Riscy Processors

A collection of open-sourced RISC-V processors

Andy Wright, Sizhuo Zhang,
Thomas Bourgeat, Murali Vijayaraghavan,
Jamey Hicks, Arvind

Computation Structures Group, CSAIL, MIT

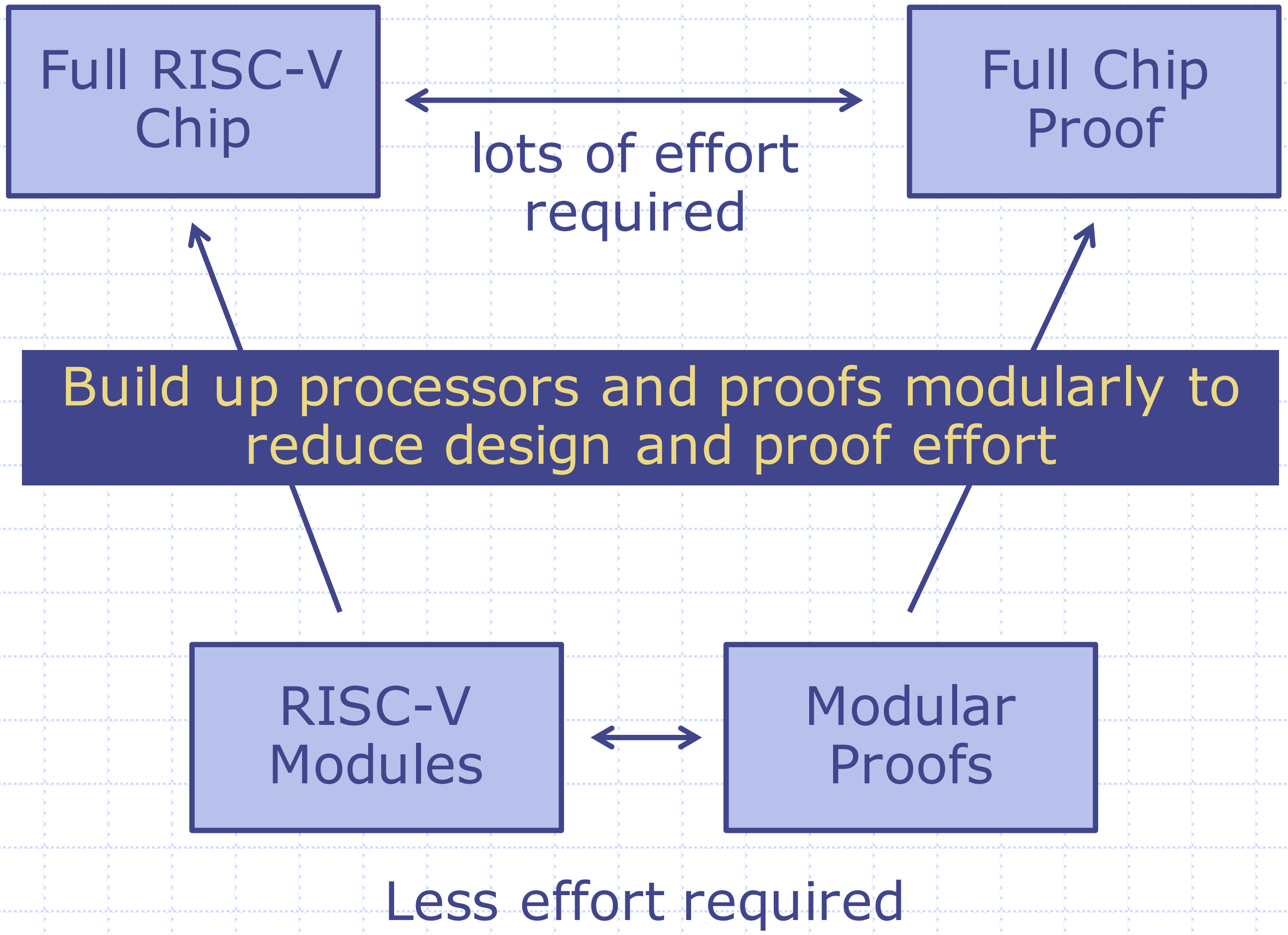
4th RISC-V Workshop

MIT's Riscy Expedition

- Advisors Profs. Arvind and Adam Chlipala
- Motivations:
  - Formal Specification
  - Formally Verified Processor Implementations
  - Memory Consistency Models
  - Accelerators
  - Microarchitectural Exploration
  - ASIC Synthesis

$\Rightarrow$ need flexible RISC-V Implementations

Chips with Proofs

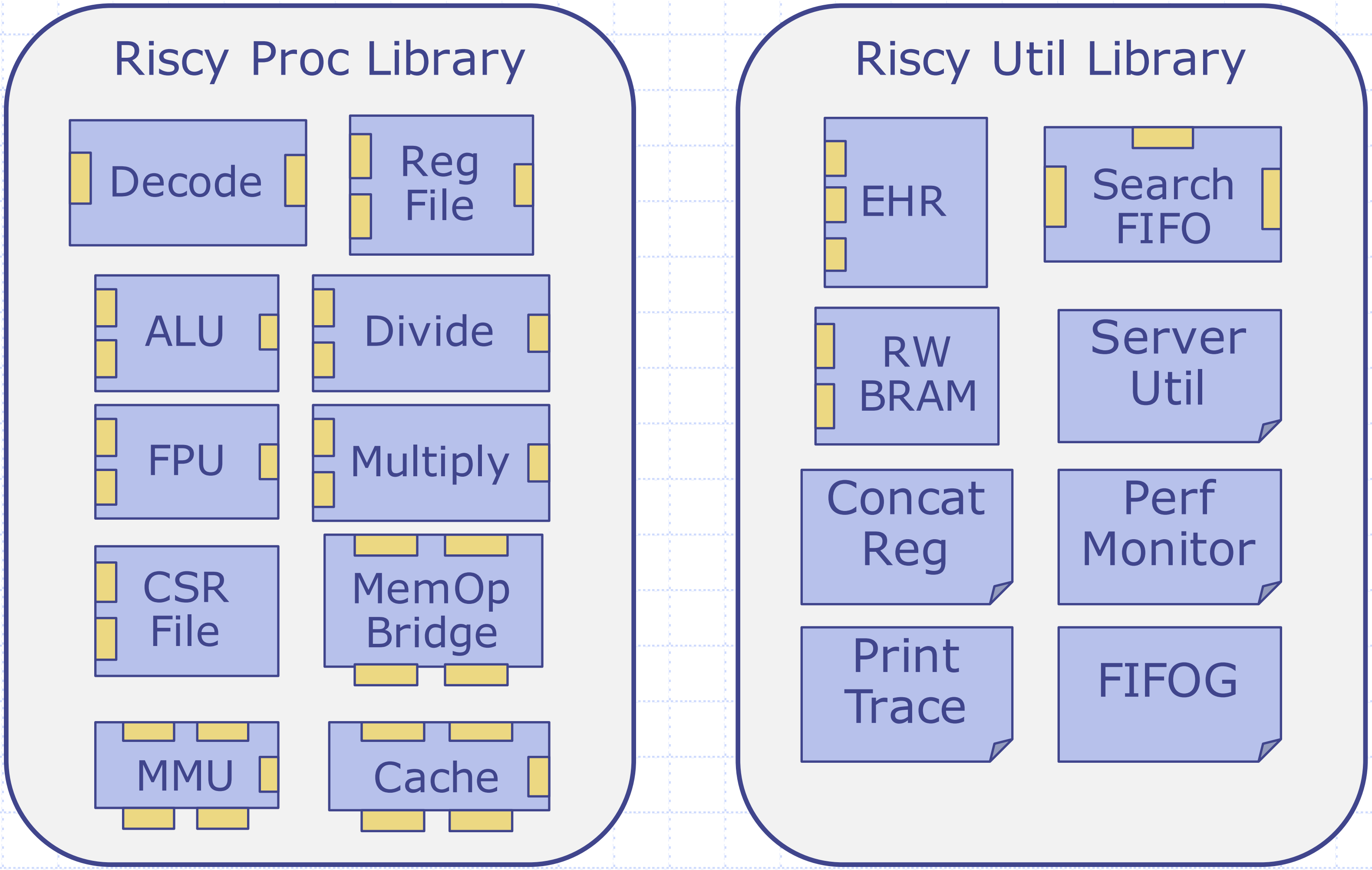


Currnet Riscy Offerings

- Building Blocks for Processor Design:
  - Riscy Processor Library
  - Riscy BSV Utility Library
- Reference Processor Implementations:
  - Multicycle
  - In-Order Pipelined
  - Out-of-Order Execution (*to be released shortly*)
  - All implementations boot Linux w/ paged virtual memory
- Infrastructure:
  - Connectal
  - Tandem Verification

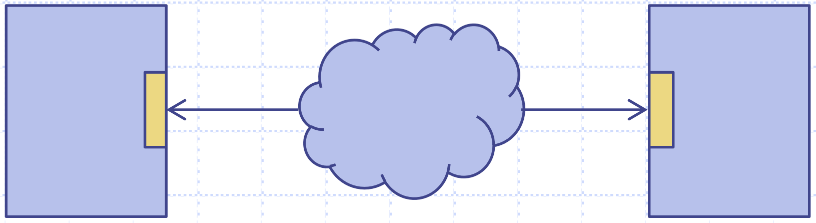
A flexible way of designing processors leveraging Bluespec System Verilog (BSV)

Processor Building Blocks



Connecting Modules

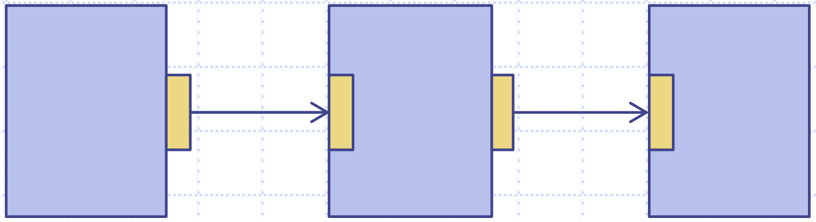
Method Definition



Connections through glue logic

Modules can act as glue logic

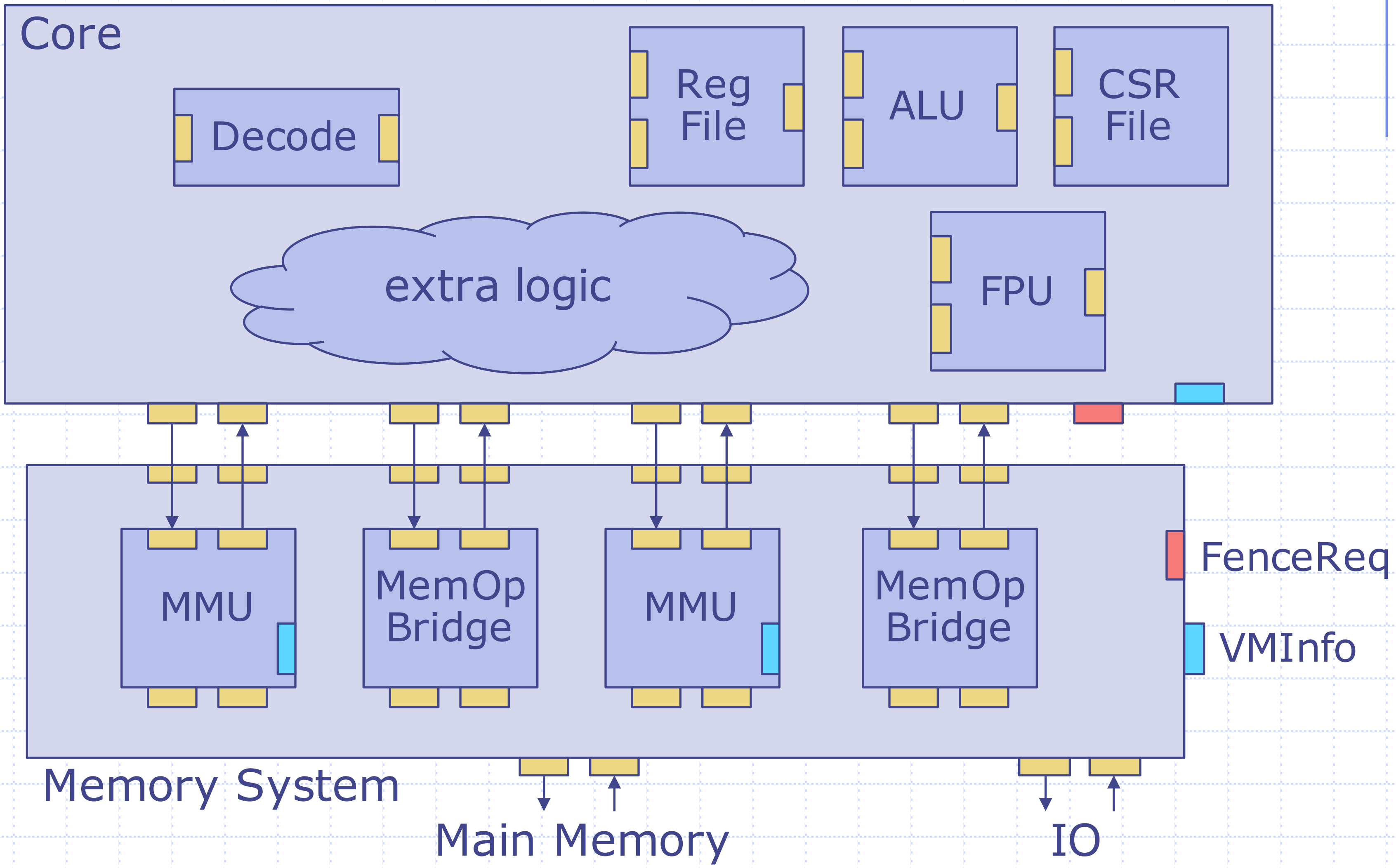
External Method Call



Direct connection

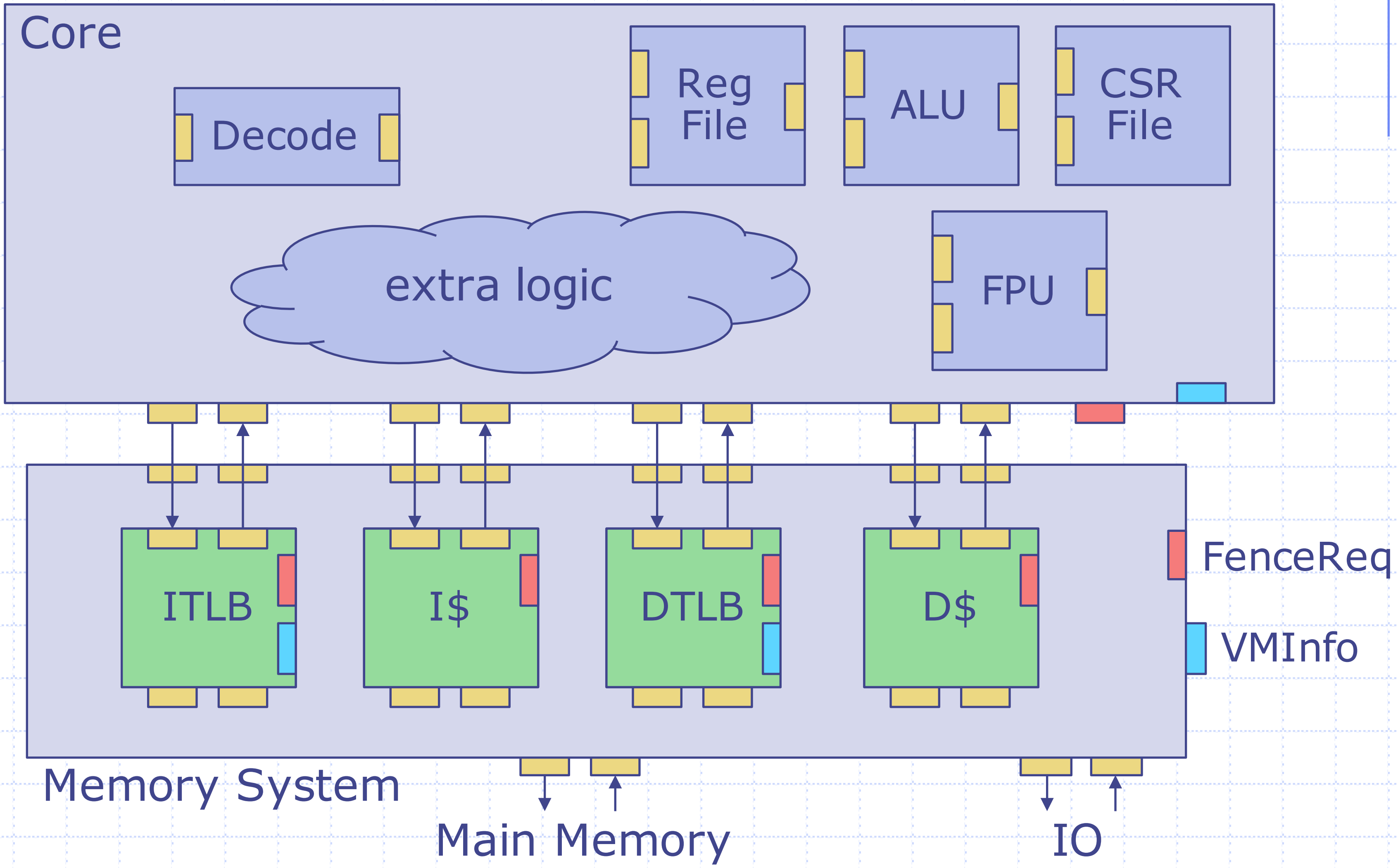
Modules with external method calls are *open modules.* Those without are *closed modules*.

Initial Processor
Multicycle Basic – FSM without Caches



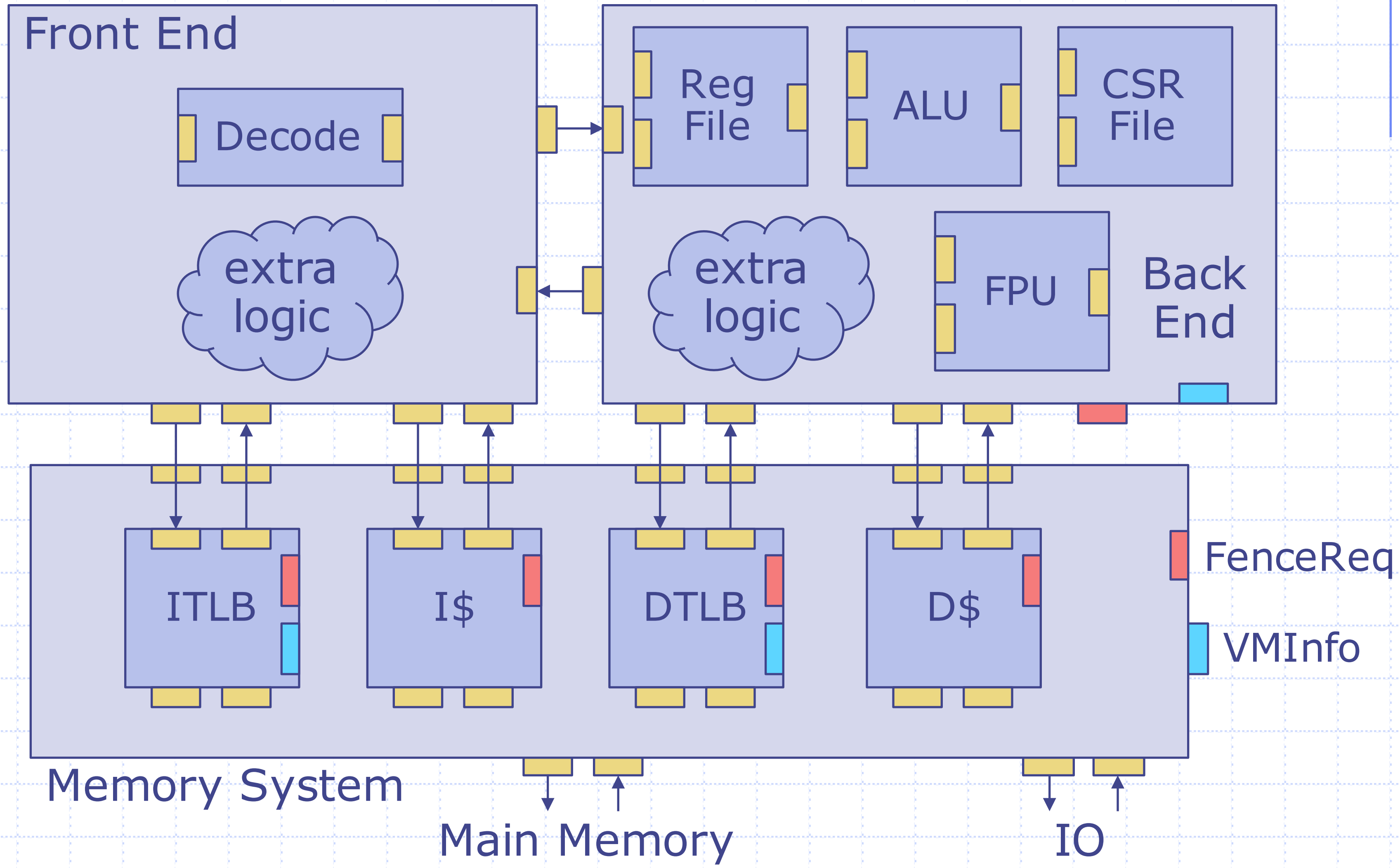
Initial Processor

Multicycle – Adding Caches and TLBs



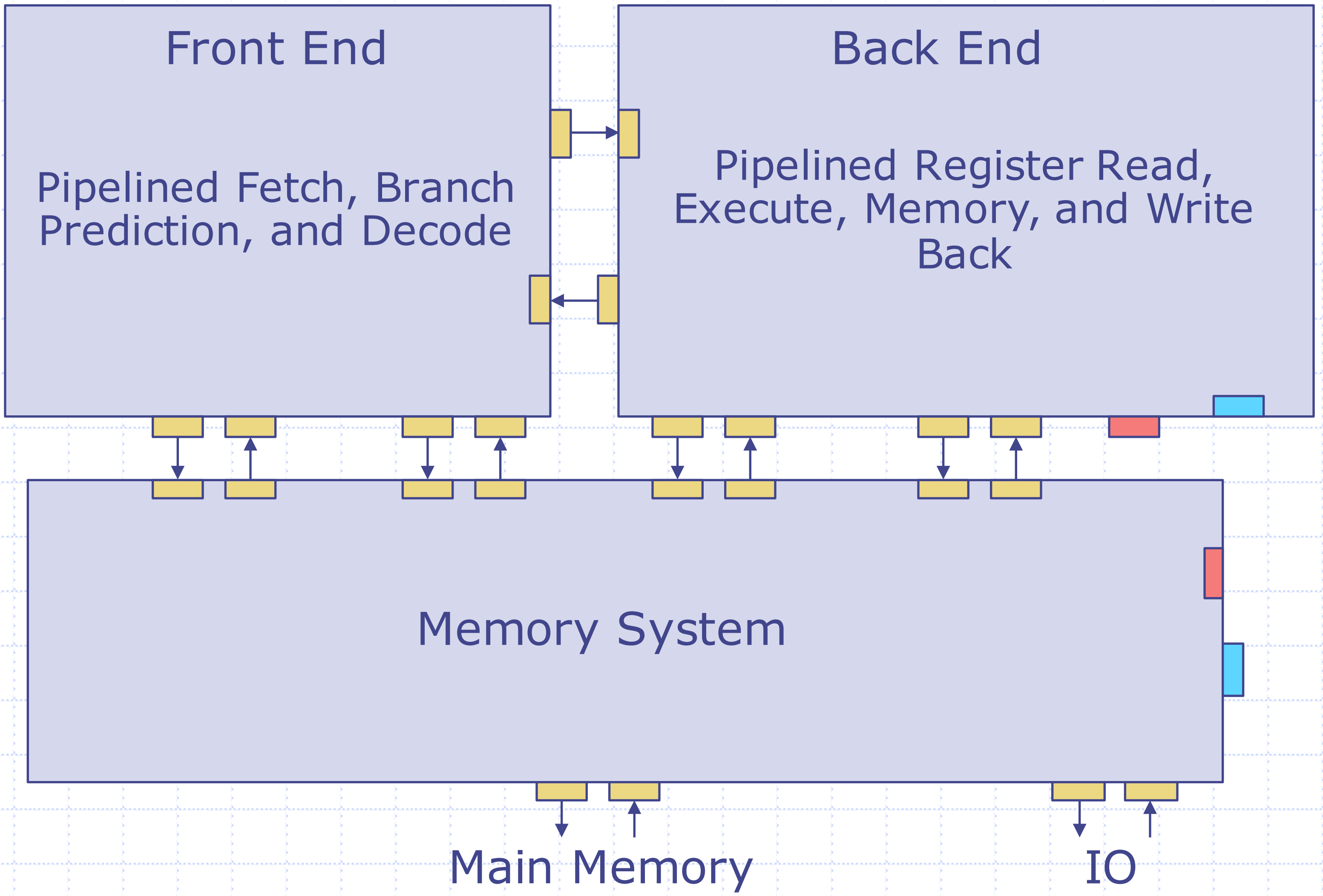
Initial Processor

Multicycle Split – Front and Back FSMs



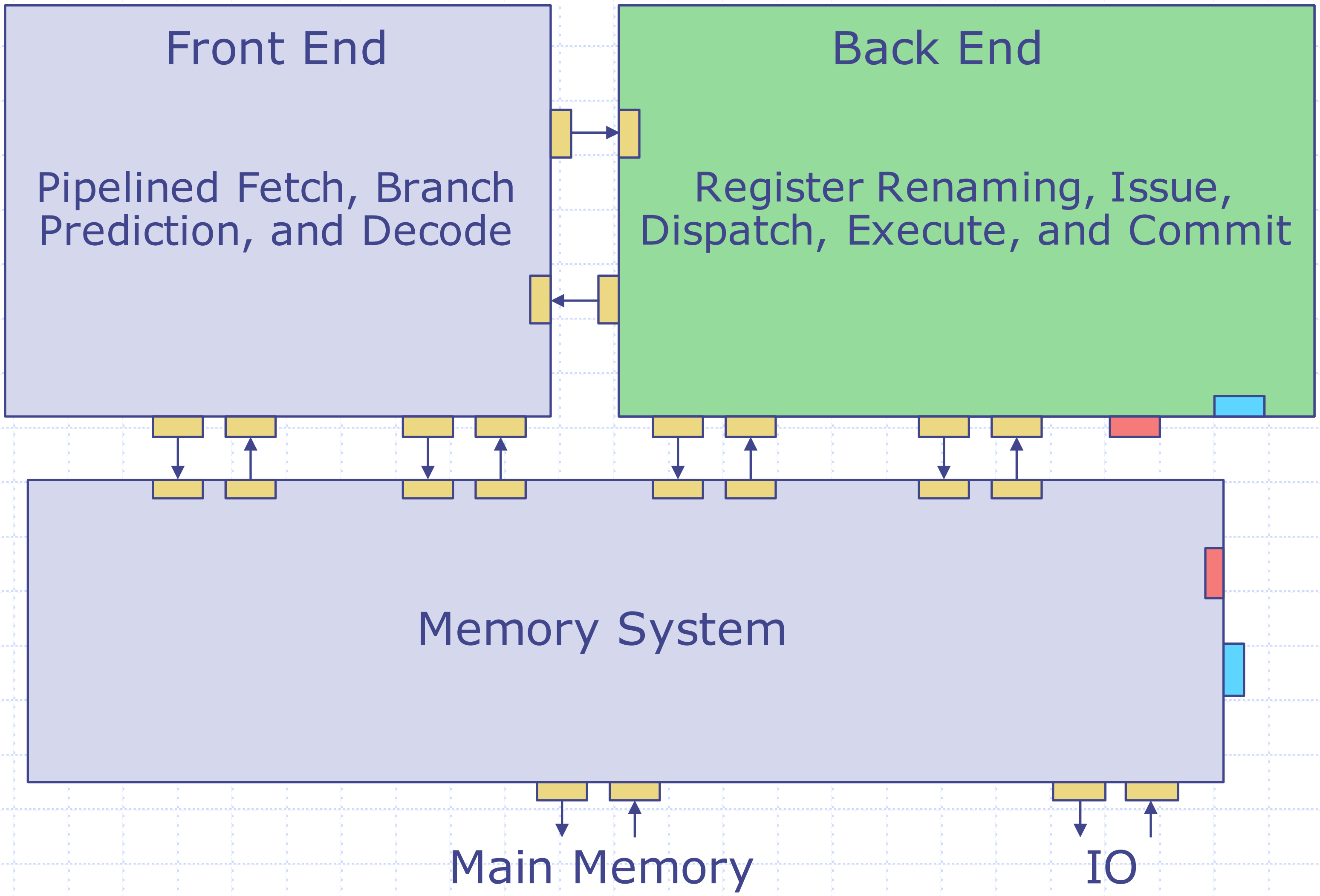
Processor Implementations

In-Order Pipelined

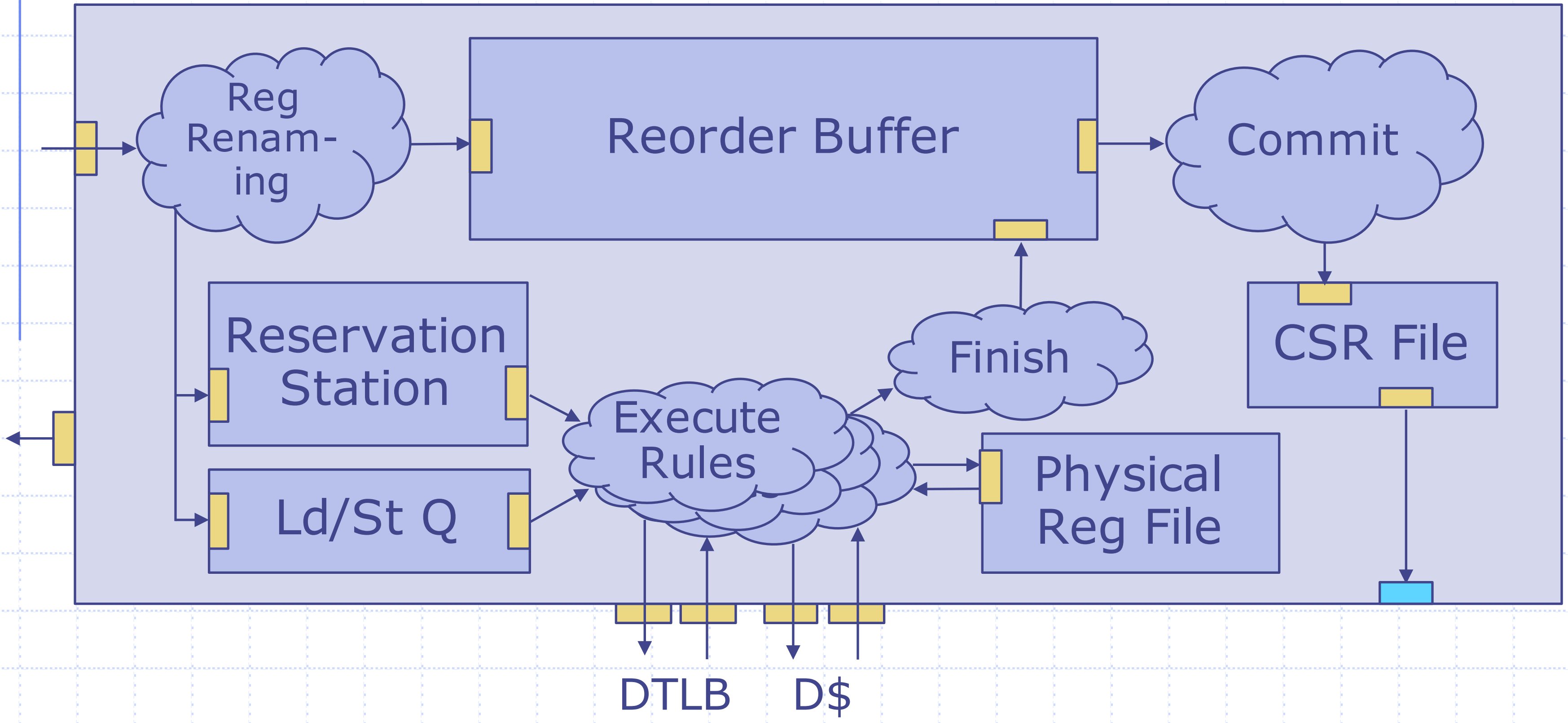

Front End
Pipelined Fetch, Branch Prediction, and Decode
Back End
Pipelined Register Read, Execute, Memory, and Write Back
Memory System
Main Memory
IO

Processor Implementations

Out-of-Order Execution


Front End
Pipelined Fetch, Branch Prediction, and Decode
Back End
Register Renaming, Issue, Dispatch, Execute, and Commit
Memory System
Main Memory
IO

Processor Implementations
Out-of-Order Execution – Back-End



How is modular design possible?

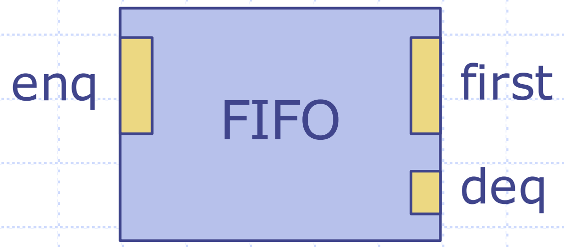
- RTL modules are not modularly refinable under composition
  - Implementation details of one module may put additional constraints on another module
- Bluespec System Verilog supports composability through:
  - Interface method abstraction
  - Implicit guards on methods
  - Guarded Atomic Actions

Bluespec System Verilog (BSV)

- High-Level Synthesis language
  - Execution model built upon guarded atomic actions (called *rules*)
  - Rule can only change the state of a module if its *guard* (or condition) is true
- The compiler adds logic to specify when rules fire
  - Rules must have its explicit guard and all implicit guards for module method calls satisfied
  - Rules must appear to fire atomically (or sequentially – one-rule-at-a-time semantics)

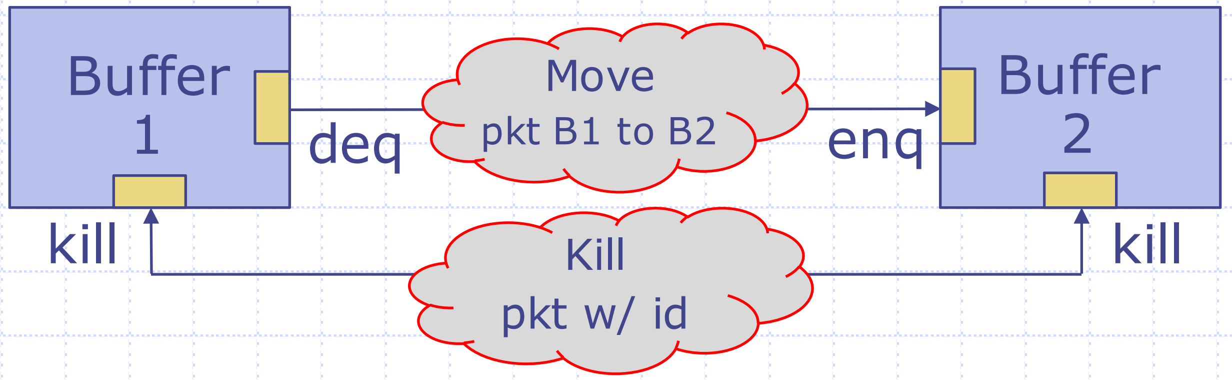
FIFO Interface

```
Interface FIFO#(type t);
  method Action enq(t x);
  method t first;
  method Action deq;
endinterface
```



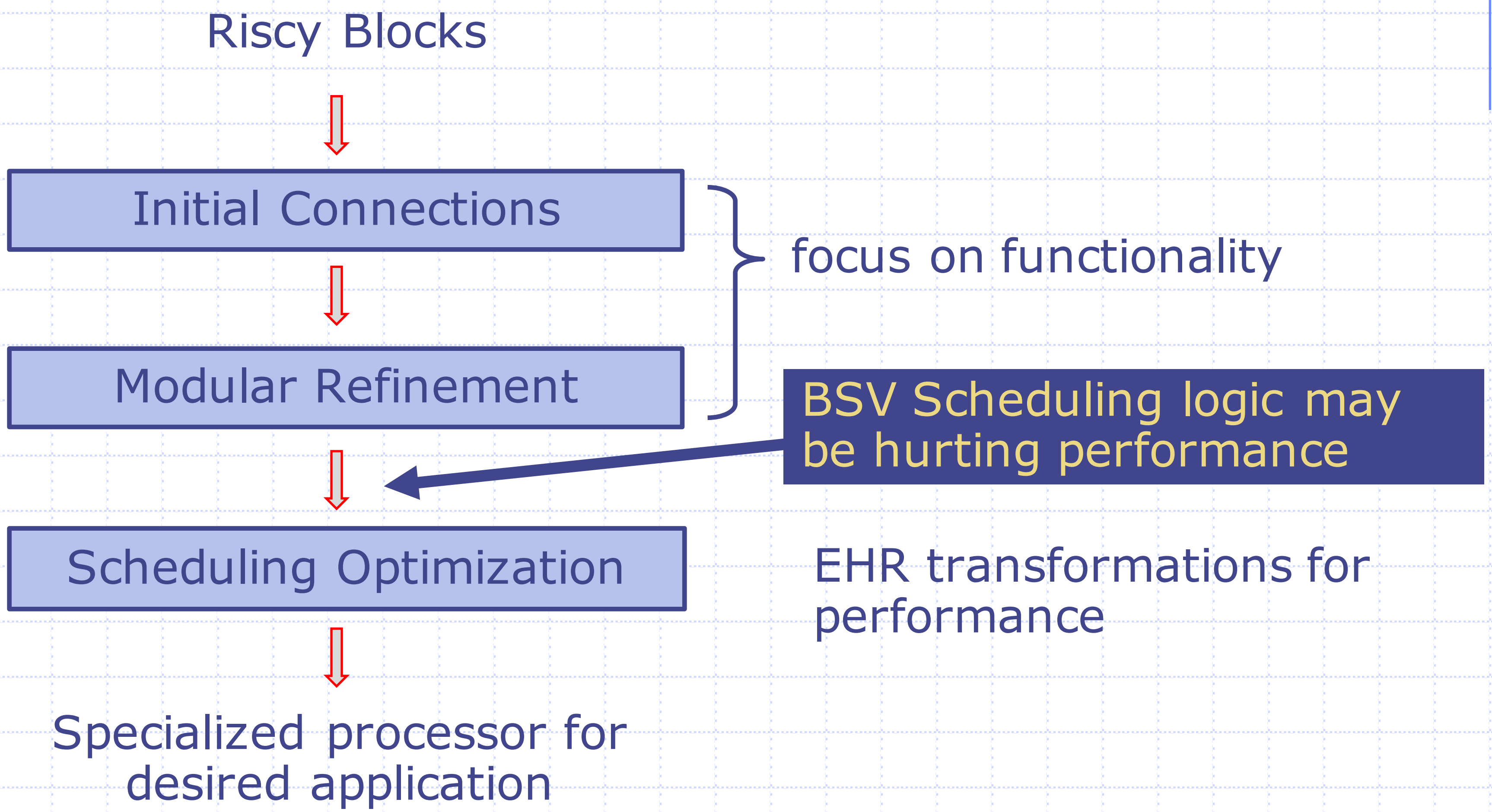
Implicit guards prevent enqueuing into a full FIFO or dequeuing from an empty FIFO

Connecting Modules with Rules

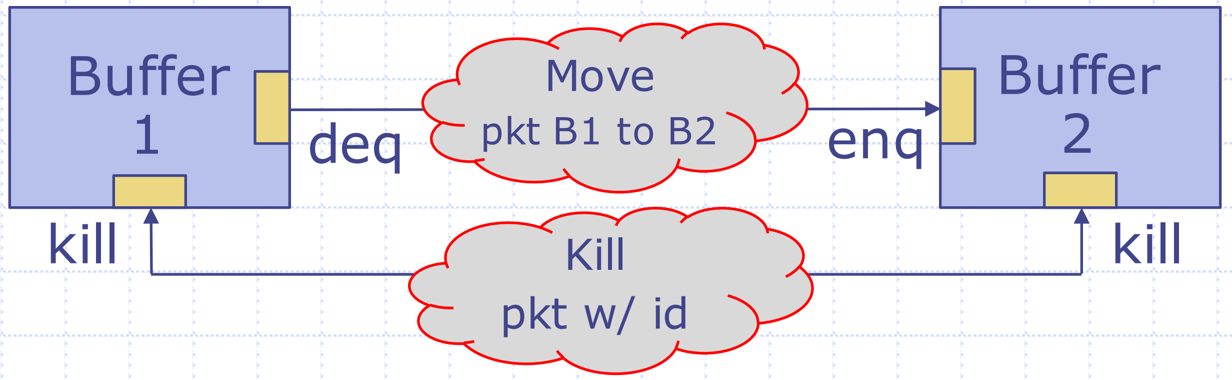


- A packet will be killed either before the move in buffer 1, or after the move in buffer 2
- Suppose neither buffer kills the current enqueuing/dequeuing packet
  - *Atomicity bug* in Verilog
  - Bluespec compiler introduces extra logic to prevent concurrent move and kill

Processor Design Flow


Riscy Blocks
Initial Connections
Modular Refinement
focus on functionality
BSV Scheduling logic may be hurting performance
Scheduling Optimization
EHR transformations for performance
Specialized processor for desired application

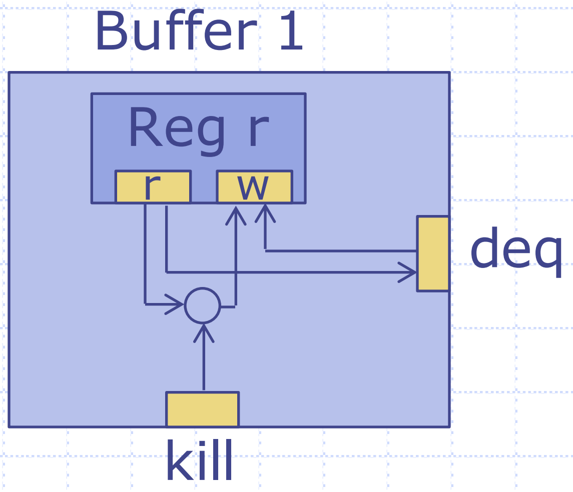
Scheduling Optimization
Introduction



- To enable Move and Kill to fire concurrently, they need an apparent ordering (or schedule)
  - If the buffer's methods don't support the schedule, use EHR refinement within the buffers to achieve the necessary scheduling

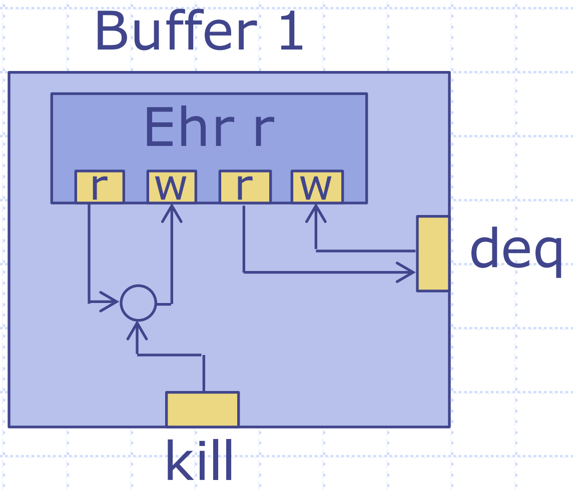
Scheduling Optimization

```
method deq if (r != invalid);
    r <= invalid;
    return r;

method kill(tag);
    if (r.tag == tag)
        r <= invalid;
```



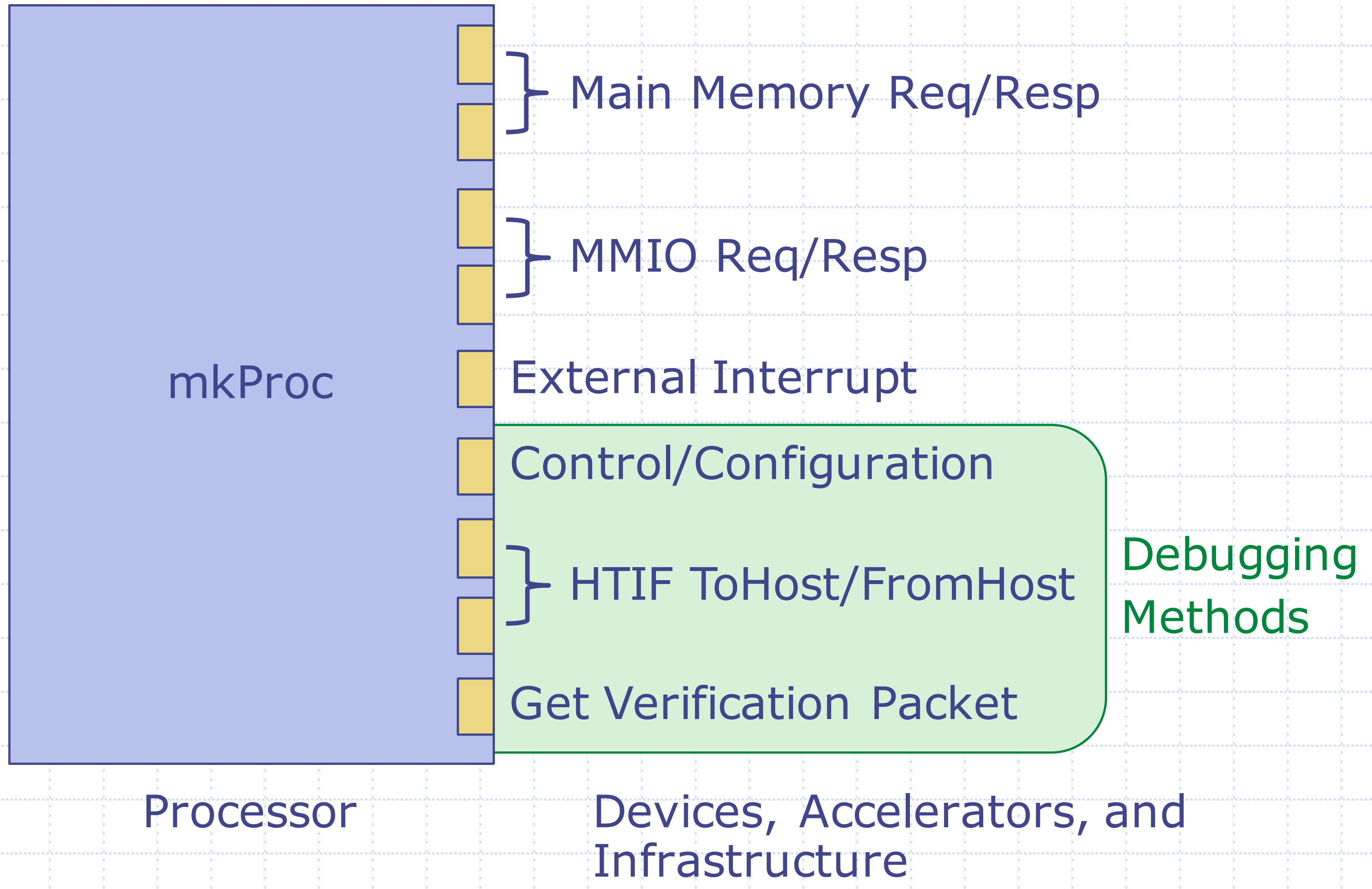
Scheduling Optimization

```
method deq if (r[1] != invalid);
    r[1] <= invalid;
    return r[1];

method kill(tag);
    if (r[0].tag == tag)
        r[0] <= invalid;
```



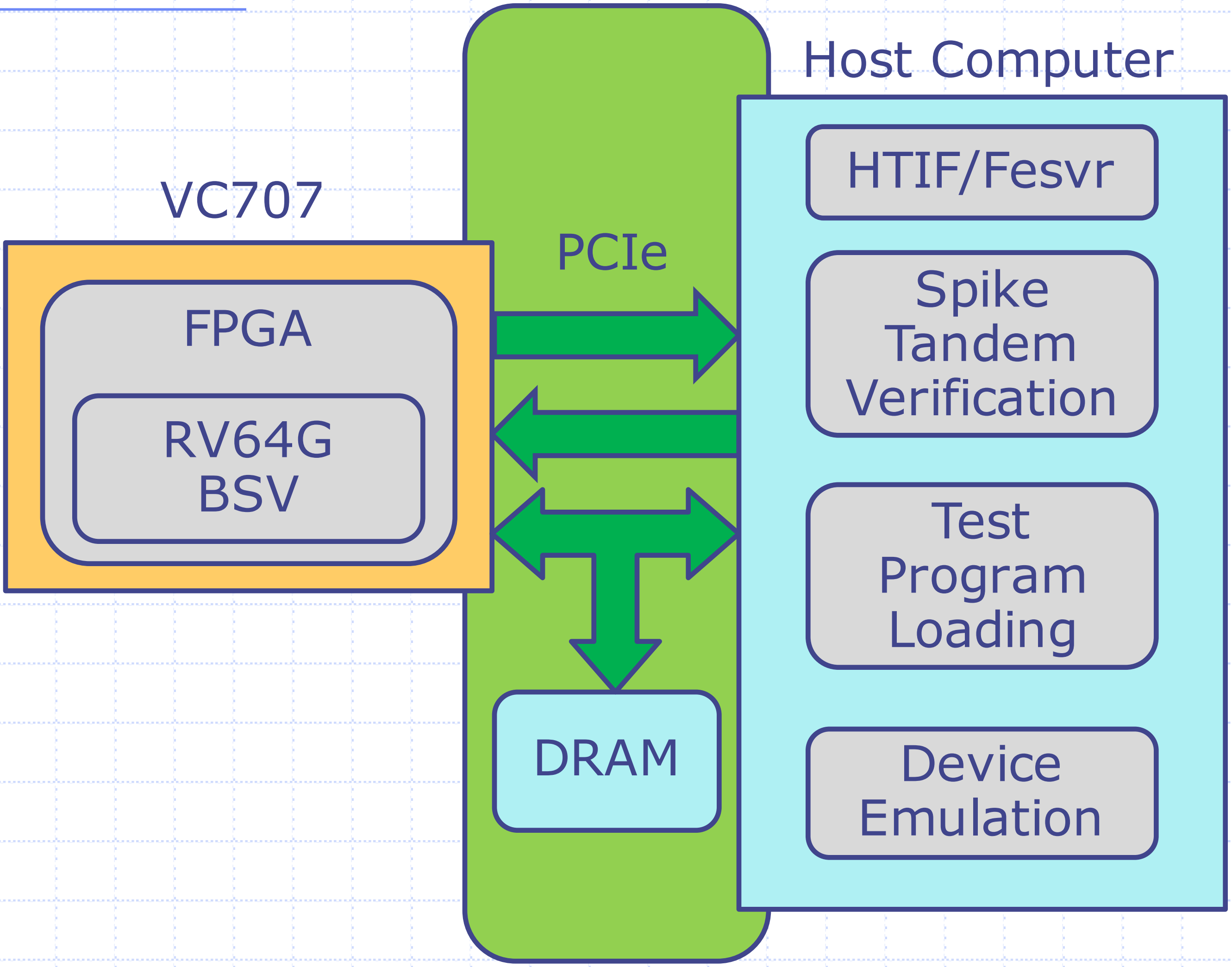
For more information about EHRs, see “The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs” by Daniel Rosenband

Processor Interface



Connectal Infrastructure

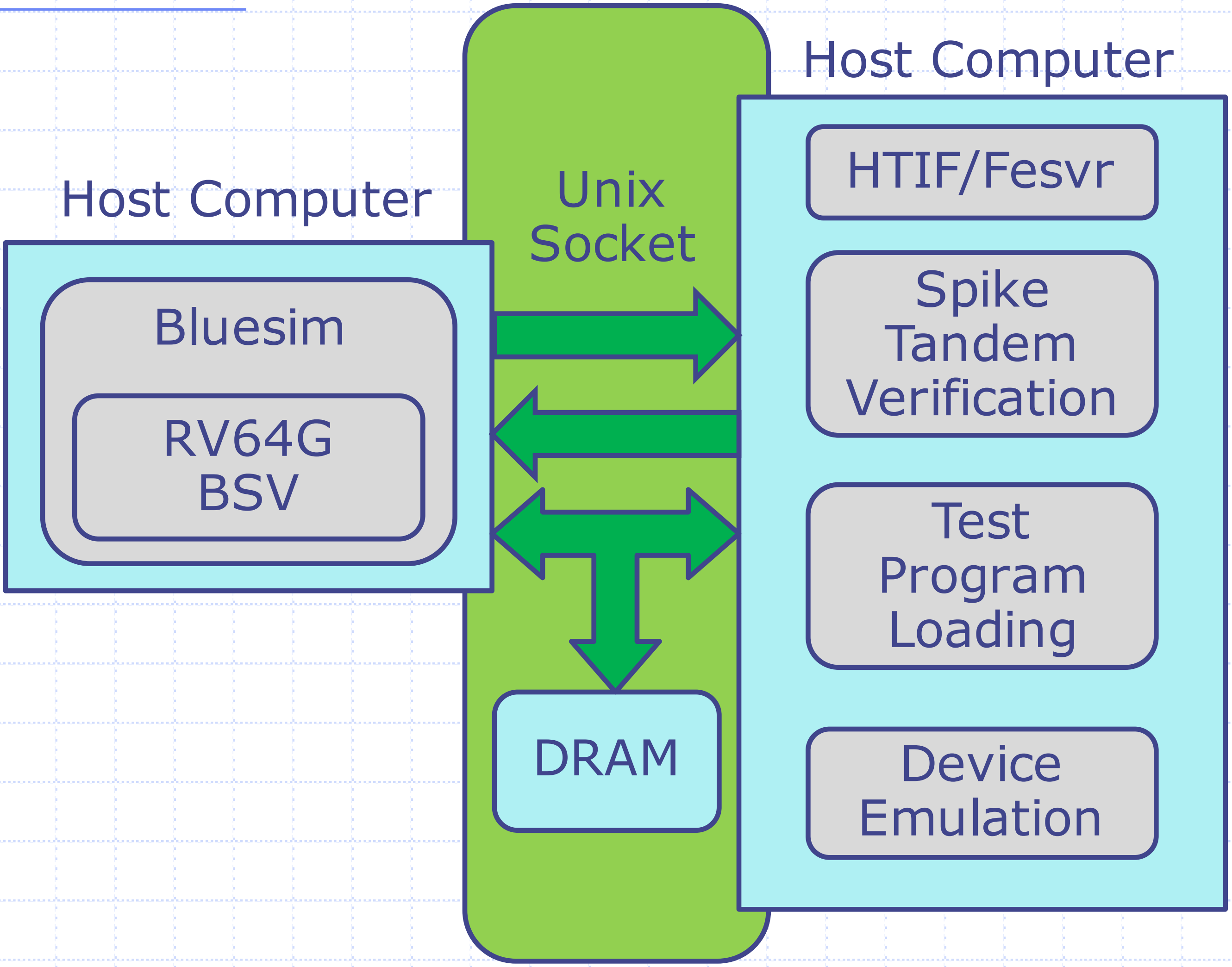
PCIe FPGA Boards



Connectal implements this connection

Connectal Infrastructure

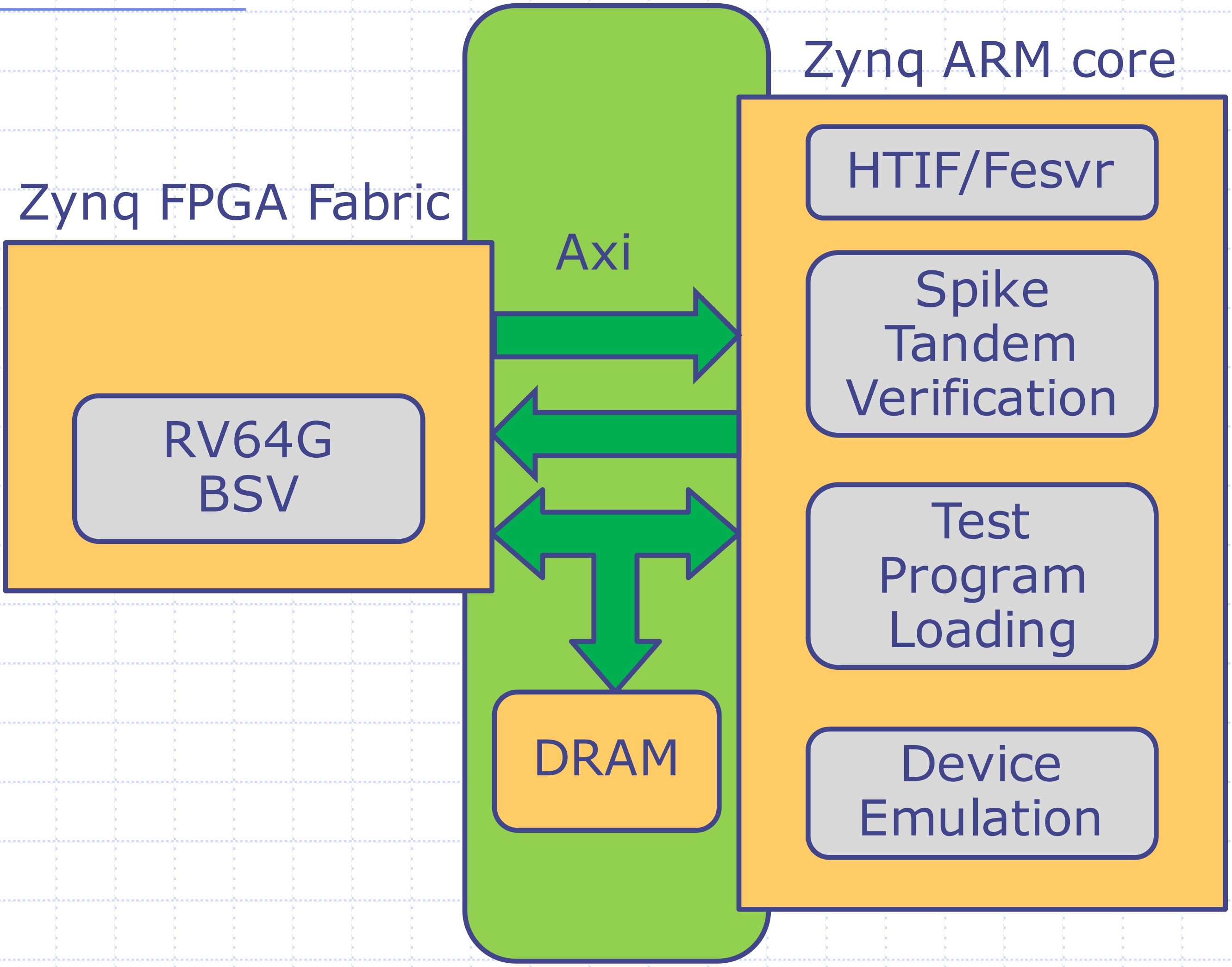
Simulation



Connectal implements this connection

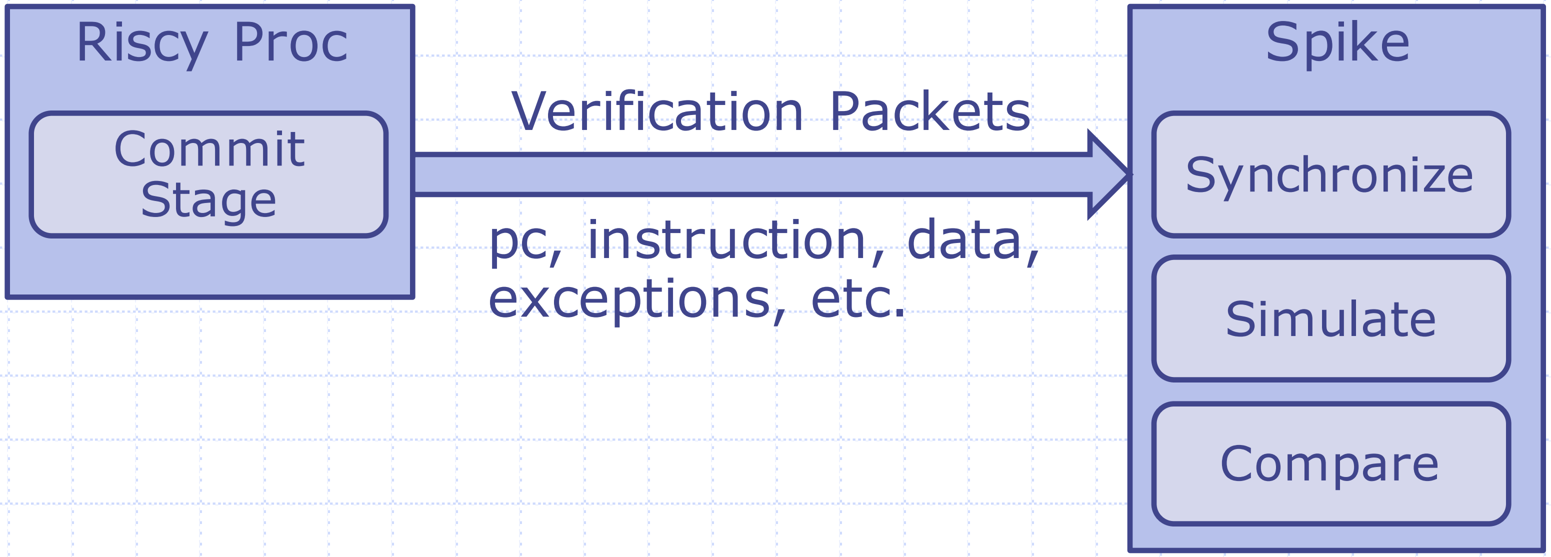
Connectal Infrastructure

Zynq Chips



ARM core running same code as host in previous slide

Tandem Verification



- Run the same program on two RISC-V implementations at once
  - Generate *verification packets* at commit stage
  - Use non-deterministic information from the implementation under test for synchronization
  - Compare results

Project Status

- Available now:
  - Riscy Processor Library
  - Riscy BSV Utility Library
  - Example multicycle processors
  - https://github.com/csail-csg/riscy
- Coming soon:
  - High-performance In-Order processor
  - Out-of-Order processor and modules
  - Multicore processors
- Planned work:
  - Formal specifications
  - Proofs for modules
  - Proofs for processors

Questions?

Backup Slides

Scheduling Optimization

In Processor Design Flow

- Choosing Schedules
  - Typically close to reverse pipeline order, but very dependent on microarchitecture
- Adapting modules to schedule
  - Registers that prevent desired schedule should be replaced with EHRs
  - EHRs add bypass paths between rules without breaking atomicity

For more information about EHRs, see "The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs" by Daniel Rosenband

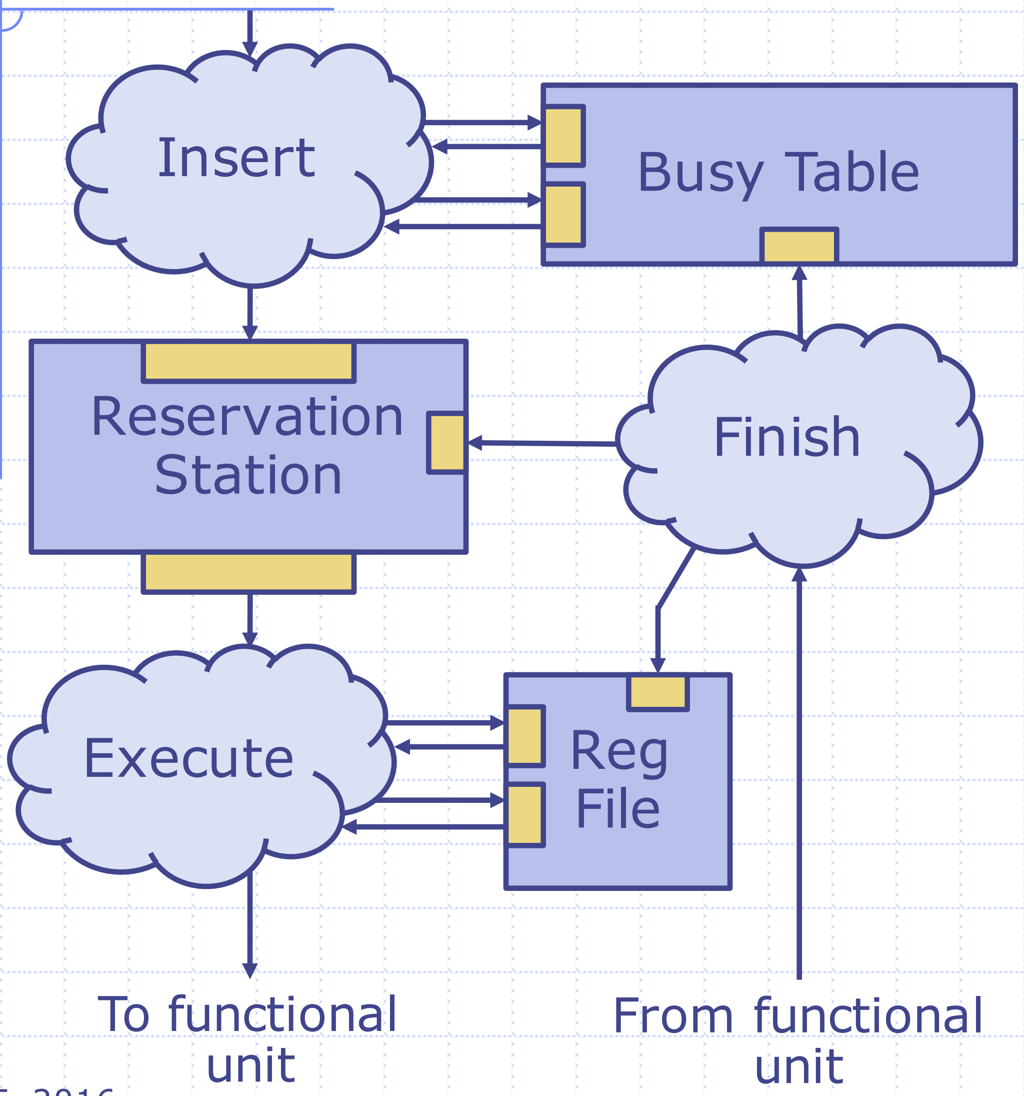
Scheduling Optimization

Consequences of Schedule Choice

- Location of concurrency logic
  - In the Move and Kill example, which buffer is in charge of killing moving instructions?
  - One location may be more efficient than another
- Existence of bypassing logic to skip latency
  - Scheduling can allow some redirecting rules to update the fetch pc in the same cycle while other redirecting rules update the fetch pc for the next cycle
- Priority between consumers of shared resources
  - Scheduling can determine priority for ports in an arbiters

Scheduling Optimization

Out-of-Order Execution Example

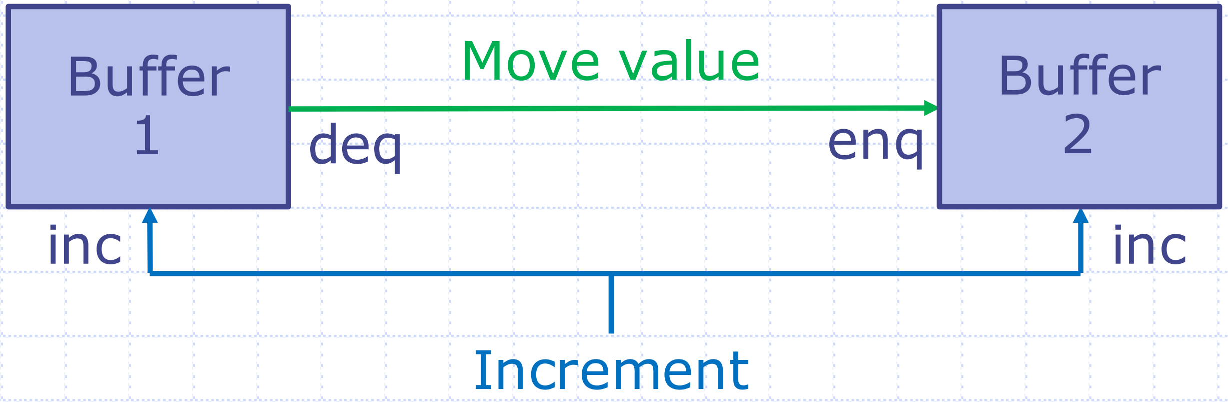


Ordering of Insert and Finish determines where bypass path is implemented

Ordering of Insert and Execute determines if an instruction can be executed in the same cycle it inserts the reservation station

Ordering of Execute and Finish determines if an instruction can be executed in the same cycle its registers become ready

Connecting Modules



Buffer 2 enq behavior

Buffer 1 deq behavior

	don't increment	increment
don't increment	+0	+1
increment	+1	+2

The implementations of these two buffers are not independent in typical HDLs. This prevents modular design/refinement.