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

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

Rishiyur S. Nikhil

L9: **BSV**: Finite State Machines/StmtFSM

bluespec

Reminders

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

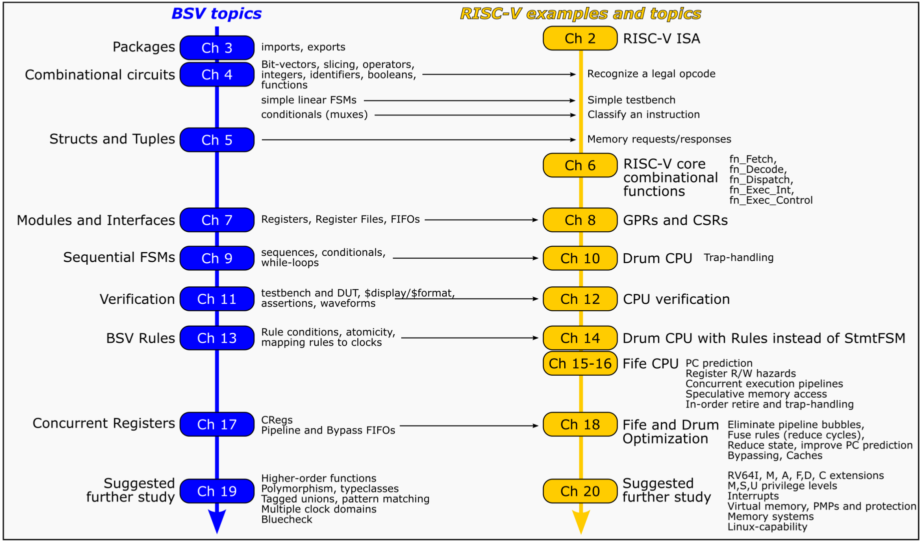
```
./Book_BLang_RISCV.pdf
 Slides/
     Slides_01_Intro.pdf
     Slides_02_ISA.pdf
     ...
 Exercises/
     Ex-03-A-Hello-World/
     Ex-03-B-Top-and-DUT/
     ...
 Code/
     src_Top/
     src_Drum/
     src_Fife/
     src_Common/
     ...
 Doc/Installing_bsc_Verilator_etc.{adoc,html}
```

- 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.

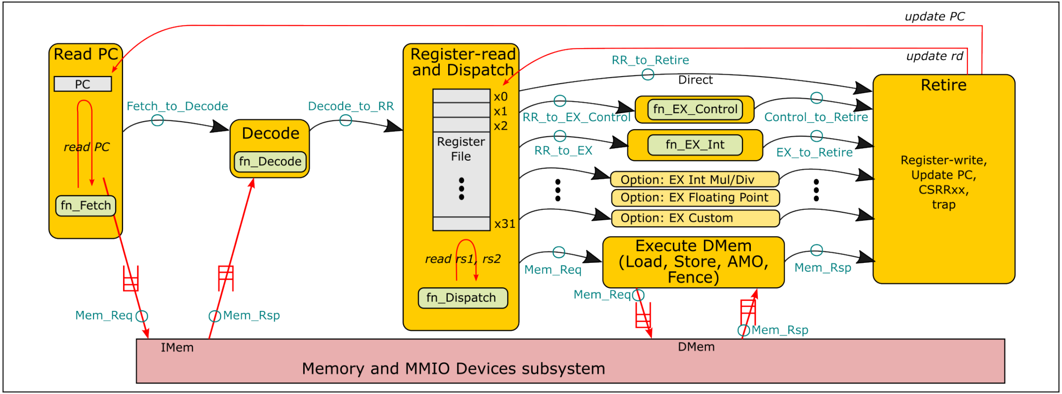
To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

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

Chapter Roadmap



Flow of information between stages in Drum and Fife

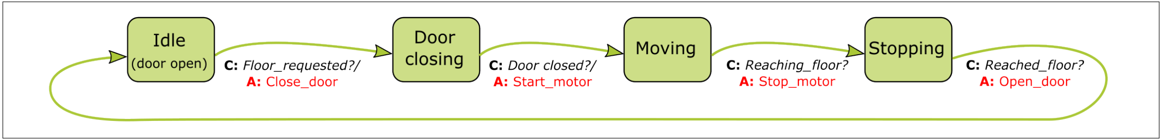


Classical Finite State Machines (FSMs): Bubble-and-Arrow diagrams

An FSM is a *process*, a behavior that evolves over time.

A classical notation for describing/specifying FSMs is the bubble-and-arrow diagram.

Here is a (greatly over-simplified) FSM spec for controlling an elevator:

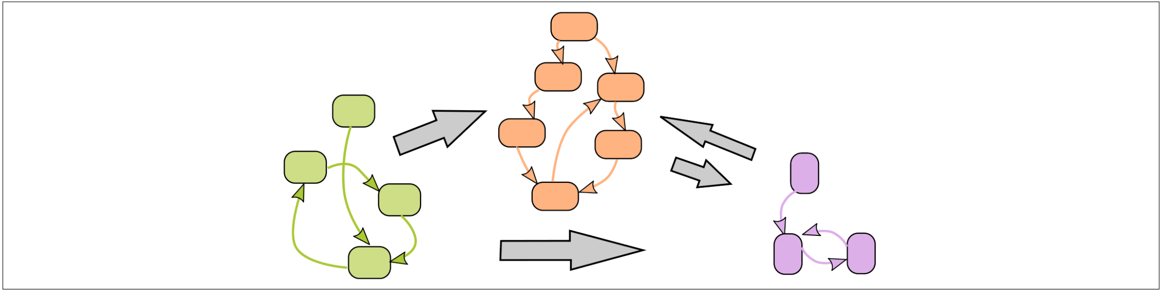


Each state of the process is depicted by a bubble. Each arrow depicts a transition to another state enabled by a condition (**C:**) and performing an action (**A:**).

The RISC-V instruction-execution flow diagram can be interpreted as an FSM bubble-and-arrow diagram, and implemented that way. This is exactlly what Drum is.

Sequential *vs.* Concurrent FSMs

Most hardware systems (except for extremely simple ones) are best viewed as *communicating, concurrent FSMs*:



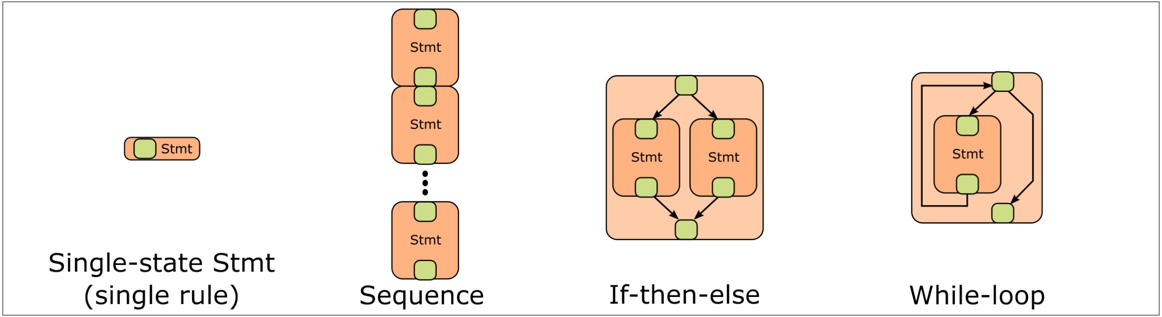
Multiple FSMs communicate with each other (*via* registers, register files, FIFOs, ...).

Note: theoretically multiple FSMS are equivalent to a single FSM, but the size of such a single-FSM description can be MUCH larger. This is because we have to describe all possible combinations of states where when one FSM is in state A_j and and another FSM is simultaneously in state B_k.

BSV: StmtFSM: sub-language for specifying structured FSMs

StmtFSM is a sub-language in **BSV** for specifying structured FSMs.

- We start with an expression of type `Action`.
- Then, we *compose* larger FSMs from smaller FSMs using sequencing, if-then-else and while-loop constructs.



Each construct produces an expression of type `Stmt`.

BSV: A single-state FSM

Example:

```
seq
   action
      rg_pc <= rg_pc + 4;           // Assignment to a register
      f_F_to_D.deq;                 // Dequeue a fifo
      f_D_to_RR.enq (v);            // Enqueue into a fifo
      $display ("Hello, World!");  // Print something (in simulation only)
   endaction
endseq
```

The `seq-endseq` construct is an expression of type `Stmt`.

All actions in an `action-endaction` block take place *simultaneously* and *instantaneously*, no matter the textual order in which they are written.

(In hardware, all their ENABLE signals are asserted simultaneously, and they are all performed on the next clock signal.)

BSV: Action-blocks can contain name-bindings

Example:

```
action
   Bit #(XLEN) next_pc = rg_pc + 4;
   rg_pc <= next_pc;
   $display ("Next PC is %08h", next_pc);
   ...
   let y <- pop_o (to_FIFOF_O (f_mem_rsps));
   ...
   $display ("mem_rsp is ", fshow (y));
endaction
```

```
seq
   ... Action or Stmt ...
   ... Action or Stmt ...
   ...
   ... Action or Stmt ...
endseq
```

Note: A sequence of *n* actions may not complete in *n* clocks.
Actions execute according to usual **BSV** semantics—an action may implicitly stall (be paused) until all the methods it invokes are READY.

Library-provided actions to explicitly pause an FSM:

```
seq
   ...
   await (... Bool expr ...)          // pause until some condition
   delay (... numeric expr ...)       // pause for n cycles
   ...
endseq
```

BSV: Conditional and loop flows

Conditional flows:

```
if (... Bool expr ...)
   ... Action or Stmt ...
else
   ... Action or Stmt ...
```

Note: “if (b) ... else ...” is used in **BSV** in two different ways:

- In computation, where they represent hardware MUXes (multilexers)
- In `StmtFSM`, where they represent alternative temporal FSM flows

But there is no ambiguity, because these are distinct contexts.

Loop flows:

```
while (... Bool expr ...)
   ... Action or Stmt ...
```

BSV: Instantiating an FSM from a `Stmt` specification

```
mkAutoFSM (... argument expression of type Stmt ...);
```

- This statement occurs inside a module, along with other sub-module instantiations.
- This statement instantiates a sub-module whose behaviour is specified by the `Stmt`
- The FSM starts running as soon as the system comes out of reset.
- If the FSM reaches its end state, it executes a `$finish()` to stop simulation.

  Note:

  - (It may not reach the end state if it has an infinite while-loop.)
  - (It may not reach the end state if some action gets stuck, *e.g.*, trying to dequeue an empty FIFO.)

BSV: `StmtFSM` final comments

- `StmtFSM` is frequently used in testbenches for sequentially producing test stimulus and squentially consuming outputs (although, in Drum, we also use it in the design).
- `StmtFSM` can only express *structured* processes (composed by nesting `seq-endseq`, `if`, `while`).

  For more complex flows, and more fine-grain concurrency, we can directly use **BSV** *Rules* (*e.g.*, we do this in Fife).

  [See also "Section 9.11 Historical Note about Structured Programming" for connections to modern software programming languages.]

- There are more ways to create `Stmt`'s; see *bsc* library document.
- There are more module-constructors (than `mkAutoFSM`); see *bsc* library document.

End