SYSTEMVERILOG
HIGH-LEVEL SYNTHESIS

Bluespec SystemVerilog™ Training

Lecture 07: Modularizing Behavior (Rules) with Interface Methods

Lecture 07: Modularizing Behavior (Rules) with Interface Methods

- Interfaces and methods; method types
- HW interpretation of methods (collection of ports)
- Examples
- Methods as transactions: Transaction Level Modeling
- Method sharing, and conflicts
- Method scheduling
- Examples

Interface declaration syntax

- An interface declaration specifies an *interface type*
- It contains one or more *method declarations*
  - (Later: can also contain sub-interfaces)

Remember: type names begin with an uppercase!

```
interface IfcName [ #( ifc type params ) ];
   method type methodName (type arg, ..., type arg);
   ...
   method type methodName (type arg, ..., type arg);
endinterface [: IfcName ]
```


Methods

- Methods and interfaces are way for a module to communicate with the outside world
- A method describes certain behavior, and the conditions for that behavior
  - Methods have rule semantics
- BSV methods encapsulate a micro-protocol:
  - Each method has an associated READY signal (output port) (this is also called its *implicit condition*)
  - Methods which perform an Action (details: next slide) have an associated ENABLE signal (input port)
  - Method arguments are input ports
  - Method results are output ports

Example (from the library)

```
interface FIFO #(type  any_t);
   method Action   enq   (any_t  x);
   method any_t    first  ();
   method Action   deq   ();
   method Action   clear ();
endinterface: FIFO
```

Module that *provides* a FIFO interface

```
module mkFIFO (FIFO#(some_t));
   ...
endmodule
```

Instantiating modules with FIFO interfaces

Using FIFO interface methods

```
module mkBaz (...);
  FIFO#(int) f1 <- mkFIFO;
  FIFO#(int) f2 <- mkFIFO;
  ...
  rule r1 (f1.first() > 8);
     f2.enq (f1.first() + 2);      f1.deq();
  endrule
endmodule
```


Three kinds of methods

```
interface FIFOwPop #(type t);
   method Action            enq   (t x);
   method t                 first  ();
   method ActionValue#(t)   pop   ();      // combines first and deq
endinterface: FIFOwPop
```

- *Value methods:* Combinational. Take 0 or more arguments and return a value. E.g., first
  - Can be used in expressions (including rule conditions)
- *Action methods:* Take 0 or more arguments and perform an action (side-effect) inside the module. E.g., enq()
  - Can be used in Action contexts (body of rules, body of other Action method)
- *ActionValue methods:* Take 0 or more arguments, perform an action, and return a result. E.g., pop()
  - Can be used in Action contexts (body of rules, body of other Action method)

Using an ActionValue method

```
module mkBaz (FIFOwPop#(int));
  …
endmodule
```

```
module mkFoo (…);
  FIFOwPop#(int) f1 <- mkBaz;
  FIFOwPop#(int) f2 <- mkBaz;
  …
  rule r1 (…);
     int x <- f1.pop();
     f2.enq (x + 2);
  endrule
endmodule
```

Notes:
Because an ActionValue method has a side-effect, it cannot be used in rule conditions (type-checking will enforce this).
It can be used in rule bodies, and its implicit condition will contribute to the rule-enabling decision.

Remember!

'=' just for variable assignment:

```
Int#(124) b = myMemory.read(addr);
```

'<=' just for writing to a register in Action blocks:

```
rule doIt;
  myRegA <= 32;
endrule
```

'<-' just for instantiations or values returned by ActionValue methods:

```
Reg#(Bit#(32))        myRegA <- mkRegA(0);
PopFIFO#(Bit#(32)) myFifo   <- mkSizedFIFO(6);
rule doIt;
  let a <- myFIFO.pop
  myRegA <= a;
endrule
```

Methods as HW ports

- Interface method types can be interpreted directly as I/O wires of a module:
  - Arguments are input signals
  - Return values are output signals
  - An implicit condition is an output "ready" signal
  - An Action or ActionValue type (side-effect) indicates an incoming "enable" signal

Methods as HW ports: FIFOwPop

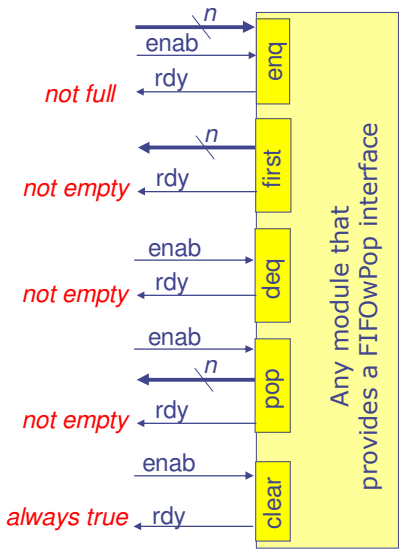
enq(t x**)**:
- n-bit argument
- has side effect (Action)

first():
- no argument
- n-bit result

deq():
- no argument
- has side effect (Action)

pop():
- n-bit result
- has side effect (Action)

clear():
- no argument
- has side effect (Action)



Methods do bidirectional communication

- A module m1 provides an interface
- A module m2 uses the interface
  - m2 "invokes" a method call in m1
- Despite the apparent above directionality, note that a single method can communicate in both directions.
  - Argument data buses go from m2 to m1
  - Result data buses go from m1 to m2
  - ENABLE signals go from m2 to m1
  - READY signals go from m1 to m2
  - There is no "pushing" or "pulling" of data—it's just HW wiring!

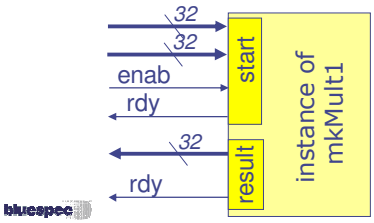
Defining interface methods

- A module that provides an interface must *define* all the methods in that interface, in the module body
  - These must be written at the end of the module body

```
module modName [#(type arg,...)] ( IfcType);
   ...
   ...
   method [ type ] methodName1 (arg, ..., arg) [ if ( cond ) ];
      ... method body ...
   endmethod
   ...
   method [ type ] methodNameN (arg, ..., arg) [ if ( cond ) ];
      ... method body ...
      return expr                    // if it is a value method
   endmethod
endmodule
```

A complete example: multiplier

```
module mkTest (Empty);

  Reg#(int) state <- mkReg(0);
  Mult_ifc  m       <- mkMult1();

  rule go (state == 0);
    m.start (9, 5);
    state <= 1;
  endrule

  rule finish (state == 1);
    $display ("Product = %d", m.result() );
    state <= 2;
  endrule

endmodule: mkTest
```




```
interface Mult_ifc;
  method Action start (int x, int y);
  method int      result ();
endinterface: Mult_ifc
```

```
module mkMult1 (Mult_ifc);
  Reg#(int) product <- mkReg(0);
  Reg#(int) d       <- mkReg(0);
  Reg#(int)  r      <- mkReg(0);

  rule cycle (r != 0);
   if (r[0] == 1) product <= product + d;
   d <= d << 1;
   r <= r >> 1;
  endrule

  method Action start (x,y) if (r == 0);
   d <= x; r <= y; product <= 0;
  endmethod

  method result () if (r == 0);
   return product;
  endmethod

endmodule: mkMult1
```

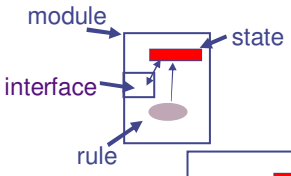

Interface methods modularize rules

- An Action method is just a part of a rule
  - Just like a rule, it has a condition (Bool expression) and a body (an Action)
  - These just become a part of any rule, both its conditions and actions, where the method is used
- A value method is also a part of a rule
  - Its condition becomes a part of the conditions of any rule where the method is used
- Thus, interface methods are also factored into rule scheduling

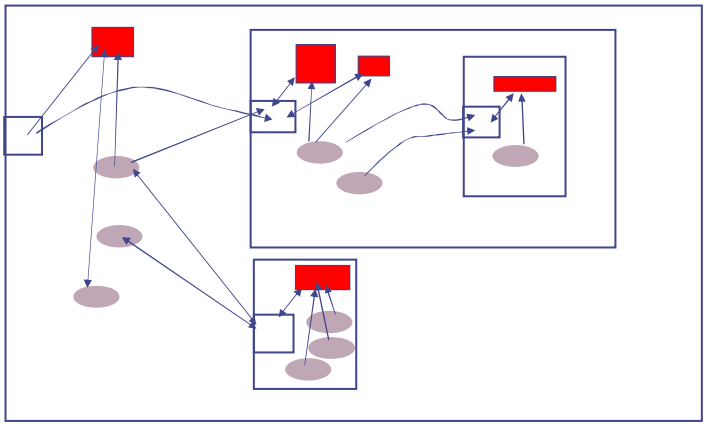
Interface method scheduling

- Because interface methods are pieces of rules, they are also factored into rule scheduling
- E.g., "methods m1 and m2 conflict"
  - Means: a rule r1 that uses m1 conflicts with a rule r2 that uses m2
- E.g., "method m1 precedes method m2"
  - Means: a rule r1 that uses m1 must logically precede a rule r2 that uses m2, when executed concurrently
- The compiler manages all this automatically

Modules, rules, interfaces, methods



The big picture: modules contain rules which use methods that are provided by sub-modules in their interfaces. Methods, too, can use other methods.



Interface methods are HW!

- Interface method declarations look like functions/ procedures in SW
- Uses of interface methods look like function/ procedure calls in SW
- But: think HW, not SW or process simulation!
- A definition of an interface method in a module is a manifest bit of circuitry behind its ports
- A use of an interface method is just a set of connections (wires) to the module interface ports
- There is no "call/execute/return", stack frame, …!

Sharing methods

```
module mkTest (…);
  …
  FIFO#(int)  f  <- mkFIFO();
  …
  rule r1 (… cond1 …);
    …
    f.enq (… expr1 …);
    …
  endrule

  rule r2 (… cond2 …);
    …
    f.enq (… expr2 …);
    …
  endrule
endmodule: mkTest
```

```
interface FIFO#(type t);
  Action enq (t n);
  …
endinterface

module mkFIFO (…);
  …
  method enq (x) if (… notFull …);
    …
  endmethod
  …
endmodule: mkFIFO
```

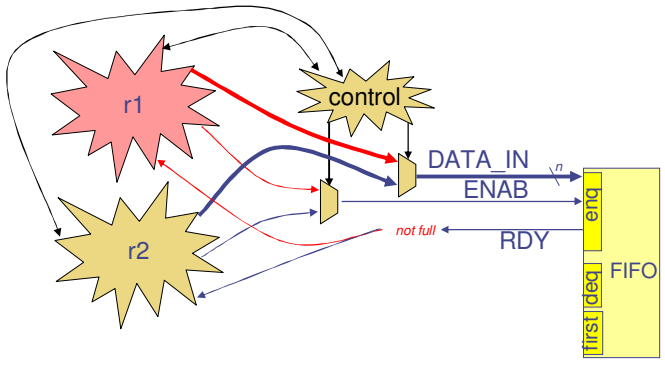
(In general the two invoking rules could be in different modules)

Sharing methods

- In SW, to call a function/procedure from two processes just means:
  - Create two instances (usually on two stacks)
- A BSV method represents real hardware
  - There is only one instance (per instantiated module)
  - It is a shared resource
  - Parallel accesses must be scheduled (controlled)
  - Data inputs and outputs must be muxed/ distributed
- The BSV compiler inserts logic to accomplish this sharing
  - This logic is not an artifact of using BSV—it is logic that the designer would otherwise have to design manually

Shared Methods: Hardware

The compiler inserts logic for sharing a method



Important special cases of sharing

- Value methods without arguments need no muxing or control, since they have no inputs into the module
  - Examples:
    - r._read() for a register
    - f.first() for a FIFO
  - Note: these methods are combinational, but they depend on the module's internal state
- Such methods can be shared freely (used in different rules)
  - It's just fan-out of output wires!

Important special cases of sharing

- All other kinds of methods
  - Value methods with at least one argument
  - Action and ActionValue methods, with or without arguments

  have input wires that need multiplexing and controlled sharing
- Hence, if used in different rules, they represent a potential resource conflict that can affect scheduling
  - Of course, this only matters if the rules could be enabled concurrently
  - Note: module inlining also may eliminate such a conflict
- (Advanced topic) BSV primitives can specify a replication factor for certain methods, so two calls to the "same" method actually get connected (automatically) to different replicas of the method, alleviating a resource bottleneck
  - E.g., a read method of a multi-ported register file

Methods and Transaction Level Modelling

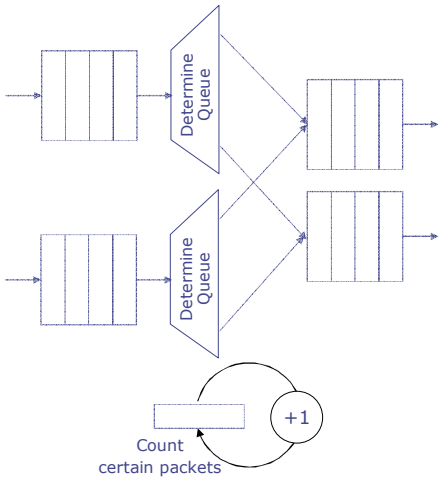
- Each method can be read as a *transaction* that can be applied against a module
- By just changing the level of abstraction of the arguments and results, we can move from realistic hardware to high-level models, using the single paradigm of methods

```
Get#(Bit#(16)) m <- mkM;
Put#(Bit#(16)) n <- mkN;

rule r1 (...);
  Bit#(16) x <- m.get();
  n.put (x);
endrule
```

```
Get#(EtherPacket) m <- mkM;
Put#(EtherPacket) n <- mkN;

rule r1 (...);
  EtherPacket x <- m.get();
  n.put (x);
endrule
```


Example: a 2x2 switch, with stats

- Packets arrive on two input FIFOs, and must be switched to two output FIFOs
- Certain "interesting packets" must be counted



2x2 switch specs

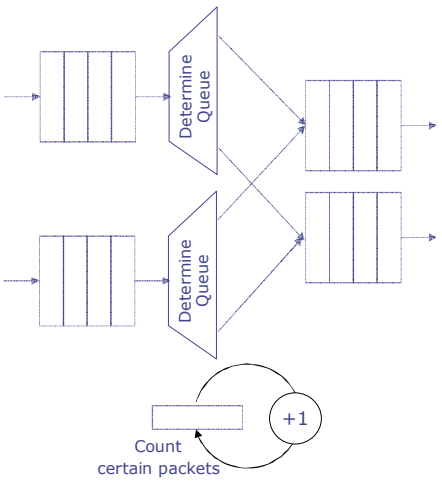
- Input FIFOs can be empty
- Output FIFOs can be full
- Shared resource collision on an output FIFO:
  - if packets available on both input FIFOs, both have same destination, and destination FIFO is not full
- Shared resource collision on counter:
  - if packets available on both input FIFOs, each has different destination, both output FIFOs are not full, and both packets are "interesting"
- Resolve collisions in favor of packets from the first input FIFO
- Must have maximum throughput: a packet must move if it can, modulo the above rules

bluespec

2x2 switch code

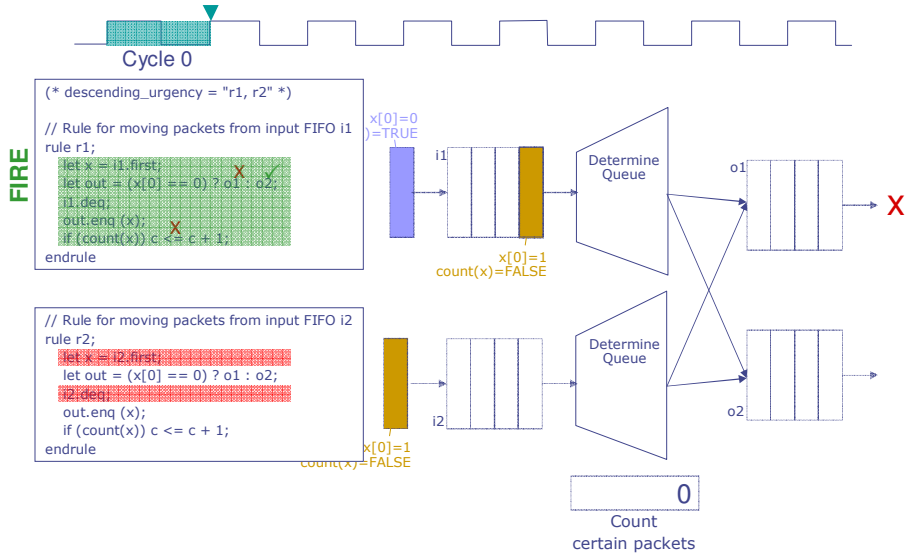
```
module mkSmallSwitch (SmallSwitch);
  FIFO#(Dat) i1 <- mkSizedFIFO(4);
  FIFO#(Dat) i2 <- mkSizedFIFO(4);
  FIFO#(Dat) o1 <- mkSizedFIFO(4);
  FIFO#(Dat) o1 <- mkSizedFIFO(4);

  (* descending_urgency = "r1, r2" *)
  rule r1;  // for packets from FIFO i1
     let x = i1.first;
     let out = (x[0] == 0) ? o1 : o2;
     i1.deq; out.enq (x);
     if (count(x)) c <= c + 1;
  endrule

  rule r2;  // for packets from FIFO i2
     let x = i2.first;
     let out = (x[0] == 0) ? o1 : o2;
     i2.deq; out.enq (x);
     if (count(x)) c <= c + 1;
  endrule
...
```

```
typedef Bit#(8) Dat;
function Bool count(int a);
  ...
endfunction
```

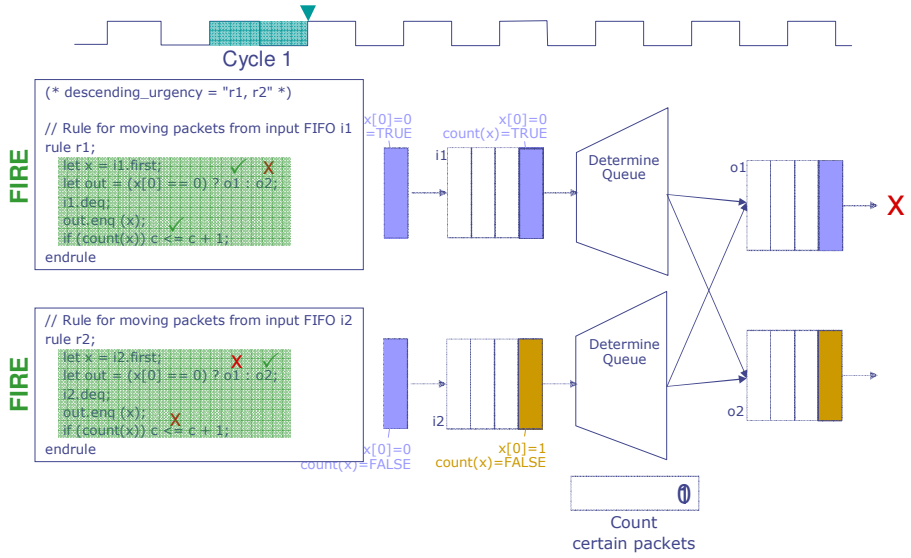


bluespec

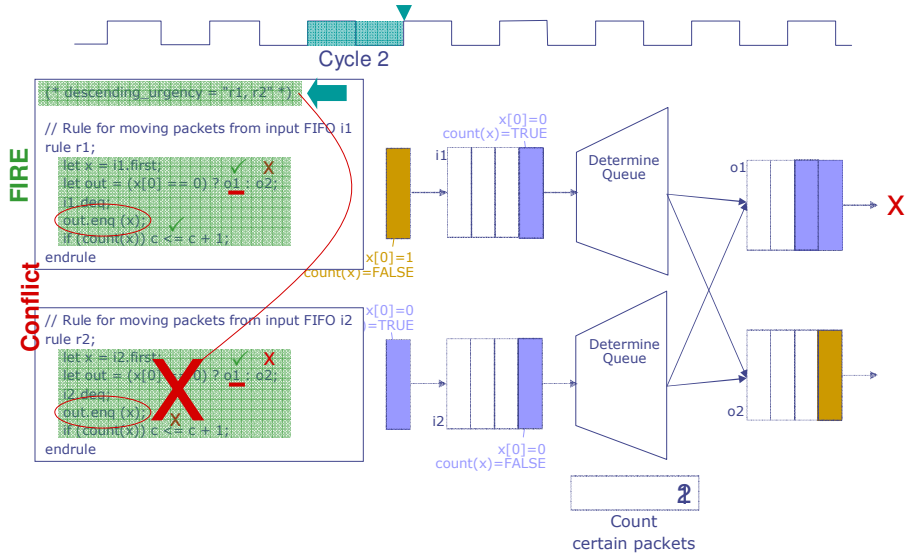
Code Explanation



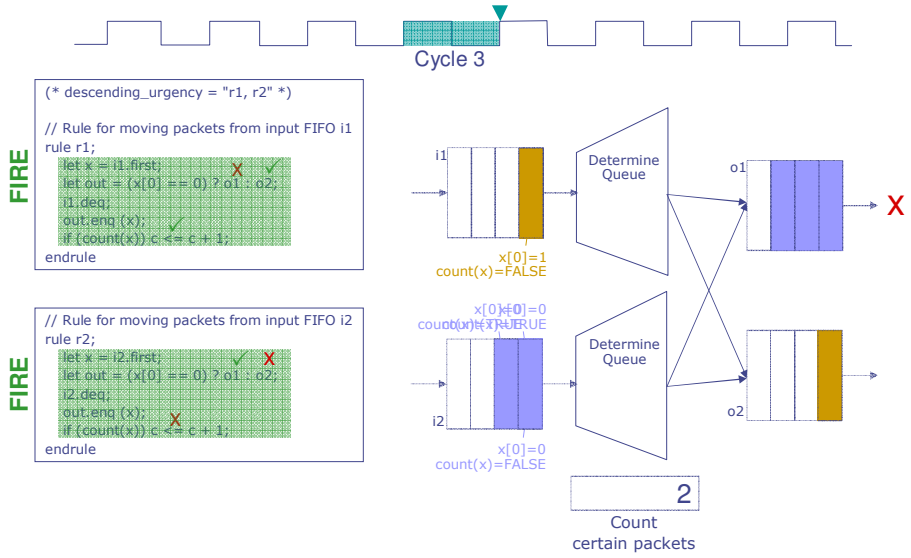
Code Explanation



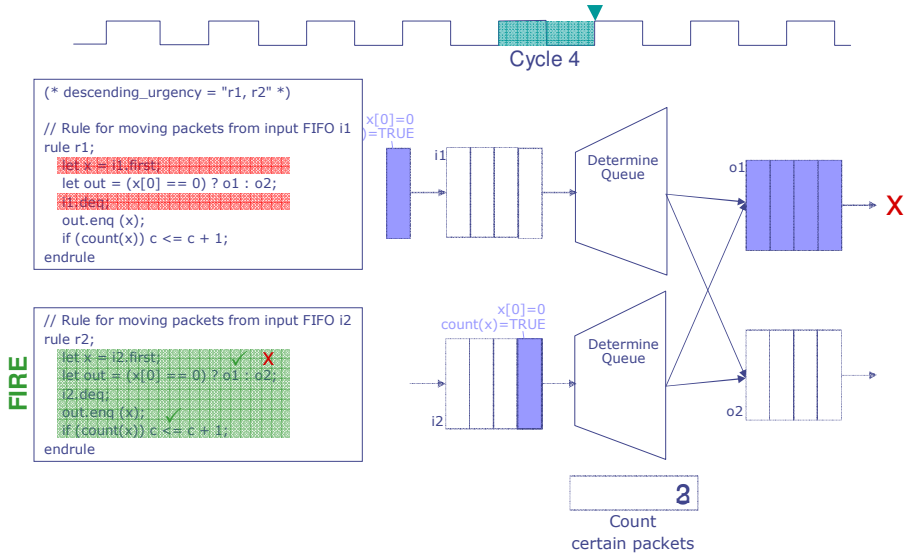
Code Explanation



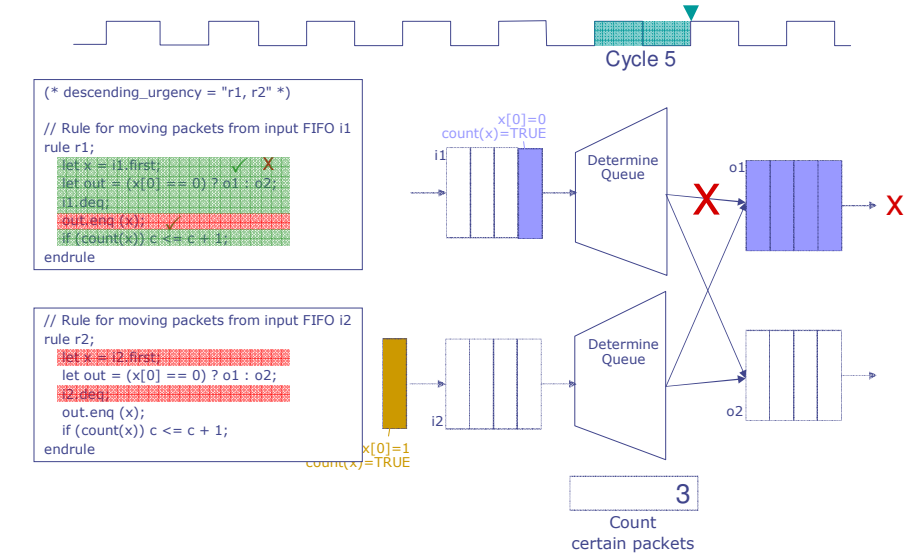
Code Explanation



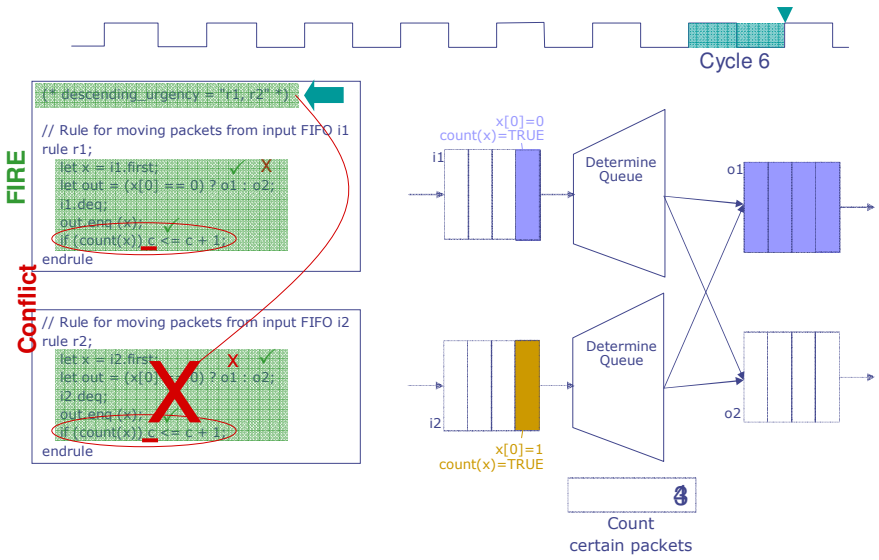
Code Explanation



Code Explanation



Code Explanation



Commentary

- Muxing into output FIFOs, and control of those muxes, automatically generated
- Automatic handling of FIFO emptiness, FIFO fullness
  - This is part of BSV's rule and interface method semantics
    - Impossible to read a junk value from an empty FIFO
    - Impossible to enqueue into a full FIFO
    - Impossible to race for multiple enqueues onto a FIFO
- All control for resource sharing handled automatically
  - Rule atomicity ensures consistency
  - The "descending_urgency" attribute resolves collisions in favor of rule r1

The BSV code directly expresses design intent
➔ *without all the clutter of control and shared-resource mgmt*
➔ *generating efficient, correct-by-construction RTL*

Managing change

- Now imagine the following changes to the existing code:
  - Some packets are multicast (go to both FIFOs)
  - Some packets are dropped (go to no FIFO)
  - More complex arbitration
    - FIFO collision: in favor of r1
    - Counter collision: in favor of r2
    - Fair scheduling
  - Several counters for several kinds of interesting packets
  - Non-exclusive counters (e.g., IP packets include TCP packets)
  - M input FIFOs, N output FIFOs (parameterized)
- Suppose these changes are required 6 months after original coding

In BSV these are easy, because
➔ *the source code remains uncluttered by all the complex control and mux logic*
➔ *atomicity ensures correctness*

Interface Methods: summary

- Interface methods are an abstraction of Verilog port lists
  - Bundle together related ports (in both directions)
  - Capture a complete "transaction"
- Interface methods fit into *Rules* with the same semantics of conditions, actions, and scheduling
  - Sharing and control logic are automatically generated by the compiler
  - "Implicit conditions" remove a lot of clutter from source code

SYSTEMVERILOG
HIGH-LEVEL SYNTHESIS

End of Lecture