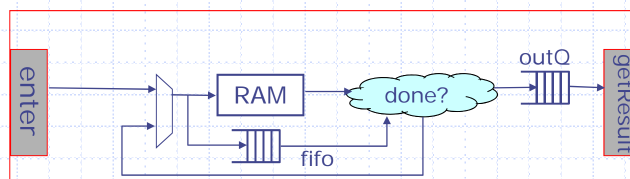
IP Lookup-2: The Completion Buffer

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

IP-Lookup module without the completion buffer




```
module mkIPLookup(IPLookup);
 rule recirculate…  ; rule exit …;
 method Action enter (IP ip);
   ram.req(ip[31:16]);
   fifo.enq(ip[15:0]);
 endmethod
 method ActionValue#(Msg) getResult();
  outQ.deq();
  return outQ.first();
 endmethod
endmodule
```

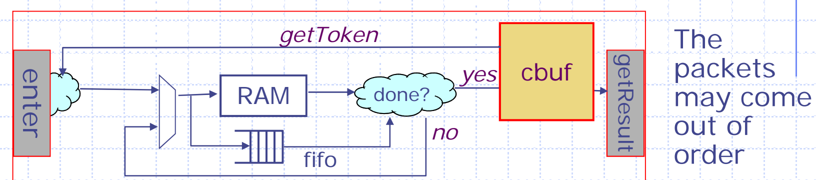
IP Lookup rules

```
rule recirculate (!isLeaf(ram.peek()));
   IP rip = fifo.first(); fifo.enq(rip << 8);
   ram.req(ram.peek() + rip[15:8]);
   fifo.deq(); ram.deq();
endrule
```

```
rule exit (isLeaf(ram.peek()));
   outQ.enq(ram.peek()); fifo.deq(); ram.deq();
endrule
```

Method `enter` and rule `exit` can be scheduled simultaneously, assuming `fifo.enq` and `fifo.deq` can be done simultaneouslyand `ram.req` and `ram.deq` can be done simultaneously

IP-Lookup module with the completion buffer

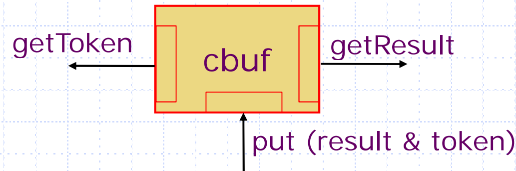


The packets may come out of order

- 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 now must also hold the "token" while the memory access is in progress: `Tuple2#(Token,Bit#(16))`

remainingIP

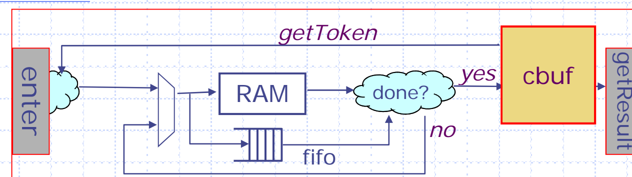
Completion buffer: Interface




```
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;
```


IP-Lookup module with the completion buffer




```
module mkIPLookup(IPLookup);
  rule recirculate…  ; rule exit …;
  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
```

for enter and getResult to execute simultaneously, cbuf.getToken and cbuf.getResult must execute simultaneously

IP Lookup rules with completion buffer

```
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
```

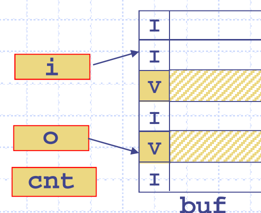
```
rule exit (isLeaf(ram.peek()));
   cbuf.put(ram.peek()); fifo.deq(); ram.deq();
endrule
```

For rule `exit` and method `enter` to execute simultaneously, `cbuf.put` and `cbuf.getToken` must execute simultaneously

$\Rightarrow$ For no dead cycles `cbuf.getToken` and `cbuf.put` and `cbuf.getResult` must be able to execute simultaneously

Completion buffer: Implementation

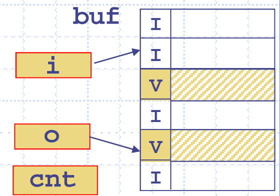
- A circular buffer with two pointers i and o, and a counter cnt
- Elements are of Maybe type




```
module mkCBuffer (CBuffer#(t))
                 provisos (Bits#(t,sz));
   RegFile#(Token, Maybe#(t)) buf <- mkRegFileFull();
   Reg#(Token)        i <- mkReg(0); //input index
   Reg#(Token)        o <- mkReg(0); //output index
   Reg#(Int#(32))   cnt <- mkReg(0); //number of filled slots
…
```

Elements must be representable as bits

Completion buffer: Concurrency Issue




```
// state elements
// buf, i, o, cnt ...
method ActionValue#(t) getToken()
                if (cnt < maxToken);
   cnt <= cnt + 1; i <= (i==maxToken) ? 0 : i + 1;
   buf.upd(i, Invalid);
   return i;
endmethod
method Action put(Token tok, t data);
   buf.upd(tok, Valid data);
endmethod
method ActionValue#(t) getResult()
     if (cnt > 0) &&&
    (buf.sub(o) matches tagged (Valid .x));
   o <= (o==maxToken) ? 0 : o + 1; cnt <= cnt - 1;
   return x;
endmethod
```

Can these methods execute concurrently?

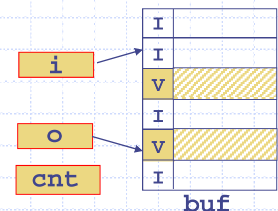
NO!

Concurrency Analysis

Problem 1

A circular buffer with two pointers i and o, and a counter cnt

Elements are of Maybe type

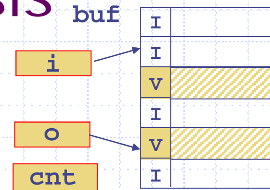


- buf must allow two simultaneous updates and one read
  - Needs a register file with one read and two write ports
- Since the updates are always to different addresses there is no data hazard and concurrent operations should be permitted

No compiler can detect that without full program analysis (i.e., understanding the use pattern)

Concurrency Analysis

Problem -2




```
// state elements
// buf, i, o, cnt ...
method ActionValue#(t) getToken()
                  if (cnt < maxToken);
   cnt <= cnt + 1; i <= (i==maxToken) ? 0 : i + 1;
   buf.upd(i, Invalid);
   return i;
endmethod
method Action put(Token tok, t data);
   buf.upd(tok, Valid data);
endmethod
method ActionValue#(t) getResult()
     if (cnt > 0) &&&
    (buf.sub(o) matches tagged (Valid .x));
   o <= (o==maxToken) ? 0 : o + 1; cnt <= cnt - 1;
   return x;
endmethod
```

Concurrent updates to cnt

A special counter module

- We often need to keep count of certain events
  - Need to read count, decrement and increment
  - Since decrementing and incrementing don't change the count we can remove some bypassing links
  - Implemented as Counter Library modules (implemented using Rwires)

Counter module

```
module mkCounter#(t v) (Counter#(t))
  provisos(Arith#(t), Literal#(t));
  Reg#(t) cnt <- mkConfigReg(v);
  RWire#(t) up <- mkRWire();
  RWire#(t) dn <- mkRWire();
  (*fire_when_enabled*)
  rule update(True);
    cnt <= cnt + fromMaybe(0, up.wget)
               - fromMaybe(0, dn.wget);
  endrule

  method t      _read()  = cnt;
  method Action incr(x) = up.wset(x);
  method Action decr(x) = dn.wset(x);
endmodule
```

Caution: rule update must fire otherwise the cnt won't be updated

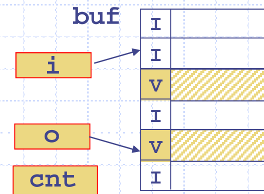
Multiported Register file

- 1R and 1W ports
  - bypass or no bypass
- 2R and 1W ports
  - bypass or no bypass
- 1R and 2W ports
  - multiple writes into the same register?
    - error vs priority
    - bypass or no bypass

All useful and each requires a different implementation

Which type of RF do we need for the completion buffer?

Completion buffer: Register File




```
// state elements i, o
// cnt <- mkCounter(0); buf <- ?
method ActionValue#(t) getToken()
                if (cnt.read() < maxToken);
   cnt.incr(); i <= (i==maxToken) ? 0 : i + 1;
   buf[i] <= Invalid;
   return i;
endmethod
method Action put(Token tok, t data);
   buf[tok] <= Valid data;
endmethod
method ActionValue#(t) getResult()
     if (cnt.read() > 0) &&&
    (buf[o] matches tagged Valid .x);
   o <= (o==maxToken) ? 0 : o + 1; cnt.decr();
   return x;
endmethod
```

buf should have 2W and 1R ports

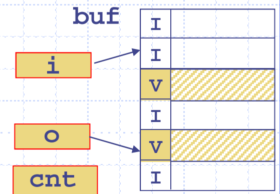
- multiple write error
- no bypassing

1R and 2W Register file

multiple writes – error; no bypassing

```
module mkRegFileFull1r1R2W(RegFile2#(Addr,Value));
  Reg#(Vector#(AddrSz, Value)) vs <- mkRegU;
  RWire#(Tuple2#(Addr, Value)) w1 <- mkRWire;
  RWire#(Tuple2#(Addr, Value)) w2 <- mkRWire;
  (* fire_when_enabed *)
  rule update(True);
    let vs_new = vs;
    case w1.wget() matches
      tagged Valid {.i1,.v1}: vs_new[i1]=v1; endcase
    case w2.wget() matches
      tagged Valid {.i2,.v2}: vs_new[i2]=v2; endcase
    vs <= vs_new;
 endrule
  method read(i) = vs[i];
  method w1(i,v) = w1.wset(tuple2(i,v));
  method w2(i,v) = w2.wset(tuple2(i,v)); endmodule
```

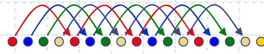
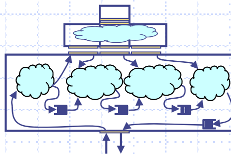
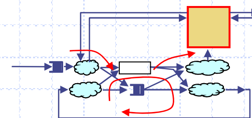
Completion buffer




```
// state elements i, o
// cnt <- mkCounter(0);
// buf <- mkRegFile1R2W;
method ActionValue#(t) getToken()
                if (cnt.read() < maxToken);
   cnt.incr(); i <= (i==maxToken) ? 0 : i + 1;
   buf.w1(i,Invalid);
   return i;
endmethod
method Action put(Token tok, t data);
   buf.w2(tok, Valid data);
endmethod
method ActionValue#(t) getResult()
      if (cnt.read() > 0) &&&
     (buf.read matches tagged Valid .x);
   o <= (o==maxToken) ? 0 : o + 1; cnt.decr();
   return x; endmethod
```

Problem solved

Longest Prefix Match for IP lookup: 3 possible implementation architectures

Rigid pipeline	Linear pipeline	Circular pipeline
		
Inefficient memory usage but simple design	Efficient memory usage through memory port replicator	Efficient memory with most complex control

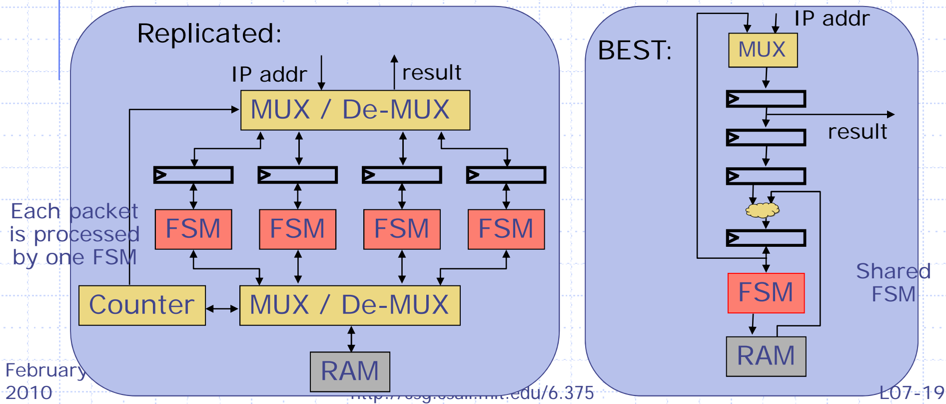
Which is "best"?

Arvind, Nikhil, Rosenband & Dave ICCAD 2004

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



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