Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L4: Structure of BSV Programs

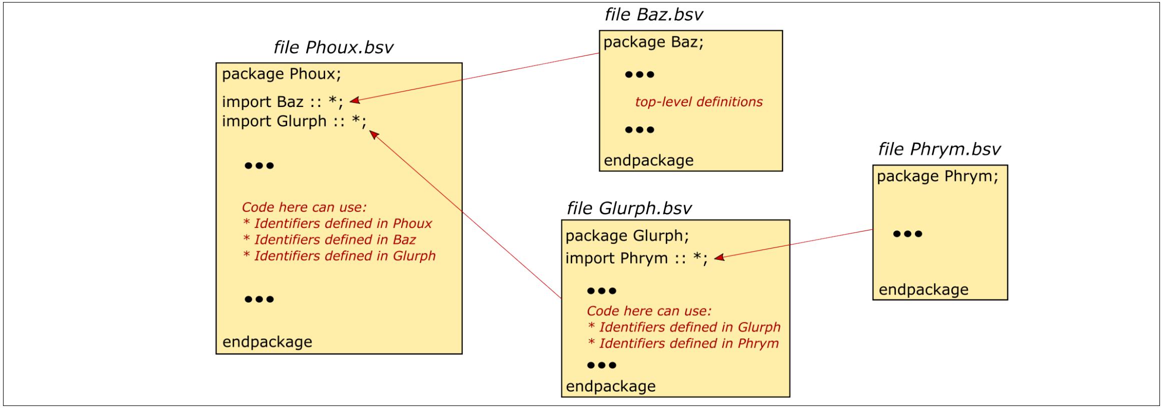
bluespec

First: a Minimal (trivial!) BSV program

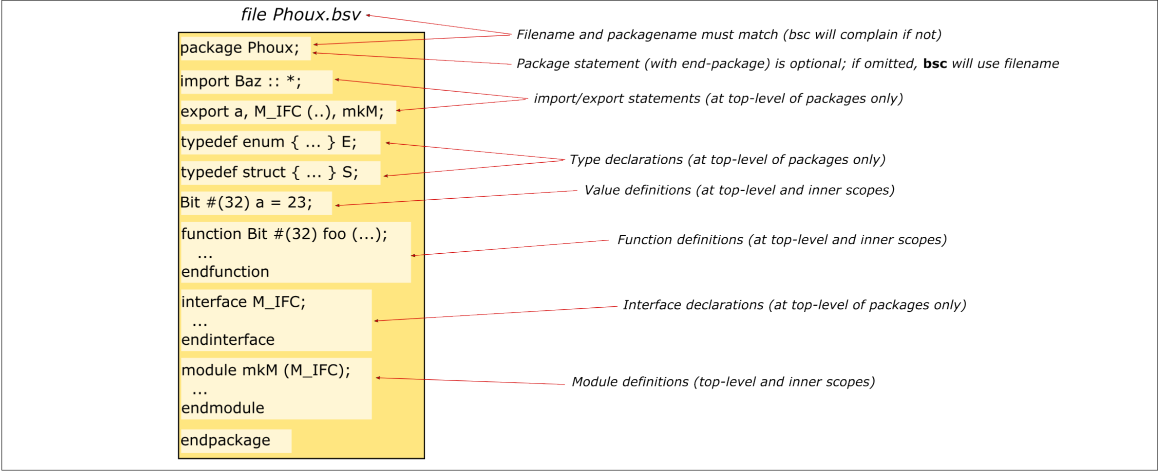
Examples

Demo: please see directory `Ex_04_01`, code and `Makefile`

File-level view of a BSV program

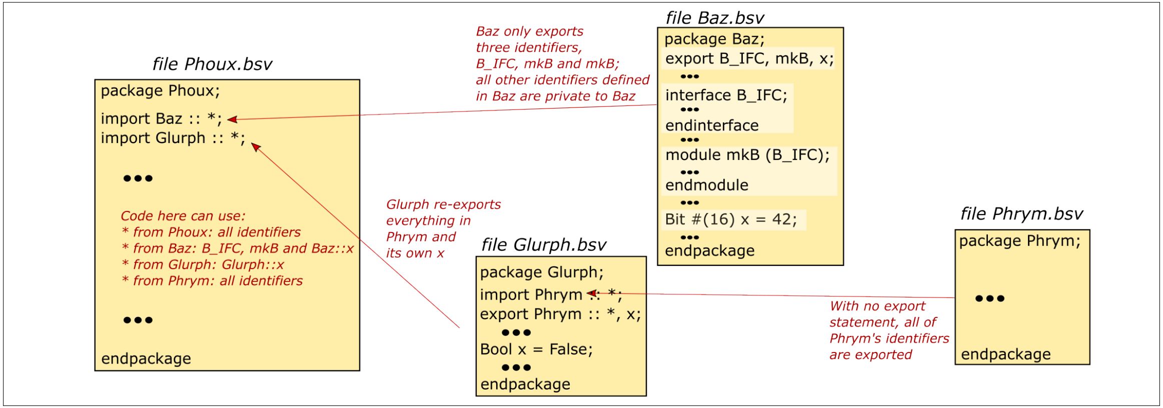


What's in a BSV package/file?


file Phoux.bsv
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)

```
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
```

Namespace control with package imports and exports



file Phoux.bsv

```
package Phoux;
import Baz :: *;
import Glurph :: *;

•••

•••
endpackage
```

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

file Baz.bsv

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

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

file Glurph.bsv

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

Glurph re-exports everything in Phrym and its own x

file Phrym.bsv

```
package Phrym;

•••

endpackage
```

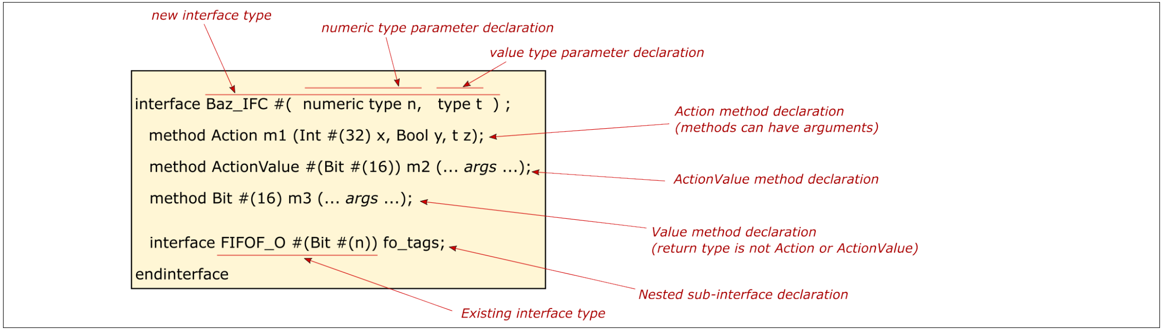
With no export statement, all of Phrym's identifiers are exported

Extending our Minimal BSV program to two packages/files

Examples

Demo: please see directory `Ex_04_02`, code and `Makefile`

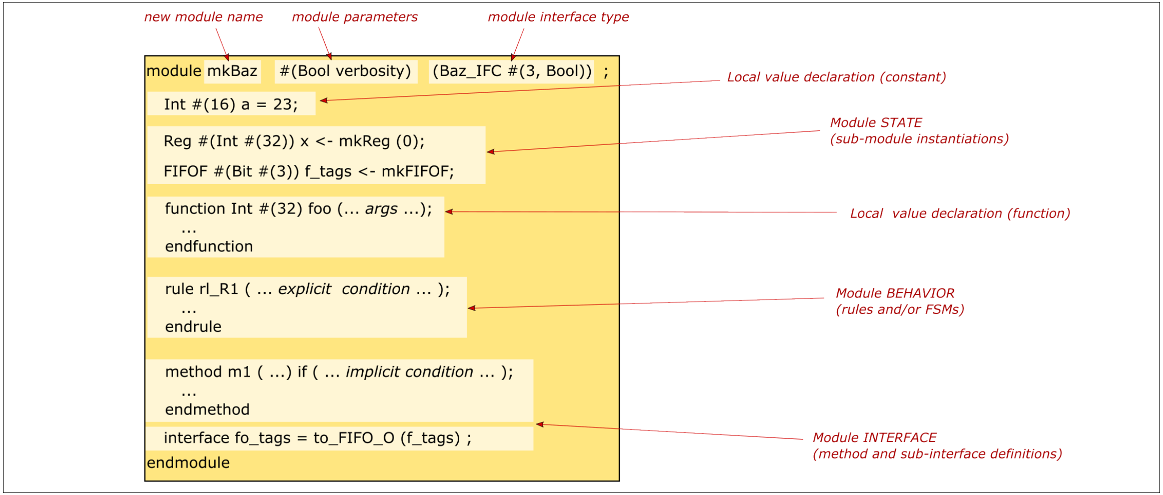
What's in an Interface Declaration?



```
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 FIFOF_O #(Bit #(n)) fo_tags;
endinterface
```

new interface type
numeric type parameter declaration
value type parameter declaration
Action method declaration (methods can have arguments)
ActionValue method declaration
Value method declaration (return type is not Action or ActionValue)
Nested sub-interface declaration
Existing interface type

What’s in a Module Declaration?


new module name
module parameters
module interface type
Local value declaration (constant)
Module STATE
(sub-module instantiations)
Local value declaration (function)
Module BEHAVIOR
(rules and/or FSMs)
Module INTERFACE
(method and sub-interface definitions)

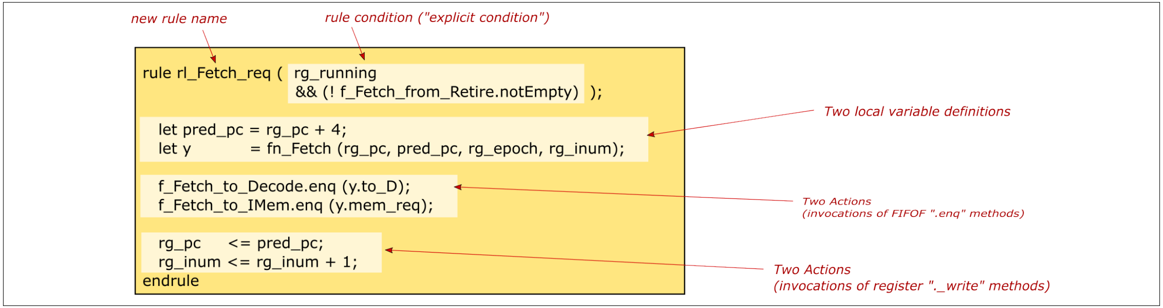
```
module mkBaz #(Bool verbosity) (Baz_IFC #(3, Bool)) ;
  Int #(16) a = 23;

  Reg #(Int #(32)) x <- mkReg (0);

  FIFOF #(Bit #(3)) f_tags <- mkFIFOF;

  function Int #(32) foo (... args ...);
    ...
  endfunction

  rule rl_R1 ( ... explicit  condition ... );
    ...
  endrule

  method m1 ( ...) if ( ... implicit condition ... );
    ...
  endmethod
  interface fo_tags = to_FIFO_O (f_tags) ;
endmodule
```

Extending our Minimal BSV program to define a module with an interface

Examples

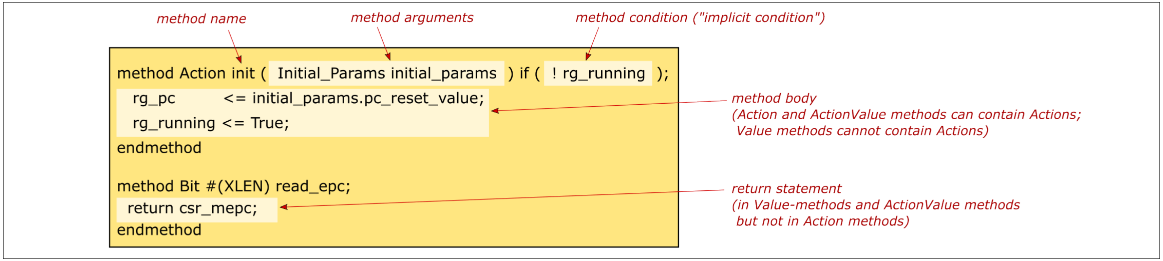
Demo: please see directory `Ex_04_04`, code and `Makefile`

What's in a Rule?


new rule name
rule condition ("explicit condition")
Two local variable definitions
Two Actions
(invocations of FIFOF ".enq" methods)
Two Actions
(invocations of register "._write" methods)

```
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
```

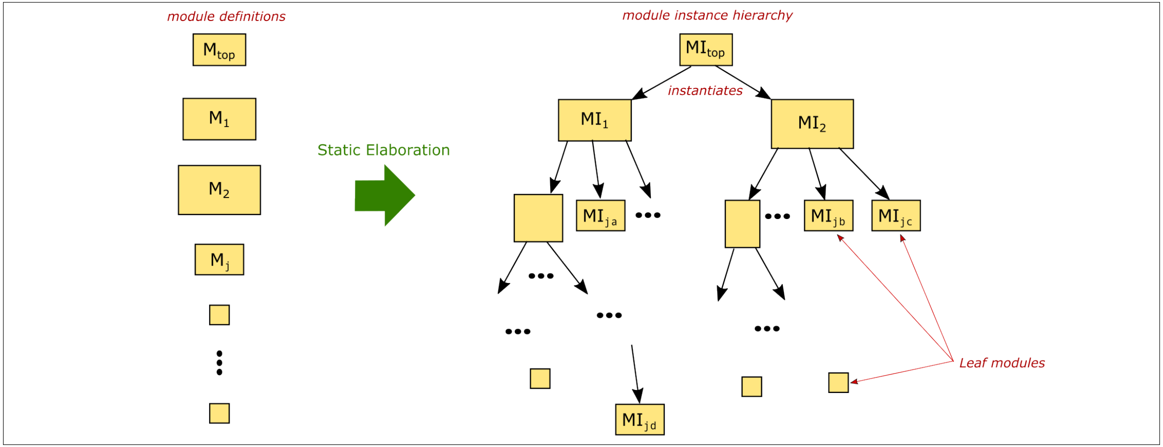
What's in an Interface Definition?



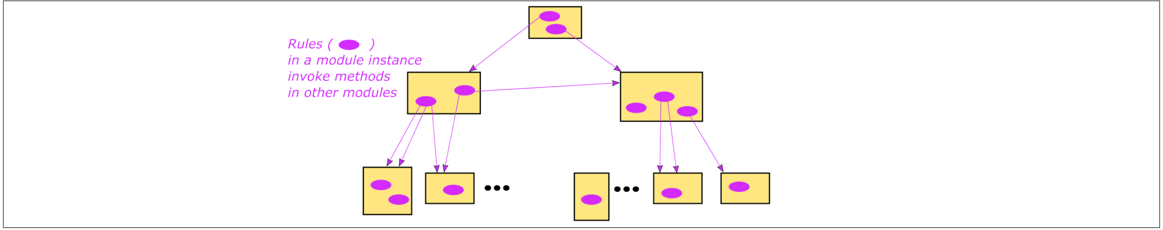
```
method Action init ( Initial_Params initial_params ) if ( ! rg_running );
   rg_pc       <= initial_params.pc_reset_value;
   rg_running <= True;
endmethod

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

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



Module interaction



End