A packet is routed based on the "Longest Prefix Match" (LPM) of its 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

<table>
<thead>
<tr>
<th>IP address</th>
<th>Result</th>
<th>M Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.13.7.3</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>10.18.201.5</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>7.14.7.2</td>
<td>F</td>
<td>4</td>
</tr>
<tr>
<td>5.13.7.2</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>10.18.200.7</td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

"C" version of LPM

```c
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

Must process a packet every 1/15 μs or 67 ns
Must sustain 3 memory dependent lookups in 67 ns

Memory latency
~30ns to 40ns
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

---

**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

```plaintext
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 Memory

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

```verilog
circular pipelin code

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

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
```

When can enter fire?

When can recirculate fire?

---

Circular Pipeline Code: discussion

```verilog
circular pipelin code

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

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
```

When can recirculate fire?
One Element FIFO

```verilog
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

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

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
```

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

Is this worth worrying about?
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?

```plaintext
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);
  else begin
    fifo.enq(rip << 8);
    ram.req(p + rip[15:8]);
  end
  fifo.deq();
endrule
```

Done?
Rule Spliting

\begin{align*}
\textit{rule} \; \text{foo} \; (\text{True}); \\
& \quad \text{if} \; (p) \; r1 <= 5; \\
& \quad \text{else} \; r2 <= 7; \\
& \quad \text{endrule}
\end{align*}

\begin{align*}
\textit{rule} \; \text{fooT} \; (p); \\
& \quad r1 <= 5; \\
& \quad \text{endrule}
\end{align*}

\begin{align*}
\textit{rule} \; \text{fooF} \; (!p); \\
& \quad r2 <= 7; \\
& \quad \text{endrule}
\end{align*}

\text{rule fooT and fooF can be scheduled independently with some other rule}

Spliting the recirculate rule

\begin{align*}
\textit{rule} \; \text{recirculate} \; (!\text{isLeaf(ram.peek())}); \\
& \quad \text{IP} \; \text{rip} = \text{fifo.first(); fifo.enq(rip} << 8); \\
& \quad \text{ram.req(ram.peek() + rip[15:8]);} \\
& \quad \text{fifo.deq(); ram.deq();} \\
& \quad \text{endrule}
\end{align*}

\begin{align*}
\textit{rule} \; \text{exit} \; (\text{isLeaf(ram.peek())}); \\
& \quad \text{outQ.enq(ram.peek()); fifo.deq(); ram.deq();} \\
& \quad \text{endrule}
\end{align*}

\begin{align*}
\textit{rule} \; \text{enter} \; (\text{True}); \\
& \quad \text{IP} \; \text{ip} = \text{inQ.first(); ram.req(ip[31:16]);} \\
& \quad \text{fifo.enq(ip[15:0]); inQ.deq();} \\
& \quad \text{endrule}
\end{align*}

\text{Now rules enter and exit can be scheduled simultaneously}
Back to the fifo problem

```verilog
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

```verilog
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

```verbatim
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
```

Problem solved!

```verbatim
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(ip[15:0]);
    inQ.deq();
endrule

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)
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) cnt <- mkReg(0); //number of filled slots
Completion buffer

```haskell
... // 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: 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
```
Implementations of Static pipelines
Two designers, two results

<table>
<thead>
<tr>
<th>LPM versions</th>
<th>Best Area (gates)</th>
<th>Best Speed (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static V (Replicated FSMs)</td>
<td>8898</td>
<td>3.60</td>
</tr>
<tr>
<td>Static V (Single FSM)</td>
<td>2271</td>
<td>3.56</td>
</tr>
</tbody>
</table>

Replicated:
- IP addr
- MUX / De-MUX
- FSM
- FSM
- FSM
- FSM
- Counter
- MUX / De-MUX
- RAM

BEST:
- IP addr
- MUX
- FSM
- RAM

Synthesis results

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

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