Exploring Microarchitectures: IP Lookup Module

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

November 9, 2006
http://csg.csail.mit.edu/6.827/

This lecture has two purposes

- Examine very different designs to solve a problem
  - Example: IP Lookups in Internet routers
- Illustrate some features of BSV
  - Rules to manage complex concurrency controls
  - Packaging designs into modules
  - Dealing with Synchronous RAMs

No prior understanding of IP Lookup problem is necessary to follow this lecture

November 9, 2006
http://csg.csail.mit.edu/6.827/
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.

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

```
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
IP Lookup:

Microarchitecture - 1
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 Pipeline Microarchitecture

- 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.
- Note the state of each register:

\[
\text{typedef enum} \{ \\
\text{Empty, Level1, Level2, Level3} \\
\} \text{State};
\]

RAMELE:RAM latency=4

Static code

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

The next function

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

The nextReq function

```c
function RegData nextReq(TableEntry p, RegData r); 
if (hasValue(r)) return r; 
else if (isLeaf(p)) 
  return RegData{value: Value value(p), 
                  addr: ?}; 
else return RegData{ 
  value: Ptr (ptr(p) + r.addr[15:8]), 
  addr: addr << 8 
};
```
Another Static Organization

Each packet is processed by one FSM

Code for Static-2 Organization

```plaintext
function Action doFSM(r, c); action
    TableEntry p;
    if (c == Level3 || c == Empty)
        if (in.notEmpty) begin
            IP ip = in.first();
            ram.req(ip[31:16]); r <= ip[15:0];
            in.deq(); c <= Level1;
        end else c <= Empty;
    else begin
        ram.req(r);
        c <= next(c);
        if (c != Empty) p <- ram.resp();
        r <= nextReq(p, r);
    end
    if (c == Level1) out.enq(rc);
endaction
rule static2(True);
    cnt <= cnt + 1;
    for (Integer i=0; i<maxLat; i=i+1)
        if (fromInteger(i) == cnt)
            doFSM(r[cnt], c[cnt]);
endrule
```

r and c are a set of 4 registers
### 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)</td>
<td>8898</td>
<td>3.60</td>
</tr>
<tr>
<td>Static V (BEST)</td>
<td>2271</td>
<td>3.56</td>
</tr>
</tbody>
</table>

### RAMs: Synchronous vs Asynchronous view

- Basic memory components are "synchronous":
  - Present a read-address \( A_J \) on clock \( J \)
  - Data \( D_J \) arrives on clock \( J+N \)
  - If you don't "catch" \( D_J \) on clock \( J+N \), it may be lost, i.e., data \( D_{J+1} \) may arrive on clock \( J+1+N \)

- This kind of synchronicity can pervade the design and cause complications
Asynchronous Interfaces for RAMs

It's easier to work with an "asynchronous" block

interface AsyncRAM\(#(type addr\_T, type data\_T)\);
  method Action req(addr\_T a);
  method ActionValue\#(data\_T) resp();
endinterface

Action Value methods

- Value method: Only reads the state; does not affect it
  - e.g. fifo.first()
- Action method: Affects the state but does not return a value
  - e.g. fifo.deq(), fifo.enq(x), fifo.clear()
- Action Value method: Returns a value but also affects the state
  - e.g. fifo.pop()
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

method ActionValue t pop() if (full);
    full <= False; return (data);
endmethod

IP Lookup:

Microarchitecture -2
Circular Pipeline
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

Circular Pipeline Code

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

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

When can these rules fire?
Completion buffer

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

module mkCBuffer (CBuffer#(any_T))
  provisos (Bits#(any_T,sz));
  RegFile#(Token, Maybe#(any_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
  ...

  method ActionValue#(any_T) getToken()
    if (cnt <= maxToken)
      cnt <= cnt + 1; i <= i + 1;
      buf.upd(i, Invalid);
    return i;
  endmethod

  method Action done(Token t, any_T data)
    return buf.upd(t, Valid data);
  endmethod

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

... // state elements buf, i, o, n...
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

Circular Pipeline Code

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

rule done (True);
    TableEntry p <= ram.resp();
    match (.rip, .t) = fifo.first();
    if (isLeaf(p)) cbuf.done(t, p);
    else begin
        fifo.enq(rip << 8, t);
        ram.req(p + signExtend(rip[15:7]));
    end
    active.deq();
endrule

(* descending_urgency = "done, enter" *)

fifo.first() < fifo.deq() < fifo.enq(...)
```

Can rules enter and done be applicable simultaneously?
Which one should go?

What is the concurrency expectation for the fifo?

Expected functionality
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

One Element FIFO

Concurrency?
- enq and deq? Conflict but they cannot be enabled together!
- first and deq? first < deq
- first and enq? Mutually exclusive
- clear and deq? deq < clear
- clear and enq? enq < clear

The good news ...

◆ It is always possible to transform your design to meet desired concurrency and functionality

How? Advanced topic!
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”?

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>229</td>
<td>2264 (very same)</td>
<td>3.32 (2% faster)</td>
<td>62.3%</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>446</td>
<td>14699</td>
<td>4.7</td>
<td>99.9%</td>
</tr>
<tr>
<td>Circular V</td>
<td>364</td>
<td>8163</td>
<td>3.62</td>
<td>99.9%</td>
</tr>
<tr>
<td>Circular BSV</td>
<td>257</td>
<td>8170 (6% larger)</td>
<td>3.32 (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

V = Verilog; BSV = Bluespec System Verilog