IP Lookup: Some subtle concurrency issues

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

but first a correction from the last lecture ...
One-Element Pipeline FIFO

module mkPipelineFIFO1 (FIFO#(t));
    Reg#(t) data <- mkRegU();
    Reg#(Bool) full <- mkReg(False);
    RWire#(void) deqEN <- mkRWire();
    Bool deqp = isValid (deqEN.wget());
method Action enq(t x) if (!full || deqp);
    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
Solution - Config registers

*Lie a little*

- ConfigReg is a Register (Reg#(a))
  
  ```
  Reg#(t) full <- mkConfigRegU;
  ```

- Same HW as Register, but the definition says read and write can happen in either order
  - However, just like a HW register, a read after a write gets the old value

- Primarily used to fool the compiler analysis to do the right thing

---

One-Element Pipeline FIFO

*A correct solution*

```
module mkLFIFO1 (FIFO#(t));
    Reg#(t) data  <- mkRegU();
    Reg#(Bool) full <- mkConfigRegU(False);
    RWire#(void) deqEN <- mkRWire();
    Bool deqp = isValid (deqEN.wget()));
    method Action enq(t x) if (!full || deqp);
        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
```

No conflicts around `full`: when both `enq` and `deq` happen; if we want `deq < enq` then `full` must be set to `True` in case `enq` occurs.

Scheduling constraint on `deqEn forces deq < enq`

- `first < enq`
- `enq < clear`
- `deq < enq`
- `deq < clear`
An aside

Unsafe modules

- Bluespec allows you to import Verilog modules by identifying wires that correspond to methods.
- Such modules can be made safe either by asserting the correct scheduling properties of the methods or by wrapping the unsafe modules in appropriate Bluespec code.

*Config Reg is an example of an unsafe module.*

back to today’s lecture ...
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.

\[ \text{line rate } \Rightarrow 15\text{Mpps for 10GE} \]

### Sparse tree representation

<table>
<thead>
<tr>
<th>IP address</th>
<th>Result</th>
<th>M Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.13.7.3</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>E</td>
<td>1</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>

In this lecture:
- Level 1: 16 bits
- Level 2: 8 bits
- Level 3: 8 bits

\[ \Rightarrow 1 \text{ to } 3 \text{ memory accesses} \]
"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 */
}
```

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

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

Memory latency ~30ns to 40ns

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:
1. Inefficient memory usage
2. Efficient memory usage
3. Efficient memory usage

Which is "best"?
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

```vhdl
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
```

Request-Response Interface for Synchronous Memory

interface Mem#(type addrT, type dataT);
    method Action req(addrT x);
    method Action deq();
    method dataT peek();
endinterface

Making a synchronous component latency-insensitive
Circular Pipeline Code:

discussion

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

When can recirculate fire?

```plaintext
ram & fifo each has an element and ram, fifo & outQ each has space
```

Is this possible?

```plaintext
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
```

Ordinary FIFO won’t work but a pipeline FIFO would
Problem solved!

```verilog
PipelineFIFO fifo <- mkPipelineFIFO;
    // use a Pipeline 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 Pipeline FIFO – users of the fifo need not be aware of RWires

Dead cycles

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

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

```verilog
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
```

Is this worth worrying about?
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?

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?
    - *descending urgency = “recirculate, enter”*

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

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

Rule Splitting

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

```plaintext
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.

Packaging a module:
Turning a rule into a method

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