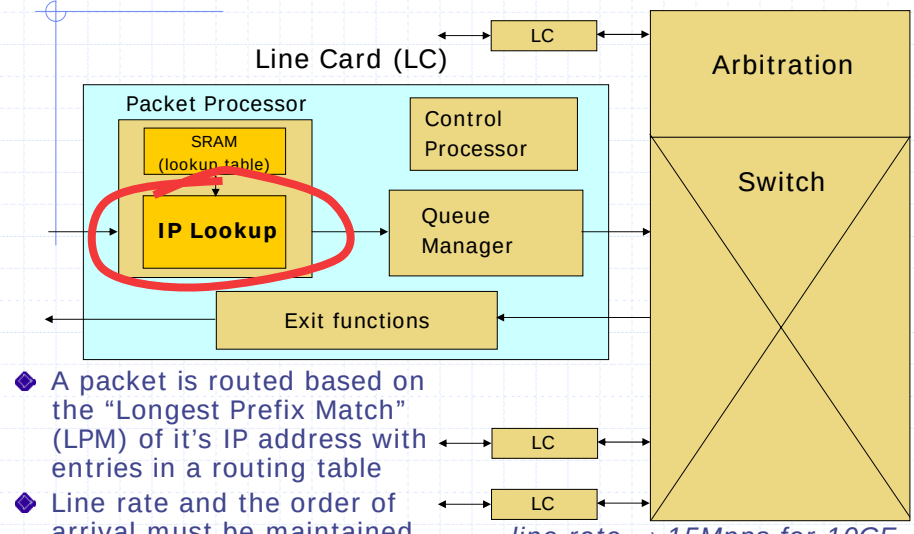
# Bluespec-4:

## Architectural exploration using IP lookup

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology



# IP Lookup block in a router

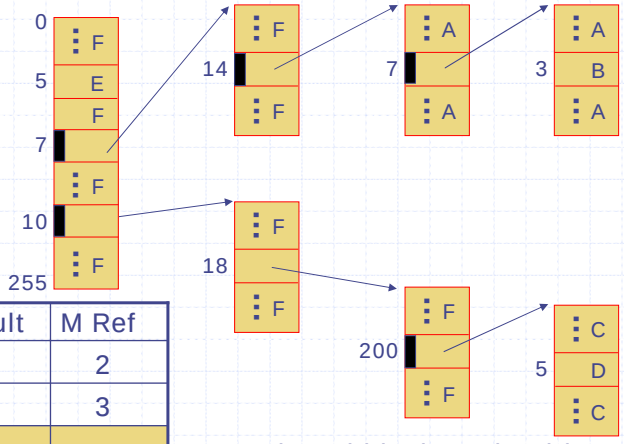



- A packet is routed based on the "Longest Prefix Match" (LPM) of it's IP address with entries in a routing table
- Line rate and the order of arrival must be maintained

*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 |

*Real-world lookup algorithms are more complex but all make a sequence of dependent memory references.*



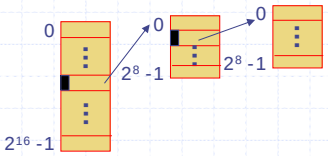
# Table representation issues

- Table size
  - Depends on the number of entries: 10K to 100K
  - Too big to fit on chip memory ➔ SRAM ➔ DRAM ➔ latency, cost, power issues
- Number of memory accesses for an LPM?
  - Too many ➔ difficult to do table lookup at line rate (say at 10Gbps)
- Control-plane issues:
  - incremental table update
  - size, speed of table maintenance software
- In this lecture (to fit the code on slides!):
  - Level 1: 16 bits, Level 2: 8 bits, Level 3: 8 bits

    ⇒ from 1 to 3 memory accesses for an LPM

# "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 */
}
```




Not obvious from the C code how to deal with
- memory latency
- pipelining

Memory latency ~30ns to 40ns

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

Must sustain 3 memory dependent lookups in 67 ns

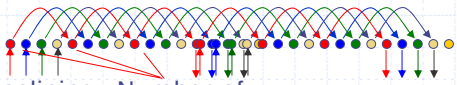


# IP Lookup:

## Microarchitecture -1
## Static Pipeline

# Static Pipeline

- Assume the memory has a latency of n (4) cycles and can accept a request every cycle
- Assume every IP look up takes exactly m (3) memory reads
- Assuming there is always an input to process:



Pipelining to deal with latency

Number of lookups per packet

The system needs space for at least n packets for full pipelining

Inefficient memory usage – unused memory slots represent wasted bandwidth

Difficult to schedule table updates



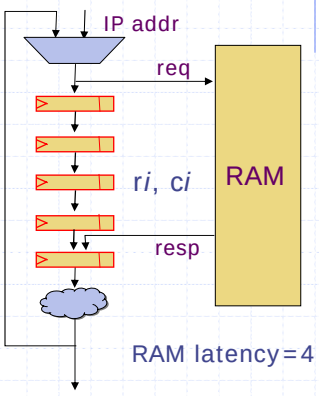
# Static (Synchronous) Pipeline Microarcitecture

- Provide n (> latency) registers; mark all of them as Empty
- Let a new message enter the system when the last register is empty or an old request leaves
- Each Register r hold either the result value or the remainder of the IP address. r5 also has to hold the next address for the memory

```
typedef union tagged {
  Value Result;
  struct{Bit#(16) remainingIP; Bit#(19) ptr;} IPptr
} regData
```

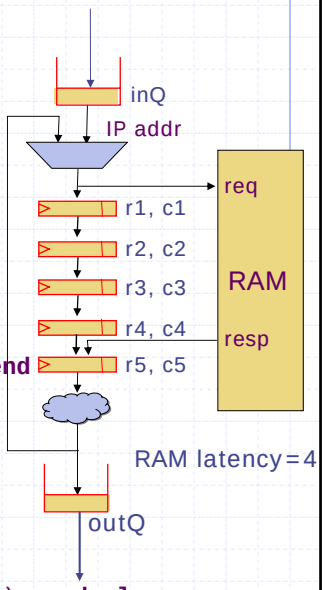
- The state c of each register is

```
typedef enum {
  Empty , Level1 , Level2 , Level3
} State;
```



# Static code

```
rule static (True);
   if (next(c5) == Empty)
      if (inQ.notEmpty) begin
       IP ip = inQ.first(); inQ.deq();
       ram.req(ext(ip[31:16]));
       r1 <= IPptr{ip[15:0],?};
       c1 <= Level1;
      end else c1 <= Empty;
   else begin
    r1 <= r5; c1 <= next(c5);
    if(!isResult(r5)) ram.req(ptr(r5));end
   r2 <= r1; c2 <= c1;
   r3 <= r2; c3 <= c2;
   r4 <= r3; c4 <= c3;
   TableEntry p;
   if((c4 != Empty)&& !isResult(r4))
        p <- ram.resp();
   r5 <= nextReq(p, r4); c5 <= c4;
   if (c5 == Level3) outQ.enq(result(r5)); endrule
```






# The next function

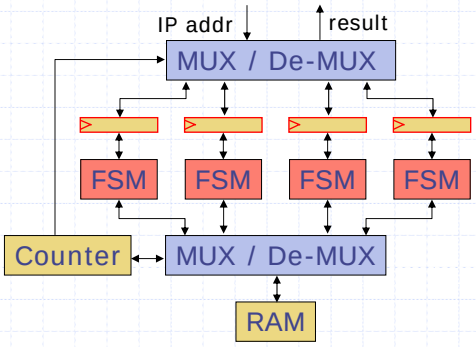
```
function State next (State c);
  case (c)
    Empty  : return(Empty);
    Level1 : return(Level2);
    Level2 : return(Level3);
    Level3 : return(Empty);
   endcase
endfunction
```

# The nextReq function

```
function RegData nextReq(TableEntry p, RegData r);
  case (r) matches
    tagged Result .*: return r;
    tagged IPptr .ip: if (isLeaf(p))
     return tagged Result value(p),
  else return tagged
          IPptr{remainingIP: ip.remainingIP << 8,
                ptr:  ptr(p) + ip.remainingIP[15:8]
               };
endcase
endfunction
```



# Another Static Organization




- Each packet is processed by its own FSM
- Counter determines which FSM gets to go

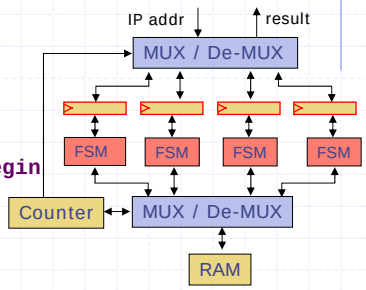
# Code for Static-2 Organization

```
function Action doFSM(r,c); action
  if (c == Empty)


  else if (c == Level1 || c == Level2) begin


  else if (c == Level3) begin


endaction endfunction
```

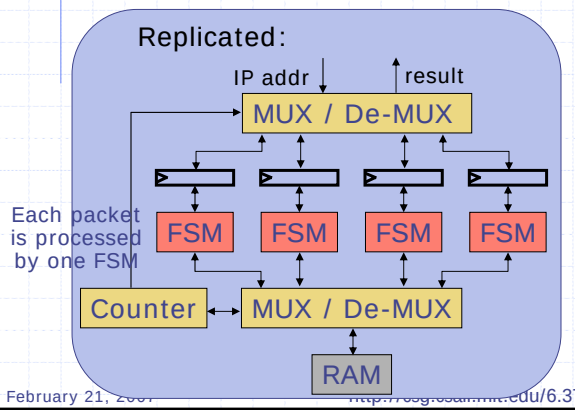


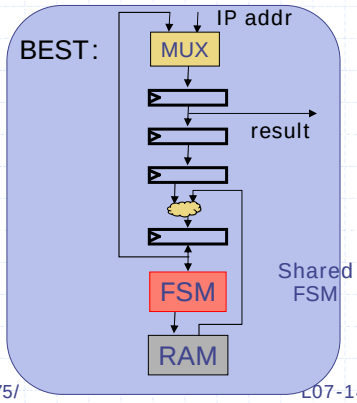



# Implementations of Static pipelines

## 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 |



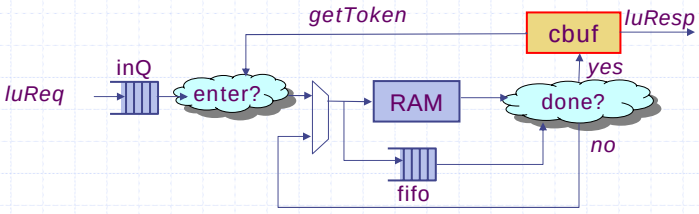
# IP Lookup:

## Microarchitecture -2
## Circular Pipeline

5-minute break to stretch you legs



# Circular pipeline

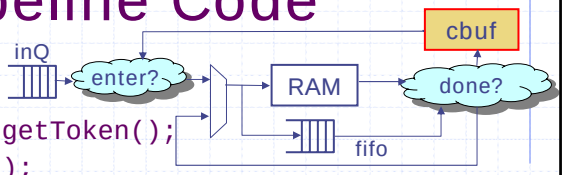



Completion buffer
- gives out tokens to control the entry into the circular pipeline
- ensures that departures take place in order even if lookups complete out-of-order

The fifo holds the token while the memory access is in progress: `Tuple2#(Bit#(16), Token)`

remainingIP

# Circular Pipeline Code



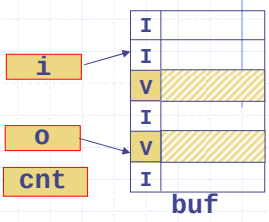

```
rule enter (True);
   Token tok <- cbuf.getToken();
   IP ip = inQ.first();
   ram.req(ext(ip[31:16]));
   fifo.enq(tuple2(ip[15:0], tok)); inQ.deq();
endrule
```

```
rule recirculate (True);
   TableEntry p <- ram.resp();
   match {.rip, .t} = fifo.first();
   if (isLeaf(p)) cbuf.put(t, p);
   else begin
       fifo.enq(tuple2(rip << 8, tok));
       ram.req(p+signExtend(rip[15:8]));
     end
   fifo.deq();
endrule
```



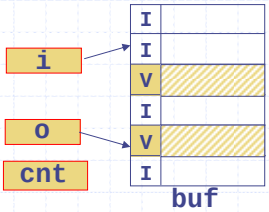
# Completion buffer

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




```
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#(Token) cnt <- mkReg(0); //number of filled slots
…
```

# Completion buffer




```
... // state elements buf, i, o, n ..
method ActionValue#(t)
          getToken() if (cnt <= maxToken);
 cnt <= cnt + 1; i <= i + 1;
 buf.upd(i, Invalid);
 return i; endmethod

method Action put(Token tok, t data);
  return buf.upd(tok, Valid data); endmethod

method ActionValue#(t) getResult() if (cnt > 0) &&&
             (buf.sub(o) matches tagged (Valid .x));
  o <= o + 1; cnt <= cnt - 1;
  return x; endmethod
```

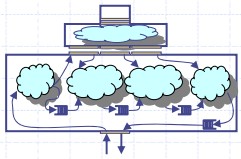


# Longest Prefix Match for IP lookup: 3 possible implementation architectures

Rigid pipeline

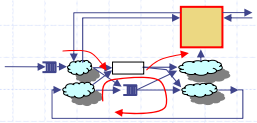
Inefficient memory usage but simple design

Linear pipeline



Efficient memory usage through memory port replicator

Circular pipeline



Efficient memory with most complex control

*Designer's Ranking:* 1 2 3

*Which is "best"?*

Arvind, Nikhil, Rosenband & Dave ICCAD 2004

# 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



# A problem ...

```
rule recirculate (True);
   TableEntry p <- ram.resp();
   match {.rip, .t} = fifo.first();
   if (isLeaf(p)) cbuf.put(t, p);
   else begin
        fifo.enq(tuple2(rip << 8, tok));
        ram.req(p+signExtend(rip[15:8]));
     end
   fifo.deq();
endrule
```

What condition does the fifo need to satisfy for this rule to fire?

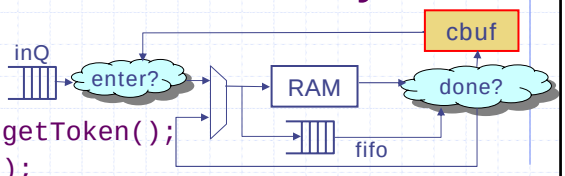
# One Element FIFO

```
module mkFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  method Action enq(t x) if (!full);
    full <= True;     data <= x;
  endmethod
  method Action deq() if (full);
    full <= False;
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod
endmodule
```

enq and deq cannot be enabled together!



# Another Problem: Dead cycle elimination




```
rule enter (True);
   Token tok <- cbuf.getToken();
   IP ip = inQ.first();
   ram.req(ext(ip[31:16]));
   fifo.enq(tuple2(ip[15:0], tok)); inQ.deq();
endrule
```

Can a new request enter the system simultaneously with an old one leaving?

*Solutions next time ...*

```
rule recirculate (True);
   TableEntry p <- ram.resp();
   match {.rip, .t} = fifo.first();
   if (isLeaf(p)) cbuf.put(t, p);
   else begin
        fifo.enq(tuple2(rip << 8, tok));
        ram.req(p+signExtend(rip[15:8]));
     end
   fifo.deq();
endrule
```