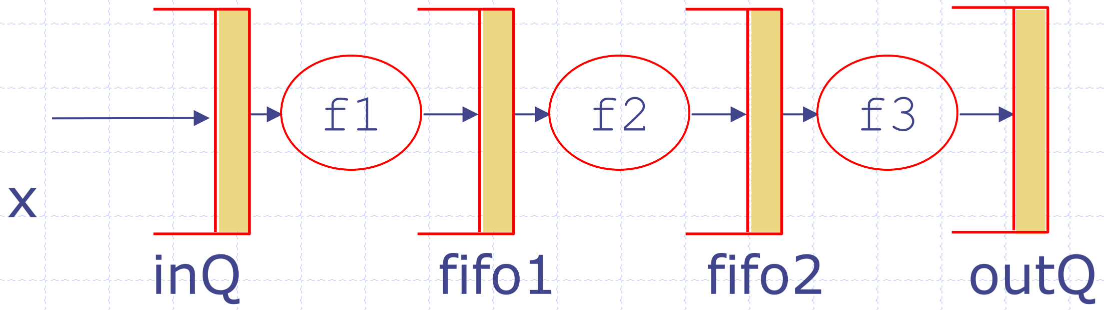
Constructive Computer Architecture:

Guards

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

Elastic pipeline




```
rule stage1;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f1(inQ.first)); inQ.deq; end endrule
rule stage2;
  if(fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f2(fifo1.first)); fifo1.deq; end endrule
rule stage3;
  if(fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f3(fifo2.first)); fifo2.deq; end endrule
```

Proper use of FIFOs always involves checking for emptiness or fullness conditions

Easy mistakes

```
rule stage1;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f1(inQ.first);
          inQ.deq; end
endrule
```

versus

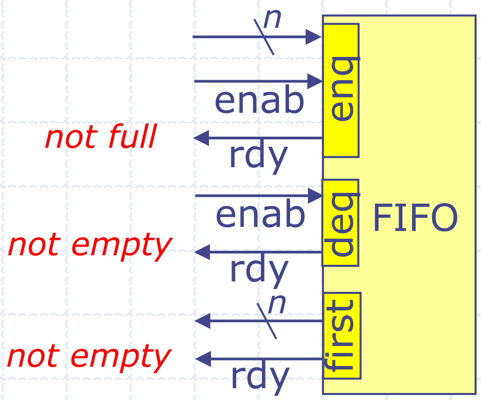
What is the difference?

```
rule stage1E;
  if(inQ.notEmpty && fifo1.notFull)
      fifo1.enq(f1(inQ.first);
  if(inQ.notEmpty) inQ.deq;
endrule
```

Guards is an abstraction to deal with such "atomicity" issues

stage1E may dequeue something even though the value read has not been processed (ie enqueued into fifo1)

FIFO Module: methods with guarded interfaces

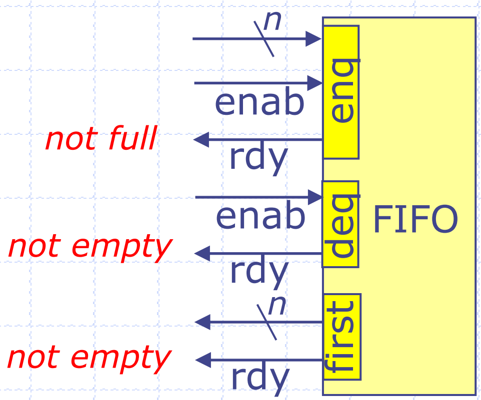



```
interface Fifo#(numeric type size,
                type t);
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface
```

- Every method has a guard (rdy wire)
- The compiler ensures that an action method is invoked (en) only if the guard is true. Similarly the value returned by a value method is meaningful only if its guard is true
- Guards make it possible to transfer the responsibility of the correct use of a method from the user to the compiler
- Guards are extraordinarily convenient for programming and also enhance modularity of the code

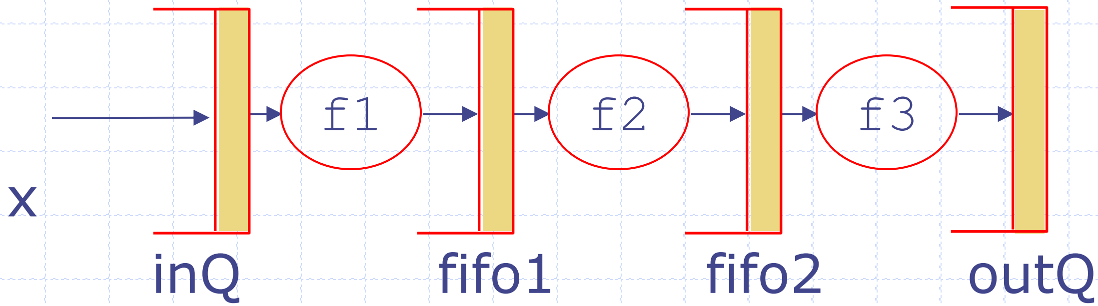
One-Element FIFO Implementation with guards

```
module mkCFFifo (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
```



Elastic pipeline

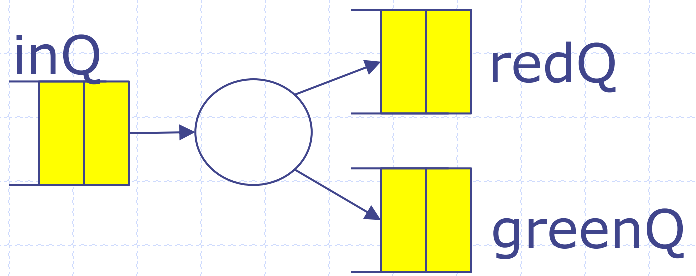
with guards




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

- When can stage1 rule fire?
  - inQ has an element
  - fifo1 has space
- The explicit guard is true but the compiler lifts all the implicit guards of the methods to the top of the rule

Switch with guards




```
rule switch (True);
  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
```

```
rule switchRed (inQ.first.color == Red);
        redQ.enq(inQ.first.value); inQ.deq;
endrule;
rule switchGreen (inQ.first.color == Green);
        greenQ.enq(inQ.first.value); inQ.deq;
endrule;
```

GCD module
with Guards

```
Reg#(Bit#(32)) x <- mkReg(0);
Reg#(Bit#(32)) y <- mkReg(0);
rule gcd (x != 0);
  if (x > y) begin
    x <= x - y; end
  else begin
    x <= y; y <= x; end
endrule
method Action start(Bit#(32) a, Bit#(32) b) if (x == 0);
  x <= a; y <= b; endmethod
method Bit#(32) result if (x == 0);
  return y; endmethod
```

If x is 0 then the rule cannot fire

- Start method can be invoked only if x is 0
- The result is available only when x is 0 is True.

All methods have implicit guards

- Every method call has an implicit guard associated with it
  - m.enq(x), the guard indicated whether one can enqueue into fifo m or not
- Methods of primitive modules like registers and EHRs have their guards always set to True
- Guards play an important role in scheduling; a rule is considered for scheduling only if its guard is true ("can fire")
- Nevertheless guards are merely syntactic sugar and are lifted to the top of each rule by the compiler

Guard Elimination

Making guards explicit in compilation

- Make the guards explicit in every method call by naming the guard and separating it from the unguarded body of the method call, i.e., syntactically replace m.g(e) by

$$m.g_B(e) \text{ when } m.g_G$$

  - Notice $m.g_G$ has no parameter because the guard value should not depend upon the input

Lifting implicit guards

```
rule foo if (True);
   (if (p) fifo.enq(8)); x.w(7)
```

All implicit guards are made explicit, and lifted and conjoined to the rule guard

rule foo if (p & $fifo.enq_G$ | !p);
   if (p) $fifo.enq_B$(8); x.w(7)

rule foo if ($fifo.enq_G$ | !p);
   if (p) $fifo.enq_B$(8); x.w(7)

Make implicit guards explicit

<a> ::= <a> | <a>
| if (<e>) <a>
~~| m.g(<e>)~~ m.g_B(<e>) when m.g_G
| let t = <e> in <a>

<a> ::= <a> | <a>
| if (<e>) <a>
| m.g(<e>) methods without guards
| let t = <e> in <a>
| <a> when <e> guarded action

The new kernel language

Concrete syntax for guards

- rule x(g);
  a
  endrule
  is the same as rule x (a when g)
- method foo(x, y) if (g);
  a
  endmethod
  is the same as
  method foo(x, y) (a when g) endmethod
- If no guard is explicitly supplied, the guard is assumed to be True

Guards vs If's

- A guard on one action of a parallel group of actions affects every action within the group

  (a1 when p1) | a2

  ==> (a1 | a2) when p1

- A condition of a Conditional action only affects the actions within the scope of the conditional action

  (if (p1) a1) | a2

  p1 has no effect on a2 ...

- Mixing ifs and whens

  (if (p) (a1 when q)) | a2

  ≡ ((if (p) a1) | a2) when ((p && q) | !p)

  ≡ ((if (p) a1) | a2) when (q | !p)

Guard Lifting Axioms
without Let-blocks

- All the guards can be "lifted" to the top of a rule
  - (a1 when p) | a2 ⇒ (a1 | a2) when p
  - a1 | (a2 when p) ⇒ (a1 | a2) when p
  - if (p when q) a ⇒ (if (p) a) when q
  - if (p) (a when q) ⇒ (if (p) a) when (q | !p)
  - (a when p1) when p2 ⇒ a when (p1 & p2)
  - $m.g_B$(e when p) ⇒ $m.g_B$(e) when p

similarly for expressions ...

  - Rule r (a when p) ⇒ Rule r (if (p) a)

We will call this guard lifting transformation WIF, for when-to-if

A complete guard lifting procedure also requires rules for let-blocks

Scheduling with guards

- At the top level a guard behaves just like an "if"
- A rule whose guard is False at a given cycle will result in no state change even if it is scheduled
- The guards of all the rules can be evaluated in parallel, often with small amount of computation
- The scheduler takes advantage of this fact by considering only those rules for scheduling in a given cycle whose guards are True

Hierarchical scheduling

- A method of scheduling is outside-in:
- Rules of the outermost modules are scheduled first, then the rules of subsequent inner modules are scheduled, as long as they can be scheduled concurrently with the called methods
- BSV also provides annotation to reverse this priority on a module basis
- It is because of scheduling complications that current BSV doesn’t allow modular compilation in the presence of interface parameters