Constructive Computer Architecture

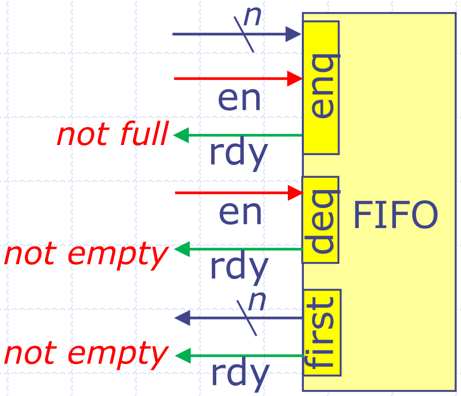
Modules with Guarded Interfaces

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Guarded interfaces

- Make the life of the programmers easier: Include some checks (readyness, fullness, ...) in the method definition itself, so that the user does not have to test the applicability of the method from outside

- Guarded Interface:
  - Every method has a *guard* (*rdy* wire)
  - 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

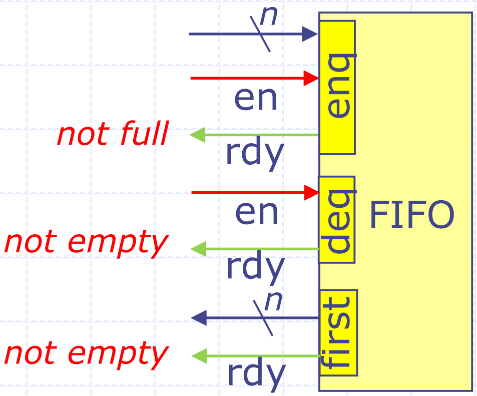


```
interface Fifo#(numeric type size, type t);
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface
```

notice, en and rdy wires are implicit

One-Element FIFO Implementation with guards

```
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
```



Notice, no semicolon turns the if into a guard

Rules with guards

- Like a method, a rule can also have a guard

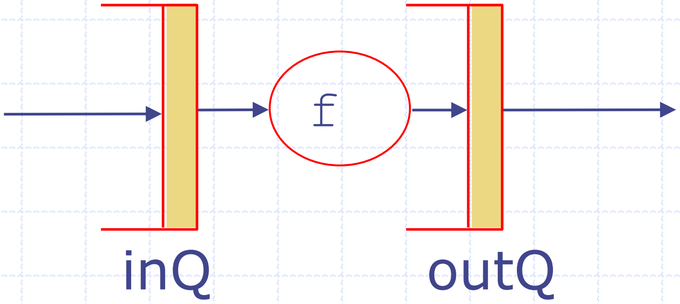
```
rule foo (p);
  begin x1 <= e1; x2 <= e2 end
endrule
```

guard

No if before the guard for rules!

- A rule can execute only if it’s guard is true, i.e., if the guard is false the rule has no effect
- True guards can be omitted

Streaming a function using a FIFO with guarded interfaces

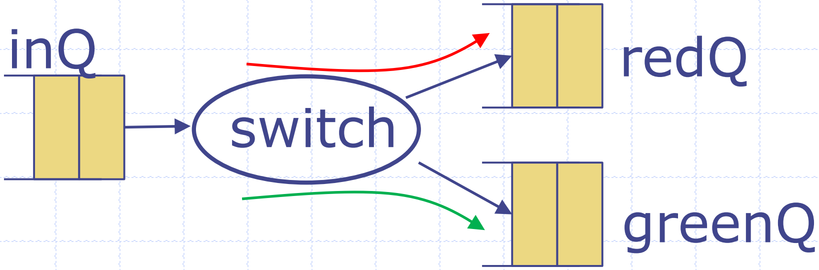


```
rule stream;
  if(inQ.notEmpty && outQ.notFull)
    begin outQ.enq(f(inQ.first)); inQ.deq; end
endrule
```

```
rule stream ~~(inQ.notEmpty && outQ.notFull)~~;
    outQ.enq(f(inQ.first)); inQ.deq;
endrule
```

The implicit guards of the method call are sufficient here

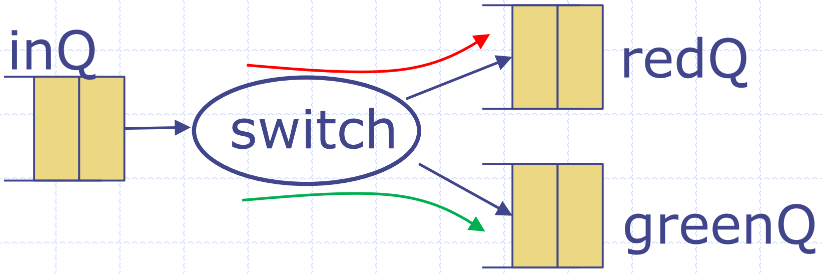
Switch using FIFOs with guarded interfaces



```
rule switch;
 if (inQ.notEmpty)
  if (inQ.first.color == Red) begin1
   if (redQ.notFull) begin2
    redQ.enq(inQ.first.value); inQ.deq;
    end2
   end1
  else begin3
   if (greenQ.notFull) begin4
    greenQ.enq(inQ.first.value); inQ.deq;
    end4
  end3
 endrule
```

All the red stuff can be deleted

Switch using FIFOs with guarded interfaces

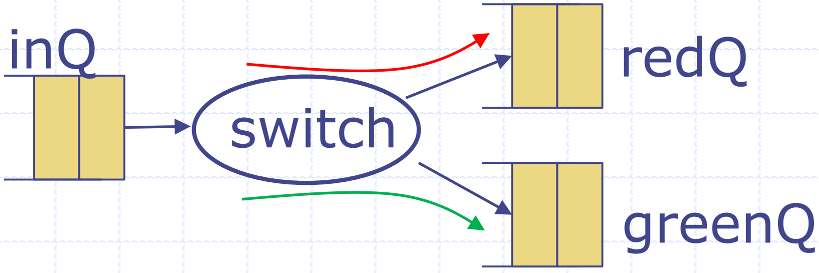


```
rule switch;
  if (inQ.first.color == Red) begin
    redQ.enq  (inQ.first.value); inQ.deq;
  end else begin
    greenQ.enq(inQ.first.value); inQ.deq;
  end
endrule
```

What is the implicit guard?

inQ.notEmpty ? (inQ.first.color == Red ? redQ.notFull
                                                         : greenQ.notFull)
                 : False

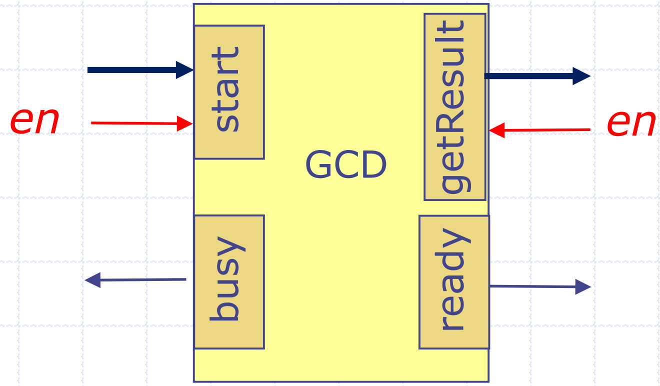
Switch using FIFOs with guarded interfaces

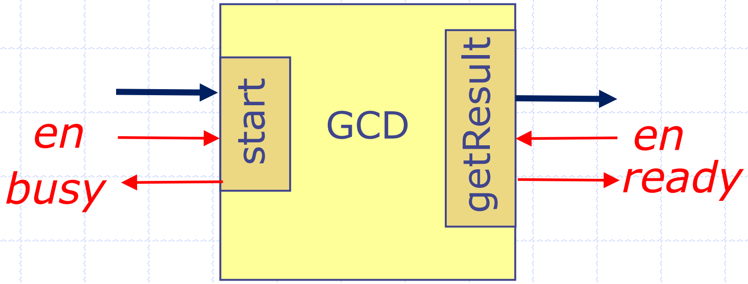


```
rule switch;
  if (inQ.first.color == Red) begin
    redQ.enq  (inQ.first.value); ~~inQ.deq;~~
  end else begin
    greenQ.enq(inQ.first.value); ~~inQ.deq;~~
  end
  inQ.deq;
endrule
```

Does this code still work?

GCD with and without guards



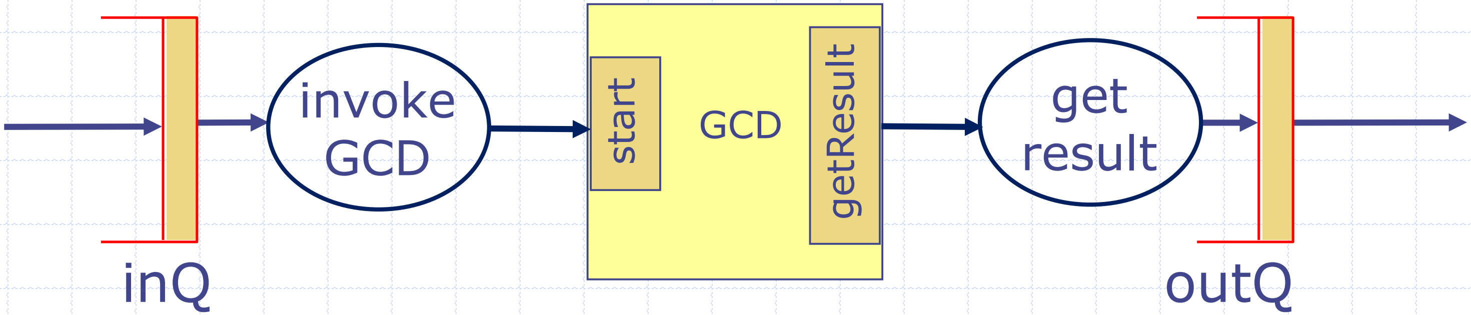


Interface without guards

Interface with guards

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

Using GCD module with guarded interfaces



```
rule invokeGCD;
  gcd.start(inQ.first); inQ.deq;
endrule;
```

```
rule getResult;
   let x <- gcd.getResult; outQ.enq(x);
endrule;
```

A rule can be executed only if guards of all of its actions are true

GCD with guarded interfaces

implementation

```
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 (x==0);
  busy <= False; return y;
endmethod
endmodule
```

Assume b /= 0

Guards vs Ifs

```
method Action enq(t x) if (!v);
   v <= True; d <= x;
endmethod
```

guard is !v; enq can be applied only if v is false

versus

```
method Action enq(t x);
   if (!v) begin v <= True; d <= x; end
endmethod
```

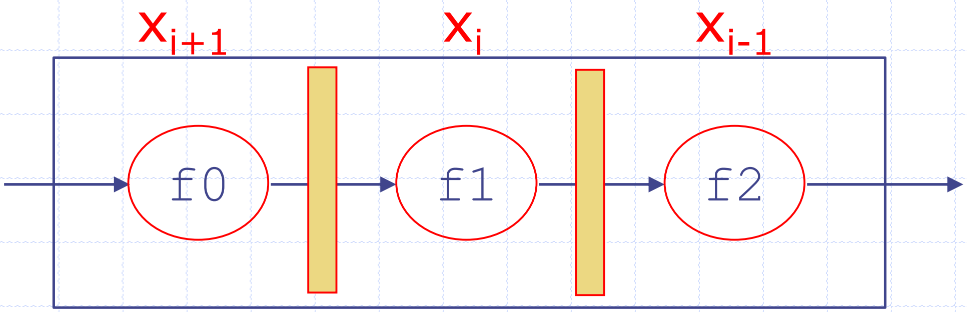
guard is True, i.e., the method is always applicable.

if v is true then x would get lost;

bad

Pipelining combinational circuits

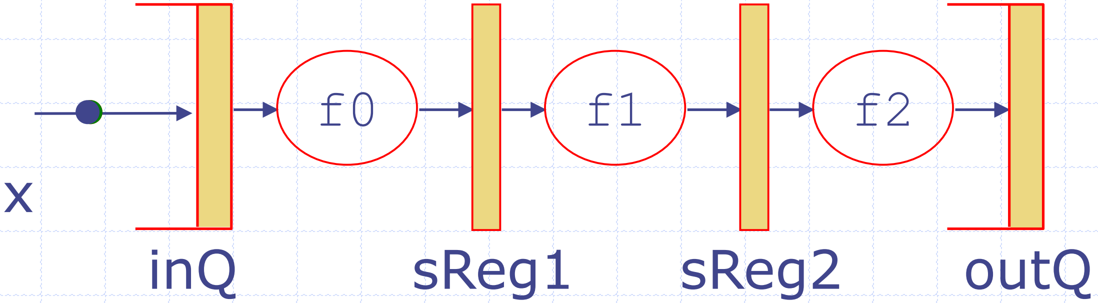
Pipelining Combinational Functions



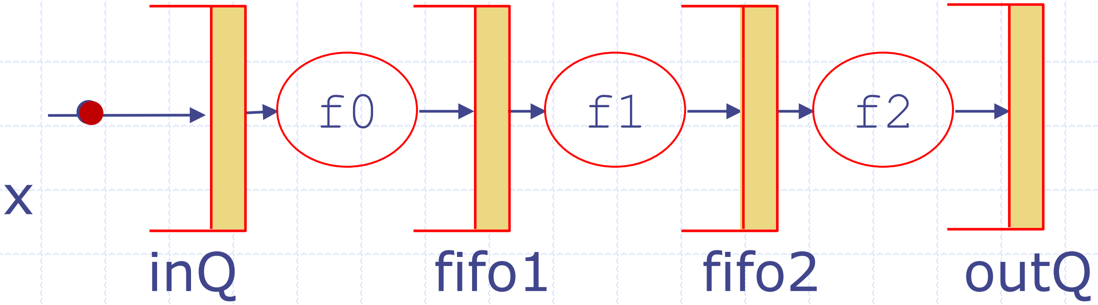
3 different datasets in the pipeline

- Lot of area and long combinational delay
- Folded or multi-cycle version can save area and reduce the combinational delay but throughput per clock cycle gets worse
- Pipelining: a method to increase the circuit throughput by evaluating multiple inputs

Inelastic vs Elastic pipeline



Inelastic: all pipeline stages move synchronously

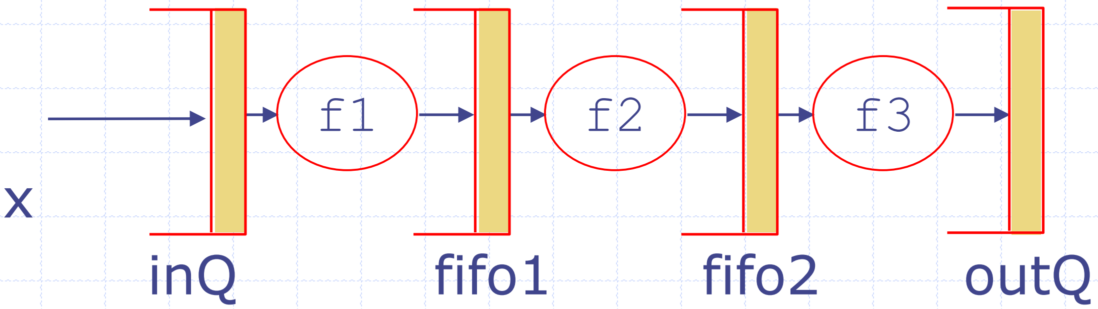


Elastic: A pipeline stage can process data if its input FIFO is not empty and output FIFO is not Full

Most complex processor pipelines are a combination of the two styles

Elastic pipeline

Use FIFOs instead of pipeline registers



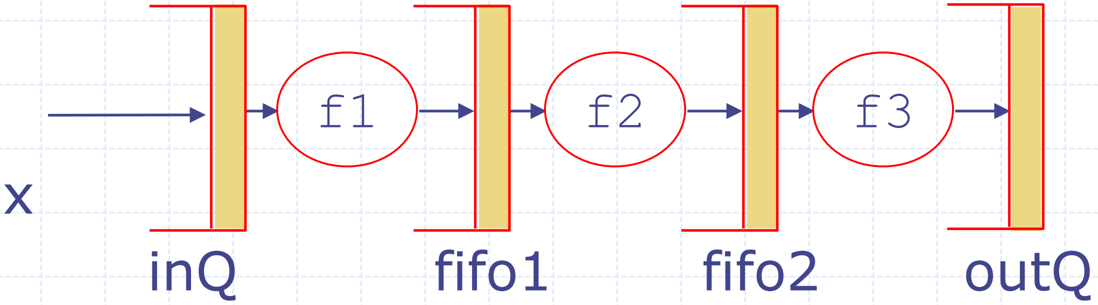
no need for valid bits

```
rule stage1;
   fifo1.enq(f1(inQ.first));
   inQ.deq();          endrule
rule stage2;
   fifo2.enq(f2(fifo1.first));
   fifo1.deq;          endrule
rule stage3;
   outQ.enq(f3(fifo2.first));
   fifo2.deq;          endrule
```

- When can stage1 rule fire?
  - inQ has an element
  - fifo1 has space
- Can tokens be left in the pipeline?

  No
- Can these rules execute concurrently?

Elastic pipeline



```
rule stage1;
  fifo1.enq(f1(inQ.first));
  inQ.deq();          endrule
rule stage2;
  fifo2.enq(f2(fifo1.first));
  fifo1.deq;          endrule
rule stage3;
  outQ.enq(f3(fifo2.first));
  fifo2.deq;          endrule
```

- If these rules cannot execute concurrently, it is hardly a pipelined system
- When can rules execute concurrently?
- What hardware is synthesized to execute rules concurrently?

Multi-rule Systems

Repeatedly:

- Select a rule to execute
- Compute the state updates
- Make the state updates

Non-deterministic choice; User annotations can be used in rule selection

One-rule-at-a-time-semantics: Any legal behavior of a Bluespec program can be explained by observing the state updates obtained by applying only one rule at a time

However, for performance we execute multiple rules concurrently whenever possible

stay tuned …