Learn RISC-V CPU Implementation and **BSV**

(**BSV**: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L11: **BSV**: Verifiying BSV Designs

bluespec

Reminders

Please git clone: `https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design`
(git pull for latest version). Repsitory structure:

```
./Book_BLang_RISCV.pdf
  Slides/
      Slides_01_Intro.pdf
      Slides_02_ISA.pdf
      ...
  Exercises/
      Ex-03-A-Hello-World/
      Ex-03-B-Top-and-DUT/
      ...
  Code/
      src_Top/
      src_Drum/
      src_Fife/
      src_Common/
      ...
  Doc/Installing_bsc_Verilator_etc.{adoc,html}
```

- Slides and Exercise are numbered in sync with book Chapter numbers.
- For Exercises, please see Appendix E of the book. Some (not all) exercises have associated code in the `Exercises/` directory.

To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

- *bsc* compiler (see `https://github.com/B-Lang-org/bsc`)
- Verilator compiler (see `https://www.verilator.org/`)

Chapter Roadmap

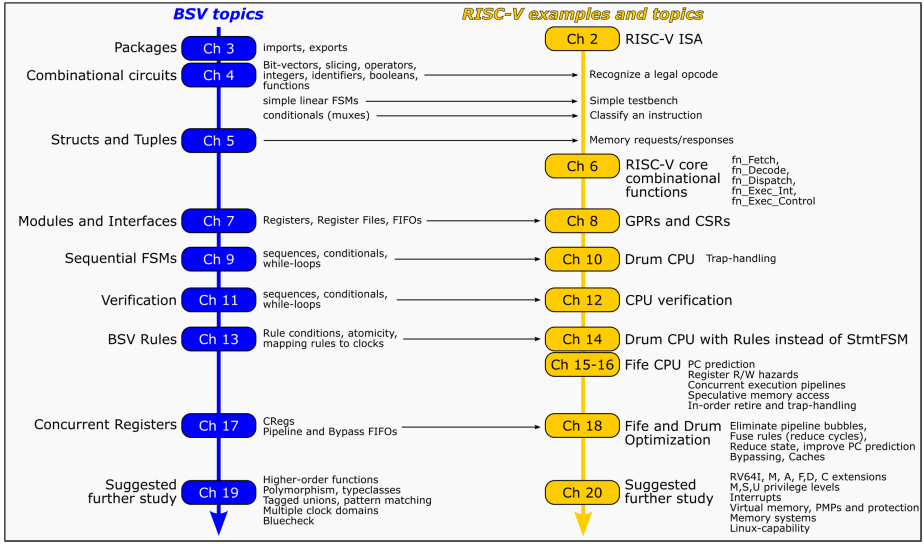
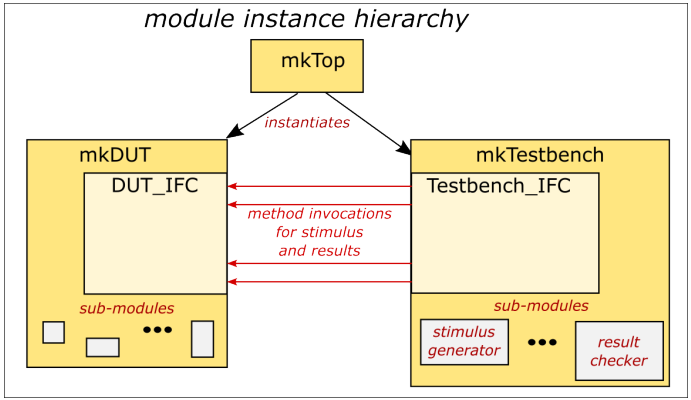


Table of Contents

1. Reminders
2. Testbenches and DUTs
3. **printf**-style debugging
   - Formatted strings
4. Dynamic Assertions
5. Waveform-style debugging
6. Final Comments

BSV: Testbenches and DUTs

Typical verification system setup



`mkTop` has `Empty` interface and merely serves to intantiate and connect:

- DUT = "Design Under Test"
- Testbench
  (Tb, Test Harness, Test Environment)

  Contains:
  - Stimulus generator: provides inputs to DUT (via DUT methods)
  - Result logger/checker: collects outputs from DUT (via DUT methods), and records/analyzes/checks them.

- Testbenches can be written in SystemVerilog (*e.g.*,, UVM, simulation only).
- Testbenches can import C code to read/write files, run generator and analysis programs, *etc.* (simulation only).
- Testbenches may be *synthesizable*, in which case the whole setup can run on FPGAs.

BSV: **printf**-style debugging

printf-style debugging

BSV has the same print-like statements as Verilog and SystemVerilog:

```
$write    (      format-string, arg, ..., arg )
$display  (      format-string, arg, ..., arg )

file <- $fopen ("log.txt", "w");
...
$fwrite   ( file, format-string, arg, ..., arg )
$fdisplay ( file, format-string, arg, ..., arg )
```

- The first two write to "standard output" (*i.e.,* the terminal); the latter two to a file. All are relevant only in simulation; no hardware is generated for any of them.
- `$write` and `$fwrite` do not append a trailing newline to the output; `$display` and `$fdisplay` do.
- The format string is a string (in double-quotes) with formatting directives for the arguments that follow (`%d` for signed integers, `%b` for binary numbers, `%h` for hexadecimal numbers, *etc.*).
- In BSV, additionally, you can interleave format strings and arguments, like this:

  $display (*format-string*, *arg*, ..., *format-string*, *arg*, ...)

Formatted strings (of type `Fmt`)

In **BSV**, arguments in `$write` and `$display` statements can also be value of type "`Fmt`", representing a formatted string.

For many struct and enum definitions, we append a "`deriving (FShow)`" clause, like this:

```
typedef enum {OPCLASS_SYSTEM,
              ...}
OpClass
deriving (Bits, Eq, FShow);
```

```
typedef struct {
    ...
} Decode_to_RR
deriving (Bits, FShow);
```

Consequently, the *bsc* compiler automatically defines a function `fshow()` that takes an argument of that type and returns a "formatted string" of type `Fmt`, which can be used in a `$display()`, like this:

```
OpClass      o  = ...
Fmt          fo = fshow (o);
$display ("opclass = ",  fo);
```

Displays the symbolic name (*e.g.,* `"OPCLASS_SYSTEM"`) instead of the bit-representation of the enum value.

```
Decode_to_RR y  = ...
Fmt          fy = fshow (y);
$display ("Decode result is ", fy);
```

Displays a formatted version of the struct, with individual struct fields, instead of the bit-representation of the whole struct value.

Complex `Fmt` values can be constructed

src_Common/Inter_Stage.bsv: line 213 ...

```
function Fmt fshow_Decode_to_RR (Decode_to_RR x);
   Fmt f = $format ("    Decode_to_RR{");
   f = f + $format ("I_%0d", x.inum);
   f = f + $format (" pc:%08h", x.pc);
   f = f + $format (" instr:%08h", x.instr);
   f = f + $format (" pred:%08h epoch:%0d\n", x.predicted_pc, x.epoch);
   f = f + $format ("            ");
   f = f + $format ("fallthru:%08h ", x.fallthru_pc);
   if (x.exception) begin
      f = f + fshow_cause (x.cause);
      f = f + $format (" tval:%0h", x.tval);
   end
   else begin
      f = f + fshow (x.opclass);
      f = f + $format (" has_{rs1,rs2,rd}:{%0d,%0d,%0d} writes_mem:%0d, imm:%0h",
                       x.has_rs1, x.has_rs2, x.has_rd, x.writes_mem, x.imm);
   end
   f = f + $format ("}");                                                    1
   return f;                                                                 2
endfunction                                                                  3
```

- `$format()` is similar to `$display()`: same format strings and arguments.
- `$format()` is a pure function, with result type `Fmt`; `$display()` is a side-effecting function, with result type `Action`.
- The "+" operator can combine two `Fmt` values, effectively concatenating the two strings that they represent.

Example usage:

```
Decode_to_RR y  = ...
Fmt          fy = fshow_Decode_to_RR (y);
$display ("Decode result is ", fy);
```

BSV: Dynamic Assertions

Dynamic Assertions

Importing the following *bsc* library:

```
import Assert ::  *;
```

makes the following library function available:

```
function Action dynamicAssert (Bool b, String s);
```

It can be used in any Action context (*e.g.*, rule body) to check an expected property each time that Action is executed. *E.g.,*

src_Drum/CPU.bsv

```
Action a_Retire_DMem =
action
   ...
   let mem_rsp <- pop_o (to_FIFOF_O (f_DMem_rsp));
   dynamicAssert ((mem_rsp.rsp_type != MEM_REQ_DEFERRED),
                  "Mem req not speculative but got DEFERRED mem response");
   ...
endaction
```

During simulation, if the boolean expression is false, it prints out the string message and terminates the simulation.

Such boolean expressions are also called a "*correctness conditions*" and "*invariants*".

This is purely a simulation facility; it does not generate any hardware.

Dynamic assertions should be used liberally in **BSV** code to verify invariants.

BSV: Waveform-style debugging

Waveform-style debugging

Many hardware designers like to debug designs using "waveforms", which are a graphical display of how values on buses (bundles of wires) in the design vary over time.

All Verilog, SystemVerilog and VHDL simulators have a facility to write out a "Value Change Dump" (VCD) file, which is a record of how each bus (bundle of wires) in the design changed over time (measured with clock ticks). VCD files can then be viewed as a graphical display in any waveform viewer. Waveform viewers are bundled with most commercial RTL simulators, but the free and open-source *gtkwave* viewer is also popular.

VCD dumping can also be controlled from within a **BSV** program using these three Actions:

`$dumpvars`	Starts writing out VCDs
`$dumpoff`	Stops writing out VCDs
`$dumpon`	Resumes writing out VCDs

This will produce a "*foo*`.vcd`" file, which can then be viewed in any waveform viewer.

Final comments on **BSV** Verification

- Verification in **BSV** is, in principle, the same as in Verilog or SystemVerilog: The DUT is instantiated along with a Testbench that provides stimulus and consumes the output for checking and analysis.

  There are many good textbooks available on verification, covering topics such as "code coverage" (how much of the DUT has been tested), constrained-random stimulus generation, so-called "*fuzzing*", creation of so-called "Verification IP" (reusable library components for testbenches), *etc.*

- The Testbench can be written in **BSV**, but it can also be written in Verilog or SystemVerilog.

  If written in **BSV**, the testbench is synthesizable, and can be executed along with the DUT on FPGAs, which is usually many orders of magnitude faster than simulation.

  If written in Verilog or SystemVerilog, the testbench may or may not be synthesizable. If not synthesizable, it can only be run in simulation.

- For **printf**-style debugging, **BSV** enhances traditional `$display()` with a powerful facility of "formatted strings" (expressions of type `Fmt`).

- **BSV** also offers dynamic assertions and VCD waveform dumping.

End