

Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L3: Structure of BSV Programs



Reminders

Please git clone or git pull: https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design

```
./
  Book_BLang_RISCV.pdf
  Code/
  Doc/
    Installing_bsc_Verilator_etc.{adoc,html}
  Exercises/
    Ex_03_B_Top_and_DUT
    Ex_03_A_Hello_World
    ...
  Slides/
    Slides\_01\_Intro.pdf
    Slides\_02\_ISA.pdf
    ...
```

Make sure you have installed:

- *bsc* compiler: <https://github.com/B-Lang-org/bsc>
- Verilator compiler: <https://www.verilator.org/>

Strategy

We start learning **BSV** “from the outside in”, and with simple exercises, so that:

- you are very quickly able to start *reading* Drum and Fife code;
- you are very quickly able to run the codes and to get in the habit of compiling-and-running; and
- you are very quickly make small modifications,

even though it will take a little longer before you are able to code things yourself from scratch.

BSV language, compiler and libraries documents

From the book, Appendix A.6.5:

- The “**BSV** Language Reference Guide”. This document describes the syntax and semantics of **BSV**.
PDF: https://github.com/B-Lang-org/bsc/releases/latest/download/BSV_lang_ref_guide.pdf
- The “BSC Libraries Reference Guide”. This document describes the extensive set of libraries and IP (Intellectual Property blocks) available to the **BSV** user.
PDF: https://github.com/B-Lang-org/bsc/releases/latest/download/bsc_libraries_ref_guide.pdf
- The “BSC User Guide”. This document describes how to use the *bsc* compiler, which compiles our hardware descriptions written in **BSV** into Verilog (which can then be simulated or synthesizes using standard Verilog tools).
PDF: https://github.com/B-Lang-org/bsc/releases/latest/download/bsc_user_guide.pdf

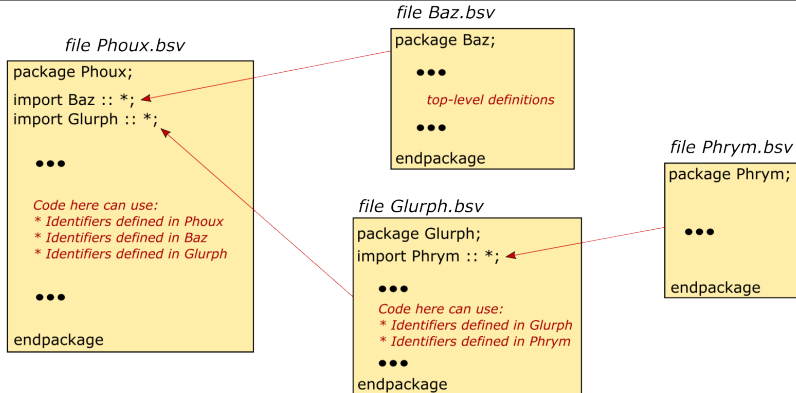
We will be using the Language Reference Guide and Librares Reference Guide extensively, so you may wish to download a copy for your laptop.



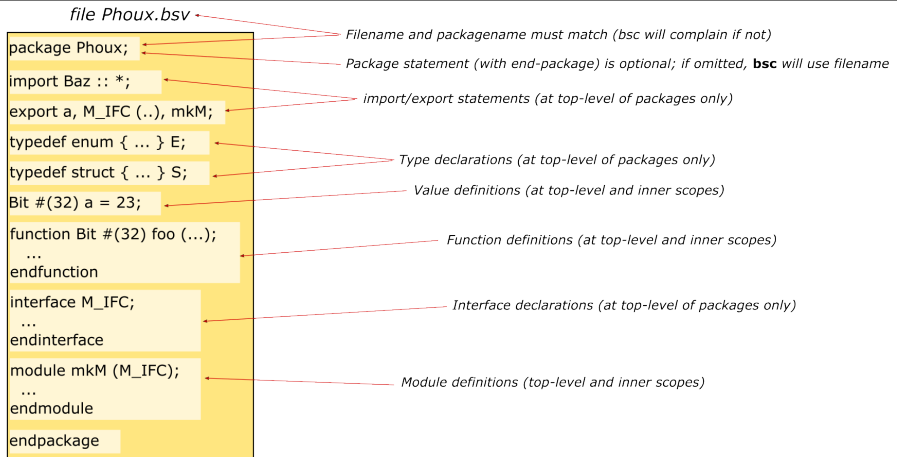
Exercise break

Please see directory: `Exercises/Ex_03_A_Hello_World/`
and its README.

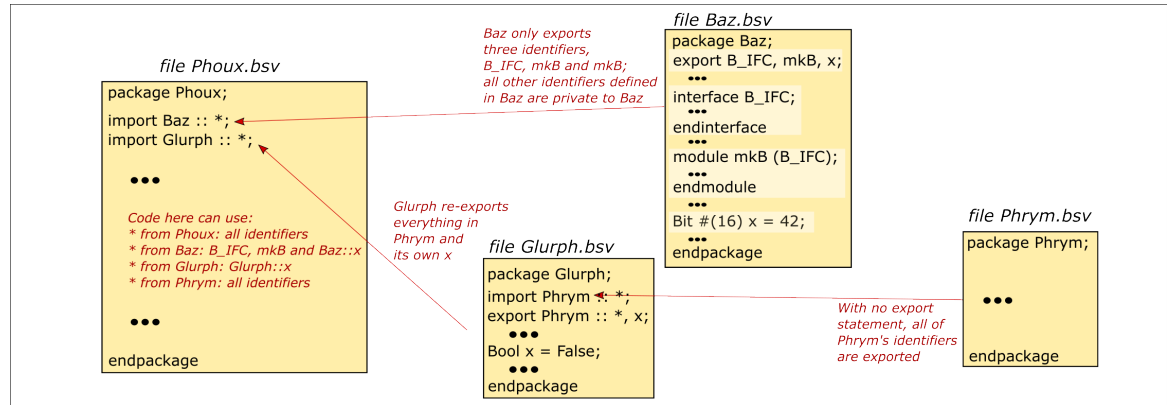
File-level view of a BSV program



What's in a BSV package/file?



Namespace control with package imports and exports





Exercise break

Please see directory: `Exercises/Ex_03_B_Top_and_DUT/`
and its README.

What's in an Interface Declaration?

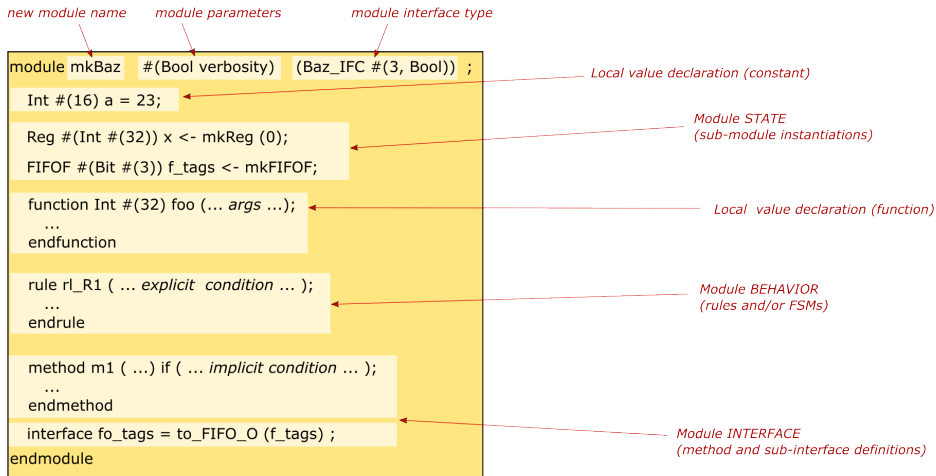
The diagram shows an interface declaration for `Baz_IFC` with several annotations pointing to its components:

- `interface`: new interface type
- `Baz_IFC`: numeric type parameter declaration
- `numeric type n`: value type parameter declaration
- `type t`: value type parameter declaration
- `method Action m1 (Int #(32) x, Bool y, t z);`: Action method declaration (methods can have arguments)
- `method ActionValue #(Bit #(16)) m2 (... args ...);`: ActionValue method declaration
- `method Bit #(16) m3 (... args ...);`: Value method declaration (return type is not Action or ActionValue)
- `interface FIFO_O #(Bit #(n)) fo_tags;`: Nested sub-interface declaration
- `endinterface`: Existing interface type

```
interface Baz_IFC #( numeric type n, type t );
  method Action m1 (Int #(32) x, Bool y, t z);
  method ActionValue #(Bit #(16)) m2 (... args ...);
  method Bit #(16) m3 (... args ...);

  interface FIFO_O #(Bit #(n)) fo_tags;
endinterface
```

What's in a Module Declaration?





Exercise break

Please see directory: `Exercises/Ex_03_C_Module_and_Interface/`
and its README.

What's in a Rule?

new rule name

rule condition ("explicit condition")

```
rule rl_Fetch_req ( rg_running  
                    && (! f_Fetch_from_Retire.notEmpty) );
```

```
let pred_pc = rg_pc + 4;  
let y      = fn_Fetch (rg_pc, pred_pc, rg_epoch, rg_inum);
```

```
f_Fetch_to_Decode.enq (y.to_D);  
f_Fetch_to_IMem.enq (y.mem_req);
```

```
rg_pc  <= pred_pc;  
rg_inum <= rg_inum + 1;
```

```
endrule
```

Two local variable definitions

*Two Actions
(invocations of FIFO ".enq" methods)*

*Two Actions
(invocations of register "._write" methods)*

What's in an Interface Definition?

The diagram illustrates the components of a BSV interface definition. It features two code snippets on a yellow background, with red arrows pointing from descriptive labels to specific parts of the code.

Method 1: Action init

```
method Action init ( Initial_Params initial_params ) if ( ! rg_running );  
  rg_pc      <= initial_params.pc_reset_value;  
  rg_running <= True;  
endmethod
```

Method 2: Bit #(XLEN) read_epc

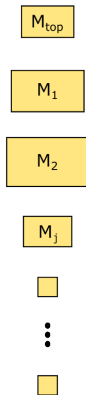
```
method Bit #(XLEN) read_epc;  
  return csr_mepc;  
endmethod
```

Annotations:

- method name*: Points to `init` in the first method.
- method arguments*: Points to `Initial_Params initial_params` in the first method.
- method condition ("implicit condition")*: Points to `if (! rg_running)` in the first method.
- method body*: Points to the body of the first method (`rg_pc <= initial_params.pc_reset_value; rg_running <= True;`). A note specifies: *(Action and ActionValue methods can contain Actions; Value methods cannot contain Actions)*.
- return statement*: Points to `return csr_mepc;` in the second method. A note specifies: *(in Value-methods and ActionValue methods but not in Action methods)*.

Static elaboration

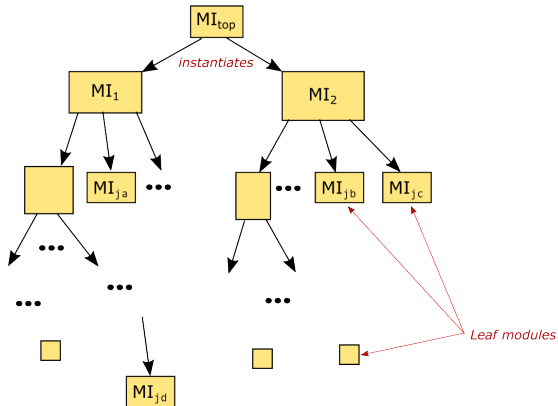
module definitions



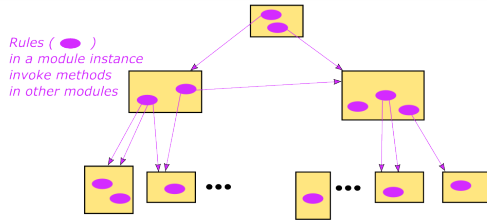
Static Elaboration



module instance hierarchy



Module interaction



End