FIFO and Concurrency Issues

Nirav Dave
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

November 14, 2006 http://csg.csail.mit.edu/6.827/

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
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>Linear V</td>
<td>410</td>
<td>14759</td>
<td>4.7</td>
<td>99.9%</td>
</tr>
<tr>
<td>Linear BSV</td>
<td>263</td>
<td>15910 (46% 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>762</td>
<td>36577</td>
<td>3.62</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

Bluespec results can match carefully coded Verilog
Compiler Argument: Within Reason
- Micro-architecture has a dramatic impact on performance

V = Verilog; BSV = Bluespec System Verilog

Circular pipeline

- 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

```verilog
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

```verilog
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
    ...
```

November 14, 2006 L18-7
http://csg.csail.mit.edu/6.827/
Completion buffer

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

Sharing Methods

- Both “done” and “recirc” make a memory request and enqueue into the fifo. How do we handle this sharing?

- Software approach: Make copies for each rule
  - How do we keep atomicity?
  - More analysis!
    - Does this scale?

- Hardware approach: Only one of them gets to go. They conflict
  - Which one get to go?
    source annotations
Simple Example

```verbatim
rule a(p);
    fifo.enq(f(r0));
    r0 <= r0 + 1;
endrule

rule b(q);
    fifo.enq(f'(r1));
    r1 <= r1 + 1;
endrule
```

Circular Pipeline Code

```verbatim
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
    fifo.deq();
endrule
```

Can rules enter and done be applicable simultaneously?

Which one should go?

What is the concurrency expectation for the fifo?
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

The good news ...

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

How? Good Question!
Register Interfaces


dq
write.x
write.en
read x
write.en

read' – returns the current state when write is not enabled
read' – returns the value being written if write is enabled

Ephemeral History Register (EHR)

read0 < write0 < read1 < write1 < ....

write0.x
write0.en
write1.x
write1.en
read0
read1

writei+1 takes precedence over writei
Textual Description

method r0w0r1w1(w0,x0,w1,x1);
    let r0 = r
    let r1 = w0 ? x0 : r0;
    let r2 = w1 ? x1 : r1;
    if (w0 | w1)
        r <= r2;
endmethod

We’ve merged all the methods together – Do the Rules also get merged together?

The One Element FIFO now

module mkFIFO1 (FIFO#(t));
    Reg#(t)    data  <- mkRegU();
    Reg#(Bool) full  <- mkReg(False);
method ActionValue#(t)
    first_deq$enq(deq,enq,enqx) if (full);
    let full' = (deq) ? False : full;
    let full'' = (enq) ? True  : full';
    if (enq)
        data <= enqx;
    if (deq | enq)
        full <= full'';
endmethod
endmodule
Compositions

This sort of composition can be generalized to arbitrary rules/methods

Can add sequential composition as well

So now...

Any questions?

There’s a quiz coming up