IP Lookup: Some subtle concurrency issues

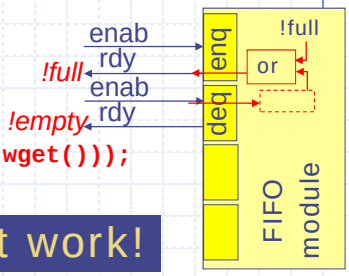
Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

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
  endmeth
  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 actually won't work!



This works correctly in both cases (fifo full and fifo empty).

first < enq
deq < enq

enq < clear
deq < clear

One-Element Pipeline FIFO

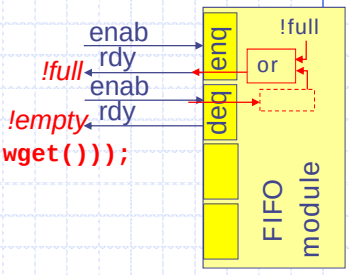
Analysis

```
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
```

Conflict analysis:



Rwire allows us to create a combinational path between enq and deq but does not affect the conflict analysis

Solution- Config registers

Lie a little

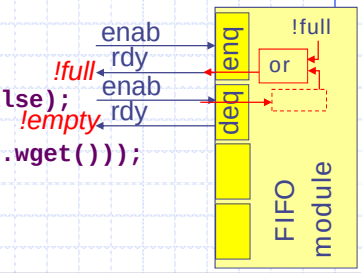
- ConfigReg is a Register (Reg#(a))

  `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  <- mkConfigReg(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
deq < enq

enq < clear
deq < clear

An aside

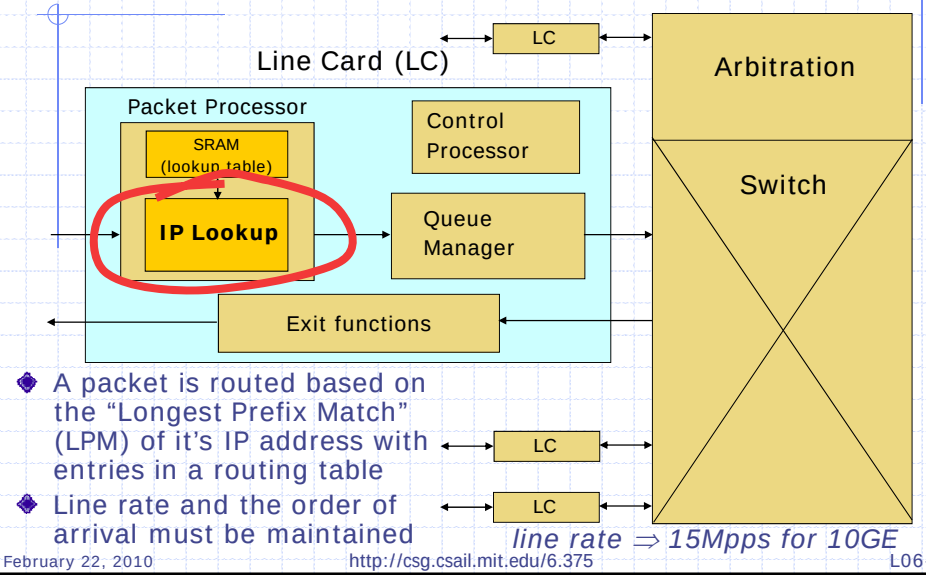
Unsafe modules

- 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 …

IP Lookup block in a router

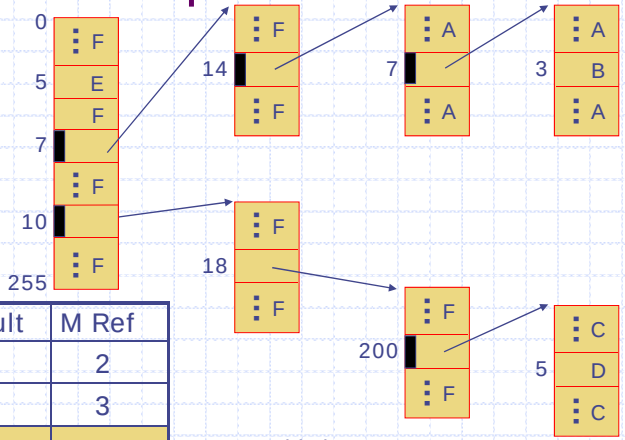


- A packet is routed based on the "Longest Prefix Match" (LPM) of it's IP address with entries in a routing table
- Line rate and the order of arrival must be maintained

line rate ⇒ 15Mpps for 10GE

Sparse tree representation

7.14.*.*	A
7.14.7.3	B
10.18.200.*	C
10.18.200.5	D
5.*.*.*	E
*	F



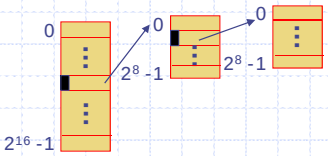
IP address	Result	M Ref
7.13.7.3	F	2
10.18.201.5	F	3
7.14.7.2		
5.13.7.2	E	1
10.18.200.7	C	4

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

⇒ 1 to 3 memory accesses

"C" version of 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 */
}
```



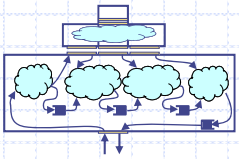
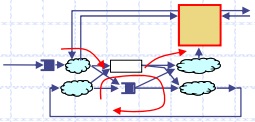
Not obvious from the C code how to deal with
- memory latency
- pipelining

Memory latency ~30ns to 40ns

Must process a packet every 1/15 μs or 67 ns

Must sustain 3 memory dependent lookups in 67 ns

Longest Prefix Match for IP lookup: 3 possible implementation architectures

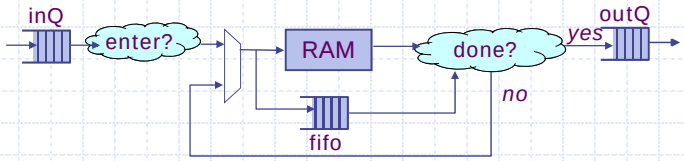
Rigid pipeline	Linear pipeline	Circular pipeline
		
Inefficient memory usage but simple design	Efficient memory usage through memory port replicator	Efficient memory with most complex control
1	2	3

Designer's Ranking:

Which is "best"?

Arvind, Nikhil, Rosenband & Dave, ICCAD 2004

Circular pipeline



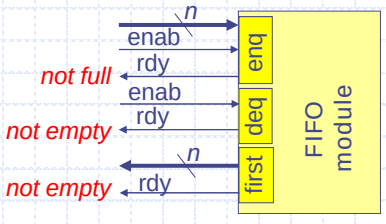
The fifo holds the request while the memory access is in progress

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.

Next lecture

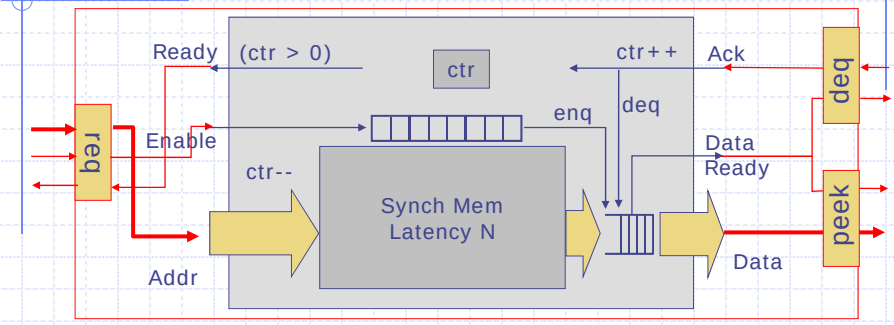
FIFO

```
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

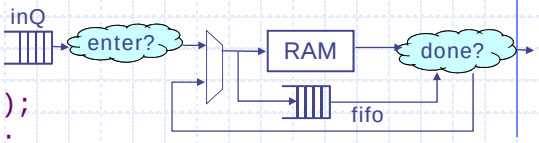
Request-Response Interface for Synchronous Memory




```
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




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

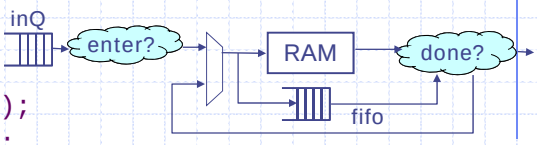
done? Is the same as isLeaf

When can enter fire?

inQ has an element and ram & fifo each has space

```
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
```

Circular Pipeline Code: *discussion*




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

When can recirculate fire?

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

Is this possible?

```
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
```

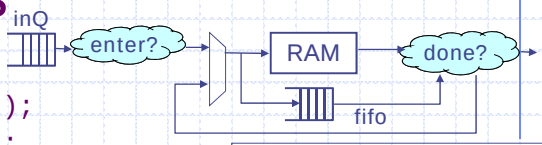

Ordinary FIFO won't work but a pipeline FIFO would

Problem solved!

```
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




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

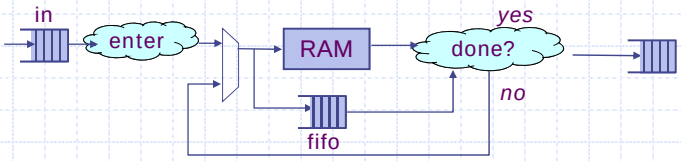
assume simultaneous enq & deq is allowed

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

Is this worth worrying about?

```
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?

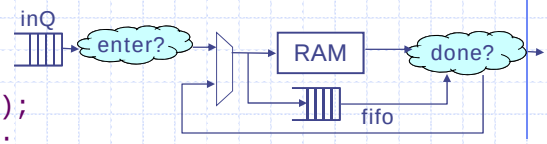
Scheduling conflicting rules

- 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?




```
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 Spliting

```
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

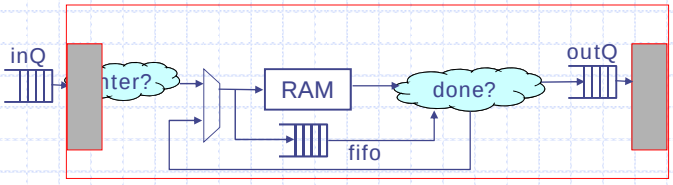
```
rule recirculate (!isLeaf(ram.peek()));
   IP rip = fifo.first(); fifo.enq(rip << 8);
   ram.req(ram.peek() + rip[15:8]);
   fifo.deq(); ram.deq();
endrule
```

```
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




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