Hardware Synthesis from Bluespec

Silvina Hanono Wachman
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

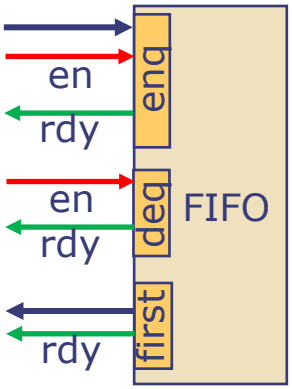
Happy 🥧 Day!

Guarded interfaces

- Modules with guarded interfaces are a new way of expressing sequential circuits
- Every method has a *guard* (*rdy* wire)
  - For some methods guard is always true
- The value returned by a method is meaningful only if its guard is true
- Every action method has an *enable signal* (*en* wire) and it can be invoked (en can be set to true) only if its guard is true
- Every method can have input data
- Value and ActionValue methods have output data

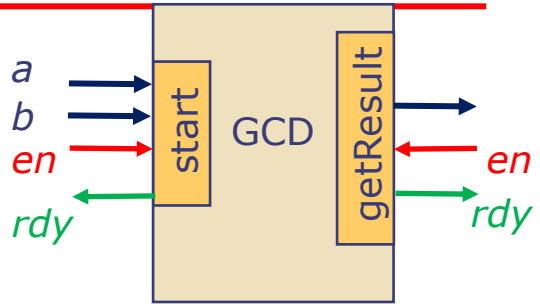
```
interface Fifo#(numeric type size, type t);
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface
```

notice, en and rdy wires are implicit



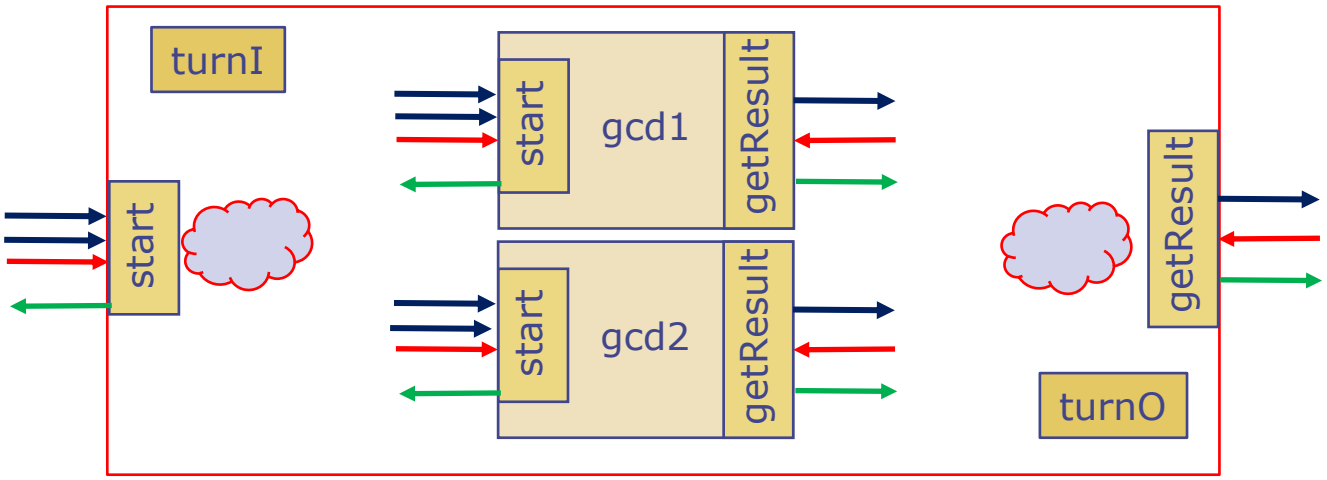
GCD Interface

```
interface GCD;
  method Action start (Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) getResult;
endinterface
```




```
module mkGCD (GCD);
  Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
  Reg#(Bool) busy_flag <- mkReg(False);
  rule gcd;
     if (x >= y) begin x <= x - y; end //subtract
     else if (x != 0) begin x <= y; y <= x; end //swap
  endrule
  method Action start(Bit#(32) a, Bit#(32) b) if(!busy_flag);
    x <= a; y <= b; busy_flag <= True;
  endmethod
  method ActionValue#(Bit#(32)) getResult if(busy_flag&&(x==0));
    busy_flag <= False; return y;
  endmethod
endmodule
```

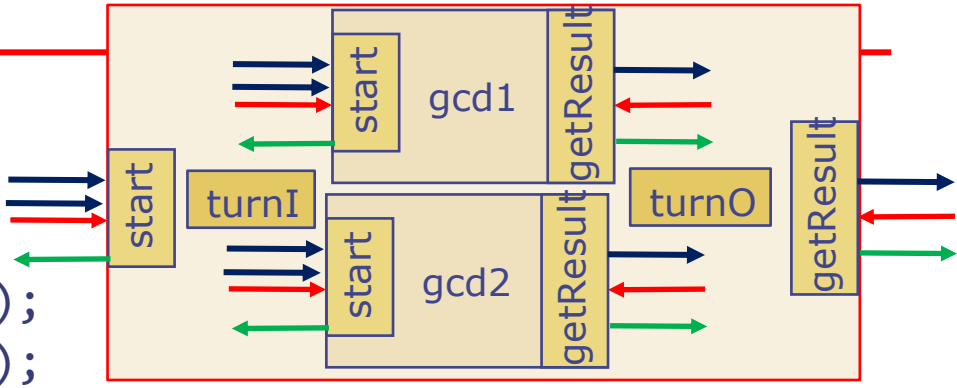
Power of Abstraction:
Another GCD implementation



- A GCD module with the same interface but with twice the throughput; uses two gcd modules in parallel
- `turnI` is used by the start method to direct the input to the gcd whose turn it is and then `turnI` is flipped
- Similarly, `turnO` is used by getResult to get the output from the appropriate gcd, and then `turnO` is flipped

```
interface GCD;
  method Action start (Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) getResult;
endinterface
```

High-throughput GCD code



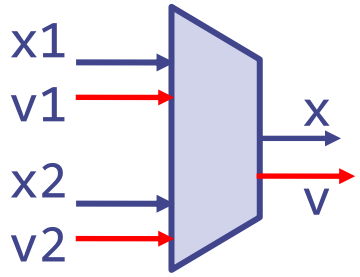

```
module mkMultiGCD (GCD);
  GCD gcd1 <- mkGCD();
  GCD gcd2 <- mkGCD();
  Reg#(Bool) turnI <- mkReg(False);
  Reg#(Bool) turnO <- mkReg(False);
  method Action start(Bit#(32) a, Bit#(32) b);
    if (turnI) gcd1.start(a,b); else gcd2.start(a,b);
    turnI <= !turnI;
  endmethod
  method ActionValue (Bit#(32)) getResult;
    Bit#(32) y;
    if (turnO) y <- gcd1.getResult
    else y <- gcd2.getResult;
    turnO <= !turnO
    return y;
  endmethod
endmodule
```

```
interface GCD;
  method Action start (Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) getResult;
endinterface
```

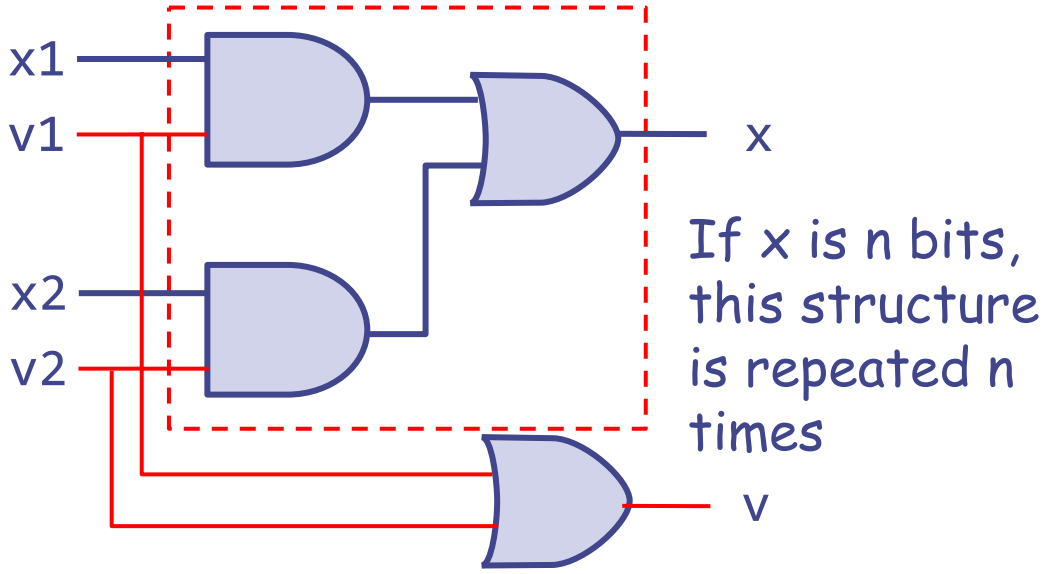
Hardware Synthesis

How is Bluespec code synthesized into hardware?

A new mux-like structure to deal with multiple sources



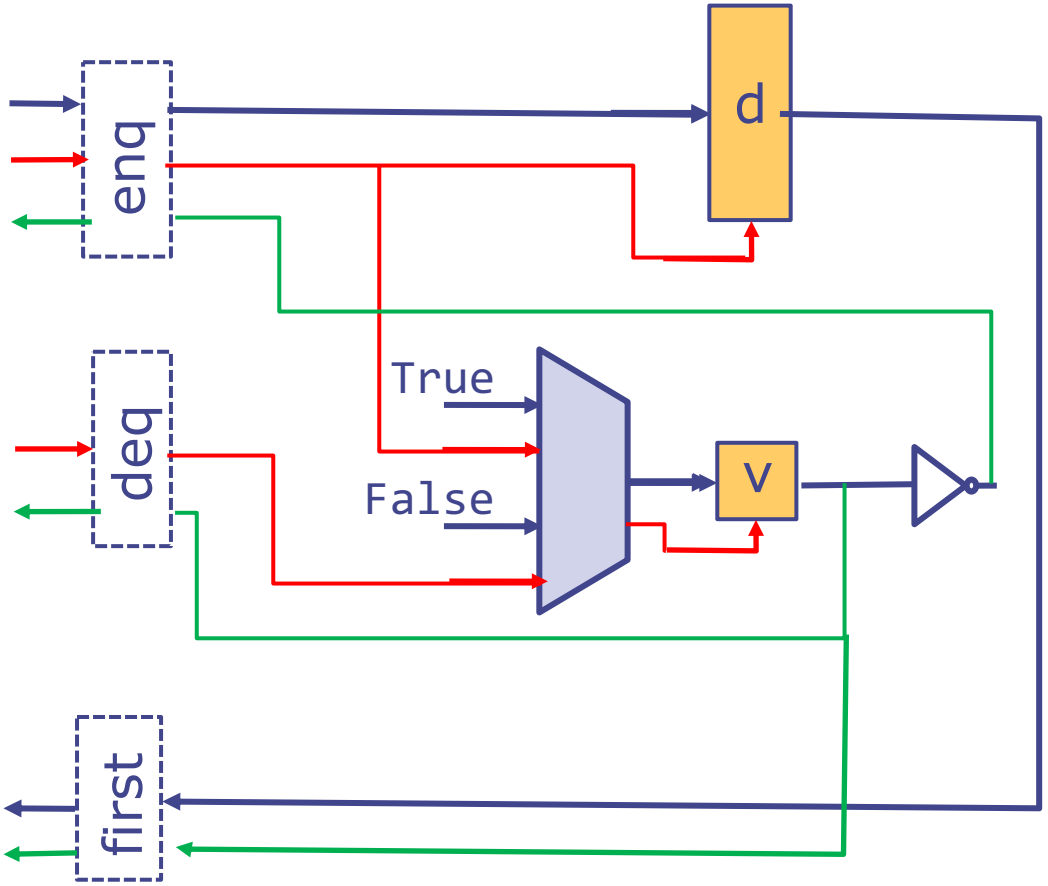
x = (v1 & x1) | (v2 & x2)
v = v1 | v2



- 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

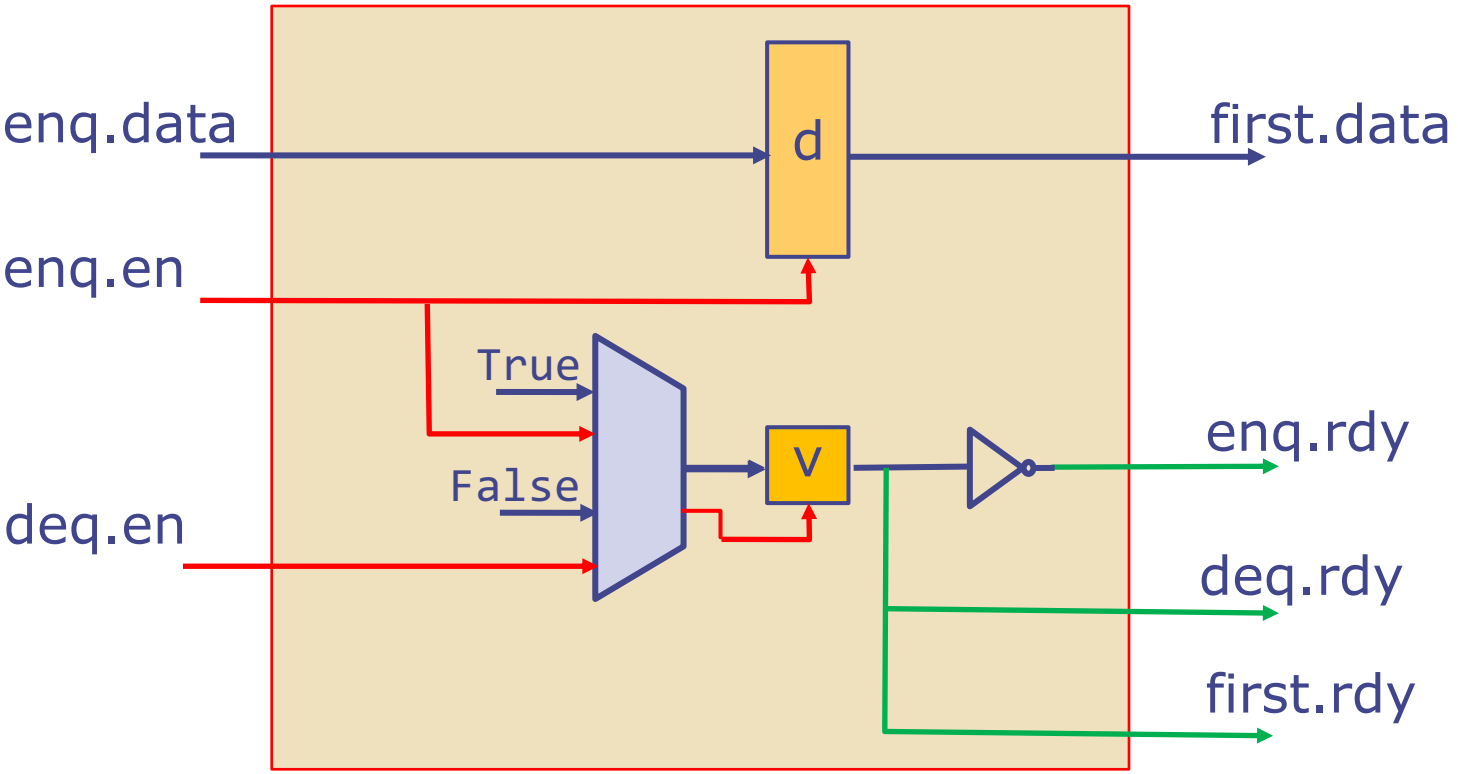
```
module mkFifo (Fifo#(1, t));
  Reg#(t)    d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(t x) if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method t first if (v);
    return d;
  endmethod
endmodule
```



- I/O: Interface
- Instantiate state
- For every register with multiple assignments, insert a mux
- Compile enq
- Compile deq
- Compile first

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

Redrawing the FIFO Circuit



A module is a sequential circuit with input and output wires corresponding to its interface methods

This is a sequential circuit because it has state elements; however, it has no cycles

Next state transition
Partial Truth Table

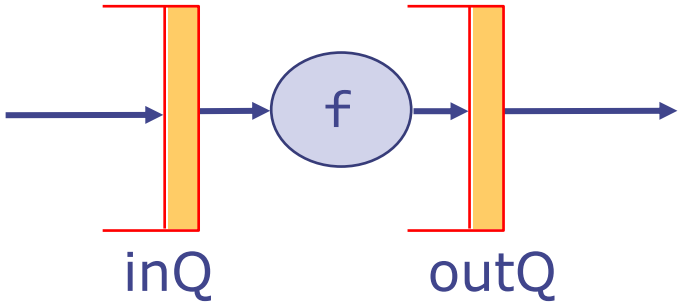
An aside

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
1				1			0			
		1		0				0		
1		1		0			1			
1		1		1				1		

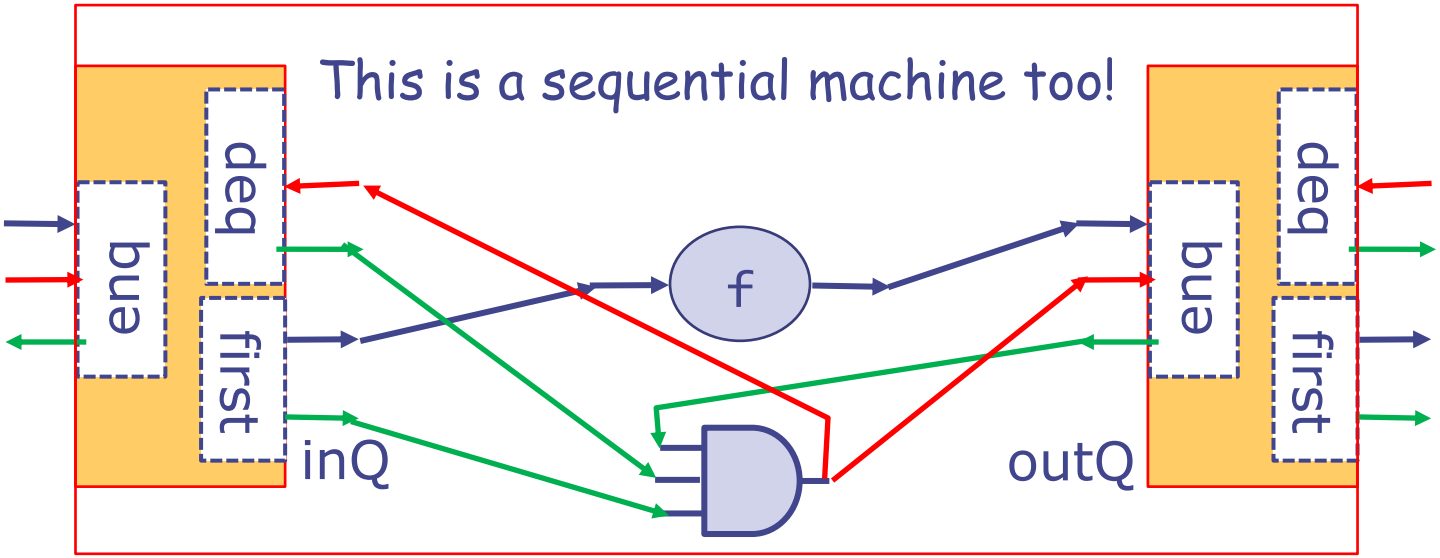
Illegal inputs

Tedious!

Streaming a function: Circuit



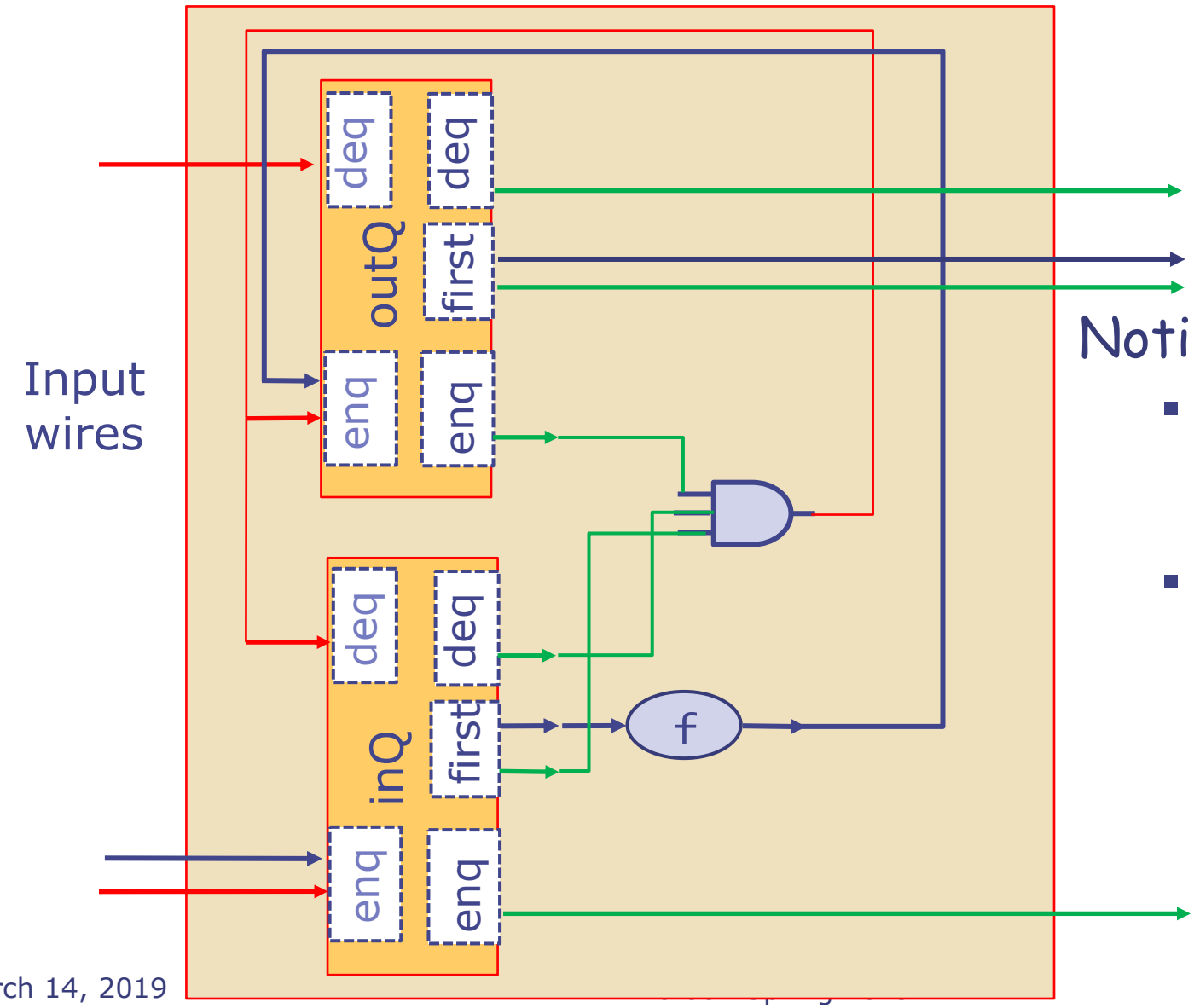

```
rule stream;
  outQ.enq(f(inQ.first));
  inQ.deq;
endrule
```



- Connect the datapath
- Collect the ready signals
- Enable the called methods

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

Redrawing the sequential circuit

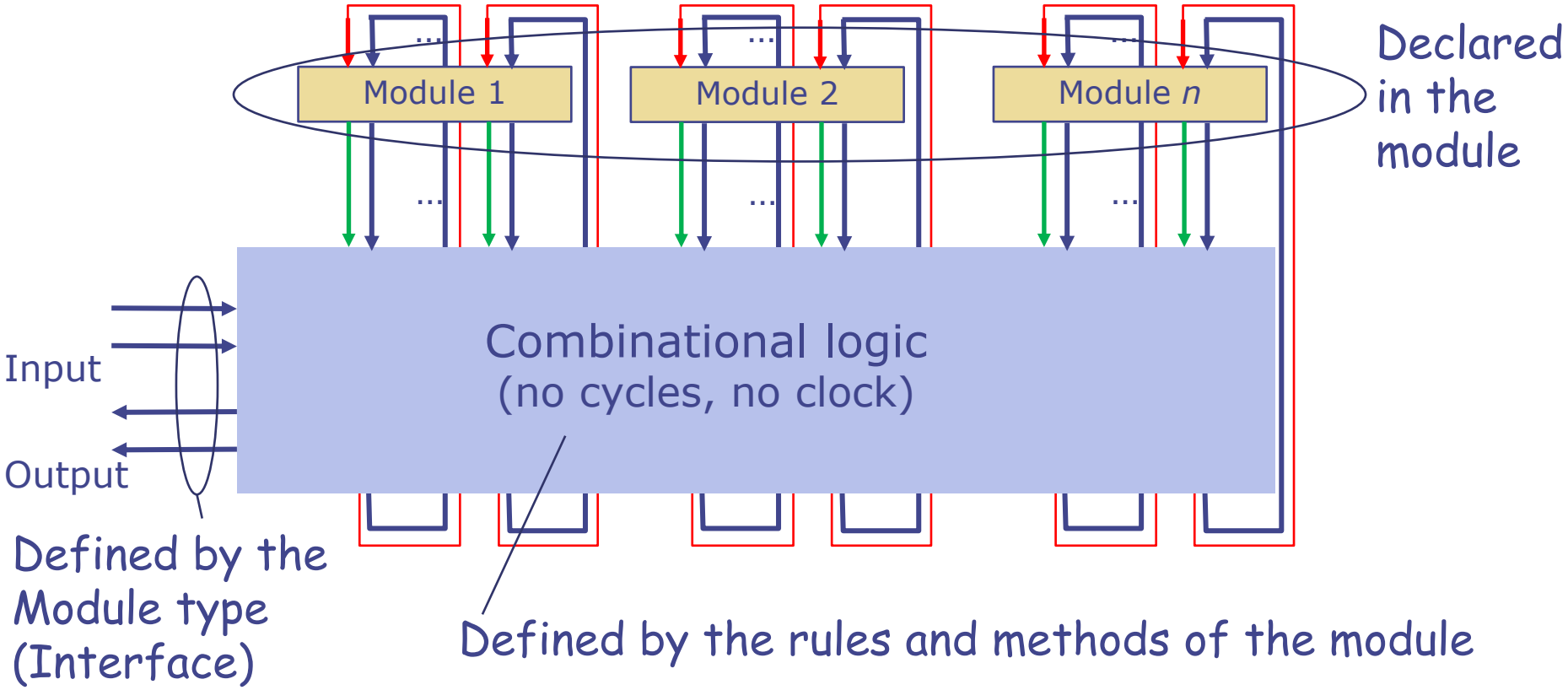


Notice:

- All input wires are red and blue
- All output wires are green and blue

Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine



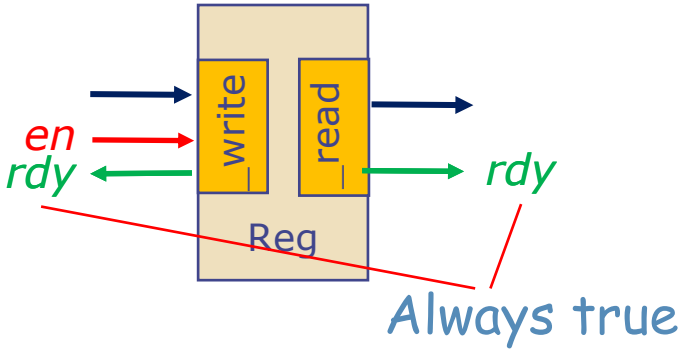
Hardware Synthesis
High-level idea

- Every module represents a sequential machine
- Register is a primitive module – its implementation is outside the language
  - Register bit width is derived from its type
- Each Register and module is instantiated explicitly
- Input/Output wires of a module are derived from its interface, i.e., type
- Rules and methods define the combinational logic to connect registers and modules

The resulting hardware is a collection of sequential machines, which itself behaves like a sequential machine

Register: The Primitive module

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

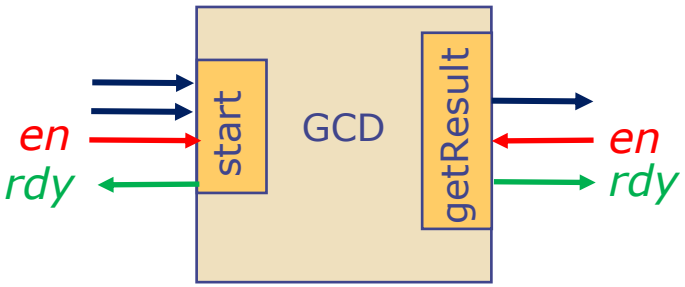


- Implementation is defined outside the language
- A register is created using mkReg or mkRegU
- The guards of _write and _read are always true and not generated
- Special syntax
  - `x <= e` instead of `x._write(e)`
  - `x` instead of `x._read` in expressions

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 an output data wire
  - An *Action method* has no output data wire

```
interface GCD;
  method Action start
      (Bit#(32) a,Bit#(32) b);
  method ActionValue(Bit#(32))
                 getResult;
endinterface
```

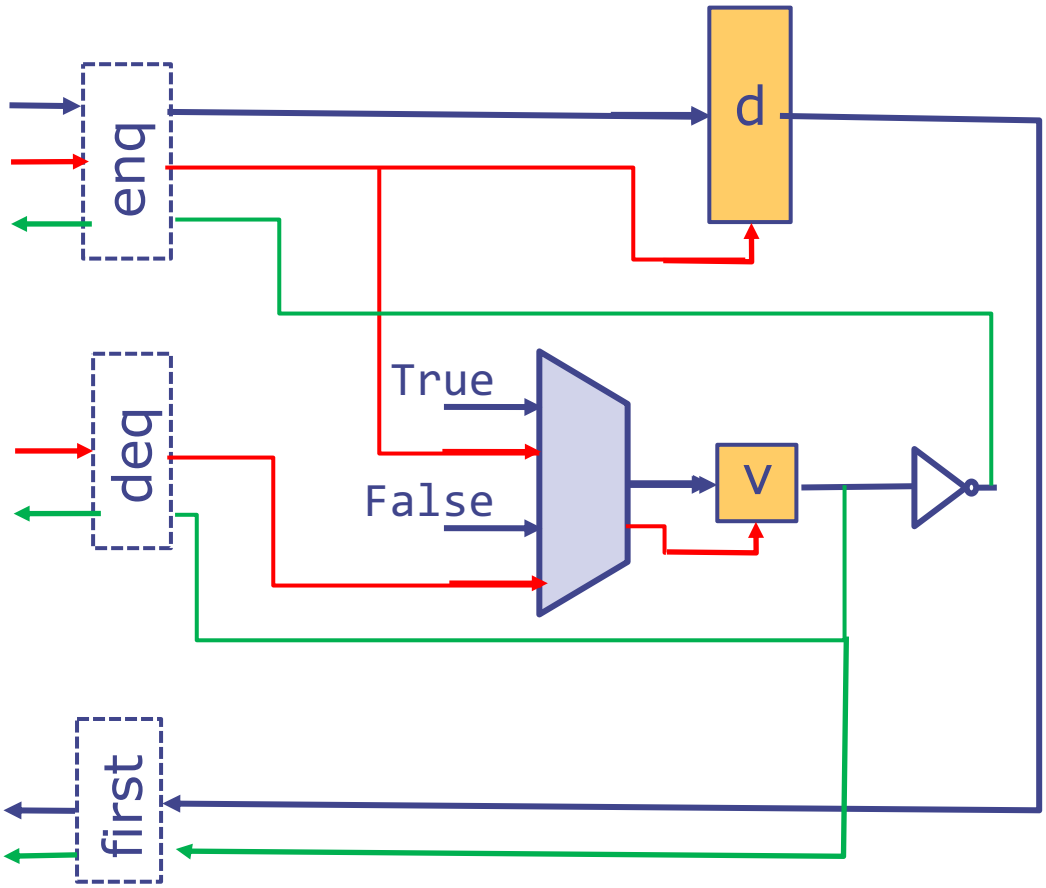


Compiling Guards

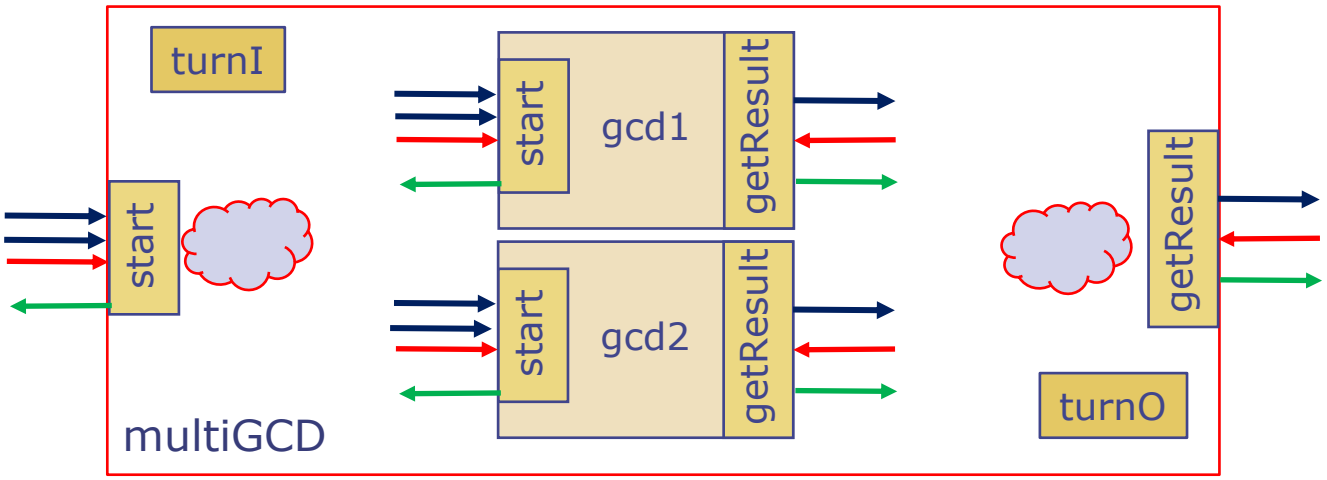
- Bluespec design specifies the guard for each method and rule
  - The guard may be stated explicitly; If there is no guard then it is assumed to be true
  - But the guard also implicitly inherits the guards of the called methods
  - The compiler generates the ready signal from these explicit and implicit guards

Guards define the ready signals

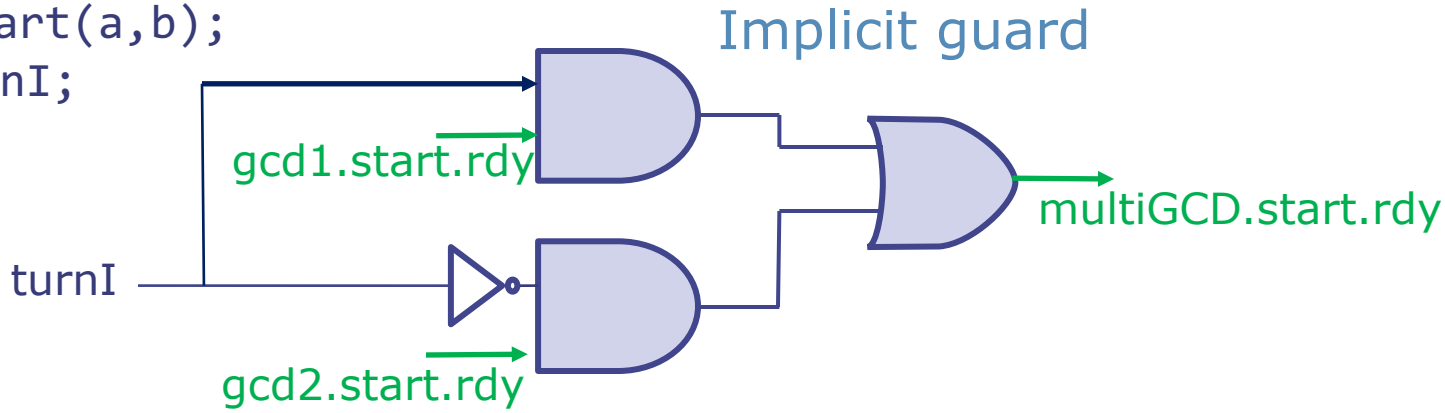
```
module mkFifo (Fifo#(1, t));
  Reg#(t)    d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(t x) if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method t first if (v);
    return d;
  endmethod
endmodule
```



Combined Guards

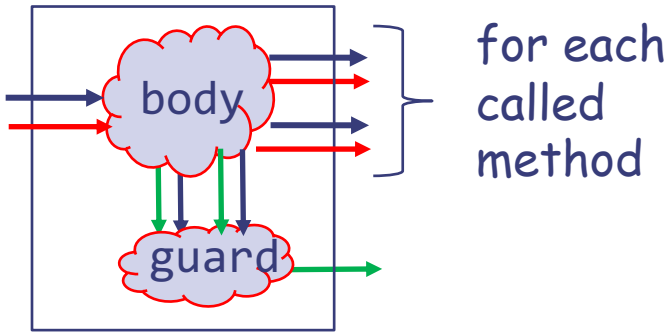



```
method Action start(Bit#(32) a, Bit#(32) b);
   if (turnI) gcd1.start(a,b);
    else gcd2.start(a,b);
   turnI <= !turnI;
endmethod
```



Generating enable signals and associated data

- Bluespec design *implicitly* specifies the enable signal for each of the called Action and ActionValue method
- It does so by propagating the incoming enable signals of the calling method or rule



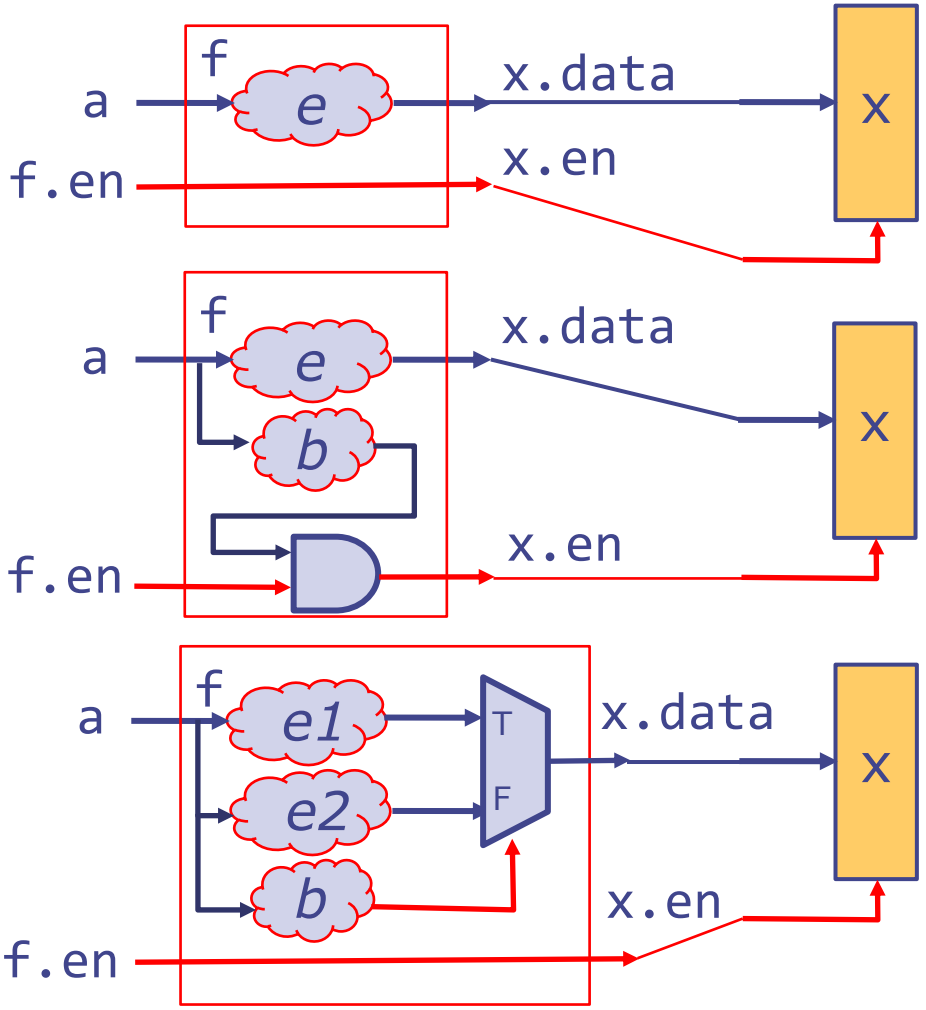
Rules and methods define the combinational logic and enables

```
module mkEx1 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
     x <= e;
  endmethod endmodule
```

```
module mkEx2 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
     if (b) x <= e;
  endmethod endmodule
```

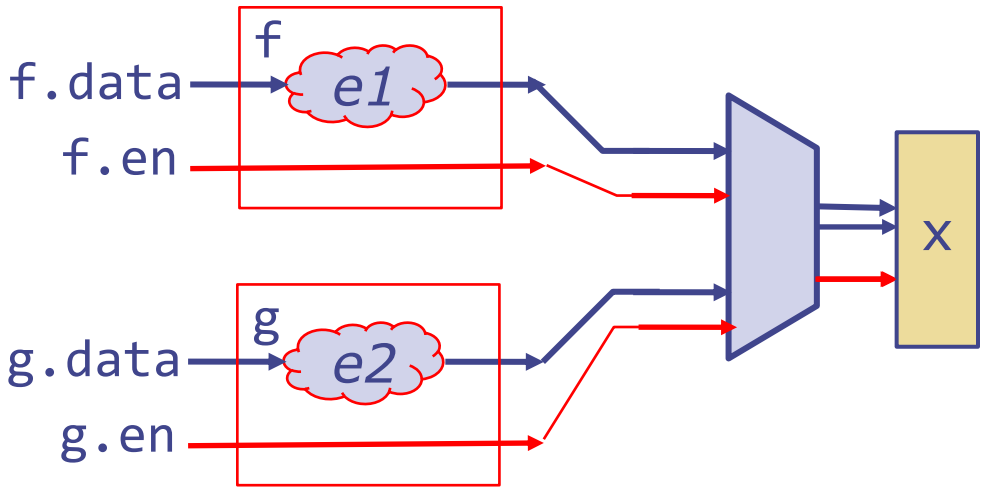
```
module mkEx3 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
     if (b) x <= e1;
     else x <= e2;
  endmethod
endmodule
```

Not shown:
f.rdy = True



Synthesis of multiple sources for register assignments

```
module mkEx4 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
     x <= e1;
  endmethod
  method Action g(t a);
     x <= e2;
  endmethod
endmodule
```



- Compiler has to ensure that both f and g can't be enabled together
- This information is provided by the compiler in the form of a *conflict matrix* (CM) for each module

Conflict Matrix for an Interface

- Conflict Matrix (CM) defines which methods of a module can be called concurrently
- CM for a register:

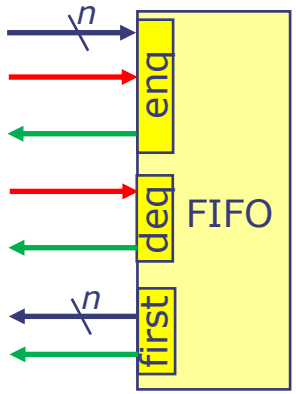
	reg.r	reg.w
reg.r	CF	<
reg.w	>	C

  - Two reads can be performed concurrently (CF – Conflict Free)
  - Two concurrent writes conflict (C) and are not permitted
  - A read and a write can be performed concurrently and it behaves as if the read happened before the write (r<w)
- CM of a register is used systematically to derive the CM for the interface of a module

CM can also be defined for any set of rules

One-Element FIFO

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



	enq	deq	first
enq	C	ME	ME
deq	ME	C	>
first	ME	<	CF

ME = mutually exclusive

a<b = a precedes b

We don't care about conflicts between mutually exclusive rules or methods because they can never be ready at the same time

Take-home problem: Draw the circuit for this GCD

```
interface GCD;
  method Action start
      (Bit#(32) a,Bit#(32) b);
  method ActionValue(Bit#(32))
                  getResult;
endinterface
```

```
module mkGCD (GCD);
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  Reg#(Bool) busy <- mkReg(False);
  rule gcd;
     if (x >= y) begin x <= x - y; end //subtract
     else if (x != 0) begin x <= y; y <= x; end //swap
  endrule
  method Action start(Bit#(32) a, Bit#(32) b) if (!busy);
    x <= a; y <= b; busy <= True;
  endmethod
  method ActionValue (Bit#(32)) getResult if (busy && (x==0));
    busy <= False; return y;
  endmethod
endmodule
```

Assume b != 0