

Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L7: RISC-V: Modules and Interfaces



Reminders

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

```
./Book_BLang_RISCV.pdf
Slides/
  Slides_01_Intro.pdf
  Slides_02_ISA.pdf
  ...
Doc/Installing_bsc_Verilator_etc.{adoc,html}
Exercises/
  Ex_03_B_Top_and_DUT/
  Ex_03_A_Hello_World/
  ...
Code/
  src_Common/
  src_Drum/
  src_Fife/
  src_Top/
  ...
```

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



Flow of information between stages in Drum and Fife

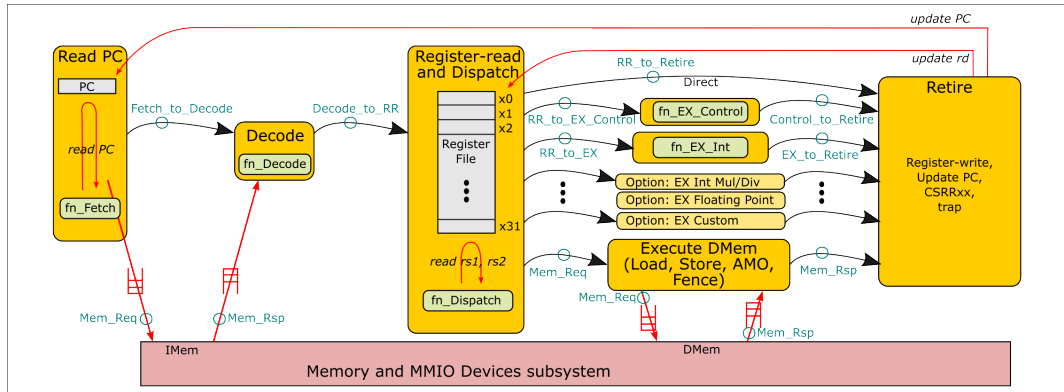
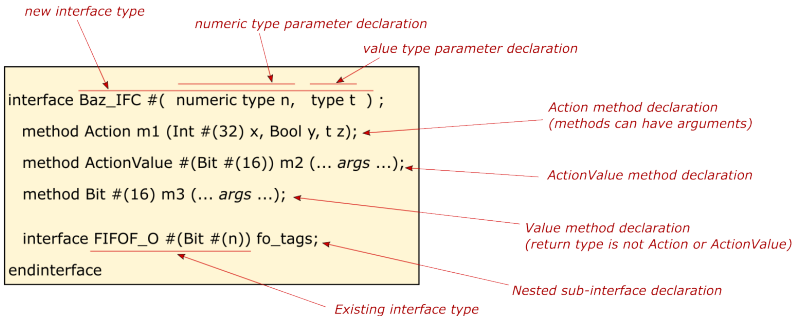


Table of Contents

- 1 Generic Information on Modules and Interfaces
- 2 Registers
- 3 Register Files
- 4 FIFOs

Generic Information on Modules and Interfaces

BSV: What's in an Interface Declaration?

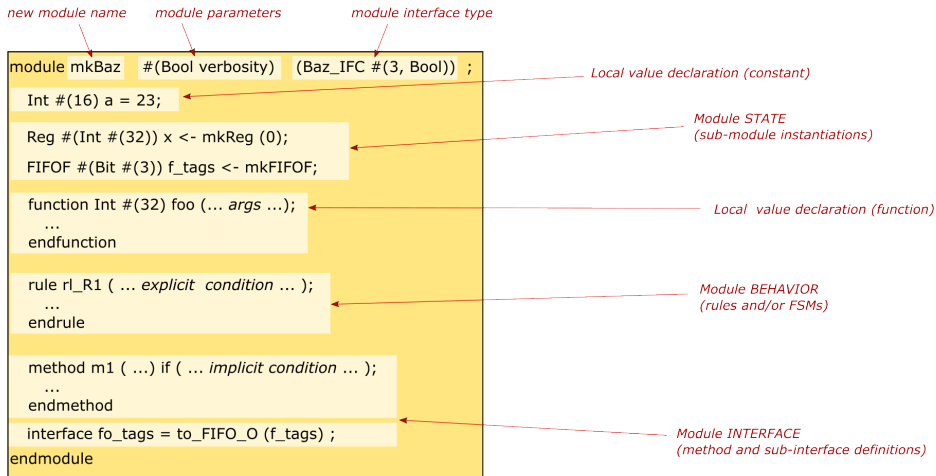


The diagram shows a BSV interface declaration for `Baz_IFC` with several annotations pointing to specific parts of the code:

- `interface`: new interface type
- `numeric type n`: numeric 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
```

BSV: What's in a Module Declaration?



BSV: 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)*

BSV: What's in an Interface Definition?

The diagram shows two BSV code snippets within a yellow box. Red arrows point from text labels to specific parts of the code. The first snippet is a method definition for 'init' with an 'Initial_Params' argument and a condition '! rg_running'. The second snippet is a method definition for 'read_epc' that returns a value. Annotations explain the components: method name, method arguments, method condition, method body, and return statement.

```
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

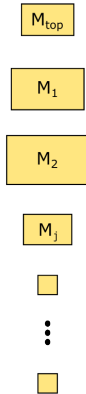
method condition ("implicit condition")

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)

BSV: Static elaboration

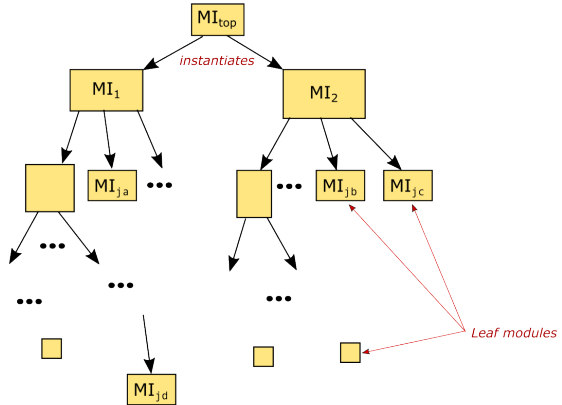
module definitions



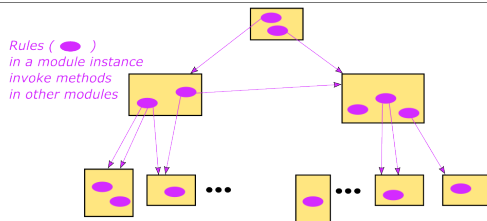
Static Elaboration



module instance hierarchy



BSV: Module interaction



BSV: Generating a Verilog module for a **BSV** module

By default, wherever we have an instantiation of a module FOO, the *bsc* compiler *inlines* the corresponding FOO module definition at that place. As a consequence, there will be no trace of module FOO in the generated Verilog.

We can place a “(* synthesize *)” attribute just before a module declaration:

```
(* synthesize *)  
module mkCPU (CPU_IFC);  
    ...  
endmodule
```

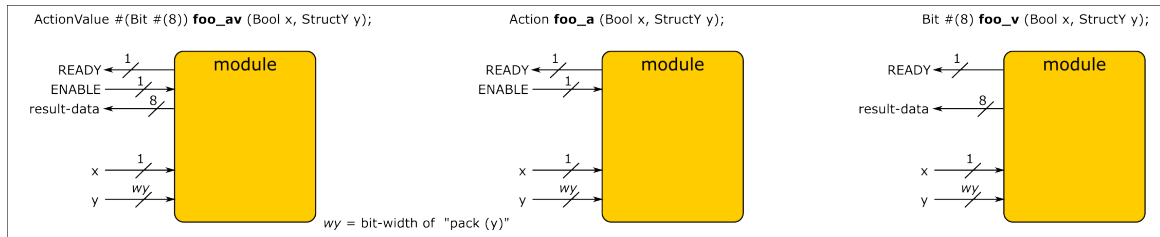
- Makes the *bsc* compiler generate a Verilog module for mkCPU.
- Wherever mkCPU is instantiated, *bsc* will instantiate that Verilog module there, instead of inlining it.

CAVEAT: all **BSV** modules can be inlined, but not all **BSV** modules can be converted into Verilog:

- The inputs and outputs of a Verilog module are all wires, carrying bit-vectors.
- In **BSV**, a parameter/result of a module may have a type with no bit-vector representation; for example, the Integer type (unbounded mathematical integers). Inlining this module may reveal that this parameter is used in a way that does not need a bit-vector.

BSV: Hardware interfaces

General hardware scheme for each **BSV** method, depending on its output type:



- All methods have an output `READY` signal. The method can be invoked only when `READY=1`.
- Side-effecting methods (output types `Action`, `ActionValue # (t)`) have an input `ENABLE` signal. The side-effect only occurs on clock edges when `ENABLE=1`.
- A method argument is an input bus.
- A method result is an output bus.

`READY/ENABLE` signaling is fundamental for implementing **BSV**'s compositional rule-based behavioral semantics (rule conditions, method conditions, rule atomicity).

BSV: Hardware interfaces: modeling Verilog/SystemVerilog ports

One can attach *attributes* “always_ready” and “always_enabled” to a **BSV** method, eliminating the READY and/or ENABLE signals, respectively (see documentation).

Note: the *bsc* compiler will check that these attribute-assertions are true.

It is easy to model Verilog/SystemVerilog module ports:

- A **BSV** method with 0 arguments (no inputs) and 1 result (output bus) that is always-ready becomes a simple Verilog/SystemVerilog output port.
- A **BSV** Action method (no output) with 1 argument (input bus), that is always-ready and always-enabled becomes a simple Verilog/SystemVerilog input port.

These are frequently used for **BSV** modules that must interface with external Verilog/SystemVerilog IP.

Registers

BSV library: Registers

Registers are the simplest “state elements” (entities that have state, *i.e.*, that store or retain values over time).

In **BSV**, registers are just pre-defined modules.

(In Verilog/System Verilog, registers are special primitives, not modules.)

The pre-defined “Reg” interface is an interface with two methods:

```
interface Reg #(t);  
  method t _read();  
  method Action _write (t x);  
endinterface
```

“mkReg” and “mkRegU” are pre-defined **BSV** modules for registers (instantiated just like user-defined modules).

Examples:

```
Reg #(Bit #(XLEN))  rg_pc <- mkReg (0);    // 0 is the reset value  
  
Reg #(Bit #(XLEN))  rg_pc <- mkRegU;       // unspecified reset value
```

BSV registers are strongly-typed.

A Reg #(Bit #(XLEN)) cannot contain a Bool value, or a Mem_Req value, or ... any value whose type is not Bit #(XLEN) (unlike Verilog/System Verilog, where all registers just hold bit-vectors).

BSV: Syntactic shorthands for reading and writing registers

Because register access is so frequent, **BSV** provides some syntactic shorthands for register-method invocations:

<code>rg_pc + 4</code>	is shorthand for	<code>rg_pc._read + 4</code>
<code>rg_pc <= v</code>	is shorthand for	<code>rg_pc._write (v);</code>

Example:

<code>rg_pc <= rg_pc + 4;</code>	is shorthand for	<code>rg_pc._write (rg_pc._read + 4);</code>
-------------------------------------	------------------	--

Register Files

BSV library: Register Files

A Register File module is an array of n registers.

Unlike a collection of n individual registers, where we can simultaneously read and write all of them, a register file is organized and accessed like a memory, through a read and write interface.

- To read the j^{th} register, we must give it the register index j (like a “memory address”).
The standard **BSV** register file module has 5 read-ports, *i.e.*, up to 5 registers can be read simultaneously (no matter how many registers are in the register file).
Note: in RISC-V, we need to read `rs1` and `rs2` simultaneously.
- To write the j^{th} register, we must give it the register index j and the value v to be written.

A synthesis tool might implement a register file using an SRAM, instead of individual registers and muxes.

BSV library: Register Files

(See “Bluespec Compiler (BSC) Libraries Reference Guide”, Section “3.1.1 Register File”.)

The RegFile interface:

```
interface RegFile #(type index_t, type data_t);  
  method Action upd (index_t addr, data_t d);      // write a register  
  method data_t sub (index_t addr);                // read a register  
endinterface: RegFile
```

“index_t” is the type for the index (for RISC-V, with 32 registers, we use Bit#(5)).

“data_t” is the type of value stored in each of the registers (for RISC-V, this is Bit#(XLEN)).

BSV register files are strongly typed, just like individual registers.

(In Verilog/SystemVerilog, where register files just hold bit-vectors.)

“mkRegFile” and “mkRegFileFull” are pre-defined **BSV** modules for register files (instantiated just like user-defined modules).

Examples:

```
RegFile #(Bit #(5), Bit #(XLEN)) gprs <- mkRegFile (1, 31);    // 31 registers; addrs 1..31  
  
RegFile #(Bit #(5), Bit #(XLEN)) gprs <- mkRegFileFull;        // 32 registers; addrs 0..31
```

FIFOs

BSV library: FIFOs

FIFO modules hold *queues* of items.

We can *enqueue* an item on one end of a FIFO.

At the other end of the FIFO, we can examine the first element, and/or *dequeue* (remove) an item.

A pre-defined FIFO interface in the **BSV** library:

```
interface FIFOF #(t);
  method Bool    notEmpty;
  method Bool    notFull;
  method t       first;
  method Action  deq;
  method Action  enq (t x);
  method Action  clear;           // Empty the FIFO
endinterface
```

“t” specifies the type of the elements in the FIFO.

BSV FIFOs are strongly typed. *E.g.*, a FIFOF #(Mem_Req) cannot hold Mem_Rsps.

The *first* and *deq* methods (“output side”) are enabled only when there is at least one item in the FIFO (it is not empty).

The *enq* method (“input side”) is enabled only when there is space available in the FIFO (it is not full).

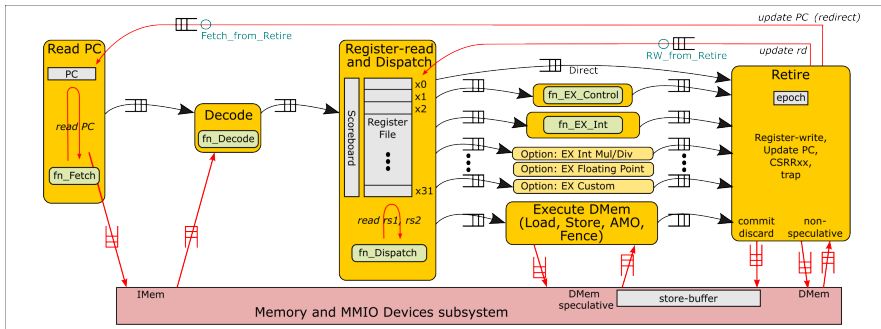
BSV library: FIFO modules

“mkFIFO” and “mkSizedFIFO” are pre-defined **BSV** modules for FIFOs (instantiated just like user-defined modules).
Examples:

```
// "small" capacity, used like single-register buffers
FIFO #(Mem_Req) f_to_IMem    <- mkFIFO;
FIFO #(Mem_Rsp) f_from_IMem <- mkFIFO;

// Higher capacity
FIFO #(RR_to_Retire) f_RR_to_Retire <- mkSizedFIFO (8);
```


RISC-V (Fife): Balancing fork-join pipeline paths



Suppose we use a small (capacity 1) FIFO *fd* on the direct path from Fetch to Decode. After Fetch enqueues one item into *fd* and to memory, it will get stuck—*fd* is full, and will remain full until Decode receives the response from memory, when it will dequeue *fd*.

For Fetch to continue, *fd* needs higher capacity—equal to the number of items that may be “in flight” on the path to memory and back (memory requests, IMem, memory-responses). In other words, the two paths must be “balanced”.

BSV library: A useful help-function

A useful function combining the `first` and `deq` methods:

```
function ActionValue #(t) pop (FIFO #(t) fifo);  
  actionvalue  
    let x = fifo.first;  
    fifo.deq;  
    return x;  
  endactionvalue  
endfunction
```

So

```
let y <- pop (fifo);
```

is equivalent to

```
let y fifo.first;  
fifo.deq;
```

BSV library: SemiFIFO interfaces for each end of a FIFO

When a FIFO is used for communication between two modules, it is enqueued in one module and dequeued in the other. Conceptually, each module uses only one end of the FIFO.

For this, it is useful to define interfaces representing one end of a FIFO:

Input end (where we enqueue):

```
interface FIFO_I #(t);  
  method Bool notFull();  
  method Action enq (t x);  
endinterface
```

Output end (where we dequeue), and associated pop_0 function:

```
interface FIFO_O #(t);  
  method Bool notEmpty();  
  method t first();  
  method Action deq();  
endinterface
```

```
function ActionValue #(t) pop_0 (FIFO_O #(t) fo);  
  actionvalue  
    let x = fo.first;  
    fo.deq;  
    return x;  
  endactionvalue  
endfunction
```

BSV library: transforming a FIFOF interface into a FIFOF_0 interface

```
function FIFOF_0 #(t) to_FIFOF_0 (FIFOF #(t) f);  
  interface FIFOF_0 #(Mem_Req) fo_IMem_req;  
    method Bool notEmpty();  
      return f.notEmpty;  
    endmethod  
  
    method t first();  
      return f.first;  
    endmethod  
  
    method Action deq();  
      f.deq;  
    endmethod  
  endinterface  
endfunction
```

We can write a similar function transforming a FIFOF interface into a FIFOF_I interface.



Exercise break

Please see Appendix E, Section Ex-07-A-Modules-Interfaces.

BSV library: Connecting FIFOs

```
interface Fetch_IFC;  
  interface FIFOF_O #(F_to_D)  
    fo_Fetch_to_Decode;  
  ...  
endinterface
```

```
interface Decode_IFC;  
  interface FIFOF_I #(F_to_D)  
    fi_Fetch_to_Decode;  
  ...  
endinterface
```

In the parent CPU module:

```
module mkCPU (CPU_IFC);  
  ...  
  // Instantiate Fetch and Decode stages  
  Fetch_IFC  stage_F  <- mkFetch;  
  Decode_IFC stage_D  <- mkDecode;  
  ...  
  // Connect the Fetch_to_Decode flow  
  Empty eifc <- mkConnection (stage_F.fo_Fetch_to_Decode, stage_D.fi_Fetch_to_Decode);  
  ...  
endmodule
```

BSV library: mkConnection

mkConnection is just another module, and could easily be written by the user if it were not pre-defined for the FIFOF_0 and FIFOF_I interfaces.

```
module mkConnection #(FIFOF_0 #(Fetch_to_Decode) f,      // module argument
                      FIFOF_I #(Fetch_to_Decode) d)      // module argument
                      (Empty);                            // module interface

  rule rl_connect;
    let x = f.first;
    f.deq;
    d.enq (x);
  endrule
endmodule
```

In general, whenever there is pair of interface types that can and are frequently connected, we recommend defining mkConnection for those interfaces.

BSV library: mkConnection

Shorthand: in this line:

```
Empty eifc <- mkConnection (stage_F.fo_Fetch_to_Decode, stage_D.fi_Fetch_to_Decode);
```

Because this is an empty interface (has no methods) and is therefore never used, in **BSV** we can just omit the “Empty eifc <-” part:

```
mkConnection (stage_F.fo_Fetch_to_Decode, stage_D.fi_Fetch_to_Decode);
```


End