Learn RISC-V CPU Implementation and BSV
(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L3: Structure of BSV Programs

bluespec ⬡

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).
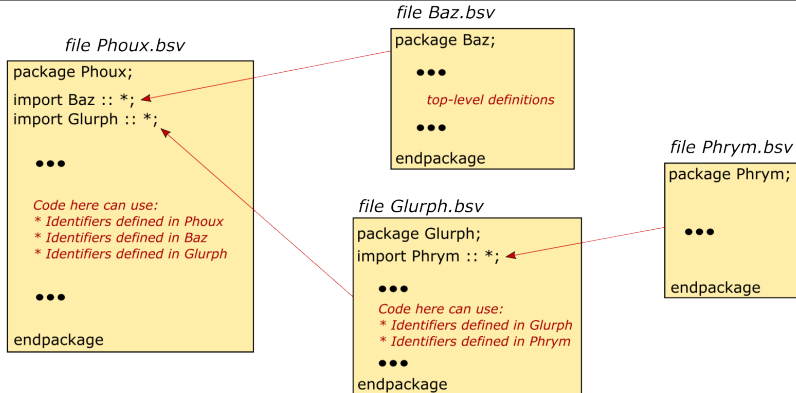 PDF: `https://github.com/B-Lang-org/bsc/releases/latest/download/bsc_user_guide.pdf`

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?

file Phoux.bsv

```
package Phoux;

import Baz :: *;

export a, M_IFC (..), mkM;

typedef enum { ... } E;

typedef struct { ... } S;

Bit #(32) a = 23;

function Bit #(32) foo (...);
   ...
endfunction

interface M_IFC;
   ...
endinterface

module mkM (M_IFC);
   ...
endmodule

endpackage
```

Filename and packagename must match (bsc will complain if not)

Package statement (with end-package) is optional; if omitted, **bsc** will use filename

import/export statements (at top-level of packages only)

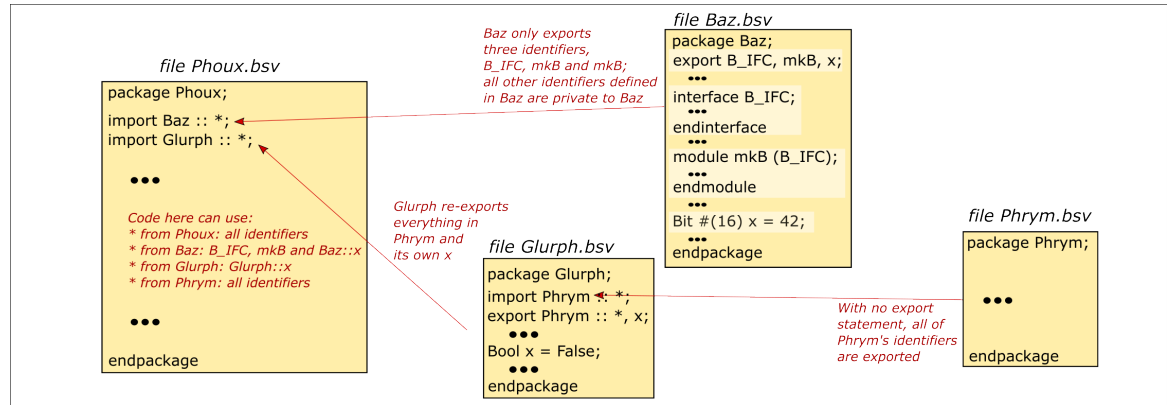Type declarations (at top-level of packages only)

Value definitions (at top-level and inner scopes)

Function definitions (at top-level and inner scopes)

Interface declarations (at top-level of packages only)

Module definitions (top-level and inner scopes)

Namespace control with package imports and exports

file Phoux.bsv

```
package Phoux;

import Baz :: *;
import Glurph :: *;

    •••

    Code here can use:
    * from Phoux: all identifiers
    * from Baz: B_IFC, mkB and Baz::x
    * from Glurph: Glurph::x
    * from Phrym: all identifiers

    •••

endpackage
```

Baz only exports
three identifiers,
B_IFC, mkB and mkB;
all other identifiers defined
in Baz are private to Baz

file Baz.bsv

```
package Baz;
export B_IFC, mkB, x;
    •••
interface B_IFC;
    •••
endinterface
    •••
module mkB (B_IFC);
    •••
endmodule
    •••
Bit #(16) x = 42;
    •••
endpackage
```

Glurph re-exports
everything in
Phrym and
its own x

file Glurph.bsv

```
package Glurph;
import Phrym :: *;
export Phrym :: *, x;
    •••
Bool x = False;
    •••
endpackage
```

file Phrym.bsv

```
package Phrym;

    •••

endpackage
```

With no export
statement, all of
Phrym's identifiers
are exported

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

What's in an Interface Declaration?


```
                    new interface type        numeric type parameter declaration
                                                      value type parameter declaration

interface Baz_IFC #( numeric type n,  type t ) ;
                                                            Action method declaration
  method Action m1 (Int #(32) x, Bool y, t z);              (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 FIFOF_O #(Bit #(n)) fo_tags;                    Nested sub-interface declaration
endinterface

                              Existing interface type
```
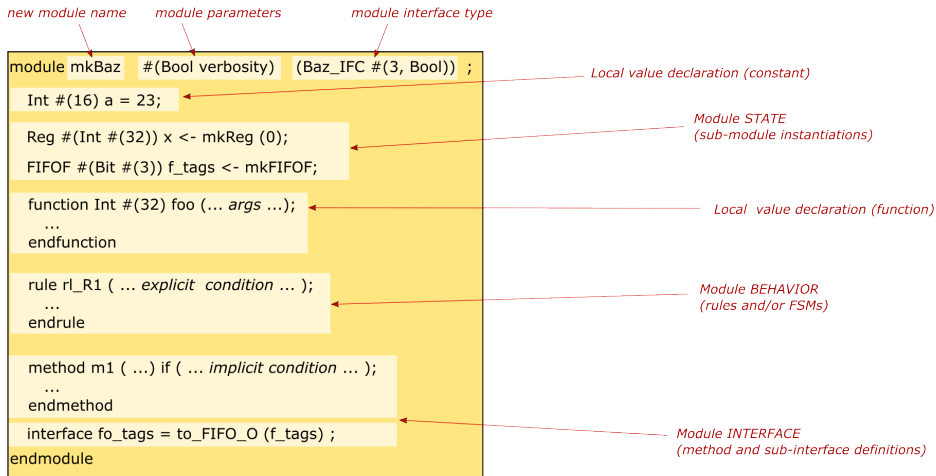
What's in a Module Declaration?

new module name   module parameters   module interface type

```
module mkBaz   #(Bool verbosity)   (Baz_IFC #(3, Bool))  ;

  Int #(16) a = 23;                                           ← Local value declaration (constant)

  Reg #(Int #(32)) x <- mkReg (0);                            ← Module STATE
  FIFOF #(Bit #(3)) f_tags <- mkFIFOF;                            (sub-module instantiations)

  function Int #(32) foo (... args ...);                      ← Local value declaration (function)
     ...
  endfunction

  rule rl_R1 ( ... explicit  condition ... );                ← Module BEHAVIOR
     ...                                                         (rules and/or FSMs)
  endrule

  method m1 ( ...) if ( ... implicit condition ... );
     ...
  endmethod

  interface fo_tags = to_FIFO_O (f_tags) ;                   ← Module INTERFACE
endmodule                                                       (method and sub-interface definitions)
```

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 FIFOF ".enq" methods)

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

What's in an Interface Definition?

method name   method arguments   method condition ("implicit condition")

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

method Bit #(XLEN) read_epc;
    return csr_mepc;
endmethod
```
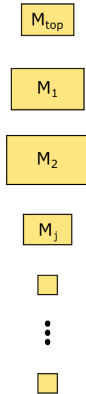
method body
(Action and ActionValue methods can contain Actions;
Value methods cannot contain Actions)

return statement
(in Value-methods and ActionValue methods
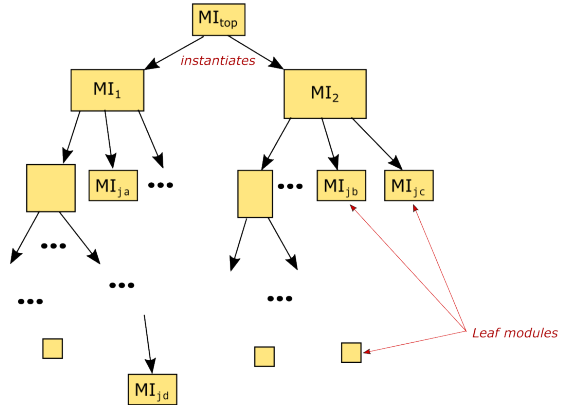but not in Action methods)

Static elaboration

Rules (●)
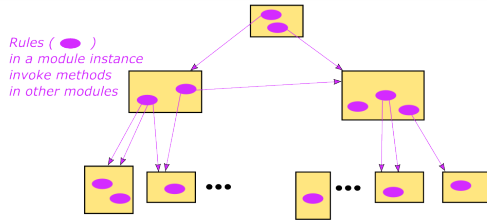in a module instance
invoke methods
in other modules

End