that becomes a priority.

Before proceeding, please review the lecture: [Lec\\_Interop\\_RTL](#)

# Eg09: Interfacing with existing HW and protocols



- The “AXI4 Stream” bus and protocol is a standard part of ARM’s AMBA™ AXI™ interconnect family
  - Ref: ARM document "AMBA 4 AXI4-Stream Protocol", AXRM IHI 0051A (ID030510)
- In this example, we show the techniques for interfacing BSV to an existing bus and protocol such as AXI4 Stream.

# The AXI4-Stream spec document

Ref: ARM document "AMBA 4 AXI4-Stream Protocol", AXRM IHI 0051A (ID030510)

## AMBA® 4 AXI4-Stream Protocol Version: 1.0 Specification



Copyright © 2010 ARM. All rights reserved.

### 2.1 Signal list

The interface signals are listed in Table 2-1. For additional information on these signals see further sections of this chapter.

Table 2-1 uses the following parameters to define the signal widths:

- n** Data bus width in bytes.
- i** **TID** width. Recommended maximum is 8-bits.
- d** **TDEST** width. Recommended maximum is 4-bits.
- u** **TUSER** width. Recommended number of bits is an integer multiple of the width of the interface in bytes.

Table 2-1 Interface signals list

Signal	Source	Description
<b>ACLK</b>	Clock source	The global clock signal. All signals are sampled on the rising edge of <b>ACLK</b> .
<b>ARESETn</b>	Reset source	The global reset signal. <b>ARESETn</b> is active-LOW.
<b>TVALID</b>	Master	<b>TVALID</b> indicates that the master is driving a valid transfer. A transfer takes place when both <b>TVALID</b> and <b>TREADY</b> are asserted.
<b>TREADY</b>	Slave	<b>TREADY</b> indicates that the slave can accept a transfer in the current cycle.
<b>TDATA[({8n-1}:0]</b>	Master	<b>TDATA</b> is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
<b>TSTRB[({n-1}:0]</b>	Master	<b>TSTRB</b> is the byte qualifier that indicates whether the content of the associated byte of <b>TDATA</b> is processed as a data byte or a position byte.
<b>TKEEP[({n-1}:0]</b>	Master	<b>TKEEP</b> is the byte qualifier that indicates whether the content of the associated byte of <b>TDATA</b> is processed as part of the data stream. Associated bytes that have the <b>TKEEP</b> byte qualifier deasserted are null bytes and can be removed from the data stream.
<b>TLAST</b>	Master	<b>TLAST</b> indicates the boundary of a packet.
<b>TID[({i-1}:0]</b>	Master	<b>TID</b> is the data stream identifier that indicates different streams of data.
<b>TDEST[({d-1}:0]</b>	Master	<b>TDEST</b> provides routing information for the data stream.
<b>TUSER[({u-1}:0]</b>	Master	<b>TUSER</b> is user defined sideband information that can be transmitted alongside the data stream.

ARM IHI 0051A

Copyright © 2010 ARM. All rights reserved.

2-2

# The AXI4-Stream spec document

Ref: ARM document "AMBA 4 AXI4-Stream Protocol", AXRM IHI 0051A (ID030510)

## 2.2 Transfer signaling

This section gives details of the handshake signaling and defines the relationship of the **TVALID** and **TREADY** handshake signals.

### 2.2.1 Handshake process

The **TVALID** and **TREADY** handshake determines when information is passed across the interface. A two-way flow control mechanism enables both the master and slave to control the rate at which the data and control information is transmitted across the interface. For a transfer to occur both the **TVALID** and **TREADY** signals must be asserted. Either **TVALID** or **TREADY** can be asserted first or both can be asserted in the same **ACLK** cycle.

A master is not permitted to wait until **TREADY** is asserted before asserting **TVALID**. Once **TVALID** is asserted it must remain asserted until the handshake occurs.

A slave is permitted to wait for **TVALID** to be asserted before asserting the corresponding **TREADY**.

If a slave asserts **TREADY**, it is permitted to deassert **TREADY** before **TVALID** is asserted.

The following sections give examples of the handshake sequence.

#### TVALID before TREADY handshake

In Figure 2-1 the master presents the data and control information and asserts the **TVALID** signal HIGH. Once the master has asserted **TVALID**, the data or control information from the master must remain unchanged until the slave drives the **TREADY** signal HIGH, indicating that it can accept the data and control information. In this case, transfer takes place once the slave asserts **TREADY** HIGH. The arrow shows when the transfer occurs.

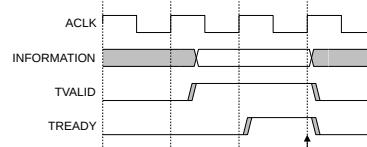


Figure 2-1 TVALID before TREADY handshake

#### TREADY before TVALID handshake

In Figure 2-2 the slave drives **TREADY** HIGH before the data and control information is valid. This indicates that the destination can accept the data and control information in a single cycle of **ACLK**. In this case, transfer takes place once the master asserts **TVALID** HIGH. The arrow shows when the transfer occurs.

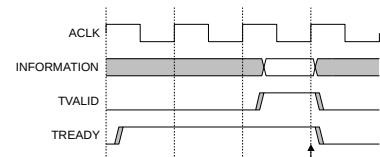


Figure 2-2 TREADY before TVALID handshake

#### TVALID with TREADY handshake

In Figure 2-3 the master asserts **TVALID** HIGH and the slave asserts **TREADY** HIGH in the same cycle of **ACLK**. In this case, transfer takes place in the same cycle as shown by the arrow.

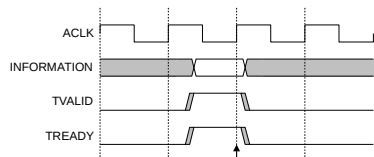
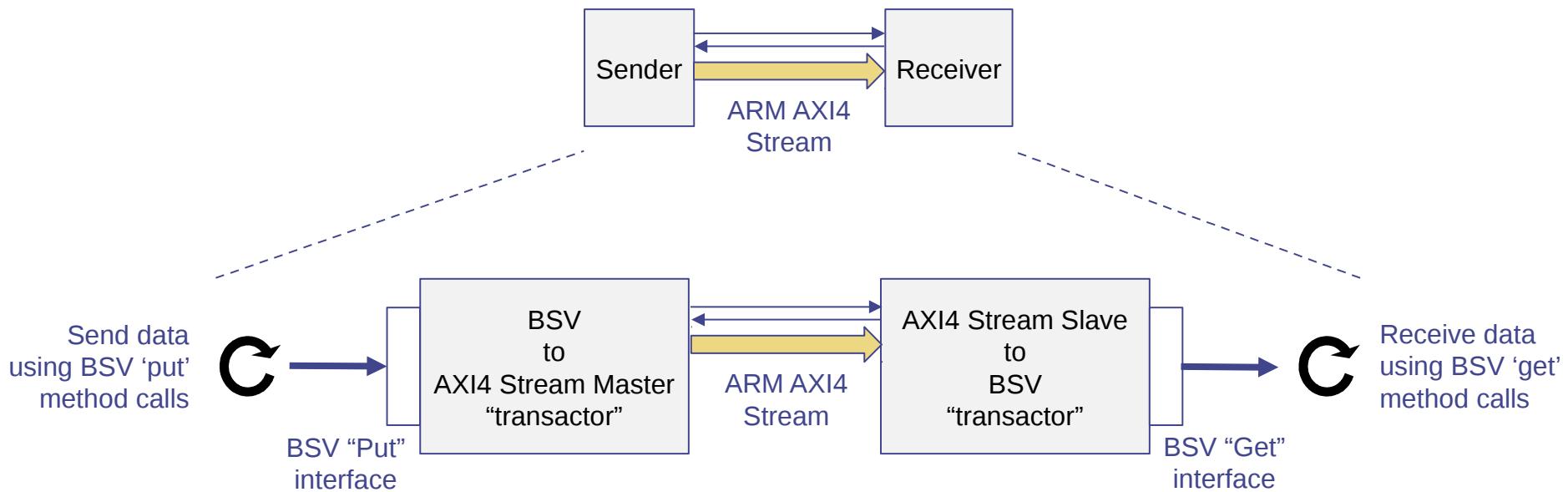


Figure 2-3 TVALID with TREADY handshake

# Overview of this example



- We will define two transactors, as shown above. Each behaves like a FIFO.
- The testbench will “put” data into the master transactor using a standard BSV Put interface. Each datum will appear on the other interface, which is a standard AXI4 Stream interface.
- The slave transactor receives data on a standard AXI4 Stream interface. The testbench “gets” each datum using a standard BSV Get interface.
- In our testbench, we simply connect the AXI4 interfaces of the two transactors, like a simple “loopback” test. When used in an SoC, these interfaces would be connected to other IP blocks with AXI4 Stream interfaces.
- The testbench puts and gets data with varying pauses, to verify that the stream is properly flow-controlled (no data is lost or overwritten).

# Directory: Eg09\_AXI4\_Stream/

Please examine the source file: src\_BSV/AXI4\_Stream.bsv

We first define a BSV struct describing the values carried on the datapath on an AXI4 stream connection:

```
typedef struct {
    Vector #(n, Bit #(8)) tDATA;
    Bit #(n)             tSTRB;
    Bit #(n)             tKEEP;
    Bool                tLAST;
    Bit #(i)              tID;
    Bit #(d)              tDEST;
    Bit #(u)              tUSER;
} AXI4_Stream_Payload #(numeric type n,           // width of TDATA (# of bytes)
                      numeric type i,            // width of TID (# of bits)
                      numeric type d,            // width of TDEST (# of bits)
                      numeric type u)           // width of TUSER (# of bits)

deriving (Bits, FShow);
```

These definitions directly follow the AXI4 Stream specification document (see previous slides).

# Directory: Eg09\_AXI4\_Stream/

Please examine the source file: src\_BSV/AXI4\_Stream.bsv

Next, we define the AXI4 Stream Master and Slave interfaces. E.g., the Master interface:

```
interface AXI4_Stream_Master_IFC #(numeric type n, numeric type i,
                                         numeric type d, numeric type u);
    (* prefix="" *)
    (* always_ready, always_enabled *) method Action put ((* port="TREADY" *) Bool tREADY);

    (* always_ready, result="TVALID" *) method Bool                      tVALID;
    (* always_ready, result="TDATA"  *) method Vector #(n, Bit #(8)) tDATA;
    (* always_ready, result="TSTRB" *) method Bit #(n)                  tSTRB;
    (* always_ready, result="TKEEP" *) method Bit #(n)                  tKEEP;
    (* always_ready, result="TLAST" *) method Bool                     tLAST;
    (* always_ready, result="TID"   *) method Bit #(i)                 tID;
    (* always_ready, result="TDEST" *) method Bit #(d)                tDEST;
    (* always_ready, result="TUSER" *) method Bit #(u)                tUSER;
endinterface
```

We use BSV “attributes” (in “(\*...\*)” brackets) to control the corresponding port signal names when bsc synthesizes Verilog for a module with this interfaces. For details, see: BSV Reference Guide, Section 13

- “prefix=”: overrides normal prefixing of Verilog signal port names with the BSV interface name
- “port=”: overrides normal use of the BSV argument name for the corresponding Verilog input port
- “result=”: overrides normal use of the BSV method name for the Verilog output port for the result
- “always\_ready”: overrides normal creation of a RDY output signal port for every BSV method, representing the method condition (*bsc* formally verifies this assertion in BSV code)
- “always\_enabled”: overrides normal creation of an ENA input signal port for Action and ActionValue methods, by which the environment signals when it is performing that action (*bsc* formally verifies this assertion in BSV code)

*Bottom line: we precisely specify which signals appear in the Verilog, and their names.*

# Directory: Eg09\_AXI4\_Stream/

Please examine the source file: src\_BSV/AXI4\_Stream.bsv

For convenience, we also define ‘mkConnection’ for AXI4 Master and Stream interfaces:

```
instance Connectable #(AXI4_Stream_Master_IFC #(n,i,d,u),
                     AXI4_Stream_Slave_IFC #(n,i,d,u));
  module mkConnection #(AXI4_Stream_Master_IFC #(n,i,d,u) axim,
                     AXI4_Stream_Slave_IFC #(n,i,d,u) axis)
    (Empty);
    (* fire_when_enabled, no_implicit_conditions *)
    rule rl_every_clock;
      axim.put (axis.tREADY);
      axis.put (axim.tVALID, axim.tDATA, axim.tSTRB, axim.tKEEP,
                axim.tLAST, axim.tID, axim.tDEST, axim.tUSER);
    endrule
  endmodule
endinstance
```

We use more BSV attributes (see Reference Guide Section 13) to express the idea that this rule fires on every clock

- “fire\_when\_enabled”: asserts that this rule does not conflict with any other rule, hence WILL\_FIRE = CAN\_FIRE.
- “no\_implicit\_conditions”: asserts that none of the methods used in this rule have method conditions.
- Together, these imply that the rule fires in every clock.

Note: bsc formally verifies these assertions in BSV code

*Bottom line: in the Verilog, we will merely have wires connecting corresponding AXI master and slave ports. No extra wires, no extra logic.*

# Directory: Eg09\_AXI4\_Stream/

Please examine the source file: src\_BSV/AXI4\_Stream.bsv

Finally, we create the desired transactors that convert from BSV Put to AXI4 Stream master and from AXI4 Stream slave to BSV Get.

Each transactor interface just has an AXI4 Stream interface and a Get/Put interface as sub-interfaces:

```
interface AXI4_Stream_Master_Xactor_IFC #(numeric type n, numeric type I,
                                         numeric type d, numeric type u);
    (* prefix="" *)
    interface AXI4_Stream_Master_IFC #(n,i,d,u)      axi_side;
    interface Put #(AXI4_Stream_Payload #(n,i,d,u)) bsv_side;
endinterface
```

In the module implementing a transactor, there is only one interesting feature, the use of mkGFIFO:

```
module mkAXI4_Stream_Master_Xactor (AXI4_Stream_Master_Xactor_IFC #(n,i,d,u));
    ...
    FIFO #(AXI4_Stream_Payload #(n,i,d,u)) fifo <- mkGFIFO(False, True);
    ...
endmodule
```

- A standard BSV FIFO is “guarded”: you cannot invoke the ‘enq’ method when it is full, and you cannot invoke the ‘first’ or ‘deq’ methods when it is empty. Thus, it is impossible to drop or overwrite data.
- The mkGFIFO is a “dangerous” FIFO that allows you to selectively disable these guards. Like RTL FIFOs, the onus is back on you to perform the full/empty checks before invoking corresponding enq/first/deq methods.
- In the above example, the “False” parameter requests that the “enq” guard is not disabled (remains guarded). This is because it is fed by a standard BSV Put interface.
- The “True” parameter requests that the “first/deq” guard is disabled (is unguarded). This is because here we are relying on the AXI4 Stream protocol, not BSV rule/method conditions, to manage flow control.
- The removal of these guards allows the module to satisfy the “always\_enabled” and “always\_ready” assertions shown earlier on the interface.

# Directory: Eg09\_AXI4\_Stream/

Please examine the source file: src\_BSV/Testbench.bsv

At the bottom of the file we use our standard trick to create separately synthesizable specializations of the polymorphic transactor modules. E.g.:

```
(* synthesize *)
module mkAXI4_Stream_Master_Xactor_2_4_4_1 (AXI4_Stream_Master_Xactor_IFC #(2,4,4,1));
    let ifc <- mkAXI4_Stream_Master_Xactor;
    return ifc;
endmodule
```

The module mkTestbench itself is very straightforward.

The conditions on the rules rl\_gen and rl\_drain are to make them pause at different times, in order to exercise positive pressure (sender ready, receiver not ready) and negative pressure (vice versa).

# Build and run the example

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier. Since the focus of this example is on interfacing with an RTL protocol, you should build and simulate in Verilog (even though it will also work in Bluesim.)
- Observe the inputs and outputs and verify that they are reasonable (all data is transmitted across the transactors without dropping, overwriting, replication, etc.)
- ‘make v\_simulate’ should create a ‘dump.vcd’ waveform file. Display this waveform to observe the AXI4 Stream signals TREADY, TVALID and TDATA, and verify that it is following the AXI4 Stream protocol according to the specification. The file “waves\_screenshot.tiff” is a picture of the waveform.
- Examine the generated Verilog files in the verilog/ directory and observe that AXI4 Stream interface signals are exactly per the specification.

# Suggested exercises

- Connect the example transactors to a real IP block of your choice that has an AXI4 Stream interface.
- The BSV software distribution contains source code for libraries for interfacing to other ARM AMBA buses (more complex than AXI4 Stream), in the directories:
  - \$BLUESPECDIR/BSVSource/Axi/
  - \$BLUESPECDIR/BSVSource/AHB/Study these codes for more examples of interfacing BSV to existing RTL buses and protocols
- Create similar transactors for some other bus of your choice (e.g., OCP)

# Summary

- Verilog ports are just a special case of BSV methods: using “always\_ready” and “always\_enabled” attributes, we can eliminate BSV’s default handshaking signals, leaving us with Verilog-like ports. We can then write explicit logic for any specified protocol or handshaking (using unguarded FIFOs, if necessary and appropriate).
- Using other attributes, we can control the exact names of interface ports in the generated Verilog
- Together, these techniques allow us to write BSV code that will interface precisely into any specified RTL interface.



End

```

import POFN.*;

typedef struct {
    int id;
    string name;
} node_t;

node_t* create_node(string name);

Integer file_length = 15;

function Int(32) determine_group(DataT);
    return {31:0};
endfunction

POFN(DataT) libname();
libname.POFN(DataT) the_Libname();
POFN(DataT) catname();
catname.POFN(DataT) the_catname();
POFN(DataT) collname();
collname.POFN(DataT) the_collname();

```

File end (This):

```

    DataT lib_file = libname();
    POFN(DataT) cat_name =
        determine_group(lib_file) == 0 ? collname : null;
    cat_name.POFN(DataT) is
    libname();
    catname = cat_name;

```

endfile; I see lib.cat is be

