IP Lookup

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 \(\Rightarrow 15 \text{Mpps for 10GE}\)
Sparse tree representation

<table>
<thead>
<tr>
<th>IP address</th>
<th>Result</th>
<th>M Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.14.<em>.</em></td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>7.14.7.3</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>10.18.200.*</td>
<td>C</td>
<td>18</td>
</tr>
<tr>
<td>10.18.200.5</td>
<td>D</td>
<td>21</td>
</tr>
<tr>
<td>5.<em>.</em>.*</td>
<td>E</td>
<td>10</td>
</tr>
<tr>
<td>*</td>
<td>F</td>
<td>255</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.
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

**Discussion**

```
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 enter fire?

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

Can a new request enter the system when an old one is leaving? Is this worth worrying about?

```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
```
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?

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

```
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);
    fifo.deq();
endrule
```
Rule Splitting

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

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

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

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

Spliting the recirculate rule

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

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

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

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

```verilog
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!

```verilog
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

```verilog
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)`
Circular Pipeline Code

with Completion Buffer

```verilog
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]);
        fifo.deq();
    end
endrule
```

Completion buffer

```verilog
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
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

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>

- 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