Hardware Synthesis: Bluespec Modules as Sequential Circuits

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Hardware Synthesis from Bluespec
High-level idea

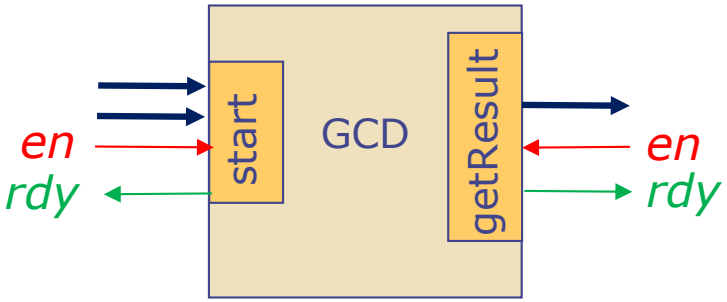
- Every module represents a sequential circuit
  - Register is a primitive module – its implementation is outside the language
- Input/Output wires of the sequential circuit corresponding to a module are derived from the module's interface
- A module contains registers and other modules that are instantiated explicitly in the module
- Each method is synthesized into a combinational circuit
  - Its inputs include the method's parameters and, in case of an action method, its enable signal
  - Outputs include the ready signal of the method, and the args and an enable signal (if needed) for each method it calls. For Value or ActionValue methods, outputs also include the returned value
- Similarly, each rule also defines a combinational circuit. The ready signal of a rule is often called a "Can Fire" signal
- Combinational logic of all the rules and methods are connected to the instantiated registers and modules using muxes

Interface defines input/output wires

- Inputs and outputs are defined by the *type* of the module, i.e., its interface definition
  - Each *method* has a output ready wire
  - Each *method* may have 0 or more input data wires
  - Each *Action method* and *ActionValue method* has an input enable wire
  - Each *value method* and *ActionValue method* has output data wires

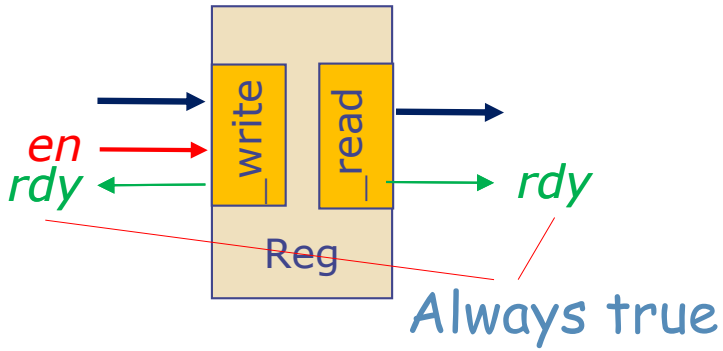
An *Action method* has no output data wire

```
interface GCD#(Bit#(n));
  method Action start
      (Bit#(n) a,Bit#(n) b);
  method ActionValue(Bit#(n))
                getResult;
endinterface
```



Register: The Primitive module

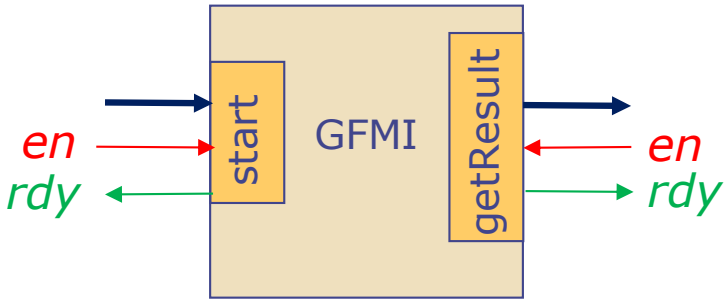
```
interface Reg#(Bit#(n);
   method Action _write(Bit#(n) x);
   method Bit#(n) _read;
endinterface
```



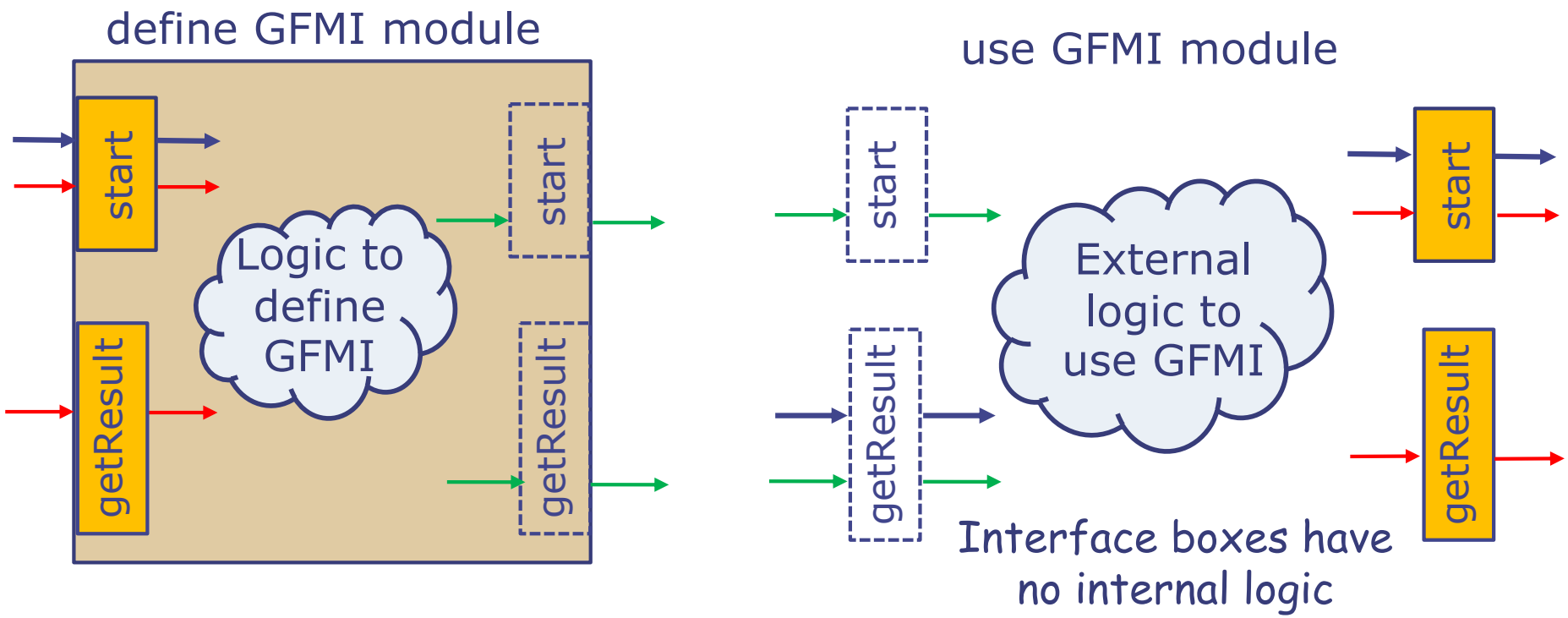
- Implementation is defined outside the language
- A register is created using mkReg or mkRegU
- The guards of _write and _read are always true and not generated
- Special syntax
  - `x <= e` instead of `x._write(e)`
  - `x` instead of `x._read` in expressions
- Since we never look inside a register, we represent it simply in terms of its input/output wires

Interface convention for drawing circuits

```
interface GFMI#(numeric type n);
  method Action start (Bit#(n) a);
  method ActionValue(Bit#(n))
                  getResult;
endinterface
```

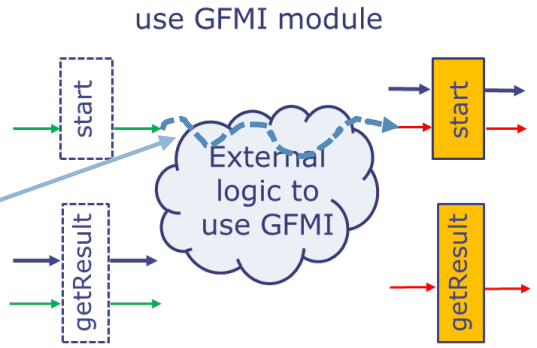


- For drawing circuits, sometimes we duplicate each interface box into two to avoid the clutter of crossing wires

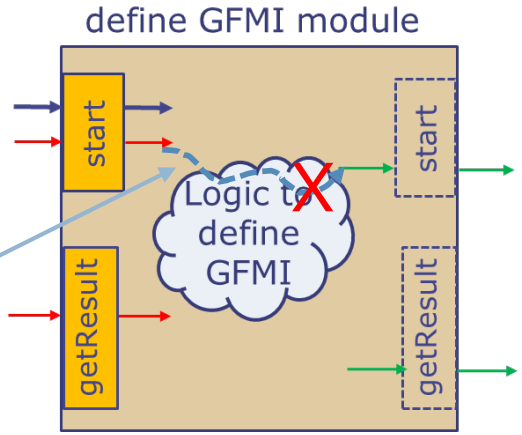


Implications of the rdy-en protocol

- rdy-en protocol in using a method implies that en of a method is not set to true unless its rdy is true
  - e.g., there must be a dependence between start.rdy and start.en

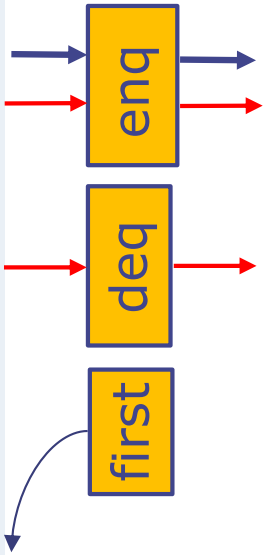


- By a similar argument one can say that inside a module the rdy of a method should *not* depend on the en of the method (otherwise we will create a combinational cycle when this module is used)
  - e.g., start.rdy must *not* depend on start.en

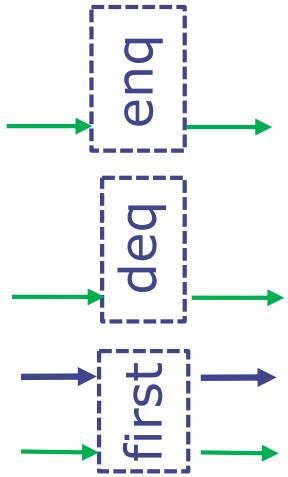


Example: FIFO Circuit
Interface wires

```
module mkFifo (Fifo#(1, Bit#(n)));
  Reg#(Bit#(n))  d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(Bit#(n) x)
                        if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method Bit#(n) first if (v);
    return d;
  endmethod
endmodule
```



No need to draw first here because it has no input wires

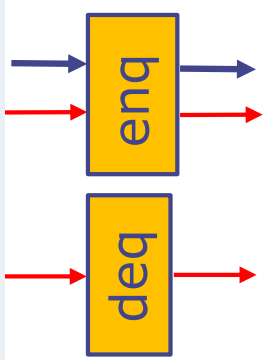


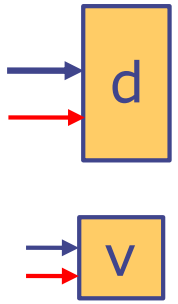
- I/O: Interface

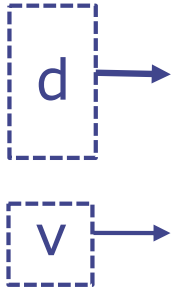
```
interface Fifo#(numeric type size, type Bit#(n));
  method Action enq(Bit#(n) x);
  method Action deq;
  method Bit#(n) first;
endinterface
```

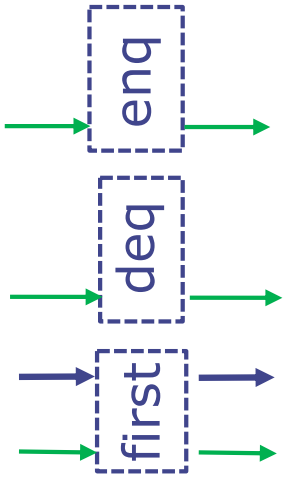
FIFO Circuit
Instantiating internal state elements

```
module mkFifo (Fifo#(1, Bit#(n)));
  Reg#(Bit#(n))  d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(Bit#(n) x)
                        if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method Bit#(n) first if (v);
    return d;
  endmethod
endmodule
```







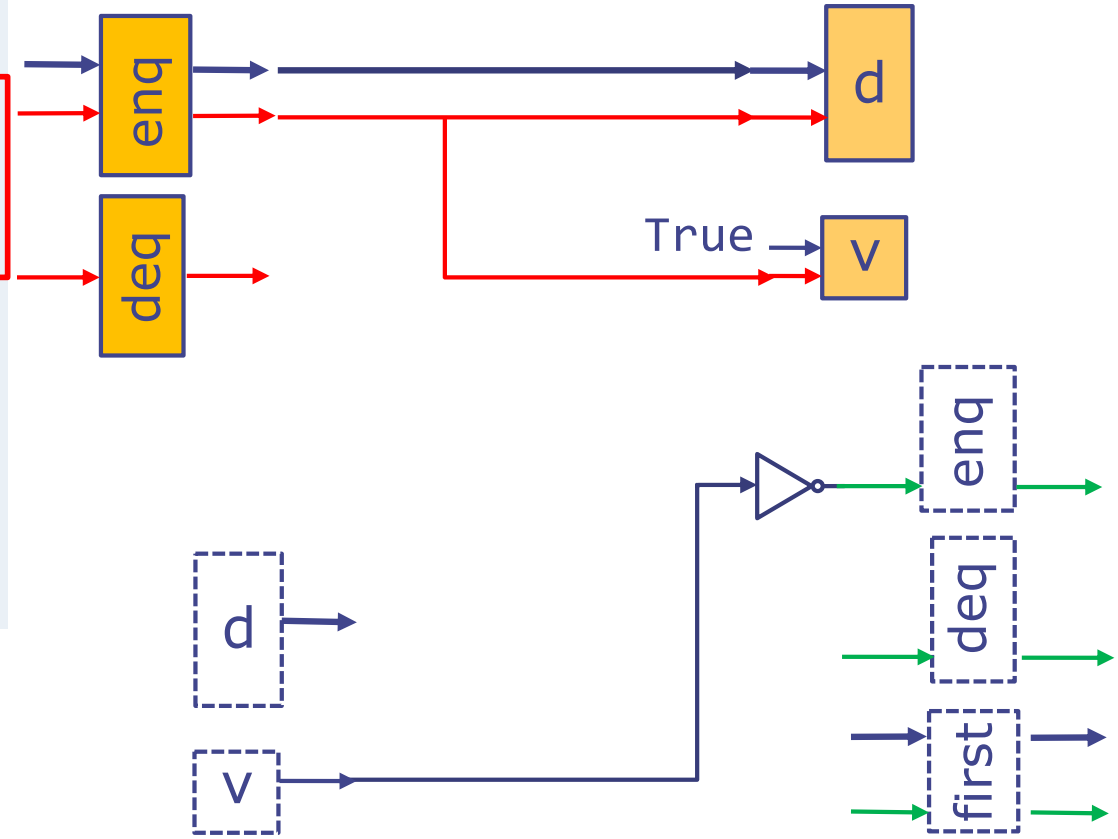


- I/O: Interface
- Instantiate state elements

```
interface Fifo#(numeric type size, type Bit#(n));
  method Action enq(Bit#(n) x);
  method Action deq;
  method Bit#(n) first;
endinterface
```

FIFO Circuit
method enq

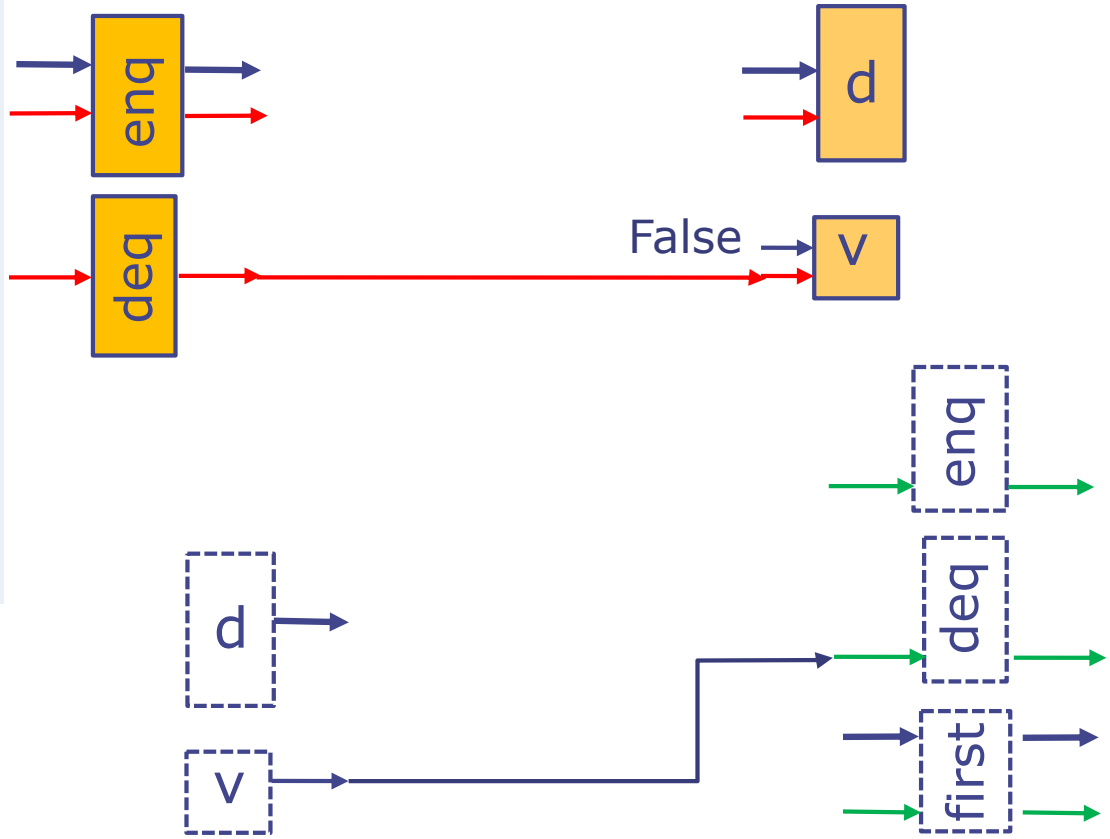
```
module mkFifo (Fifo#(1, Bit#(n)));
  Reg#(Bit#(n))  d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(Bit#(n) x)
                          if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method Bit#(n) first if (v);
    return d;
  endmethod
endmodule
```



- I/O: Interface
- Instantiate state elements
- Compile methods: enq

FIFO Circuit
method deq

```
module mkFifo (Fifo#(1, Bit#(n)));
  Reg#(Bit#(n))  d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(Bit#(n) x)
                        if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method Bit#(n) first if (v);
    return d;
  endmethod
endmodule
```

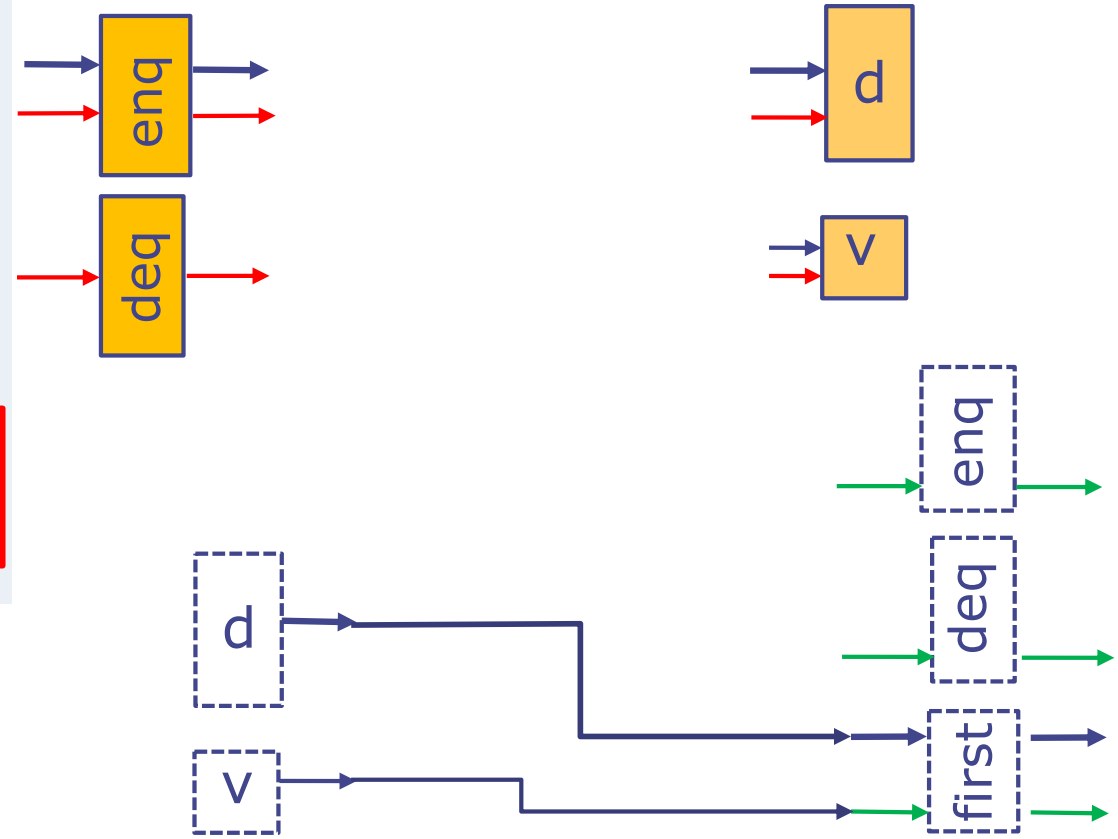


- I/O: Interface
- Instantiate state elements
- Compile methods: deq

FIFO Circuit
method first

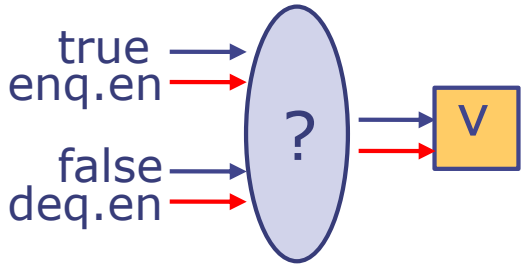
```
module mkFifo (Fifo#(1, Bit#(n)));
  Reg#(Bit#(n))  d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(Bit#(n) x)
                        if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method Bit#(n) first if (v);
    return d;
  endmethod
endmodule
```

- I/O: Interface
- Instantiate state elements
- Compile methods: first

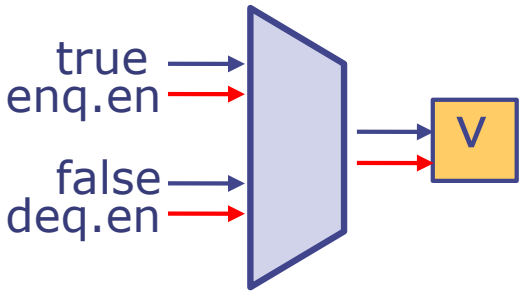


Combing the methods into a one circuit

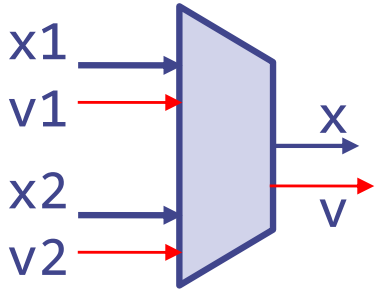
- An issue arises in combing these circuits if an input port has several sources, e.g., inputs to register v



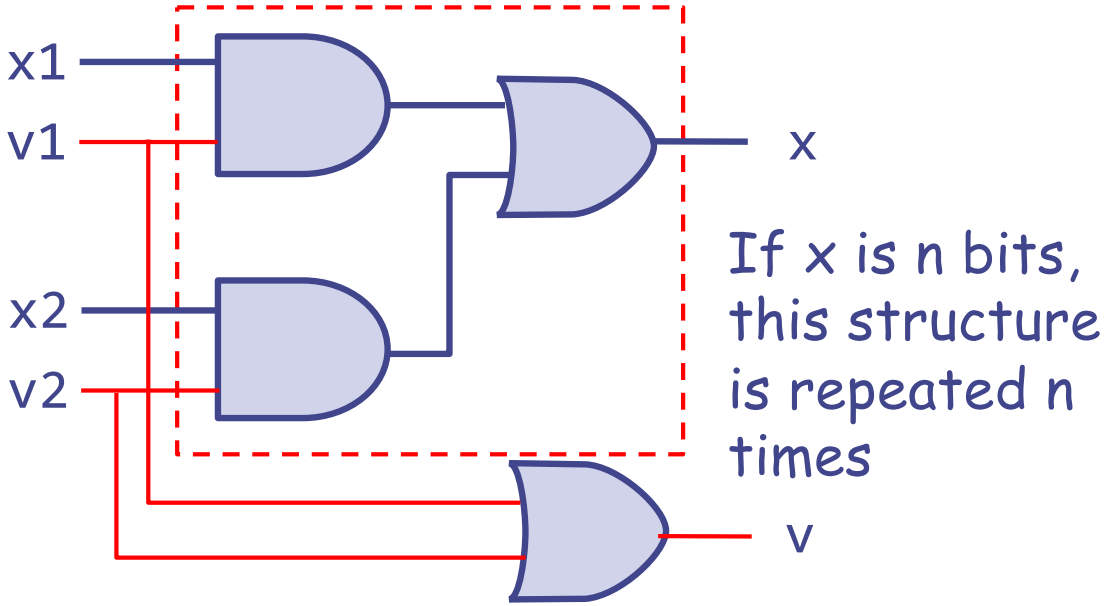
- We introduce a new type of mux for this purpose (we will call it emux for mux-with-enable)



emux to deal with multiple sources



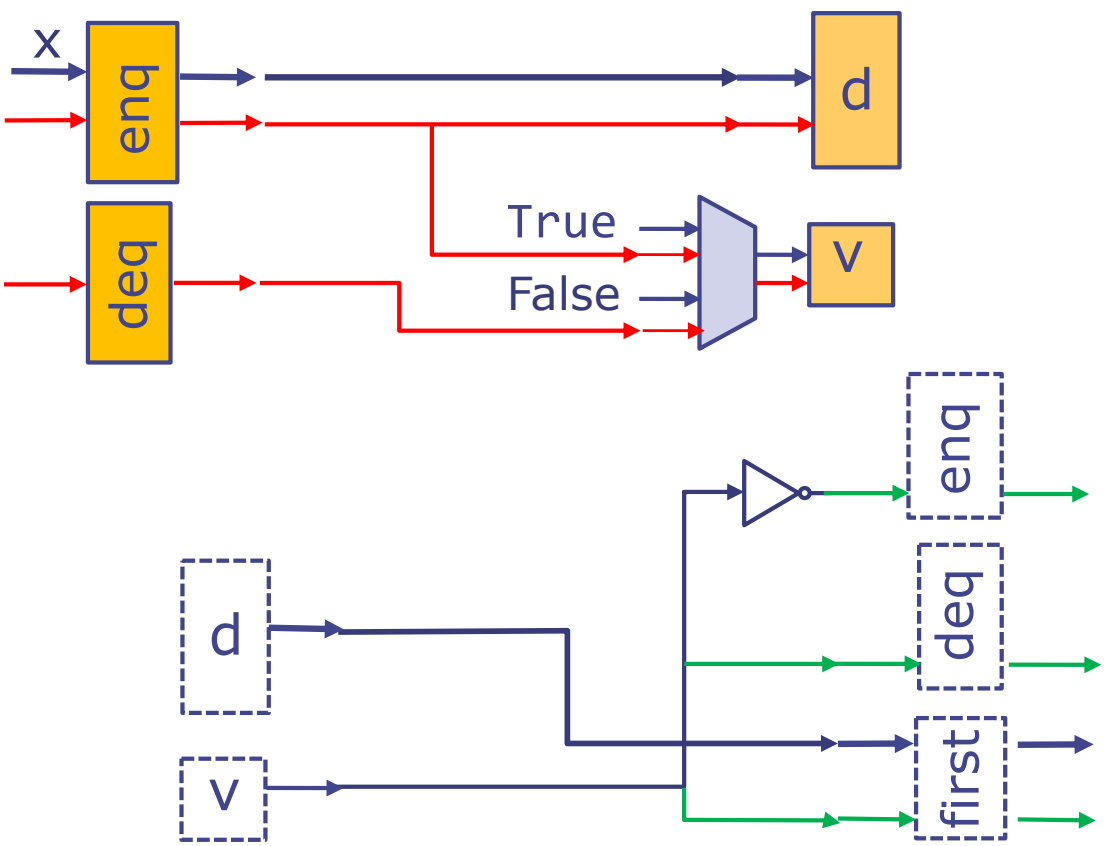
x = (v1 & x1) | (v2 & x2)
v = v1 | v2



- x*i* has a meaningful value only if its corresponding v*i* is true
- Compiler has to ensure that at most one v*i* input to the mux is true at any given time; the circuit will behave unpredictably if multiple input signals are valid

FIFO Circuit

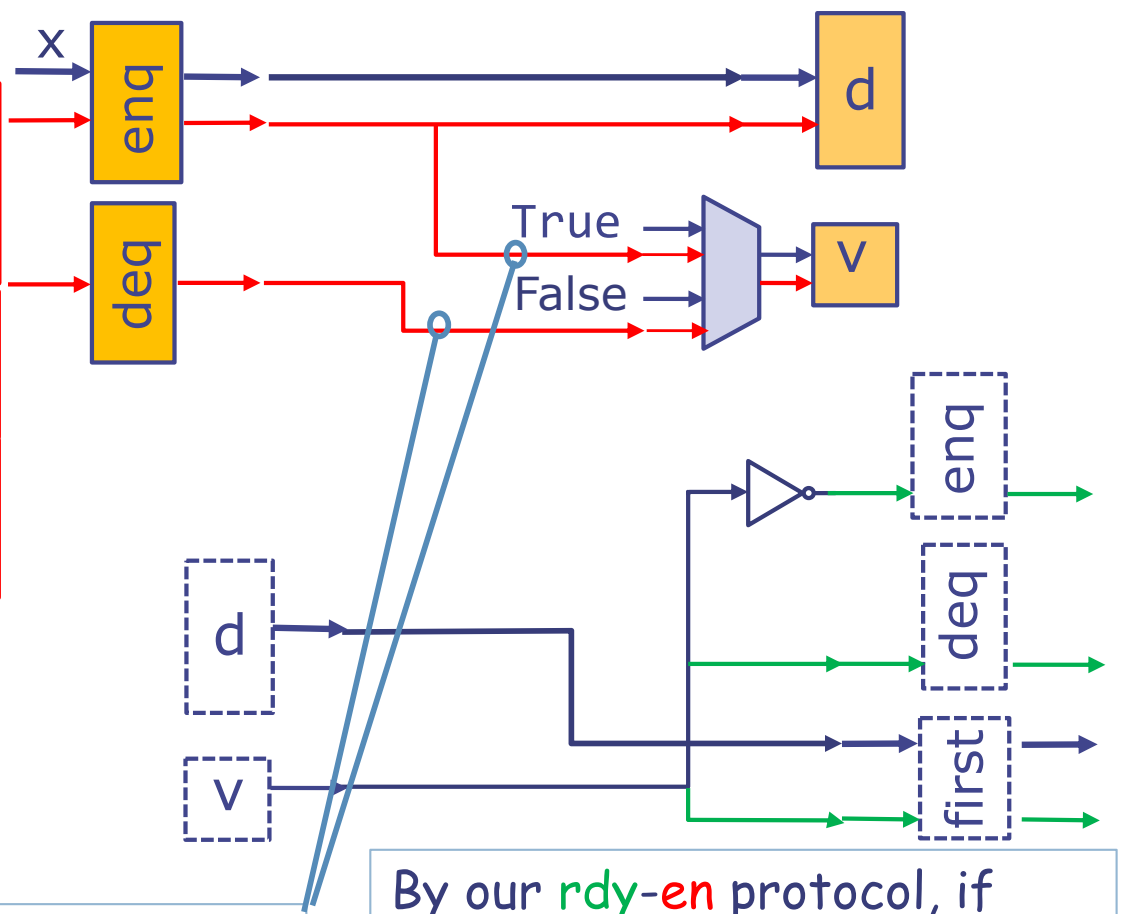
```
module mkFifo (Fifo#(1, Bit#(n)));
  Reg#(Bit#(n))  d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(Bit#(n) x)
                          if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method Bit#(n) first if (v);
    return d;
  endmethod
endmodule
```



- I/O: Interface
- Instantiate state elements
- Compile methods
- Combines the methods by inserting muxes

FIFO Circuit
a correctness issue

```
module mkFifo (Fifo#(1, Bit#(n)));
  Reg#(Bit#(n))  d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(Bit#(n) x)
                          if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method Bit#(n) first if (v);
    return d;
  endmethod
endmodule
```

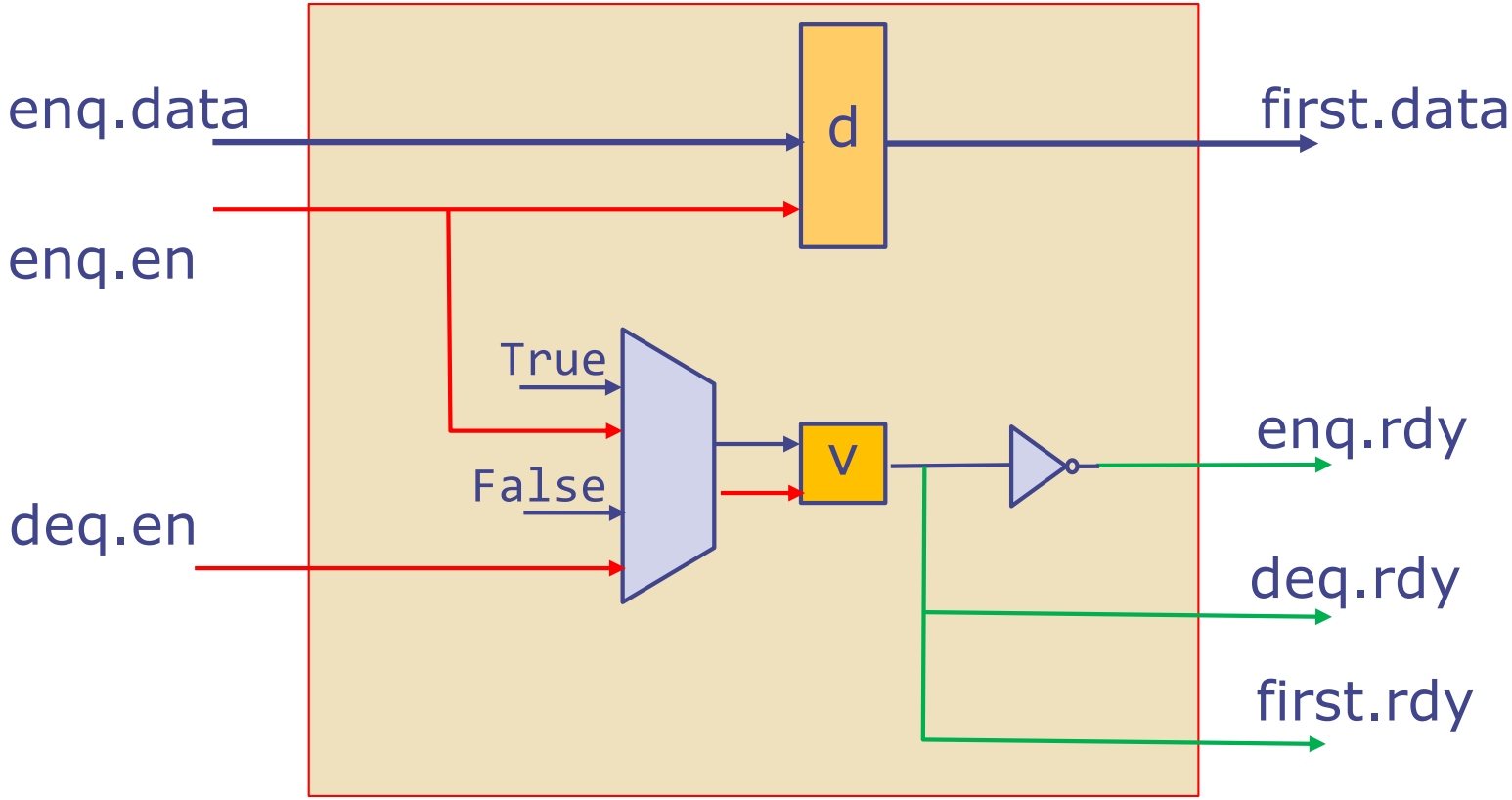


- I/O: Interface
- Instantiate state elements
- Compile methods
- Combines the methods by inserting muxes

We need to guarantee that both enq.en and deq.en cannot be True at the same time

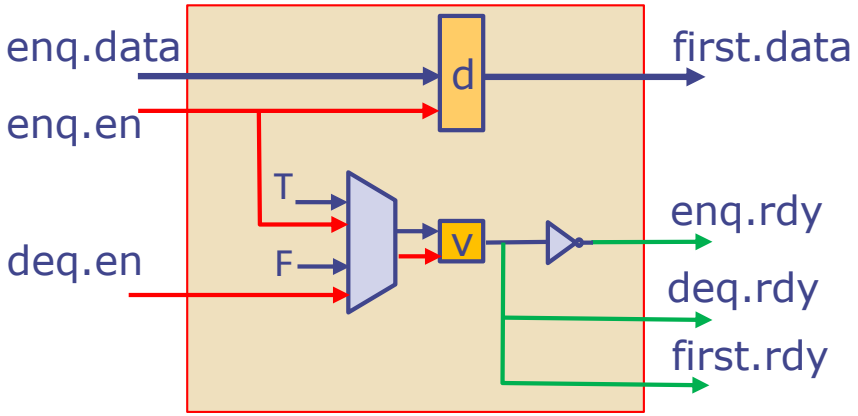
By our rdy-en protocol, if enq.en and deq.en are both True then so must be enq.rdy and deq.rdy, but this is not possible

Redrawing the FIFO Circuit
without interfaces



- The sequential circuit corresponding to a one-element FIFO; It has no cycles but it is a sequential circuit nevertheless because it has state elements.
- Interface boxes in our diagrams have no internal logic; they merely represent the ports of a sequential circuit

Ready signals and guards



- We can see that in this example the readiness of each method depends only on the internal state of the module
- rdy signals are derived from guards and therefore, guard expressions should be written to avoid any dependence on inputs

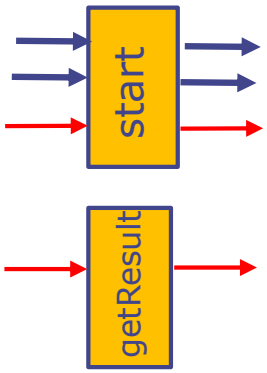
Next state transition
Partial Truth Table

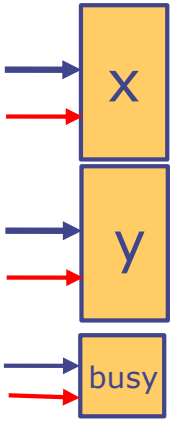
An aside

inputs			state		next state		outputs			
enq. en	enq. data	deq. en	d^t	v^t	d^{t+1}	v^{t+1}	enq. rdy	deq. rdy	first. rdy	first. data
0	x	0	x	0	x	0	1	0	0	-
1	d	0	x	0	d	1	0	1	1	-
0	x	0	d	1	d	1	0	1	1	d
0	x	1	d	1	-	0	1	0	0	d
Illegal inputs										
1				1			0			
		1		0				0		
1		1		0			1		Tedious!	
1		1		1				1		

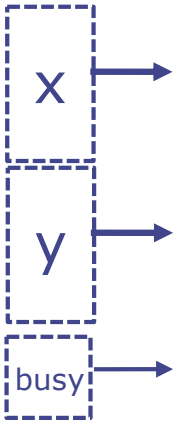
GCD interface and internal registers

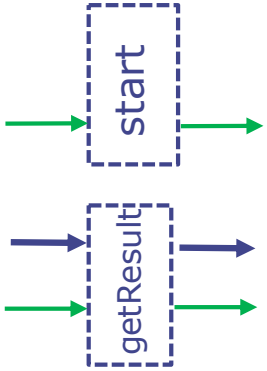
```
module mkGCD (GCD#(Bit#(n));
  Reg#(Bit#(n)) x <- mkReg(0);
  Reg#(Bit#(n)) y <- mkReg(0);
  Reg#(Bool) busy <- mkReg(False);
  rule gcd if (busy);
     if (x >= y) x <= x - y;
     else if (x != 0)
      begin x <= y; y <= x; end
  endrule
  method Action start(Bit#(n) a,
      Bit#(n) b) if (!busy);
    x <= a; y <= b; busy <= True;
  endmethod
  method ActionValue#(Bit#(n))
     getResult if (busy&&(x==0));
       busy <= False; return y;
  endmethod
endmodule
```





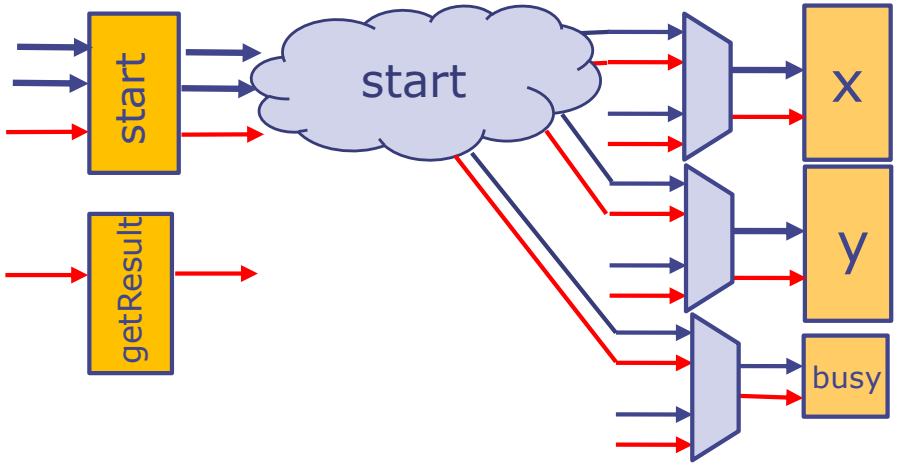

```
interface GCD#(Bit#(n));
  method Action start (Bit#(n) a,
                Bit#(n) b);
  method ActionValue#(Bit#(n))
                getResult;
endinterface
```



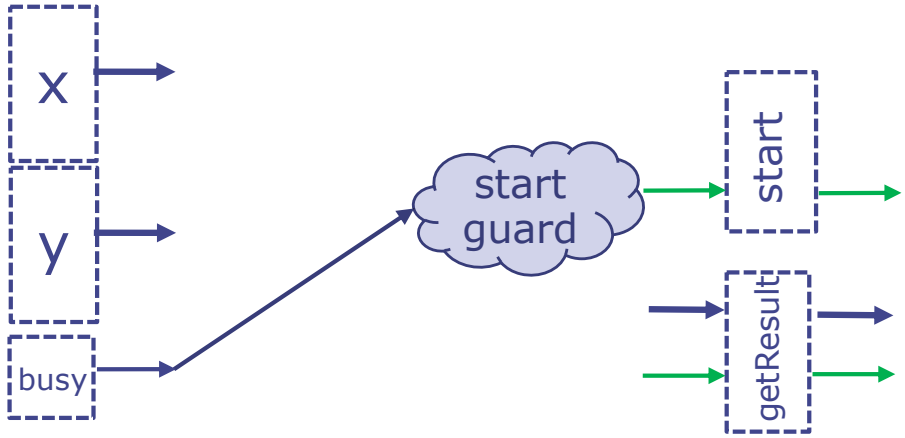


GCD start

```
module mkGCD (GCD#(Bit#(n)));
  Reg#(Bit#(n)) x <- mkReg(0);
  Reg#(Bit#(n)) y <- mkReg(0);
  Reg#(Bool) busy <- mkReg(False);
 rule gcd if (busy);
     if (x >= y) x <= x - y;
     else if (x != 0)
      begin x <= y; y <= x; end
  endrule
  method Action start(Bit#(n) a,
      Bit#(n) b) if (!busy);
    x <= a; y <= b; busy <= True;
  endmethod
  method ActionValue#(Bit#(n))
     getResult if (busy&&(x==0));
       busy <= False; return y;
  endmethod
endmodule
```

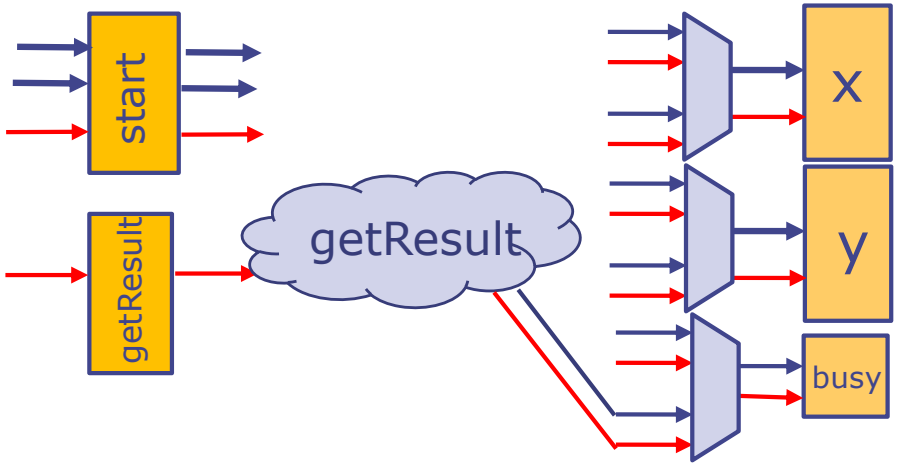



```
interface GCD#(Bit#(n));
  method Action start (Bit#(n) a,
                       Bit#(n) b);
  method ActionValue#(Bit#(n))
                       getResult;
endinterface
```

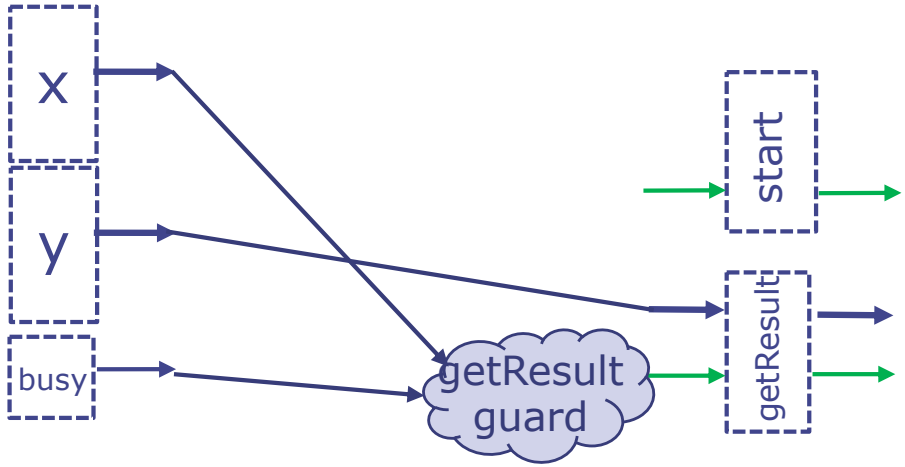


GCD getResult

```
module mkGCD (GCD#(Bit#(n)));
  Reg#(Bit#(n)) x <- mkReg(0);
  Reg#(Bit#(n)) y <- mkReg(0);
  Reg#(Bool) busy <- mkReg(False);
 rule gcd if (busy);
     if (x >= y) x <= x - y;
     else if (x != 0)
      begin x <= y; y <= x; end
  endrule
  method Action start(Bit#(n) a,
      Bit#(n) b) if (!busy);
    x <= a; y <= b; busy <= True;
  endmethod
  method ActionValue#(Bit#(n))
     getResult if (busy&&(x==0));
       busy <= False; return y;
  endmethod
endmodule
```



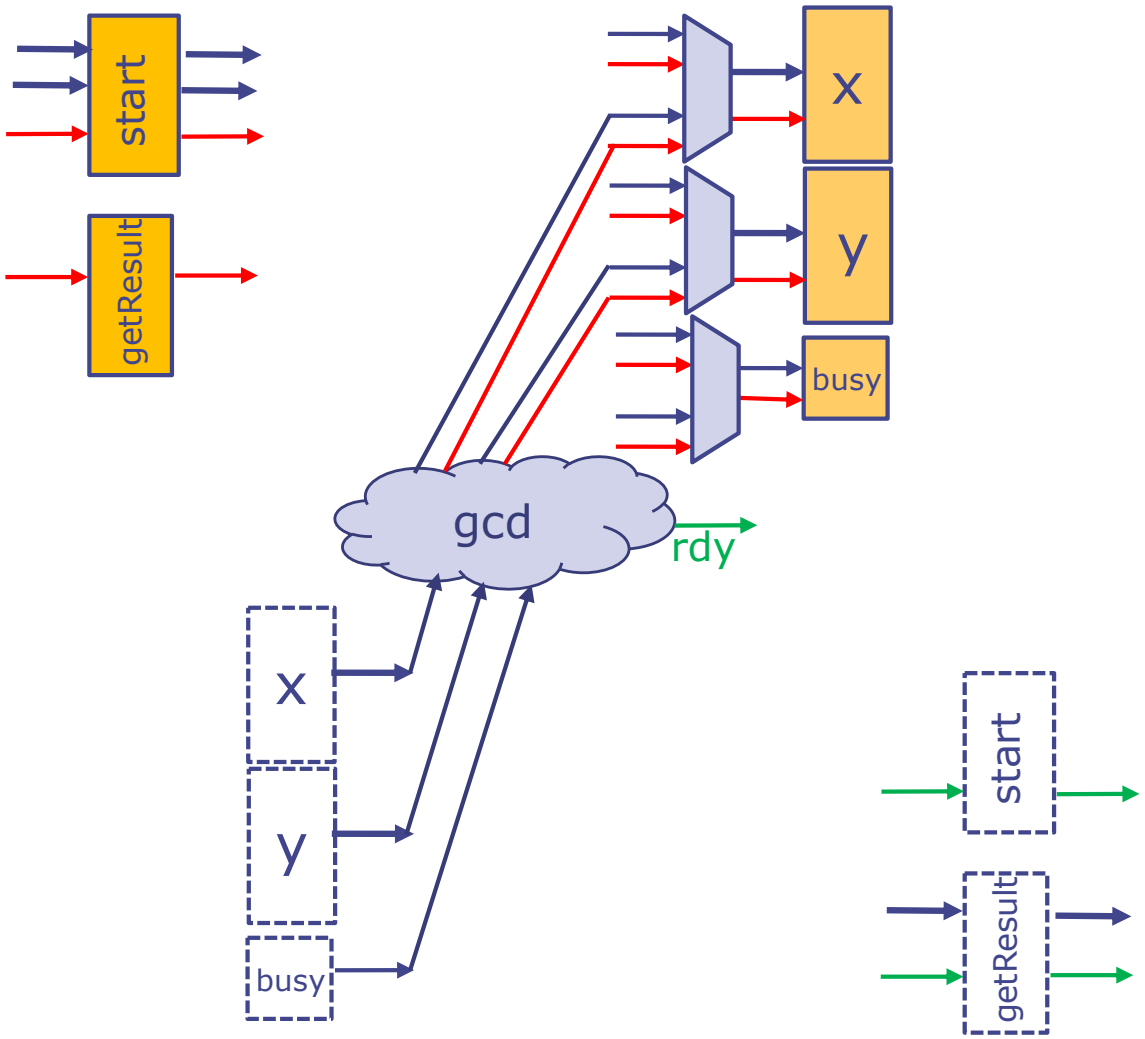

```
interface GCD#(Bit#(n));
  method Action start (Bit#(n) a,
                       Bit#(n) b);
  method ActionValue#(Bit#(n))
                       getResult;
endinterface
```



GCD rule

```
module mkGCD (GCD#(Bit#(n)));
  Reg#(Bit#(n)) x <- mkReg(0);
  Reg#(Bit#(n)) y <- mkReg(0);
  Reg#(Bool) busy <- mkReg(False);
  rule gcd if (busy);
     if (x >= y) x <= x - y;
     else if (x != 0)
      begin x <= y; y <= x; end
  endrule
  method Action start(Bit#(n) a,
      Bit#(n) b) if (!busy);
    x <= a; y <= b; busy <= True;
  endmethod
  method ActionValue#(Bit#(n))
     getResult if (busy&&(x==0));
       busy <= False; return y;
  endmethod
endmodule
```

```
interface GCD#(Bit#(n));
  method Action start (Bit#(n) a,
                       Bit#(n) b);
  method ActionValue#(Bit#(n))
                       getResult;
endinterface
```

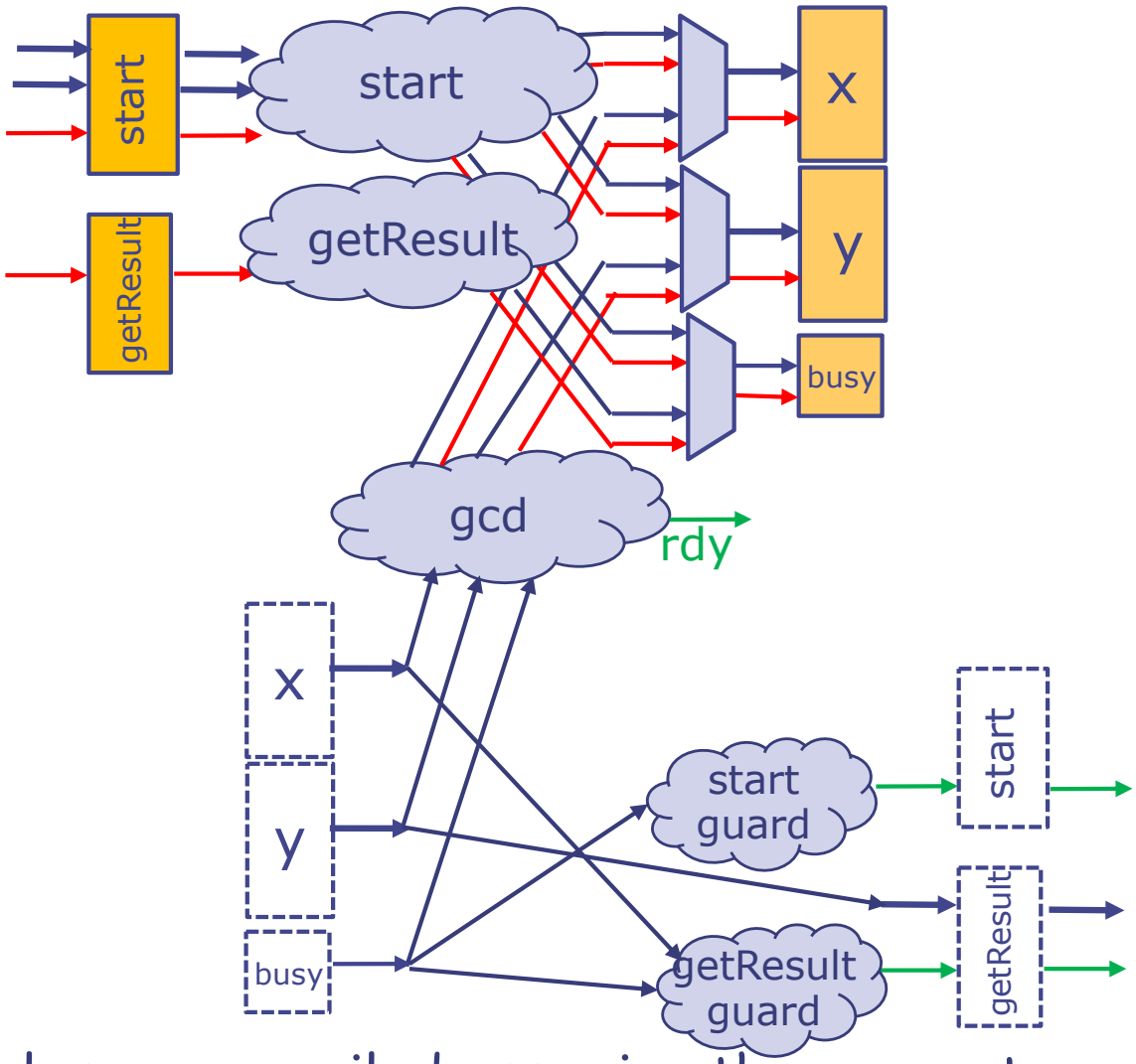


Rules are compiled to ensure that the output en signals are true only when the rule guard rdy is true

GCD

```
module mkGCD (GCD#(Bit#(n)));
  Reg#(Bit#(n)) x <- mkReg(0);
  Reg#(Bit#(n)) y <- mkReg(0);
  Reg#(Bool) busy <- mkReg(False);
 rule gcd if (busy);
     if (x >= y) x <= x - y;
     else if (x != 0)
      begin x <= y; y <= x; end
  endrule
  method Action start(Bit#(n) a,
      Bit#(n) b) if (!busy);
    x <= a; y <= b; busy <= True;
  endmethod
  method ActionValue#(Bit#(n))
     getResult if (busy&&(x==0));
       busy <= False; return y;
  endmethod
endmodule
```

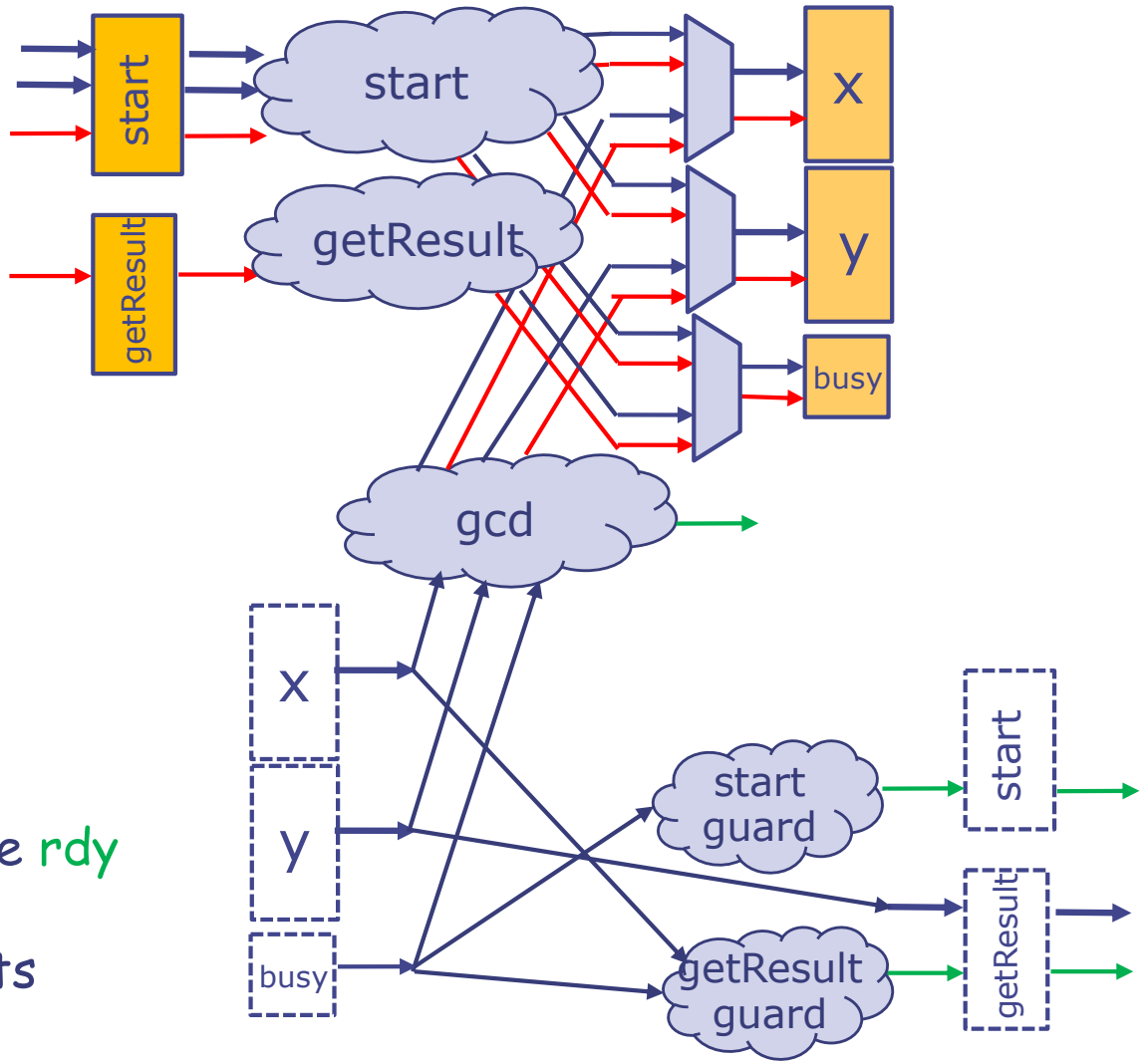
```
interface GCD#(Bit#(n));
  method Action start (Bit#(n) a,
                       Bit#(n) b);
  method ActionValue#(Bit#(n))
                       getResult;
endinterface
```



Rules are compiled assuming they execute every cycle; later we will see how to suppress the execution of a rule if needed

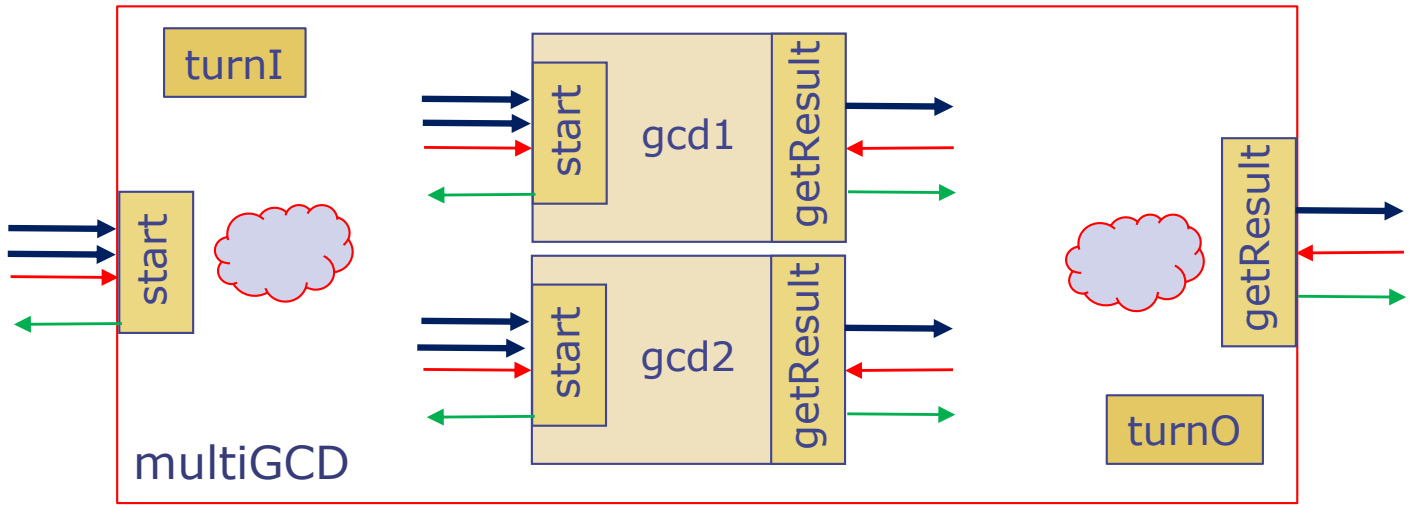
GCD: Are emux inputs guaranteed to be disjoint?

```
module mkGCD (GCD#(Bit#(n)));
  Reg#(Bit#(n)) x <- mkReg(0);
  Reg#(Bit#(n)) y <- mkReg(0);
  Reg#(Bool) busy <- mkReg(False);
 rule gcd if (busy);
     if (x >= y) x <= x - y;
     else if (x != 0)
      begin x <= y; y <= x; end
  endrule
  method Action start(Bit#(n) a,
      Bit#(n) b) if (!busy);
    x <= a; y <= b; busy <= True;
  endmethod
  method ActionValue#(Bit#(n))
     getResult if (busy&&(x==0));
       busy <= False; return y;
  endmethod
endmodule
```

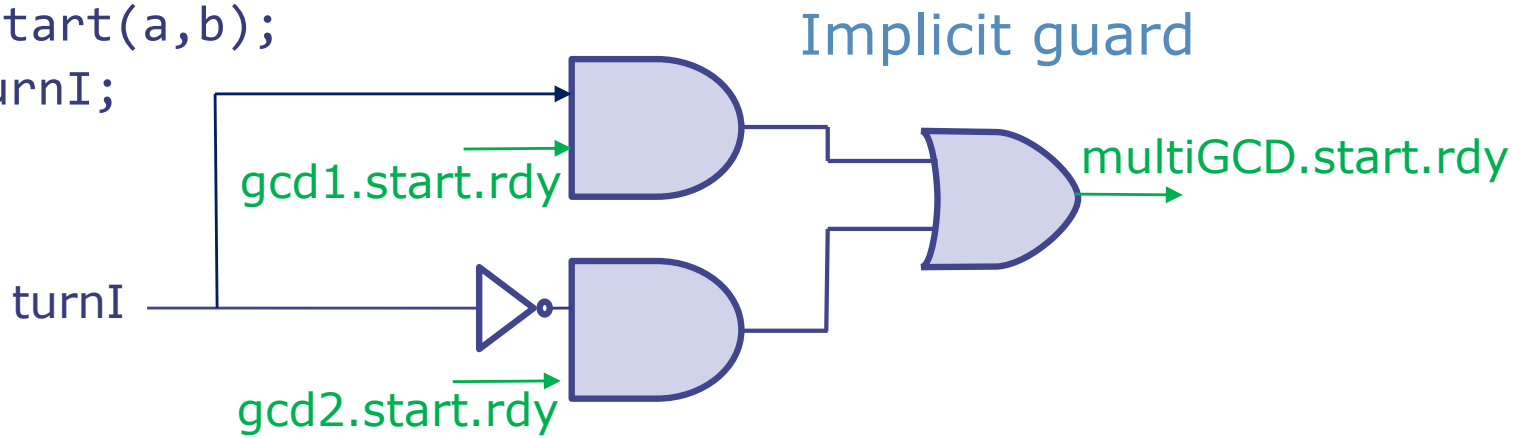


For both x emux and y emux, the rdy of start and gcd are disjoint
For busy emux: earlier arguments apply, i.e., the rdy of start and getResult are disjoint

Ready Signal in Multi GCD



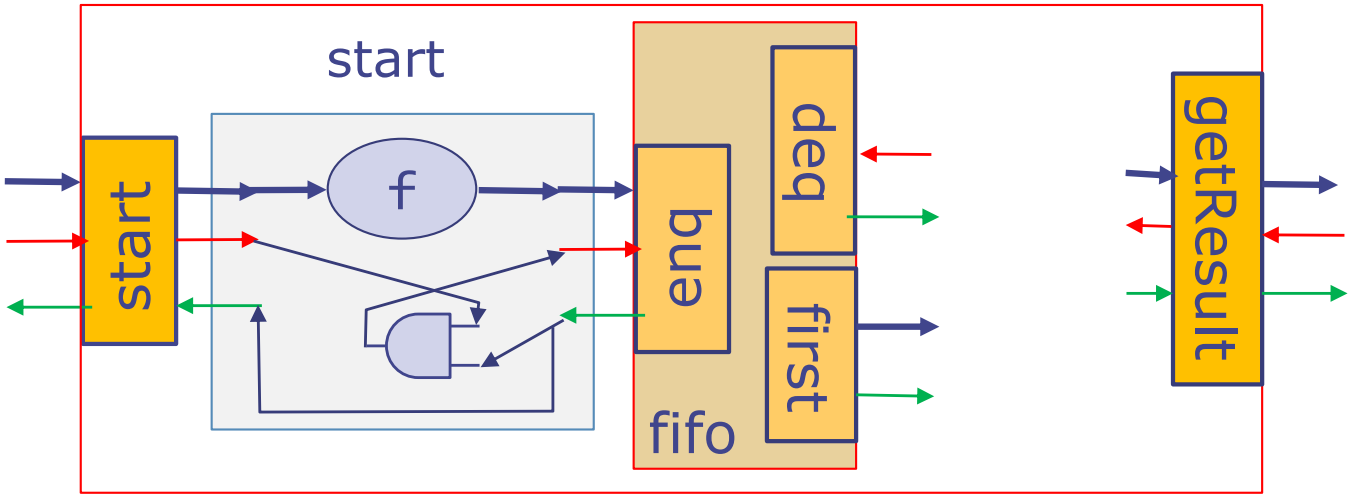

```
method Action start(Bit#(n) a, Bit#(n) b);
   if (turnI) gcd1.start(a,b);
    else gcd2.start(a,b);
   turnI <= !turnI;
endmethod
```



Streaming a function Circuit

```
interface GMFI#(numeric n);
  method Action start (Bit#(n) a);
  method ActionValue(Bit#(n))
                  getResult;
endinterface
```

```
module mkstreamf (GMFI#(n));
  Fifo#(1, Bit#(n)) fifo  <- mkFifo;
  method Action start(Bit#(n)  x);
    fifo.enq(f(x));
  endmethod
  method ActionValue (Bit#(n)) getResult;
    fifo.deq;
    return fifo.first();
  endmethod
endmodule
```



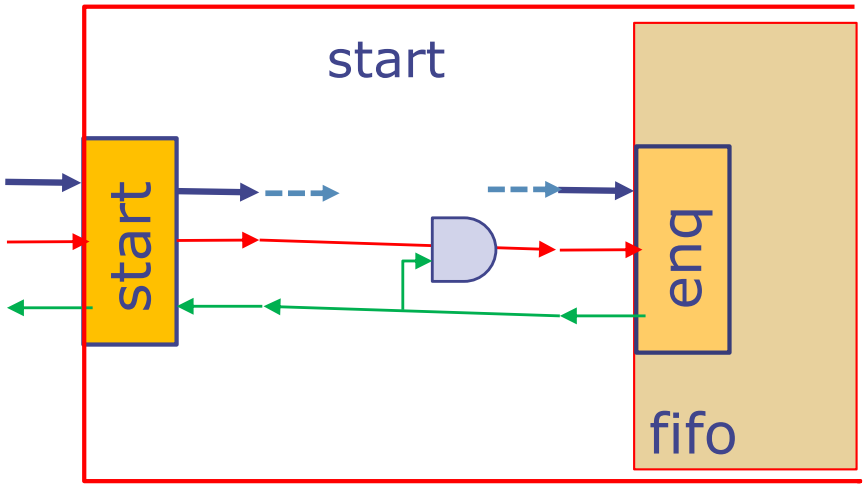
- Compiling a method
  - Generate data and enable for the called methods
  - Generate the ready signal

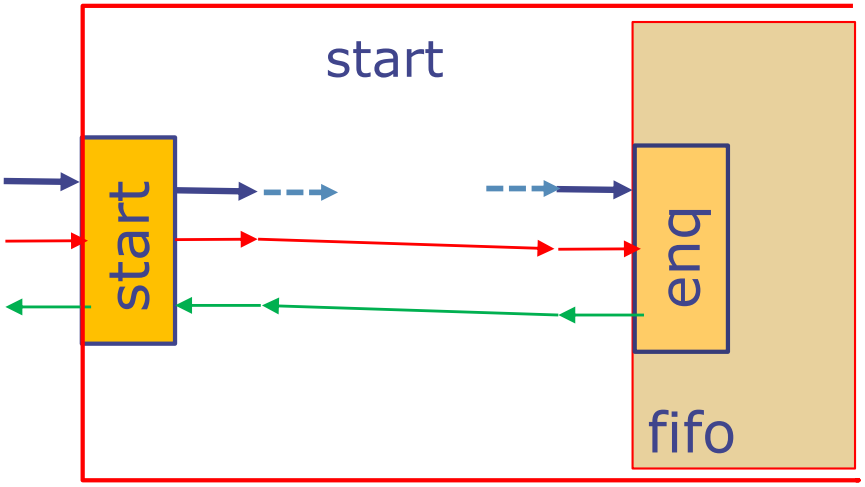
Notice that enq.en cannot be True unless enq.rdy is true;

The circuit could be simpler

```
method Action start(Bit#(n)  x);
    fifo.enq(f(x));
  endmethod
```

When method A calls method B, there is no need to combine the A.en to B.rdy



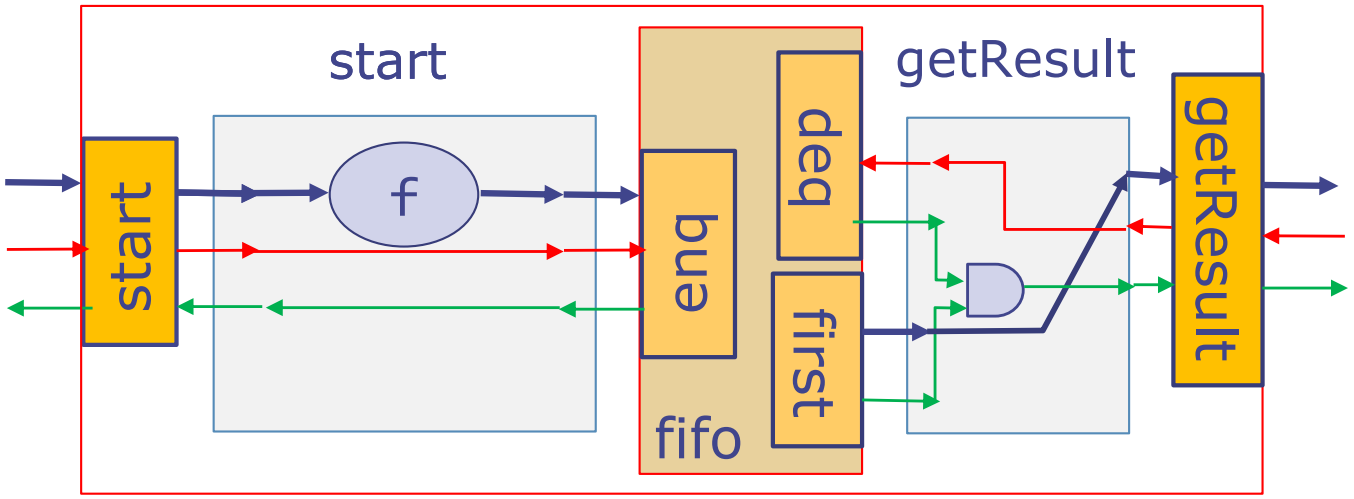


- The And gate is not needed because
  - start.en can't be true unless start.rdy is true
  - start.rdy can't be true unless its called method is ready, i.e., enq.rdy is true
  - therefore enq.en can't be true unless enq.rdy is true

Streaming a function Circuit

```
interface GMFI#(numeric n);
  method Action start (Bit#(n) a);
  method ActionValue(Bit#(n))
                  getResult;
endinterface
```

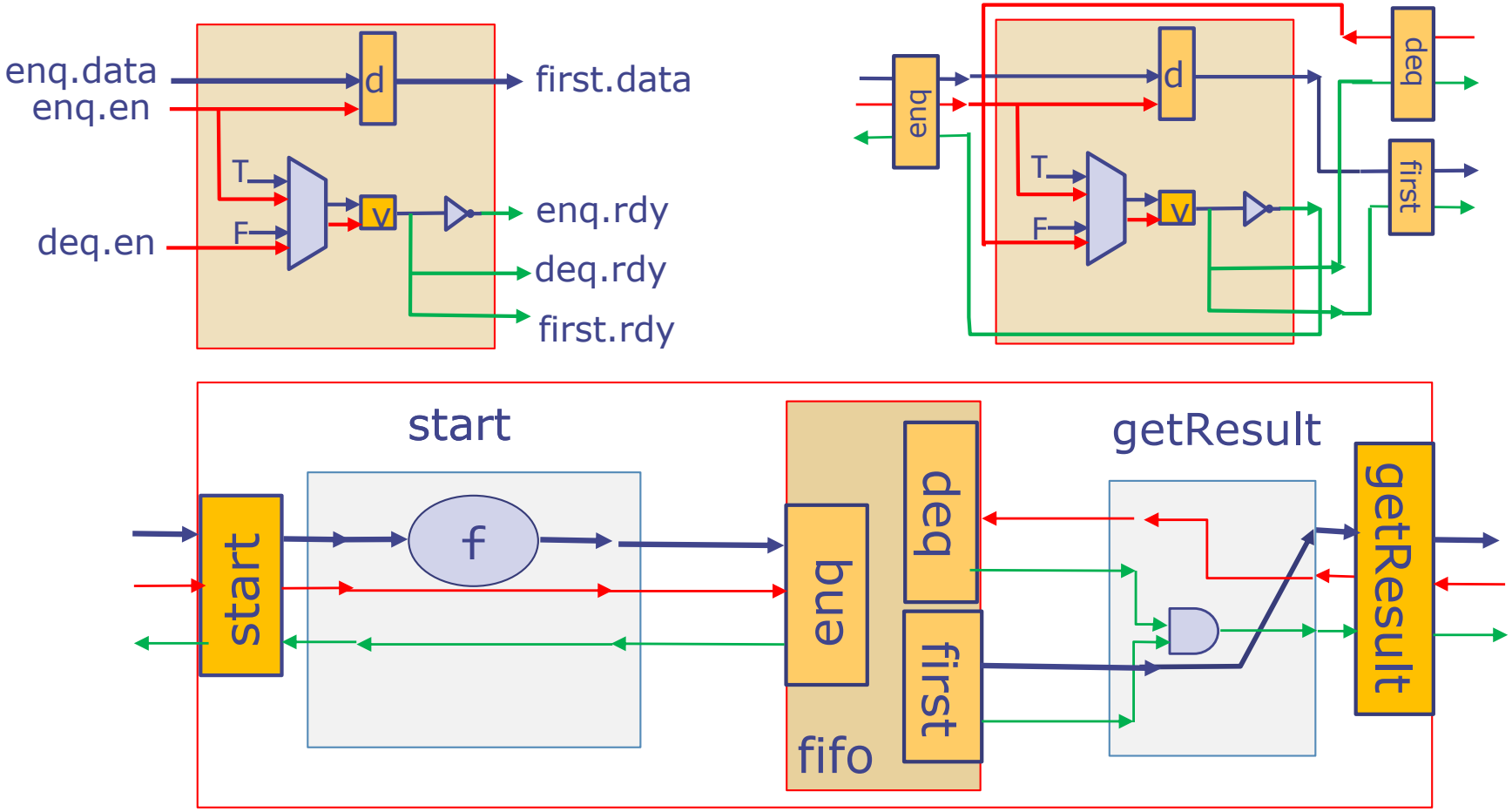
```
module mkstreamf (GMI#(n));
  Fifo#(1, Bit#(n)) fifo  <- mkFifo;
  method Action start(Bit#(n)  x);
    fifo.enq(f(x));
  endmethod
  method ActionValue (Bit#(n)) getResult;
    fifo.deq;
    return fifo.first();
  endmethod
endmodule
```



- Compiling a method
  - Generate data and enable for the called methods
  - Generate the ready signal

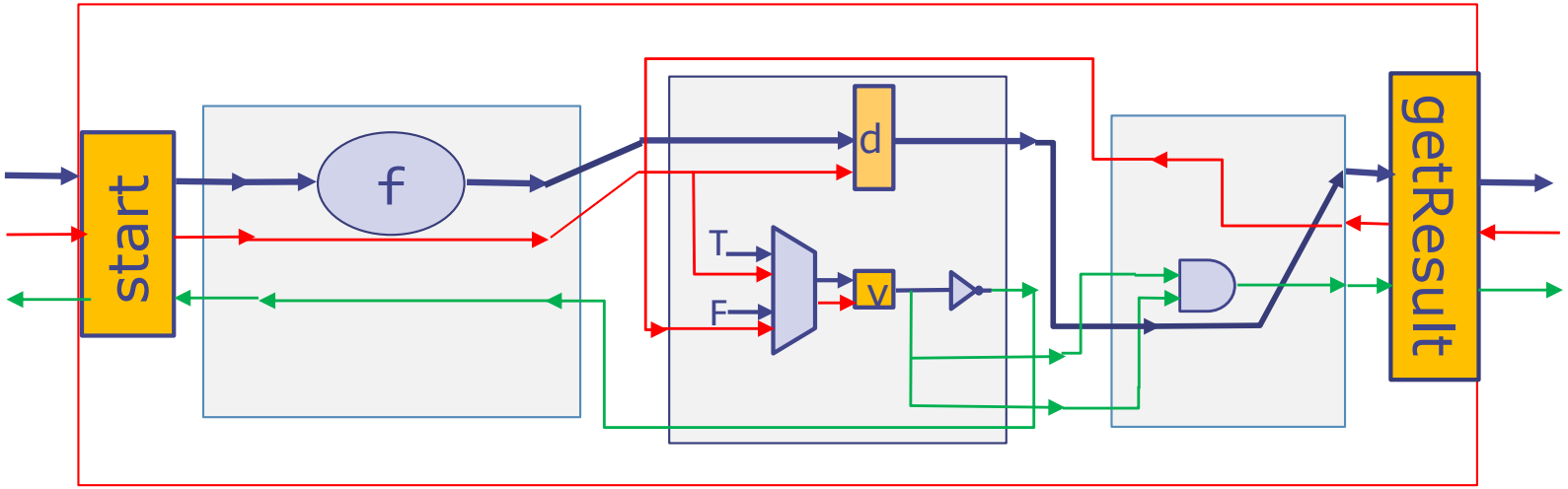
Notice, deq.en cannot be True unless deq.rdy is true

Substituting the fifo circuit



- We can "in-line" the fifo module by eliminating the interface boxes and connecting the wires appropriately

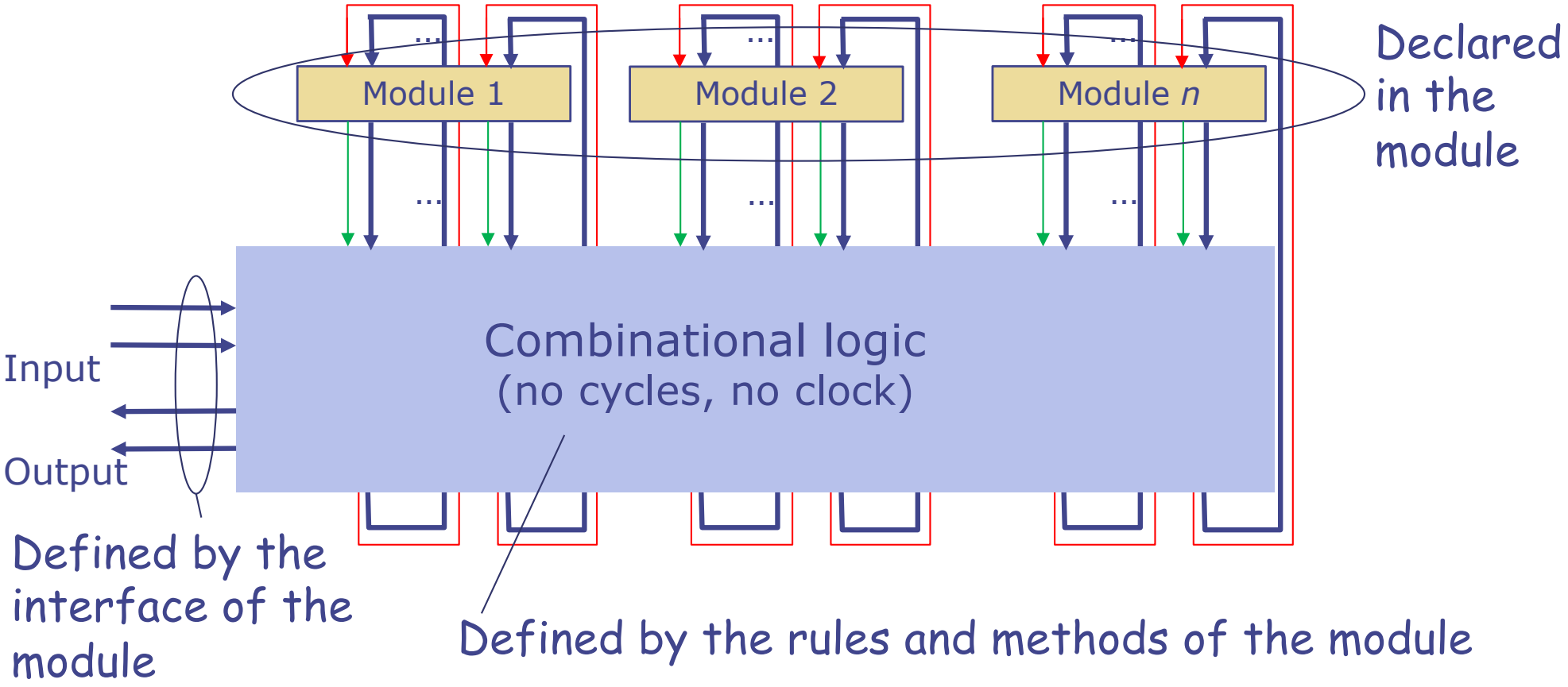
After substituting the fifo circuit



- This is a sequential machine with two registers
- Composition of modules, i.e., sequential machines, results in a module, i.e., a sequential machine
  - Notice, that the guards of both start and gerResult methods depend upon the value of the v register only

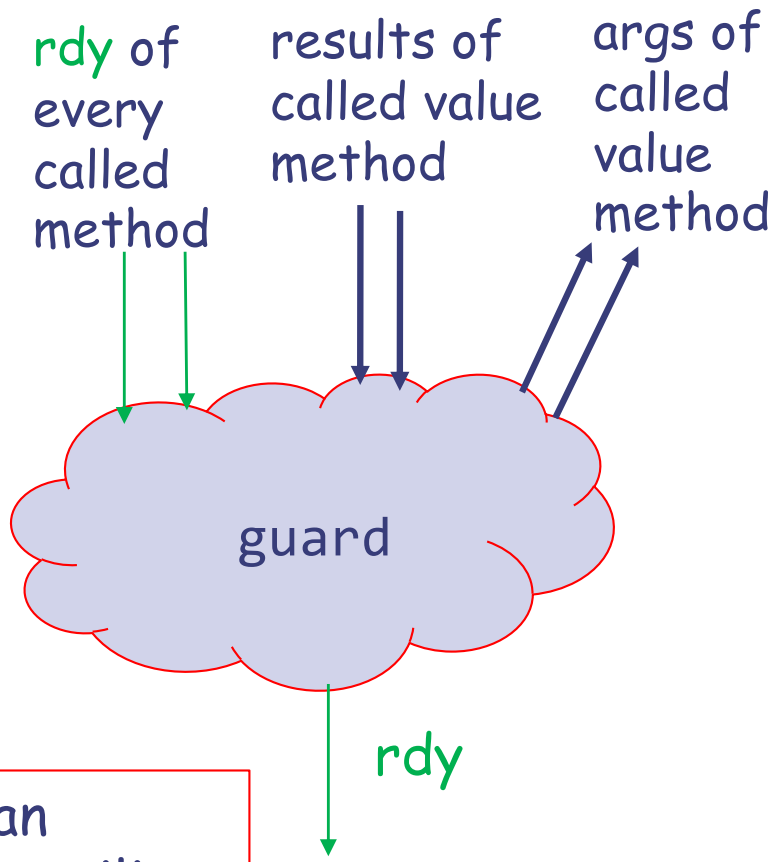
Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine



Compiling Summary method's guard

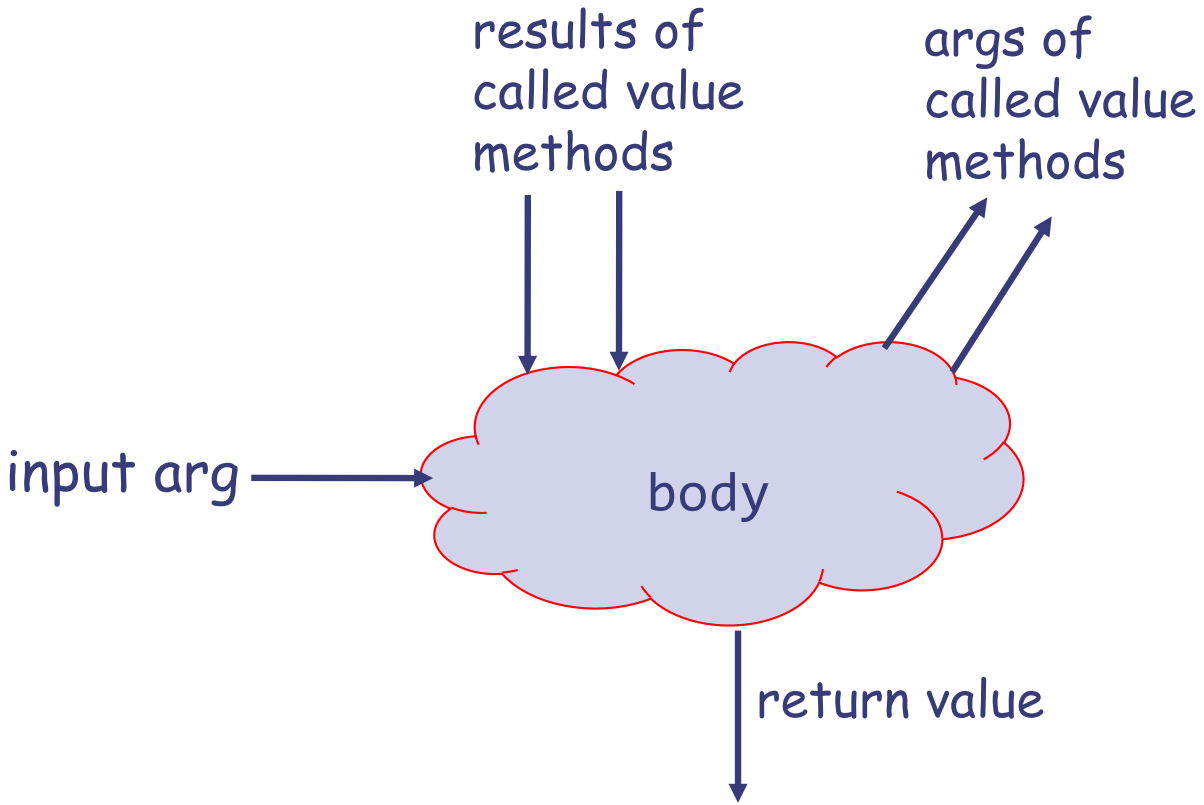
- Generate the ready signal for a method by
  - Compiling the guard expression to generate a ready signal
    - If no guard expression is given the ready signal is true
  - Combine the ready signals of the called methods with the guard expression's ready signal



The input argument and the value from an ActionValue method cannot be used in compiling guards; this is to avoid the creation of combinational cycles

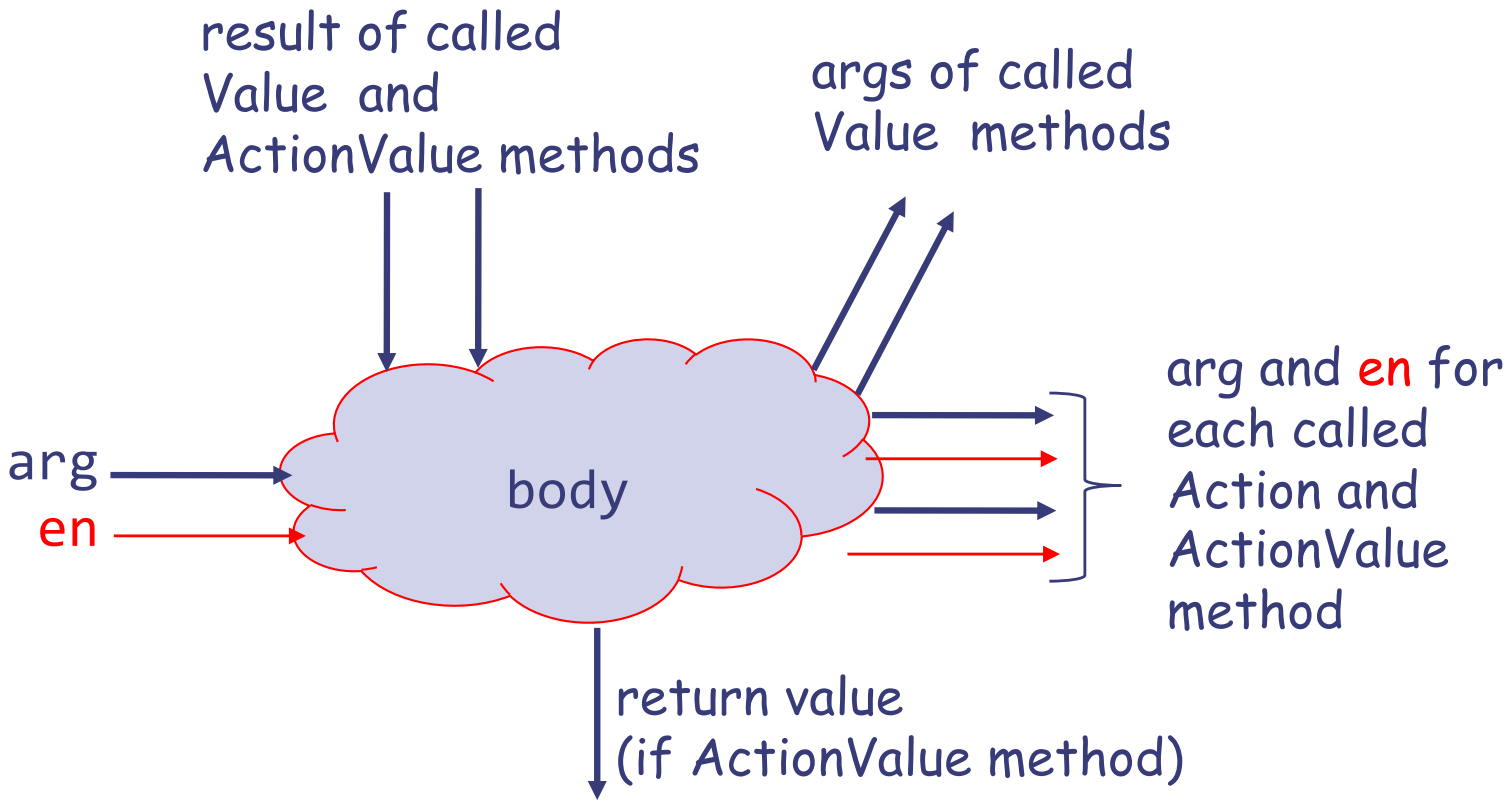
BSV syntax prohibits calling action value methods in a guard expression

Compiling a value method's body



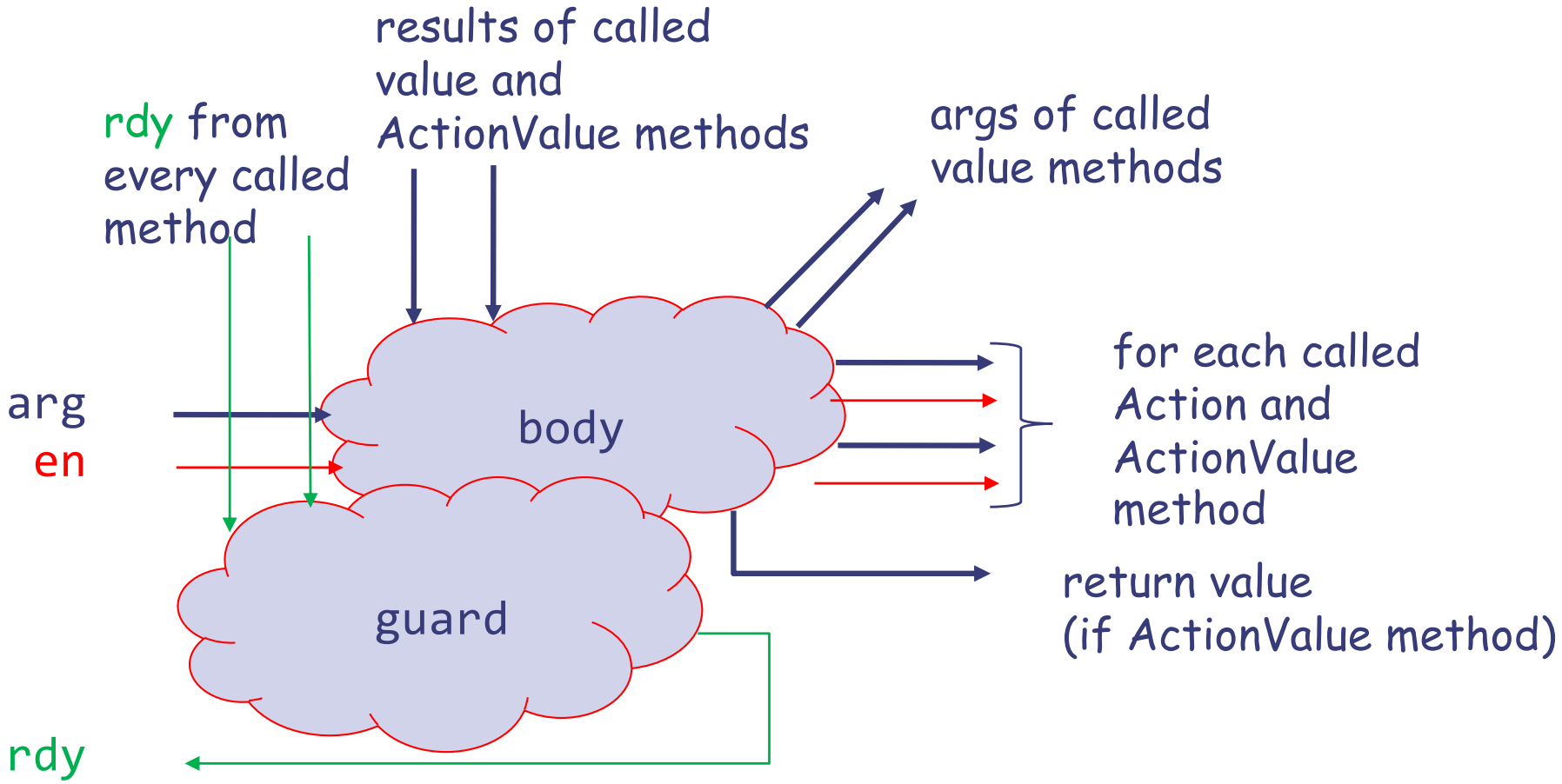
- Compiling a value method is like compiling an expression

Compiling an Action or ActionValue method's body



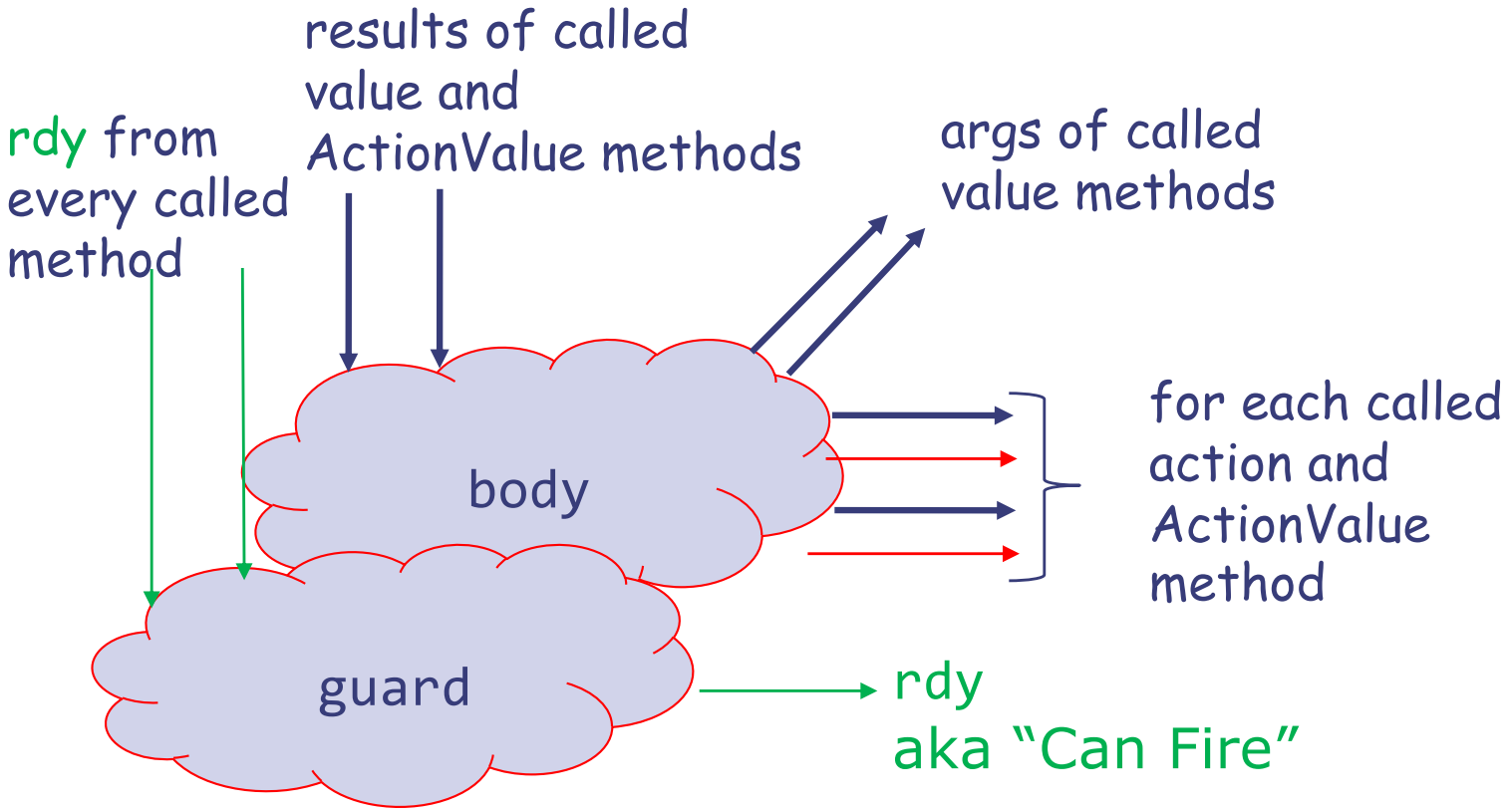
- For each of the *called* Action and ActionValue method, generate the argument data and associated enable
- The enable signal of a method is also used as an input in compiling the method's body but not in compiling the method's guard

Putting it all together



- The arguments of a value method or ActionValue method cannot depend upon its own result
  - Bluespec syntax guarantees this

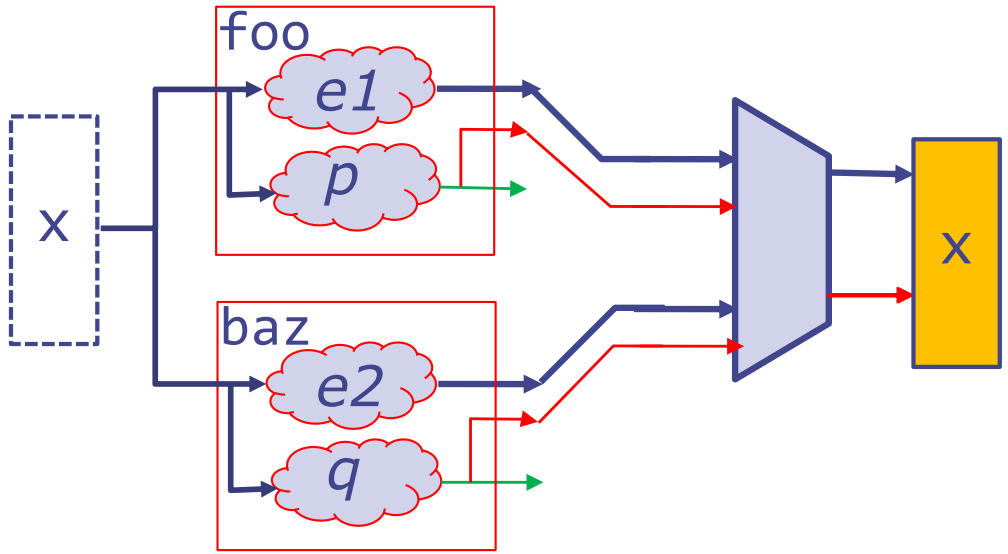
Compiling rules



- Rules are compiled similarly to Action methods: a rule has no input argument or return value
- A rule's rdy signal is known as "Can Fire"
- The en for none of the Action and ActionValue methods can be true if "Can Fire" is False

An issue in combining multiple sources

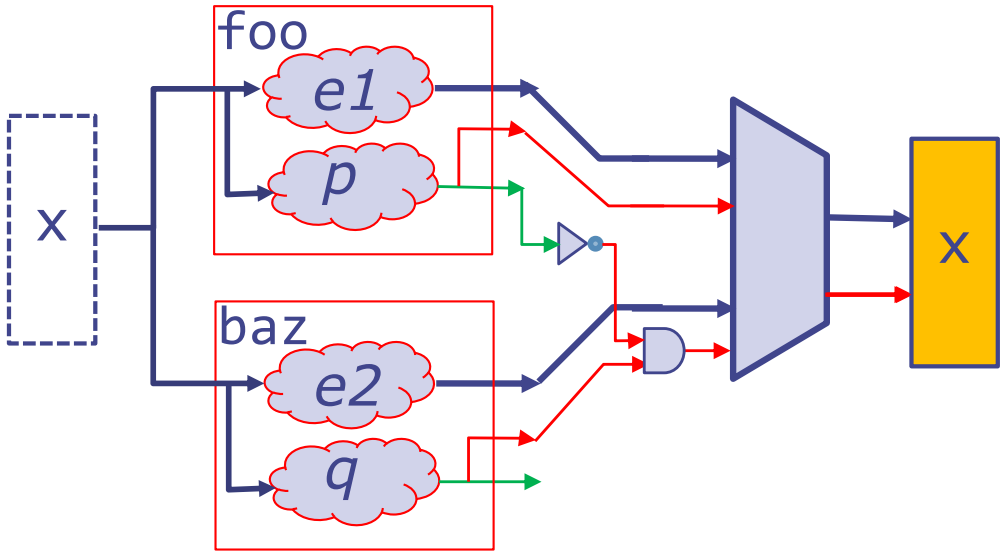
```
module mkEx (...);
  Reg#(Bit#(n)) x <- mkRegU;
  rule foo if p(x);
     x <= e1;
  endrule
  rule baz if q(x);
     x <= e2;
  endmethod
endmodule
```



- The procedure we have given will result in the above circuit and will execute rules foo and baz concurrently
- But to avoid a double write error, the compiler has to ensure that
  - Either p and q are mutually exclusive and thus, rules foo and baz will not be rdy to execute at the same time,
  - Otherwise the compiler must prevent one of the rules from executing

Preventing a rule from from executing

```
module mkEx (...);
  Reg#(Bit#(n)) x <- mkRegU;
  rule foo if p(x);
     x <= e1;
  endrule
  rule baz if q(x);
     x <= e2;
  endmethod
endmodule
```



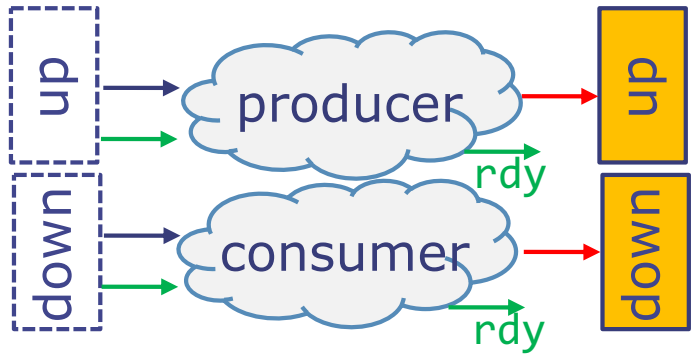
- Suppose p and q can be true simultaneously. We can give priority to (say) rule foo over baz by preventing the execution of baz if foo "can fire"
- Preventing a rule from firing is the same as not letting it update any state.
- Rule baz can execute only when the "can fire" signal of rule foo is false

Up-Down counter

```
module mkUpDownCounter (UpDownCounter);
   Reg#(Bit#(8)) ctr <- mkReg (0);
   method ActionValue#(Bit#(8)) up if (ctr < 255);
      ctr <=  ctr+1; return ctr;
   endmethod
   method ActionValue#(Bit#(8)) down if (ctr > 0);
      ctr <=  ctr-1; return ctr;
   endmethod
endmodule
```

Using the counter

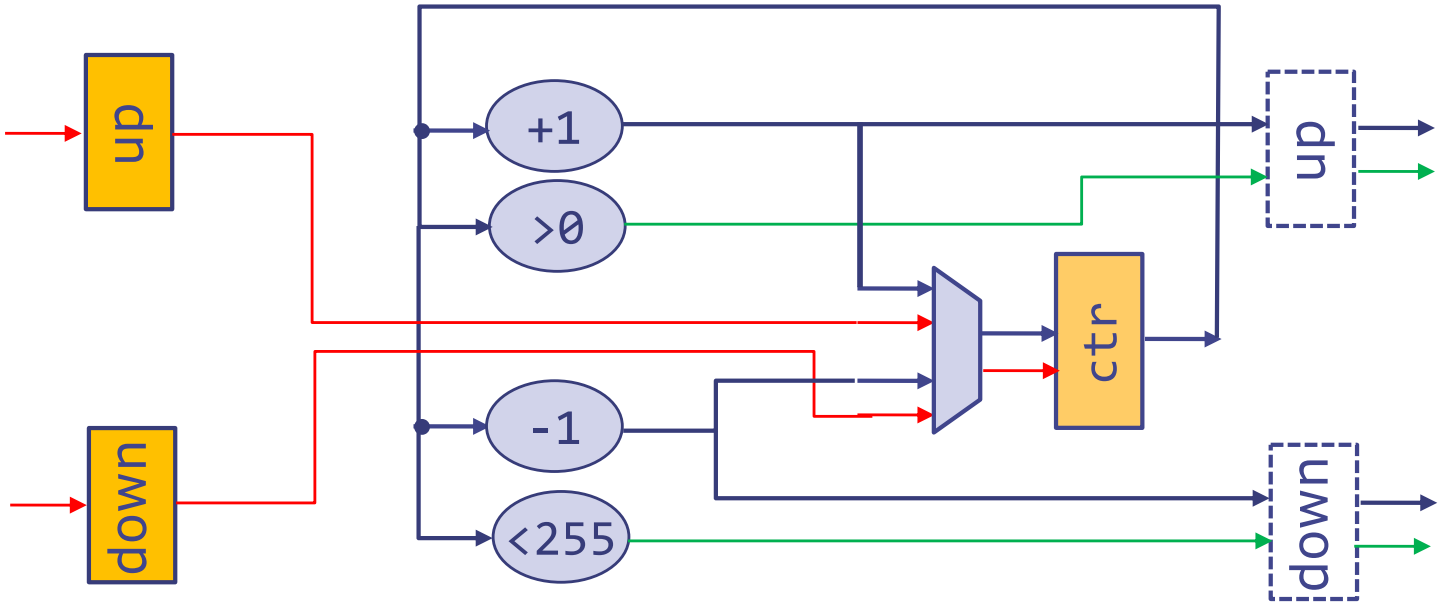
```
   UpDownCounter Bit#(8) x
            <- mkUpDownCounter;
   rule producer;
     ... x.up ...;
   endmethod
   rule consumer;
     ... x.down ...;
   endmethod
```



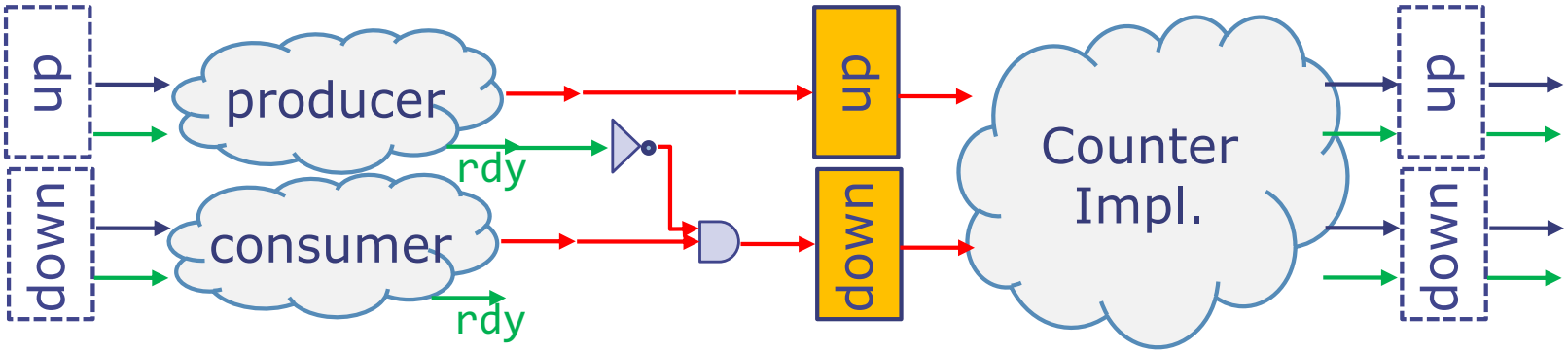
Concurrent execution of producer and consumer will cause a double write error, and thus, must be prevented

Inside the Up-Down counter

```
module mkUpDownCounter (UpDownCounter);
   Reg#(Bit#(8)) ctr <- mkReg (0);
   method ActionValue#(Bit#(8)) up if (ctr < 255);
      ctr <=  ctr+1; return ctr;
   endmethod
   method ActionValue#(Bit#(8)) down if (ctr > 0);
      ctr <=  ctr-1; return ctr;
   endmethod
endmodule
```



Up-Down counter
How to avoid the double write error?




```
UpDownCounter Bit#(8) x
        <- mkUpDownCounter;
rule producer;
  ... x.up ...;
endmethod
rule consumer;
  ... x.down ...;
endmethod
```

When producer's rdy is True, it makes consumer's en False, preventing it from making any state updates, and hence, no double write error

Preserving atomicity while preventing a rule from firing

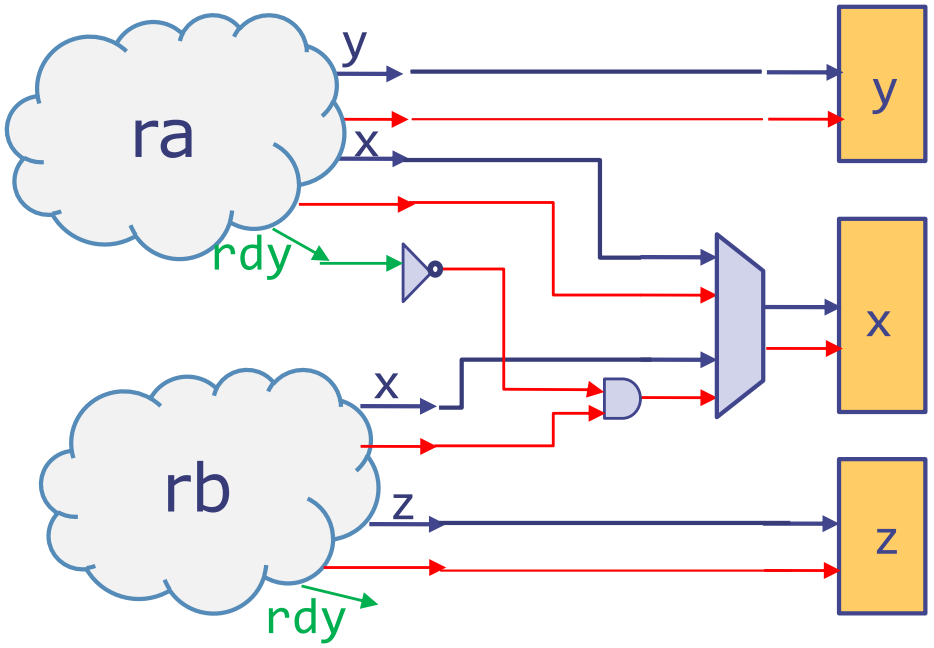
```
rule ra;
   x <= e1; y <= e2;
endmethod
rule rb;
   x <= e3; z <= e4;
endmethod
```

- ra and rb conflict because of a double write in x
- Suppose we want to prevent rb from firing

What is wrong with this circuit?

The atomicity of rule rb is violated: y may be updated without x being updated!

fix?



Preserving atomicity while preventing a rule from firing

```
rule ra;
   x <= e1; y <= e2;
endmethod
rule rb;
   x <= e3; z <= e4;
endmethod
```

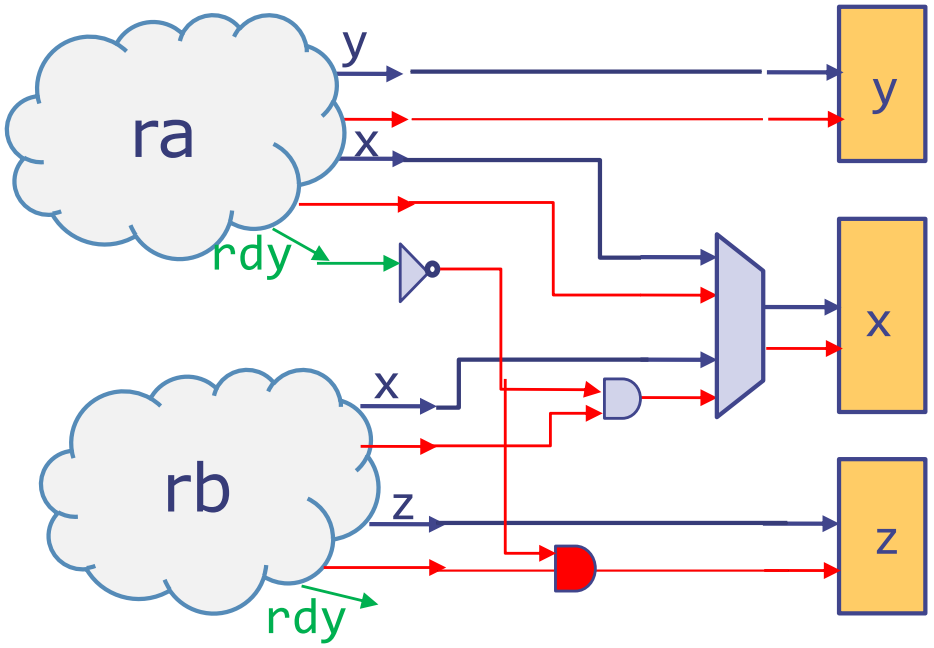
- ra and rb conflict because of a double write in x
- Suppose we want to prevent rb from firing

What is wrong with this circuit?

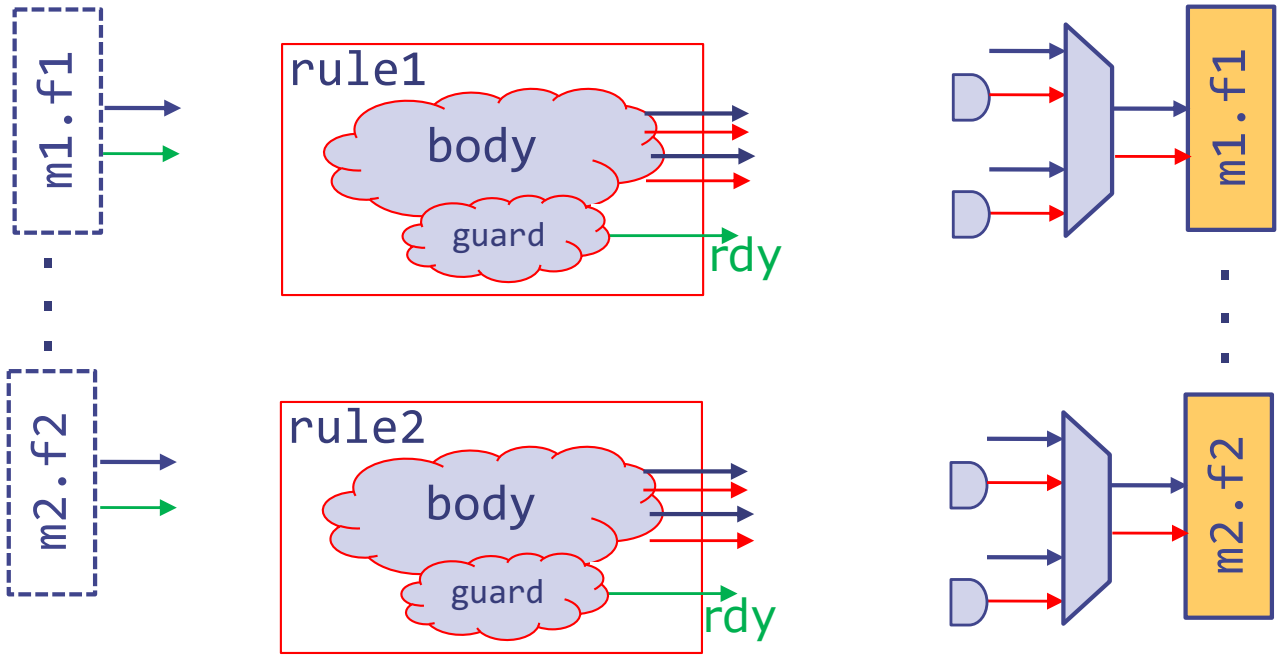
The atomicity of rule rb is violated: y may be updated without x being updated!

fix?

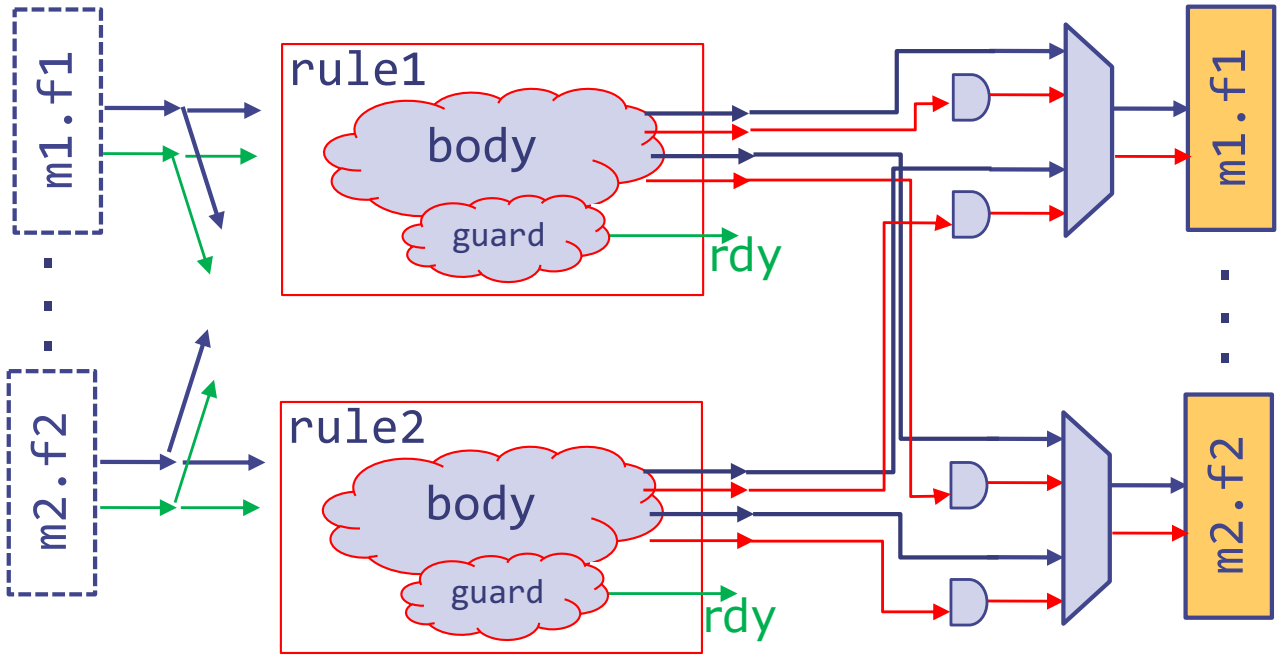
When we do not want a rule to fire all its state updates must be stopped



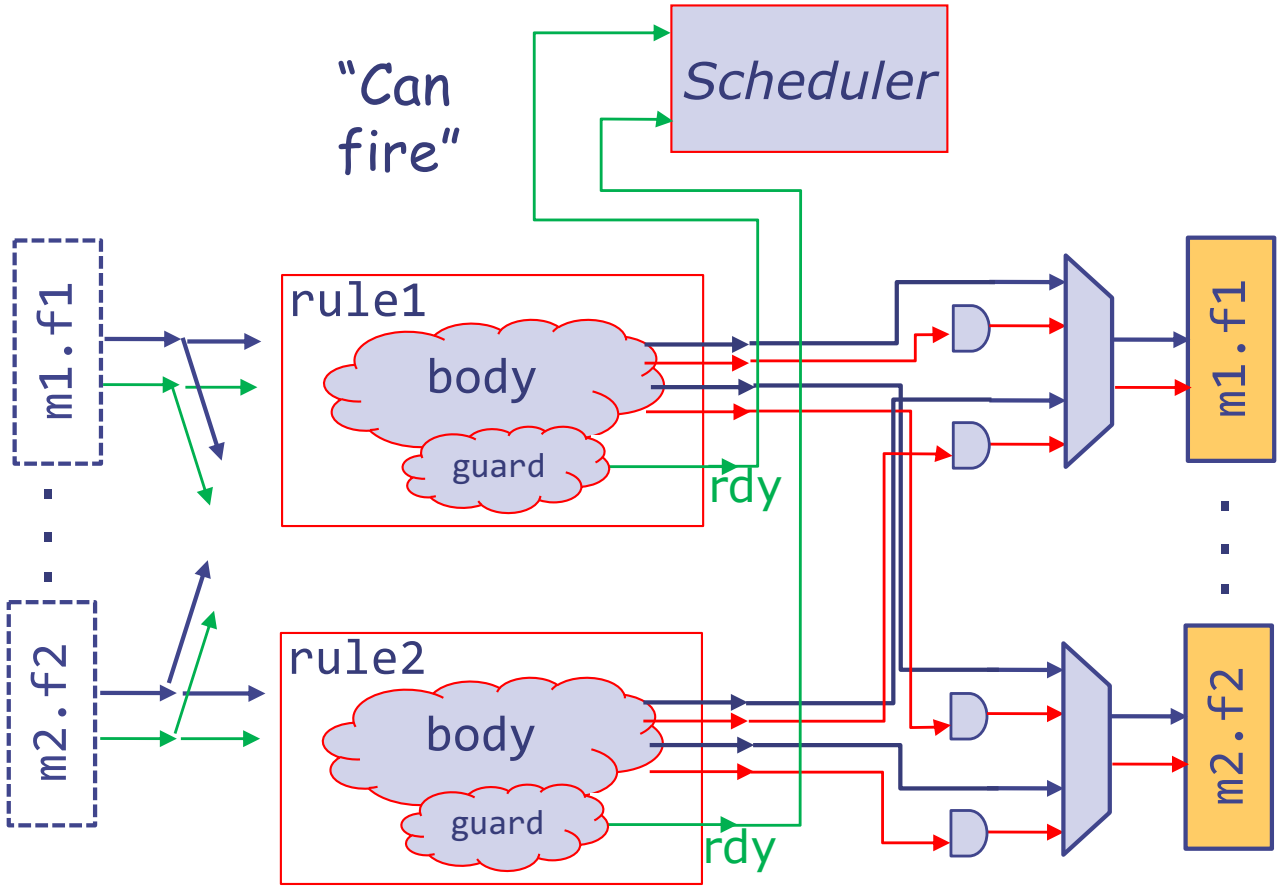
A general method for inhibiting rule execution



A general method for inhibiting rule execution

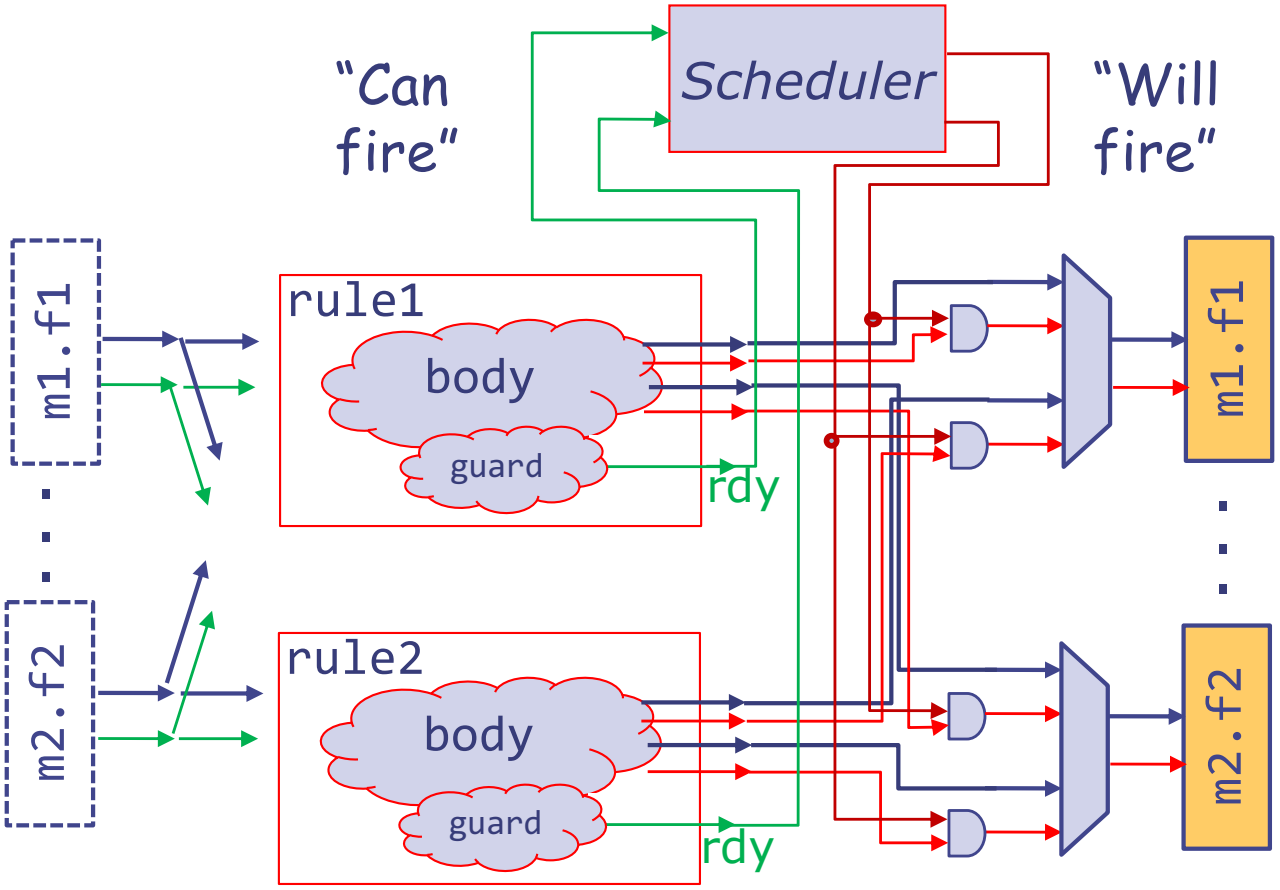


A general method for inhibiting rule execution



- We introduce a scheduler to control which rules among the ready rules should execute
  - We feed it the rdy signals of all the rules

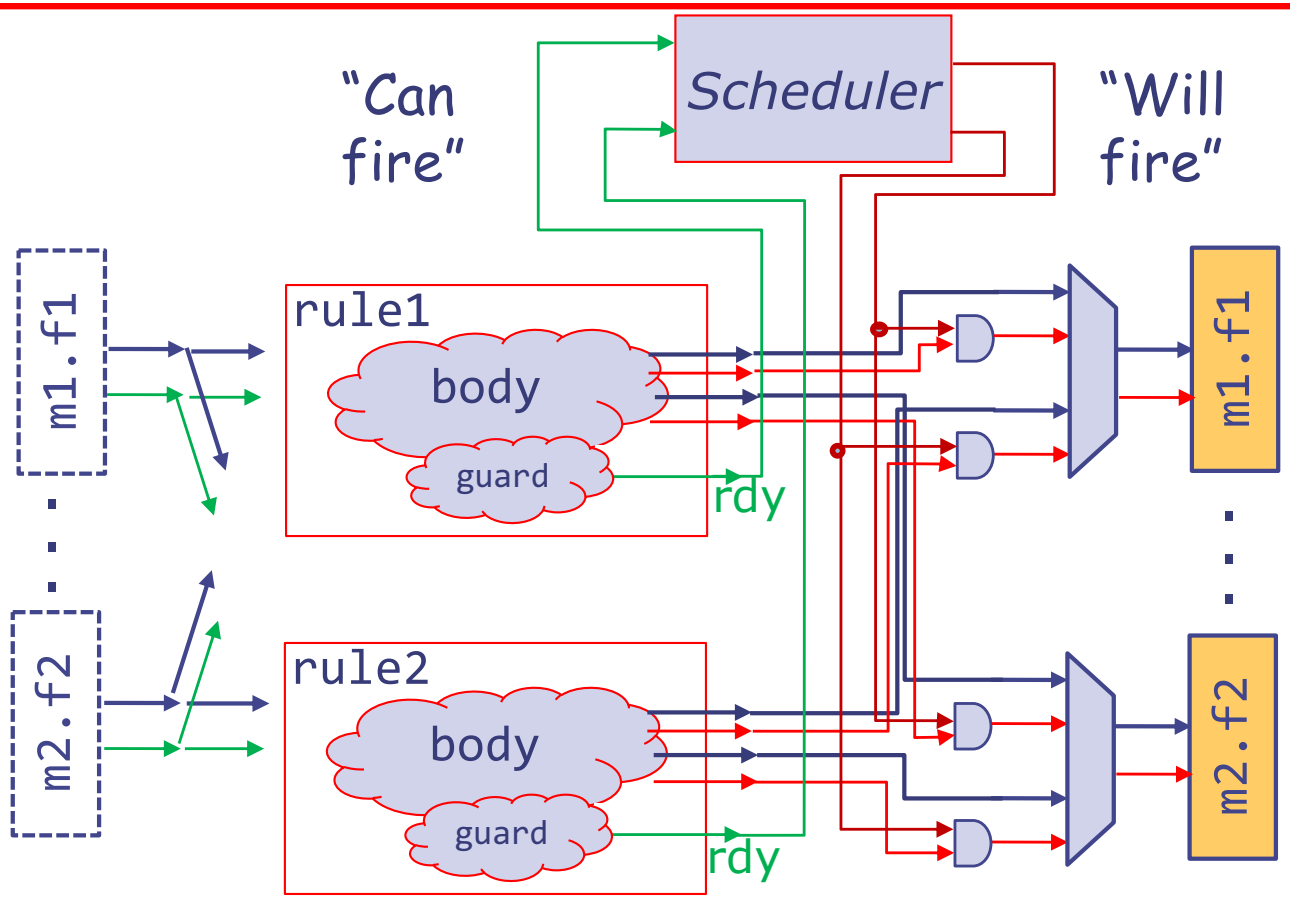
A general method for inhibiting rule execution



- We introduce a scheduler to control which rules among the ready rules should execute
  - We feed it the rdy signals of all the rules
- The scheduler lets only *non-conflicting* rules proceed
  - It turns off some of the "can fire" signals

A general method for inhibiting rule execution

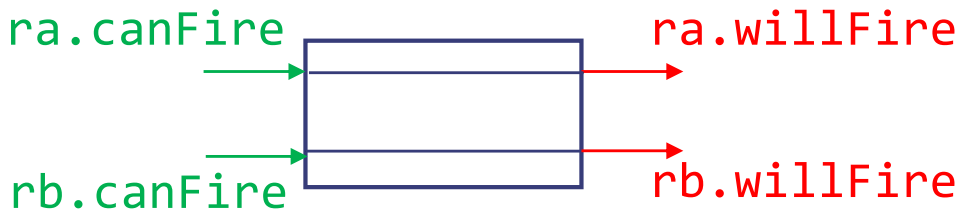
Scheduler is a pure combinational circuit with a small number of gates



- We introduce a scheduler to control which rules among the ready rules should execute
  - We feed it the rdy signals of all the rules
- The scheduler lets only *non-conflicting* rules proceed
  - It turns off some of the "can fire" signals

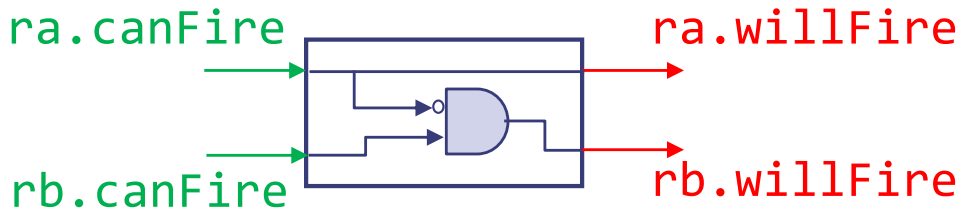
What is inside the scheduler

- Suppose rules ra and rb can be executed concurrently – no double write
  - Scheduler

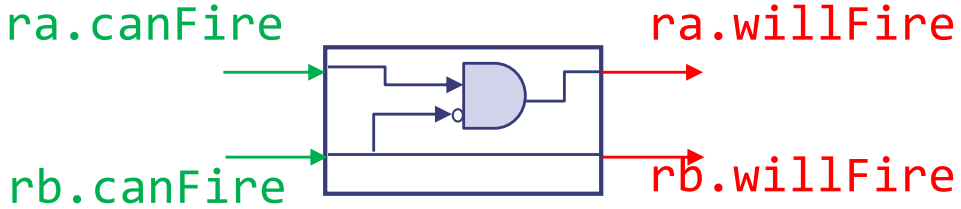


What is inside the scheduler

- Suppose rules ra and rb should not be executed concurrently
  - Schedule 1: rule ra has priority, i.e.,
    if ra can fire rb will not fire



  - Schedule 2: rule rb has priority, i.e.,
    if rb can fire ra will not fire



The choice is specified by scheduling annotations in the BSV program

Summary

- We have shown how to generate a sequential machine corresponding to any module
- Sequential machines are connected to each other by atomic rules and methods
- Sometimes we have to prevent the execution of a rule which is ready to execute to avoid double write errors
- We can easily design hardware schedulers to intervene and prevent the execution of any set of ready rules for whatever reason we want
  - Indeed, we will make use of this facility to enforce some more desirable properties of digital designs