Bluespec-4
The IP Lookup Problem

Arvind
Laboratory for Computer Science
M.I.T.

January 14, 2003

http://www.csg.lcs.mit.edu/IAPBlue

The IP lookup problem

- An IP lookup table contains IP *prefixes* and associated data
- The problem: given an IP address, return the data associated with the *longest prefix match* ("LPM")

Example Table

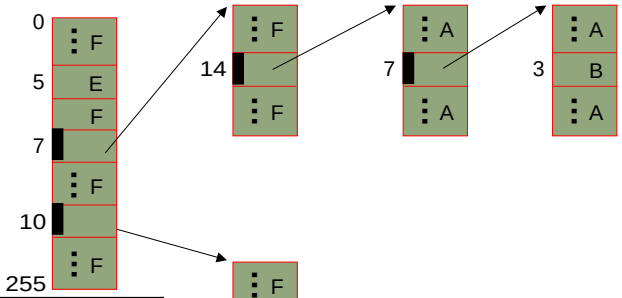
Prefix	*Data*
7.14.*.*	A
7.14.7.3	B
10.18.200.*	C
10.18.200.5	D
5.*.*.*	E
*	F

IP address	Result
7.13.7.3	F
10.7.12.15	F
10.18.201.5	F
7.14.7.2	A
5.13.7.2	E
8.0.0.0	F
10.18.200.7	C

Example lookups

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	A	4
5.13.7.2	E	1
10.18.200.7	C	4

Table representation issues

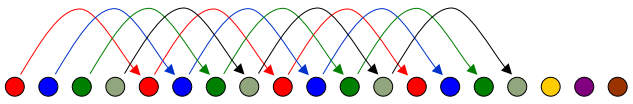
- LPM is used for CIDR (Classless Inter-Domain Routing)
- Number of memory accesses for an LPM?
  - Too many ➔ difficult to do table lookup at line rate (say at 10Gbps)
- Table size
  - Too big ➔ bigger SRAM ➔ more latency, cost, power
- Control-plane issues:
  - incremental table update
  - size, speed 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

Outline

- Example: IP Lookup √
- Three solutions ⇐
  - Statically scheduled memory pipeline
  - Straight pipeline with uncoordinated memory references
  - Circular pipeline for 100% memory utilization
- Modeling RAMs
  - Synchronous vs. Asynchronous view
- Bluespec coding for the circular pipeline
  - completion buffer

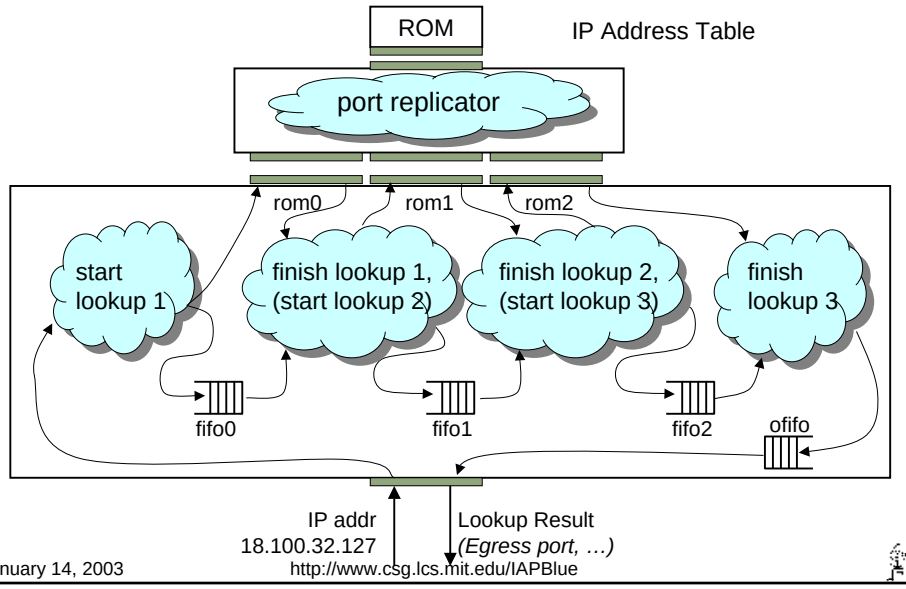
Static scheduling solution

- Assume the SRAM containing the table has latency of *n* cycles, lay out a pipeline so that the memory accesses are precisely scheduled

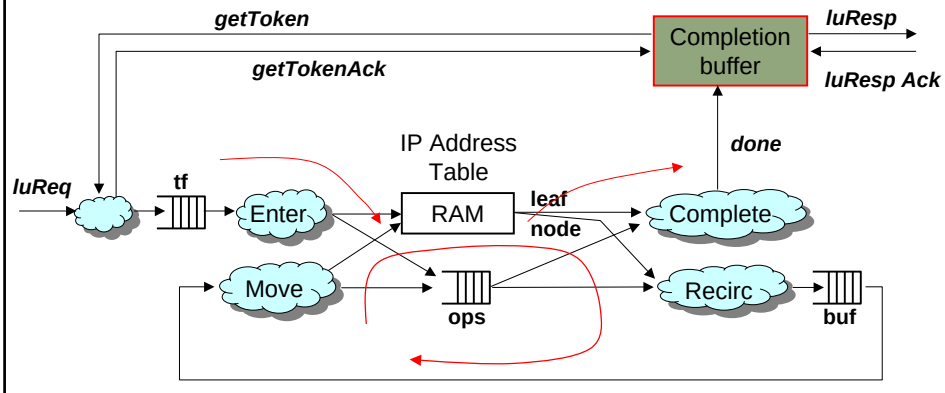


- Issues:
  - Since an LPM may take 1-3 mem accesses, unused slots may be left idle
  - May have to replan the pipeline for a different latency memory
  - Very difficult to plan if memory is also to be used for some unrelated task.

LPM: Straight Pipeline Solution



LMP: Circular pipeline solution



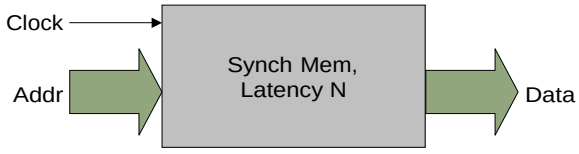
Completion buffer
- gives out tokens to control the entry into the circular pipeline
- ensures that departures take place in order even if lookups complete out-of-order

Outline

- Example: IP Lookup √
- Three solutions √
  - Statically scheduled memory pipeline
  - Straight pipeline with uncoordinated memory references
  - Circular pipeline for 100% memory utilization
- Modeling RAMs ⇐
  - Synchronous vs. Asynchonous view
- Bluespec coding for circular pipeline

RAMs

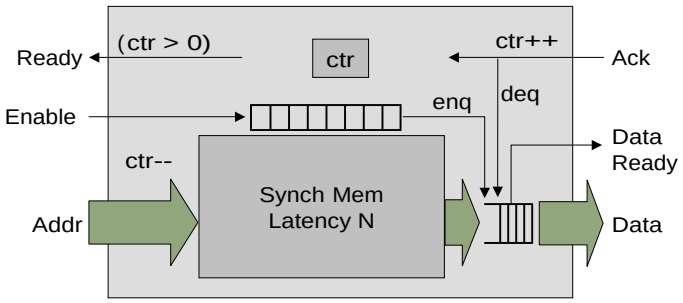
- Basic memory components are "synchronous":
  - Present a read-address A_J on clock J
  - Data D_J arrives on clock J+N
  - If you don't "catch" D_J on clock J+N, it may be lost, i.e., data D_{J+1} may arrive on clock J+1+N



- This kind of synchronicity can pervade the design and cause complications

Asynchronous RAMs

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



RAMs: Synchronous vs Asynchronous

- The async memory has interface:

```
interface AsyncRAM lat addr data =
    read   :: addr -> Action
    result :: data
    ack    :: Action
```

- A sync memory can be converted into an async memory with a Bluespec function:

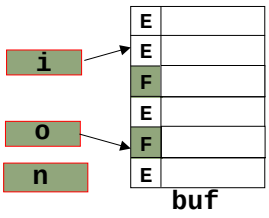
```
syncToAsync :: SyncRAM lat addr data ->
                              AsyncRAM lat+1 addr data
```


Outline

- Example: IP Lookup √
- Three solutions √
  - Statically scheduled memory pipeline
  - Straight pipeline with uncoordinated memory references
  - Circular pipeline for 100% memory utilization
- Modeling RAMs √
  - Synchronous vs. Asynchonous view
- Bluespec coding for circular pipeline ⇐

Completion buffer

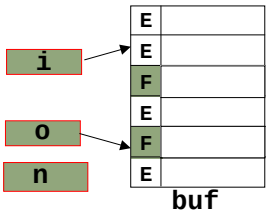
```
interface CBuffer n a =
    getToken    :: CBToken n
    getTokenAck :: Action
    done        :: CBToken n ->
                   a -> Action
    get         :: a
    ack         :: Action
```



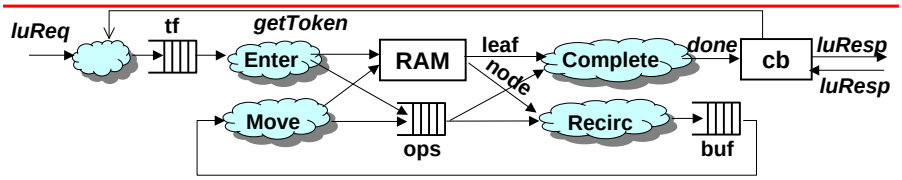

```
data CBToken n = CBToken (Bit (TLog n))

mkCBuffer =
  module
  buf :: Array (Bit ln) (Maybe a) <- mkArray 0 hi
    i :: Reg (Bit ln) <- mkReg 0      -- input index
    o :: Reg (Bit ln) <- mkReg 0      -- output index
    n :: Counter ln1  <- mkCounter 0 -- number of filled slots
```


Completion buffer

```
mkCBuffer :: (Log n ln, Add 1 ln ln1) => Module (CBuffer n a)
mkCBuffer =
  module ... state elements buf, i, o, n ...
    interface
      getToken = CBToken i
                   when n.value <= hi

      getTokenAck = action incr i
                           n.up
                           buf.upd i Nothing
                    when n.value <= hi

      done (CBToken t) a = buf.upd t (Just a)

      get = x  when n.value > 0, Just x <- buf.sub o

      ack = action  incr o
                    n.down
              when  n.value > 0, Just x <- buf.sub o
```

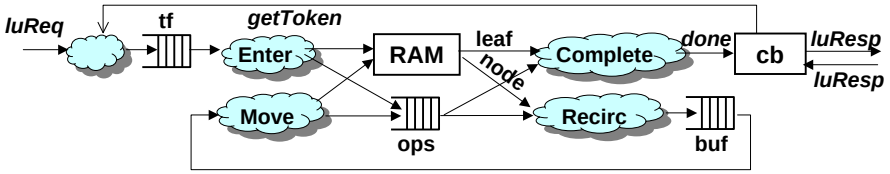


Bluespec code: Circular pipeline



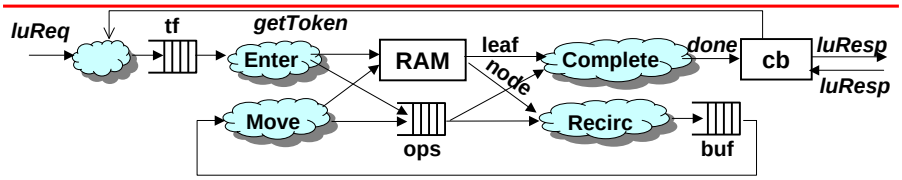

```
mkLPM :: AsyncRAM lat LuAddr LuData -> Module LPM
mkLPM ram =
  module
    cb  :: CBuffer NStg LuResult   <- mkCBuffer
    tf  :: FIFO ( CBToken, IPaddr) <- mkFIFO
    ops :: FIFO ( CBToken, IPaddr) <- mkSizedFIFO (lat+1)
    buf :: FIFO ((CBToken, IPaddr), LuAddr)
                                   <- mkSizedFIFO (NStg+1)
    rules ...
    interface
      luReq ipa = action tf.enq (cb.getToken, ipa)
                         cb.getTokenAck
      luResp    = cb.get
      luRespAck = cb.ack
```


Circular pipeline rules



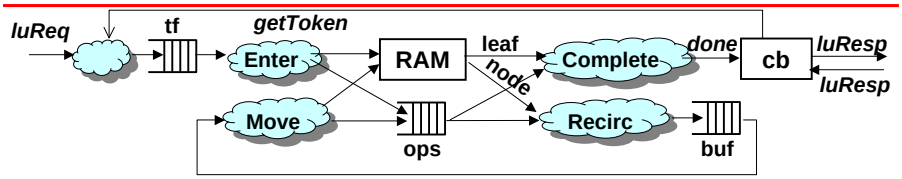

```
"Enter":
      when (tok, ipa) <- tf.first
       ==> action  tf.deq
                   ram.read (zeroExt ipa[31:16])
                   ops.enq (tok, ipa << 16)
```


Circular pipeline rules *Continued*




```
"Complete":
      when (Leaf res) <- ram.result,
           (tok, ipa) <- ops.first
       ==> action  ram.ack
                   ops.deq
                   cb.done tok res

"Recirculate":
      when (Node addr) <- ram.result,
           (tok, ipa)  <- ops.first
       ==> action  ram.ack
                   ops.deq
                   buf.enq ((tok, ipa << 8),
                            addr+(zeroExt ipa[31:24])
```


Circular pipeline rules *Continued-2*




```
"Move":
       when (tokipa, addr) <- buf.first
        ==> action  buf.deq
                    ram.read addr
                    ops.enq tokipa

"Enter":
       when (tok, ipa) <- tf.first
        ==> action  tf.deq
                    ram.read (zeroExt ipa[31:16])
                    ops.enq (tok, ipa << 16)
```

notice the conflict!

Some observations

- Rules conflicts should be carefully studied to make sure that there are no races
  - Atomic firing of rules is very helpful in reasoning about such matters
- Buffer sizes have to be set appropriately otherwise RAM may not be fully utilized
- Rule priorities have to be specified correctly to avoid deadlocks
- Timing Closure may require insertion of more pipeline stages (i.e. FIFO buffers)!
- Circular pipeline solution extends to IPv6 in a straightforward manner