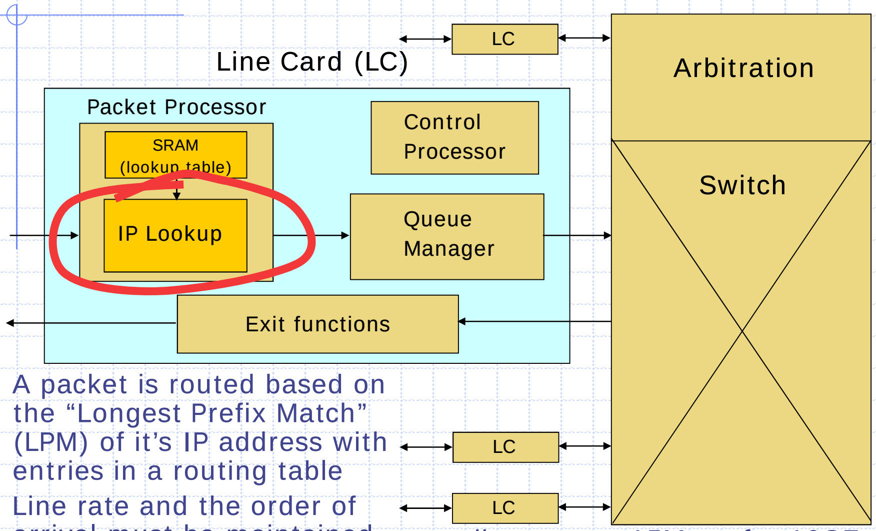
IP Lookup: Some subtle concurrency issues

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

IP Lookup block in a router

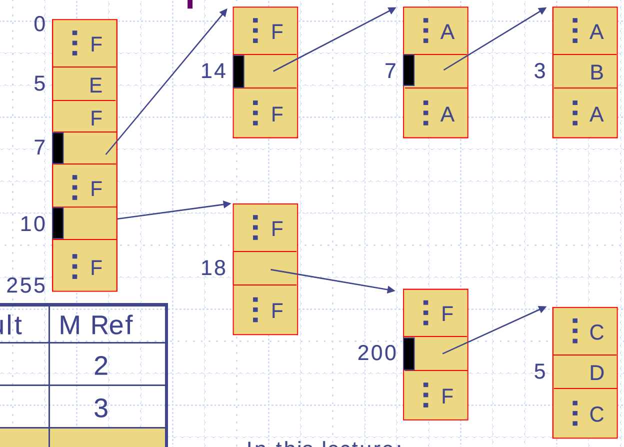


- A packet is routed based on the "Longest Prefix Match" (LPM) of it's IP address with entries in a routing table
- Line rate and the order of arrival must be maintained

line rate ⇒ 15Mpps for 10GE

Sparse tree representation

7.14.*.*	A
7.14.7.3	B
10.18.200.*	C
10.18.200.5	D
5.*.*.*	E
*	F



IP address	Result	M Ref
7.13.7.3	F	2
10.18.201.5	F	3
7.14.7.2		
5.13.7.2	E	1
10.18.200.7	C	4

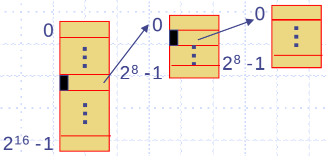
In this lecture:

Level 1: 16 bits
Level 2: 8 bits
Level 3: 8 bits

⇒ 1 to 3 memory accesses

"C" version of LPM

```
int
lpm (IPA ipa)
/*  3 memory lookups */
{  int p;
    /*  Level 1: 16 bits  */
    p = RAM [ipa[31:16]];
    if (isLeaf(p)) return value(p);
    /*  Level 2: 8 bits  */
    p = RAM [ptr(p) + ipa [15:8]];
    if (isLeaf(p)) return value(p);
    /*  Level 3:  8 bits  */
    p = RAM [ptr(p) + ipa [7:0]];
    return value(p);
     /* must be a leaf */
}
```

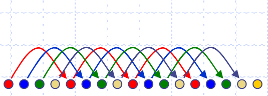


Must process a packet every 1/15 μs or 67 ns

Must sustain 3 memory dependent lookups in 67 ns

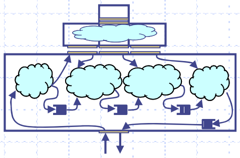
Longest Prefix Match for IP lookup: 3 possible implementation architectures

Rigid pipeline



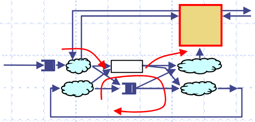
Inefficient memory usage but simple design

Linear pipeline



Efficient memory usage through memory port replicator

Circular pipeline



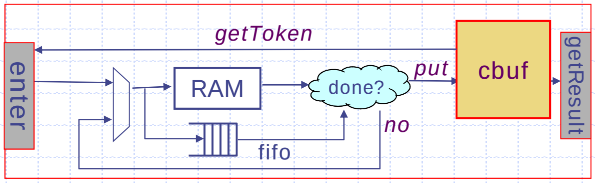
Efficient memory with most complex control

Designer's Ranking:

Which is "best"?

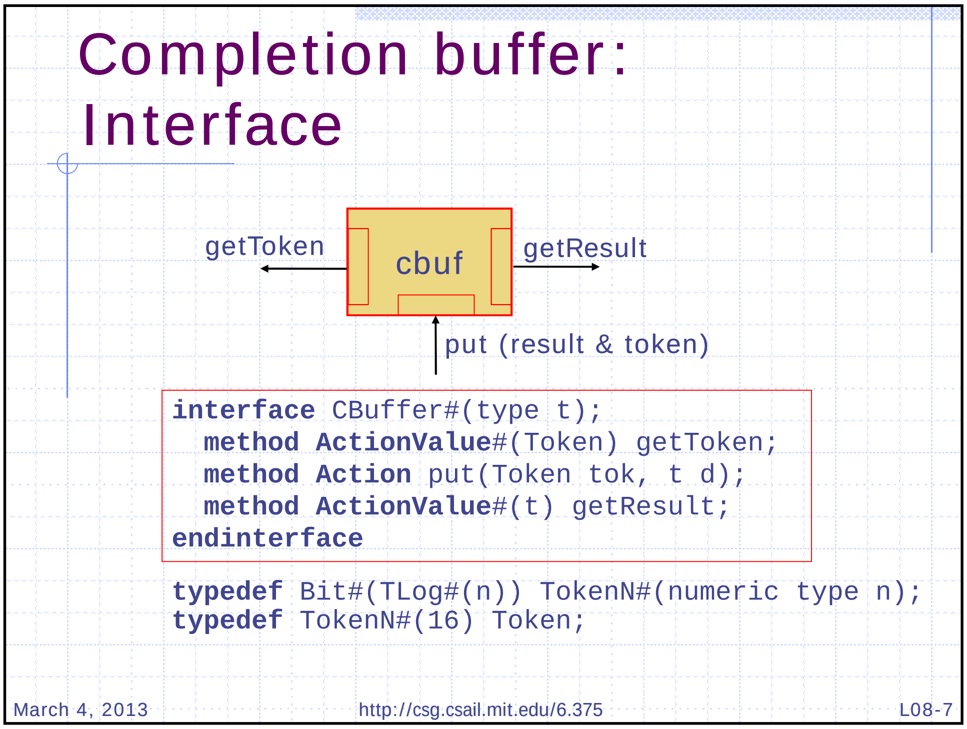
Arvind, Nikhil, Rosenband & Dave [ICCAD 2004]

IP-Lookup module: Circular pipeline



- Completion buffer ensures that departures take place in order even if lookups complete out-of-order
- Since cbuf has finite capacity it gives out tokens to control the entry into the circular pipeline
- The fifo must also hold the "token" while the memory access is in progress: `Tuple2#(Token,Bit#(16))`

remainingIP

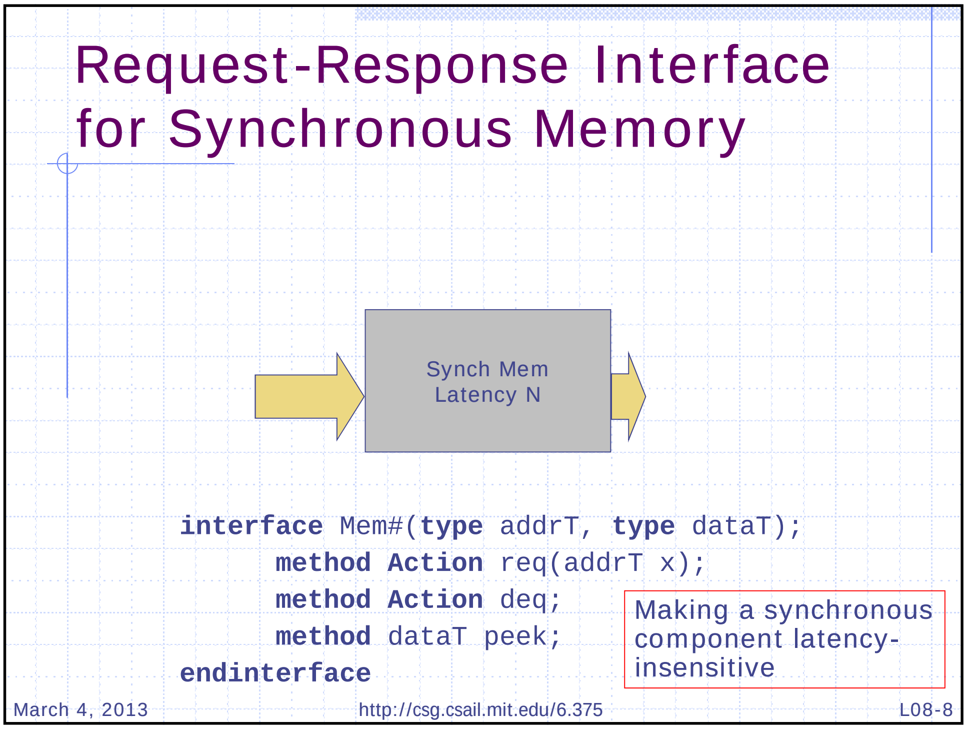


Completion buffer: Interface

getToken
cbuf
getResult
put (result & token)

```
interface CBuffer#(type t);
  method ActionValue#(Token) getToken;
  method Action put(Token tok, t d);
  method ActionValue#(t) getResult;
endinterface
```

```
typedef Bit#(TLog#(n)) TokenN#(numeric type n);
typedef TokenN#(16) Token;
```

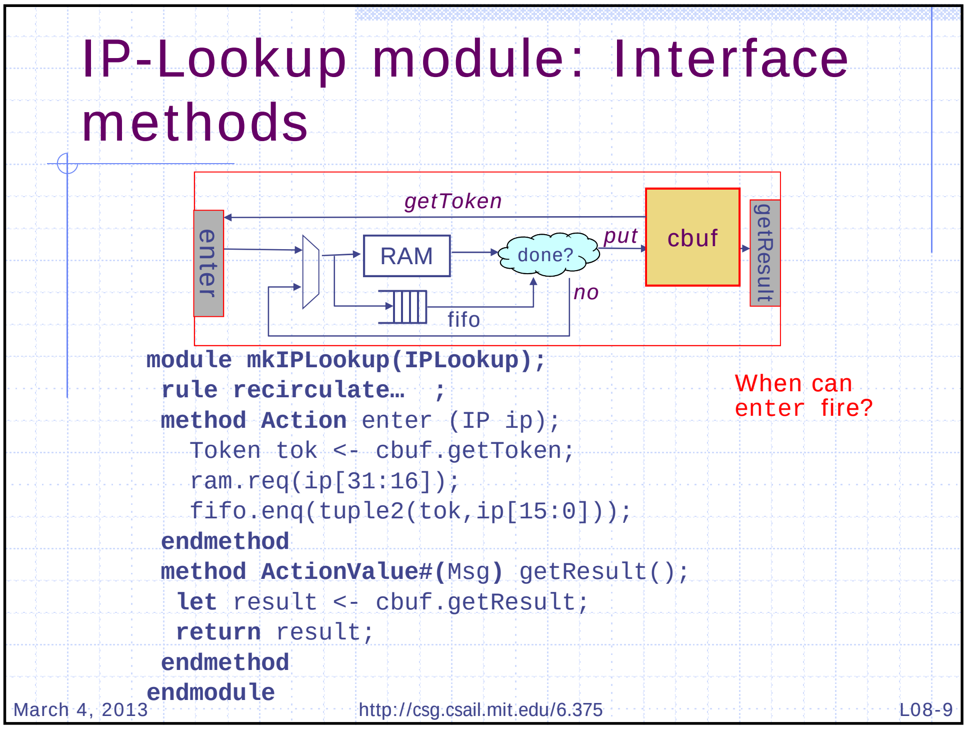


Request-Response Interface for Synchronous Memory

Synch Mem
Latency N

```
interface Mem#(type addrT, type dataT);
      method Action req(addrT x);
      method Action deq;
      method dataT peek;
endinterface
```

Making a synchronous component latency-insensitive

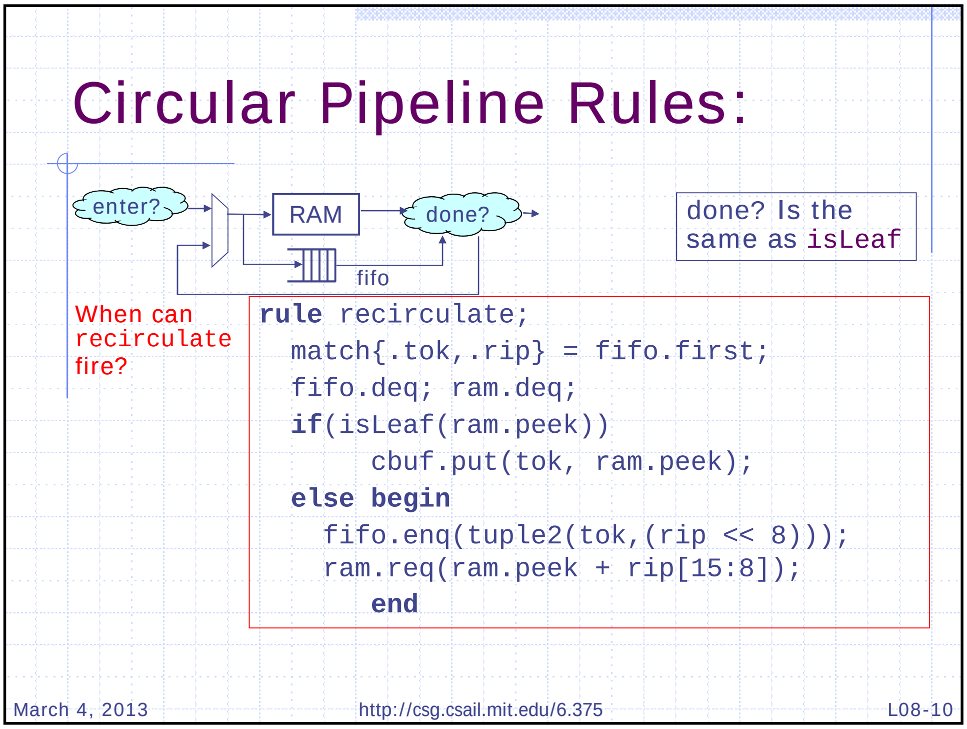


IP-Lookup module: Interface methods

getToken
enter
RAM
done?
put
cbuf
getResult
no
fifo

```
module mkIPLookup(IPLookup);
 rule recirculate…  ;
 method Action enter (IP ip);
   Token tok <- cbuf.getToken;
   ram.req(ip[31:16]);
   fifo.enq(tuple2(tok,ip[15:0]));
 endmethod
 method ActionValue#(Msg) getResult();
  let result <- cbuf.getResult;
  return result;
 endmethod
endmodule
```

When can enter fire?



Circular Pipeline Rules:

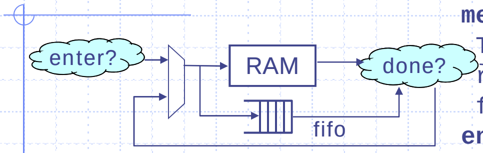
enter?
RAM
done?
fifo

done? Is the same as isLeaf

When can recirculate fire?

```
rule recirculate;
  match{.tok,.rip} = fifo.first;
  fifo.deq; ram.deq;
  if(isLeaf(ram.peek))
       cbuf.put(tok, ram.peek);
  else begin
     fifo.enq(tuple2(tok,(rip << 8)));
     ram.req(ram.peek + rip[15:8]);
       end
```

Dead Cycles



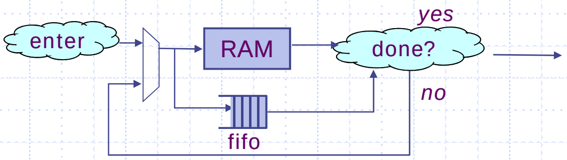
```
method Action enter (IP ip);
 Token tok <- cbuf.getToken;
 ram.req(ip[31:16]);
 fifo.enq(tuple2(tok,ip[15:0]));
endmethod
```

Can a new request enter the system when an old one is leaving?

```
rule recirculate;
  match{.tok,.rip} = fifo.first;
  fifo.deq; ram.deq;
  if(isLeaf(ram.peek))
       cbuf.put(tok, ram.peek);
  else begin
    fifo.enq(tuple2(tok,(rip << 8)));
    ram.req(ram.peek + rip[15:8]);
       end
```

Is this worth worrying about?

The Effect of Dead Cycles

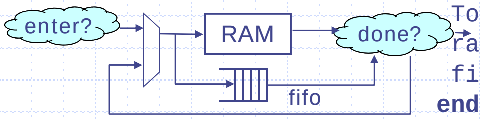


Circular Pipeline

- RAM takes several cycles to respond to a request
- Each IP request generates 1-3 RAM requests
- FIFO entries hold base pointer for next lookup and unprocessed part of the IP address

What is the performance loss if “exit” and “enter” don’t ever happen in the same cycle?

So is there a dead cycle?



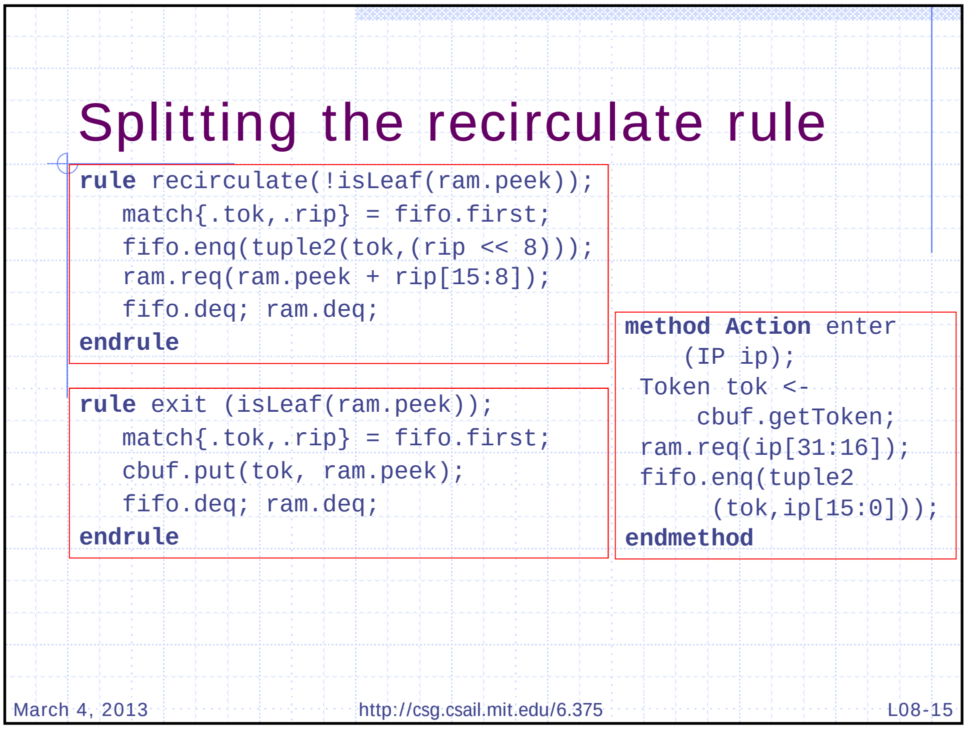
```
method Action enter (IP ip);
  Token tok <- cbuf.getToken;
  ram.req(ip[31:16]);
  fifo.enq(tuple2(tok,ip[15:0]));
endmethod
```

```
rule recirculate;
  match{.tok,.rip} = fifo.first;
  fifo.deq; ram.deq;
  if(isLeaf(ram.peek))
       cbuf.put(tok, ram.peek);
  else begin
     fifo.enq(tuple2(tok,(rip << 8)));
     ram.req(ram.peek + rip[15:8]);
       end
```

Rule Spliting

```
rule foo (True);
   if (p) r1 <= 5;
   else r2 <= 7;
endrule
```

≡

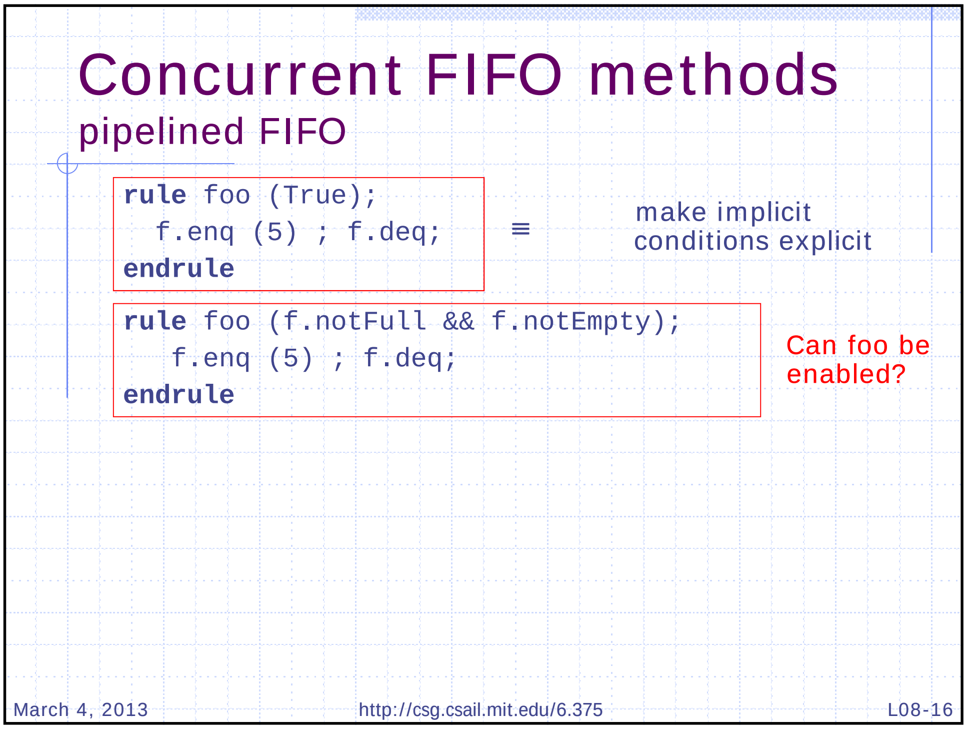


Splitting the recirculate rule

```
rule recirculate(!isLeaf(ram.peek));
   match{.tok,.rip} = fifo.first;
   fifo.enq(tuple2(tok,(rip << 8)));
   ram.req(ram.peek + rip[15:8]);
   fifo.deq; ram.deq;
endrule
```

```
rule exit (isLeaf(ram.peek));
   match{.tok,.rip} = fifo.first;
   cbuf.put(tok, ram.peek);
   fifo.deq; ram.deq;
endrule
```

```
method Action enter
     (IP ip);
 Token tok <-
     cbuf.getToken;
 ram.req(ip[31:16]);
 fifo.enq(tuple2
      (tok,ip[15:0]));
endmethod
```



Concurrent FIFO methods

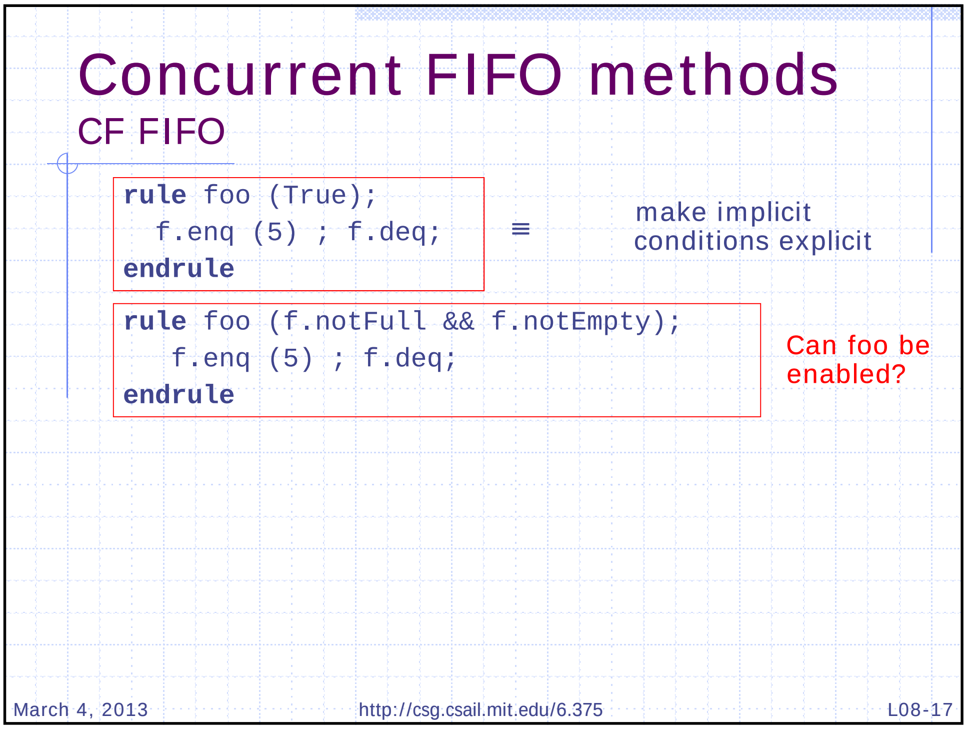
pipelined FIFO

```
rule foo (True);
  f.enq (5) ; f.deq;
endrule
```

≡ make implicit conditions explicit

```
rule foo (f.notFull && f.notEmpty);
   f.enq (5) ; f.deq;
endrule
```

Can foo be enabled?



Concurrent FIFO methods

CF FIFO

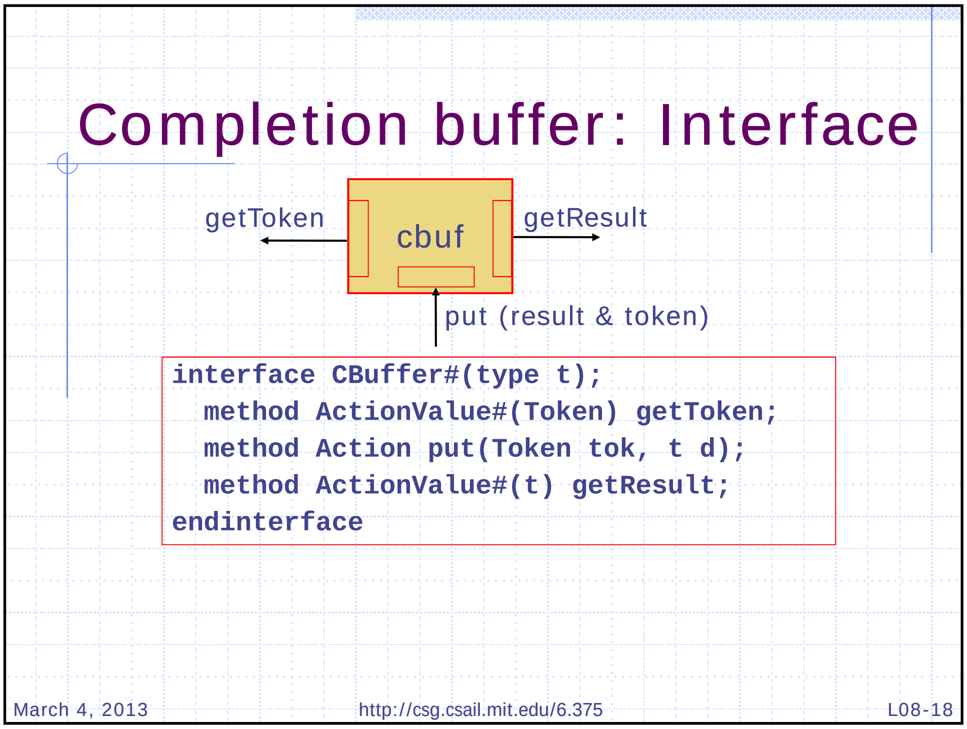
```
rule foo (True);
  f.enq (5) ; f.deq;
endrule
```

≡

make implicit
conditions explicit

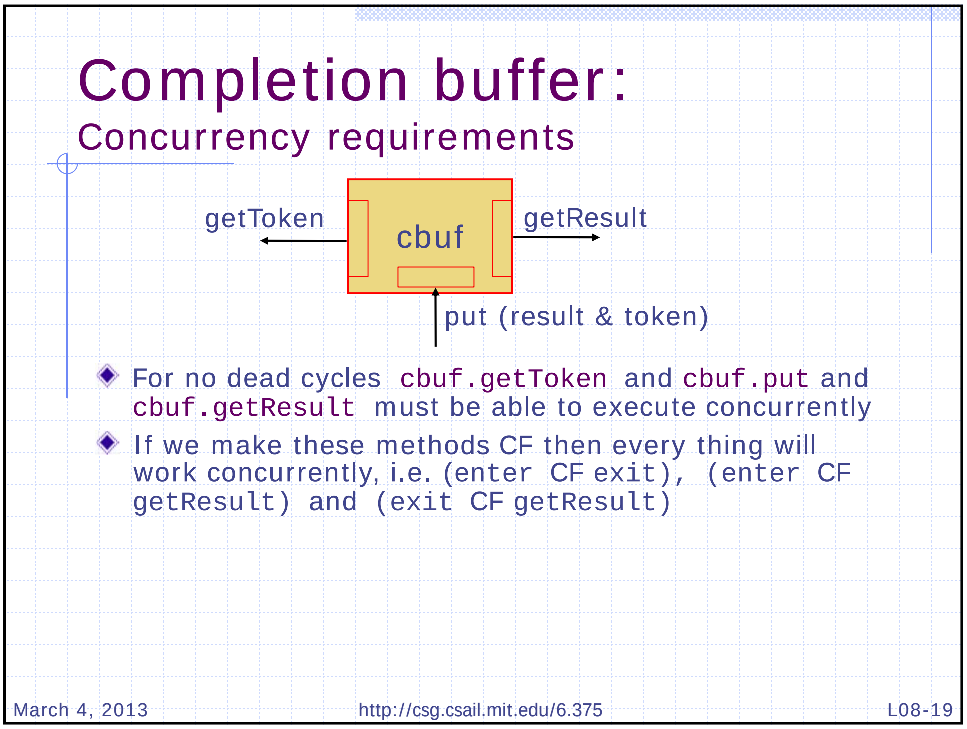
```
rule foo (f.notFull && f.notEmpty);
   f.enq (5) ; f.deq;
endrule
```

Can foo be
enabled?



Completion buffer: Interface

```
interface CBuffer#(type t);
  method ActionValue#(Token) getToken;
  method Action put(Token tok, t d);
  method ActionValue#(t) getResult;
endinterface
```

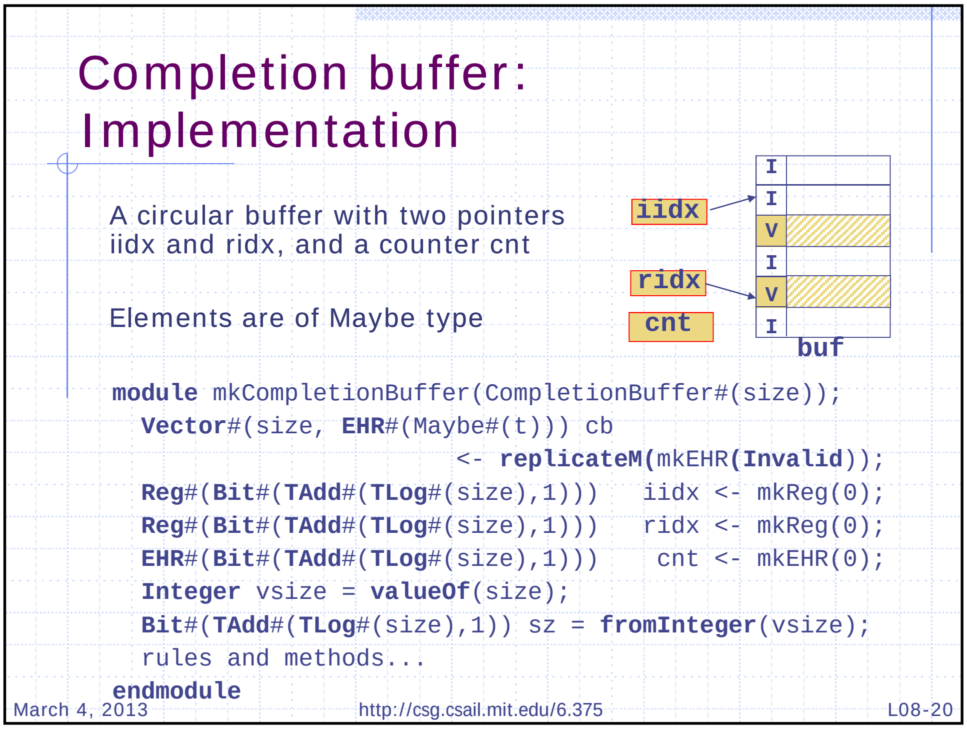


Completion buffer:

Concurrency requirements

getToken
cbuf
getResult
put (result & token)

- For no dead cycles cbuf.getToken and cbuf.put and cbuf.getResult must be able to execute concurrently
- If we make these methods CF then every thing will work concurrently, i.e. (enter CF exit), (enter CF getResult) and (exit CF getResult)



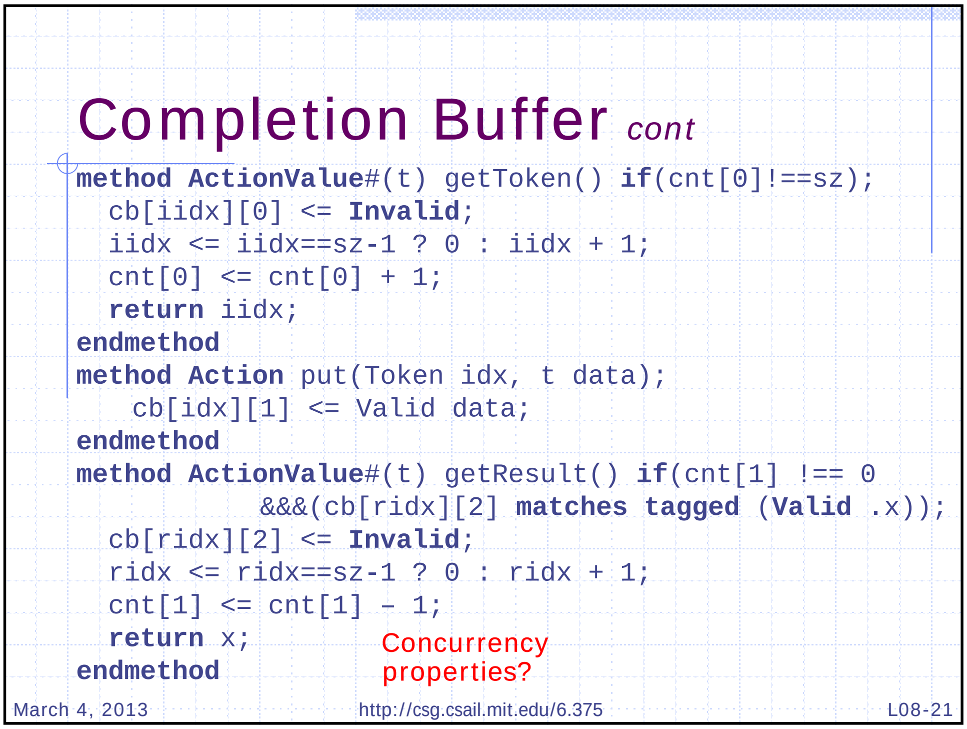
Completion buffer: Implementation

A circular buffer with two pointers iidx and ridx, and a counter cnt

Elements are of Maybe type

iidx
ridx
cnt
I
I
V
I
V
I
buf

```
module mkCompletionBuffer(CompletionBuffer#(size));
  Vector#(size, EHR#(Maybe#(t))) cb
                           <- replicateM(mkEHR(Invalid));
  Reg#(Bit#(TAdd#(TLog#(size),1)))   iidx <- mkReg(0);
  Reg#(Bit#(TAdd#(TLog#(size),1)))   ridx <- mkReg(0);
  EHR#(Bit#(TAdd#(TLog#(size),1)))    cnt <- mkEHR(0);
  Integer vsize = valueOf(size);
  Bit#(TAdd#(TLog#(size),1)) sz = fromInteger(vsize);
  rules and methods...
endmodule
```

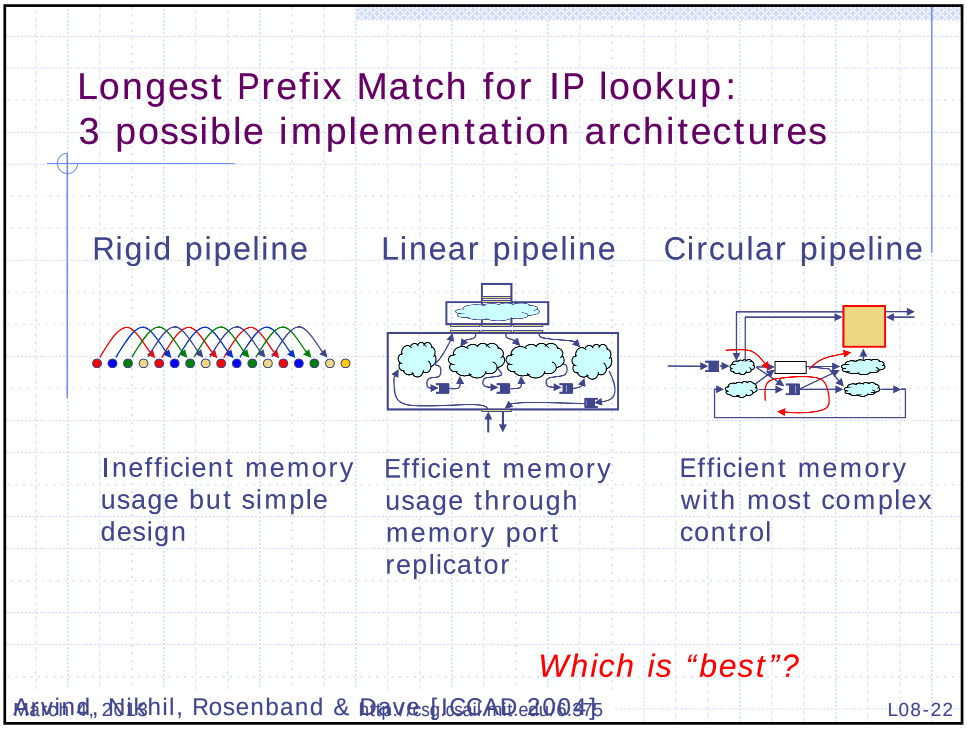


Completion Buffer *cont*

```
method ActionValue#(t) getToken() if(cnt[0]!==sz);
  cb[iidx][0] <= Invalid;
  iidx <= iidx==sz-1 ? 0 : iidx + 1;
  cnt[0] <= cnt[0] + 1;
  return iidx;
endmethod
method Action put(Token idx, t data);
   cb[idx][1] <= Valid data;
endmethod
method ActionValue#(t) getResult() if(cnt[1] !== 0
             &&&(cb[ridx][2] matches tagged (Valid .x));
  cb[ridx][2] <= Invalid;
  ridx <= ridx==sz-1 ? 0 : ridx + 1;
  cnt[1] <= cnt[1] – 1;
  return x;
endmethod
```

Concurrency properties?

March 4, 2013 http://csg.csail.mit.edu/6.375 L08-21



Longest Prefix Match for IP lookup: 3 possible implementation architectures

Rigid pipeline

Inefficient memory usage but simple design

Linear pipeline

Efficient memory usage through memory port replicator

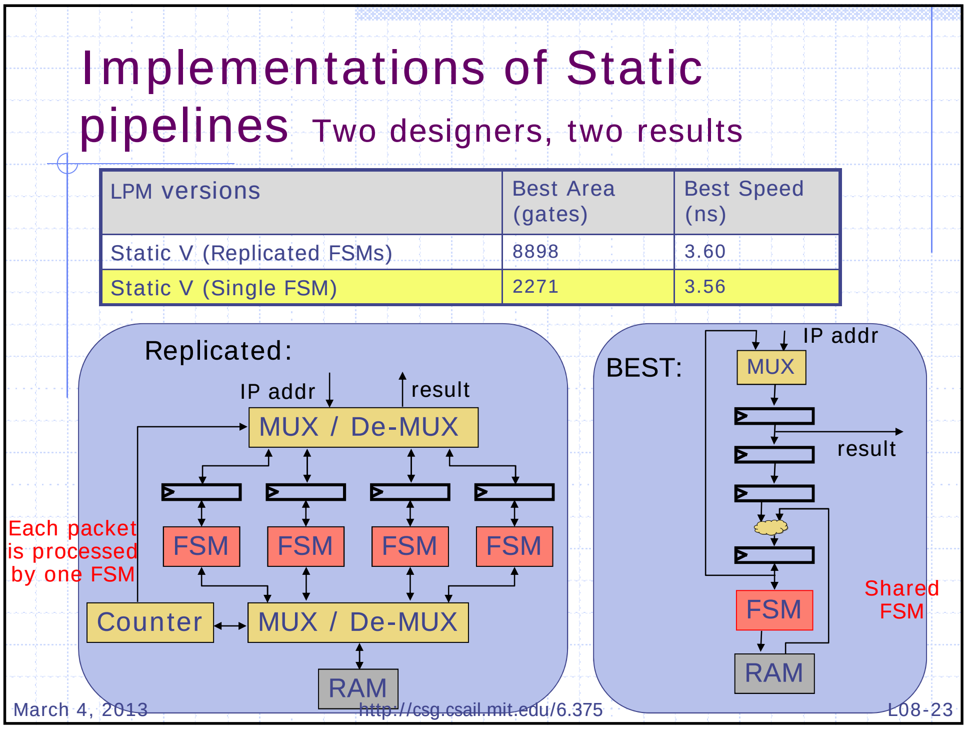
Circular pipeline

Efficient memory with most complex control

Which is "best"?

Arvind, Nikhil, Rosenband & Dave [ICCAD 2004]

March 4, 2013 http://csg.csail.mit.edu/6.375 L08-22

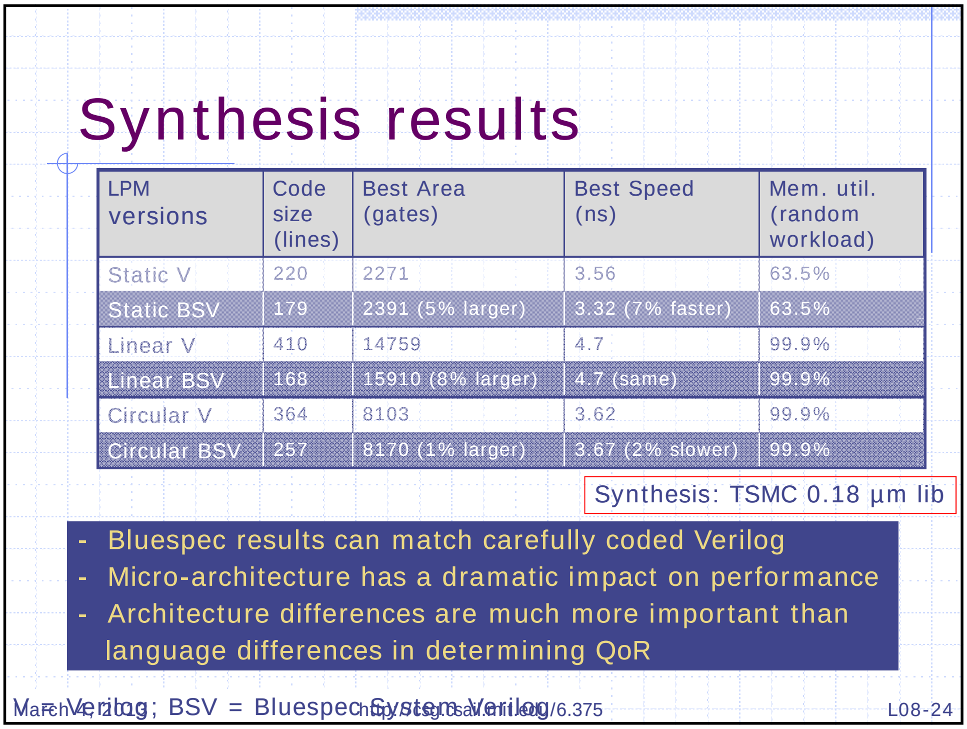


Implementations of Static pipelines Two designers, two results

LPM versions	Best Area (gates)	Best Speed (ns)
Static V (Replicated FSMs)	8898	3.60
Static V (Single FSM)	2271	3.56

Each packet is processed by one FSM

Shared FSM



Synthesis results

LPM versions	Code size (lines)	Best Area (gates)	Best Speed (ns)	Mem. util. (random workload)
Static V	220	2271	3.56	63.5%
Static BSV	179	2391 (5% larger)	3.32 (7% faster)	63.5%
Linear V	410	14759	4.7	99.9%
Linear BSV	168	15910 (8% larger)	4.7 (same)	99.9%
Circular V	364	8103	3.62	99.9%
Circular BSV	257	8170 (1% larger)	3.67 (2% slower)	99.9%

Synthesis: TSMC 0.18 µm lib

- Bluespec results can match carefully coded Verilog
- Micro-architecture has a dramatic impact on performance
- Architecture differences are much more important than language differences in determining QoR

V = Verilog; BSV = Bluespec System Verilog