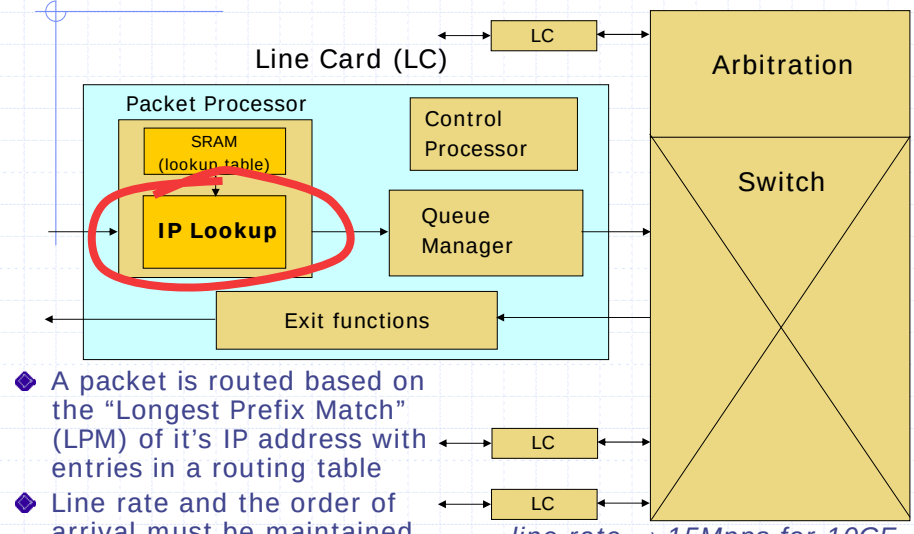
IP Lookup

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

IP Lookup block in a router

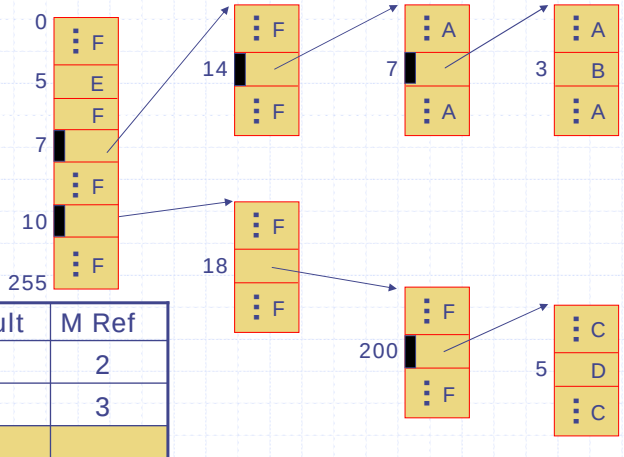


- A packet is routed based on the "Longest Prefix Match" (LPM) of it's IP address with entries in a routing table
- Line rate and the order of arrival must be maintained

line rate ⇒ 15Mpps for 10GE

Sparse tree representation

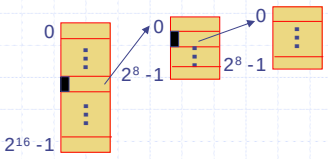
7.14.*.*	A
7.14.7.3	B
10.18.200.*	C
10.18.200.5	D
5.*.*.*	E
*	F



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

"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 */
}
```



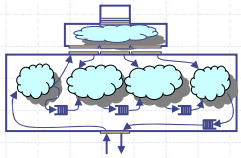
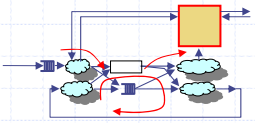
Not obvious from the C code how to deal with
- memory latency
- pipelining

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

Longest Prefix Match for IP lookup: 3 possible implementation architectures

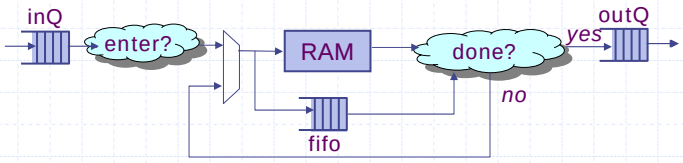
Rigid pipeline	Linear pipeline	Circular pipeline
		
Inefficient memory usage but simple design	Efficient memory usage through memory port replicator	Efficient memory with most complex control

Designer's Ranking:

Which is "best"?

Arvind, Nikhil, Rosenband & Dave ICCAD 2004

Circular pipeline

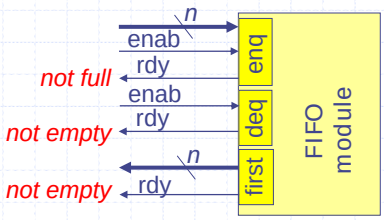


The fifo holds the request while the memory access is in progress

The architecture has been simplified for the sake of the lecture. Otherwise, a "completion buffer" has to be added at the exit to make sure that packets leave in order.

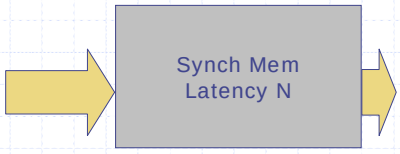
FIFO

```
interface  FIFO#(type t);
  method Action enq(t x);  // enqueue an item
  method Action deq();     // remove oldest entry
  method t first();        // inspect oldest item
endinterface
```



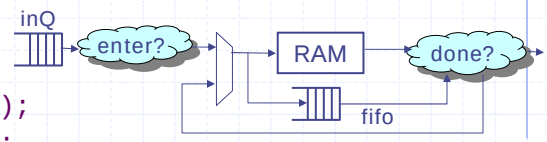
n = # of bits needed to represent a value of type t

Request-Response Interface for Synchronous Memory



Making a synchronous component latency-insensitive

Circular Pipeline Code



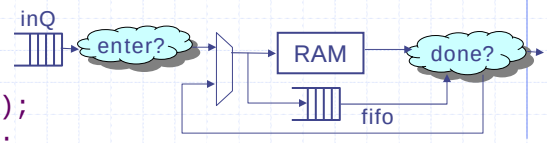

```
rule enter (True);
   IP ip = inQ.first();
   ram.req(ip[31:16]);
   fifo.enq(ip[15:0]);
   inQ.deq();
endrule
```

done? Is the same as isLeaf

When can enter fire?

```
rule recirculate (True);
   TableEntry p = ram.peek(); ram.deq();
   IP rip = fifo.first();
   if (isLeaf(p)) outQ.enq(p);
   else begin
       fifo.enq(rip << 8);
       ram.req(p + rip[15:8]);
     end
   fifo.deq();
endrule
```


Circular Pipeline Code: *discussion*



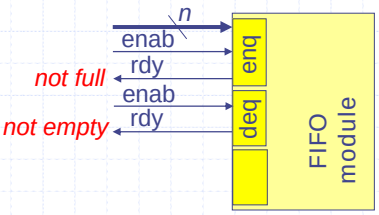

```
rule enter (True);
   IP ip = inQ.first();
   ram.req(ip[31:16]);
   fifo.enq(ip[15:0]);
   inQ.deq();
endrule
```

When can recirculate fire?

```
rule recirculate (True);
   TableEntry p = ram.peek(); ram.deq();
   IP rip = fifo.first();
   if (isLeaf(p)) outQ.enq(p);
   else begin
       fifo.enq(rip << 8);
       ram.req(p + rip[15:8]);
     end
   fifo.deq();
endrule
```

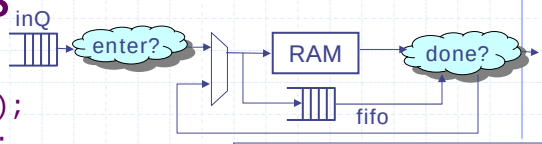
One Element FIFO

```
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
```



Dead cycles

```
rule enter (True);
   IP ip = inQ.first();
   ram.req(ip[31:16]);
   fifo.enq(ip[15:0]); inQ.deq();
endrule
```



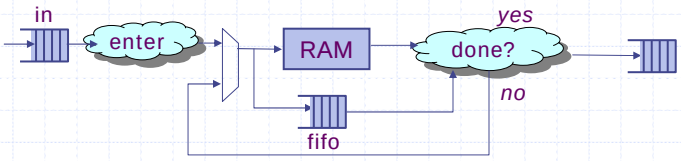
assume simultaneous enq & deq is allowed

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

Is this worth worrying about?

```
rule recirculate (True);
   TableEntry p = ram.peek(); ram.deq();
   IP rip = fifo.first();
   if (isLeaf(p)) outQ.enq(p);
   else begin
        fifo.enq(rip << 8);
        ram.req(p + rip[15:8]);
     end
   fifo.deq();
endrule
```

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?

The compiler issue

- Can the compiler detect all the conflicting conditions?
- Does the compiler detect conflicts that do not exist in reality?

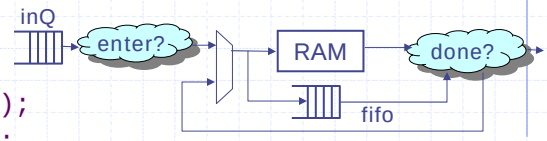
Scheduling conflicting rules

- When two rules conflict on a shared resource, they cannot both execute in the same clock
- The compiler produces logic that ensures that, when both rules are applicable, only one will fire
  - Which one?

source annotations

```
(* descending_urgency = "recirculate, enter" *)
```


So is there a dead cycle?




```
rule enter (True);
   IP ip = inQ.first();
   ram.req(ip[31:16]);
   fifo.enq(ip[15:0]); inQ.deq();
endrule
```

In general these two rules conflict but when isLeaf(p) is true there is no apparent conflict!

```
rule recirculate (True);
   TableEntry p = ram.peek(); ram.deq();
   IP rip = fifo.first();
   if (isLeaf(p)) outQ.enq(p);
   fifo.deq();
endrule
```

Rule Spliting

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

≡

```
rule fooT (p);
   r1 <= 5;
endrule

rule fooF (!p);
   r2 <= 7;
endrule
```

rule fooT and fooF can be scheduled independently with some other rule

Spliting the recirculate rule

```
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
```

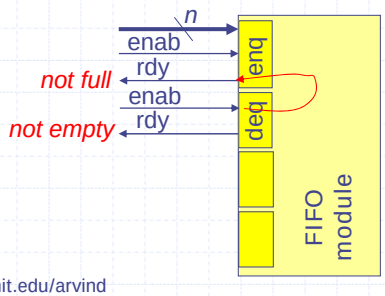
```
rule enter (True);
   IP ip = inQ.first(); ram.req(ip[31:16]);
   fifo.enq(ip[15:0]); inQ.deq();
endrule
```

Now rules `enter` and `exit` can be scheduled simultaneously, assuming `fifo.enq` and `fifo.deq` can be done simultaneously

Back to the fifo problem

```
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
```

The functionality we want is as if deq "happens" before enq; if deq does not happen then enq behaves normally



RWire to rescue

```
interface RWire#(type t);
      method Action wset(t x);
      method Maybe#(t) wget();
endinterface
```

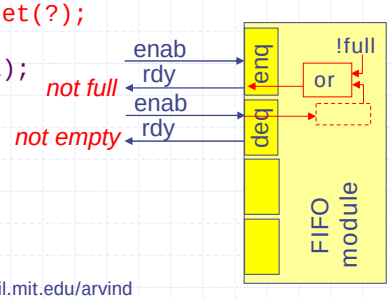
Like a register in that you can read and write it but unlike a register

- read happens after write
- data disappears in the next cycle

One Element "Loopy" FIFO

```
module mkLFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  RWire#(void) deqEN <- mkRWire();
  method Action enq(t x) if
              (!full || isValid (deqEN.wget()));
    full <= True;     data <= x;
  endmethod
  method Action deq() if (full);
    full <= False; deqEN.wset(?);
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod
endmodule
```

This works correctly in both cases (fifo full and fifo empty).

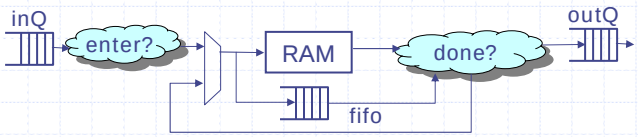


Problem solved!

```
LFIFO fifo <- mkLFIFO;
      // use a loopy fifo
rule recirculate (True);
 TableEntry p = ram.peek();
 ram.deq();
 IP rip = fifo.first();
 if (isLeaf(p)) outQ.enq(p);
 else
  begin
   fifo.enq(rip << 8);
   ram.req(p + rip[15:8]);
  end
 fifo.deq();
endrule
```

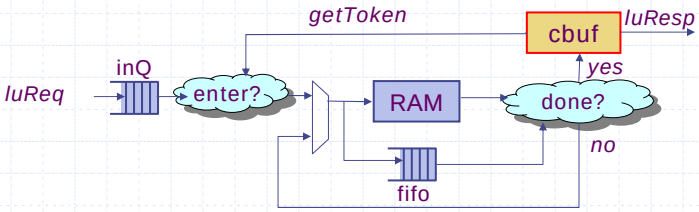
- RWire has been safely encapsulated inside the Loopy FIFO – users of Loopy fifo need not be aware of RWires

Packaging a module:
Turning a rule into a method




```
rule enter (True);
   IP ip = inQ.first();
   ram.req(ip[31:16]);
   fifo.enq(p[15:0]);
   inQ.deq();
endrule
```


Circular pipeline
with Completion Buffer



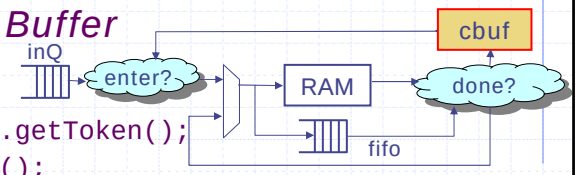
Completion buffer
- gives out tokens to control the entry into the circular pipeline
- ensures that departures take place in order even if lookups complete out-of-order

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

remainingIP

Circular Pipeline Code

with Completion Buffer

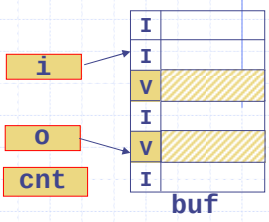



```
rule enter (True);
   Token tok <- cbuf.getToken();
   IP ip = inQ.first();
   ram.req(ip[31:16]);
   fifo.enq(tuple2(ip[15:0], tok)); inQ.deq();
endrule
```

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

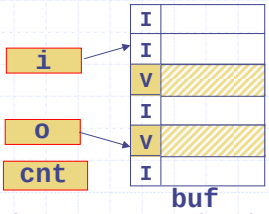

Completion buffer

```
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#(Int#(32))  cnt <- mkReg(0); //number of filled slots
…
```

Completion buffer



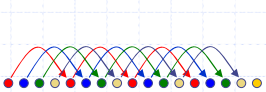
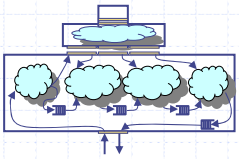
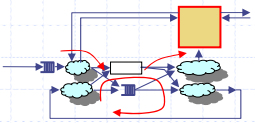

```
// state elements
// buf, i, o, n ...
method ActionValue#(t) getToken() if (cnt < maxToken);
 cnt <= cnt + 1; i <= i + 1;
 buf.upd(i, Invalid);
 return i; endmethod

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
```

Home work: Think about concurrency Issues, i.e., can these methods be executed concurrently? Do they need to?

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

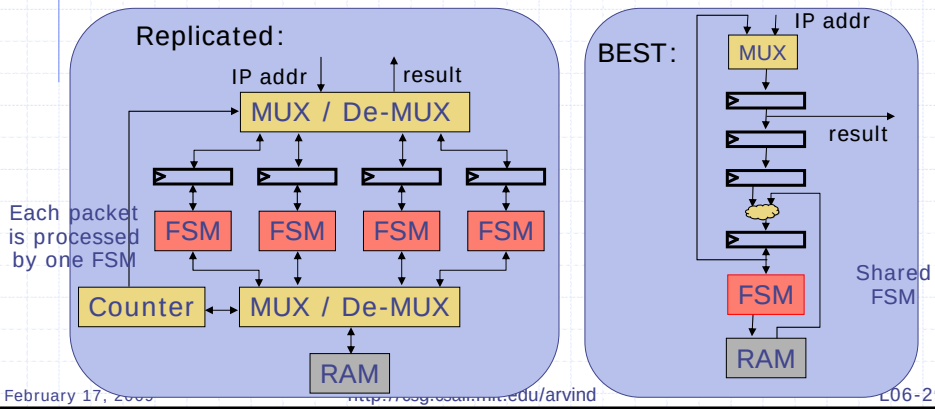
Which is "best"?

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