

# Bluespec-3

## The IP Lookup Problem

Arvind  
Laboratory for Computer Science  
M.I.T.

Lecture 19

<http://www.csg.lcs.mit.edu/6.827>

## 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

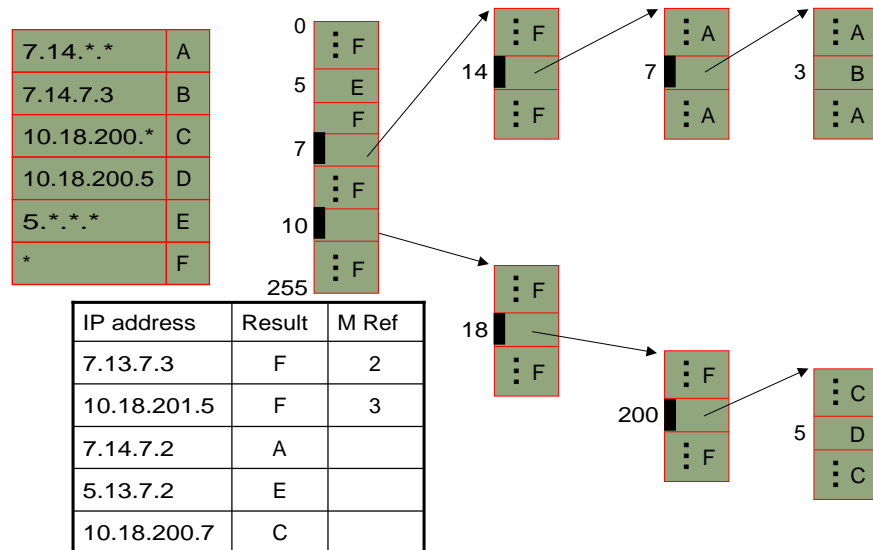
Prefix	Data
7.14.*.*	A
7.14.7.3	B
10.18.200.*	C
10.18.200.5	D
5.*.*.*	E
*	F

IP address	Result
7.13.7.3	F
10.7.12.15	F
10.18.201.5	F
7.14.7.2	
5.13.7.2	
8.0.0.0	
10.18.200.7	

Example  
lookups

<http://www.csg.lcs.mit.edu/6.827>

## Sparse tree representation



<http://www.csg.lcs.mit.edu/>



## Table representation issues

- LPM is used for CIDR (Classless Inter-Domain Routing)
- Number of memory accesses for an LPM?
  - Too many → difficult to do LPMs at line rate
- 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 (so code will fit on slides!):
  - Level 1: 16 bits, Levels 2 and 3: 8 bits
  - So: from 1 to 3 memory accesses for an LPM

<http://www.csg.lcs.mit.edu/6.827>



## 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
  - Port replicator
- Bluespec coding for straight pipeline
- Bluespec coding for circular pipeline
- Phase 1 compilation

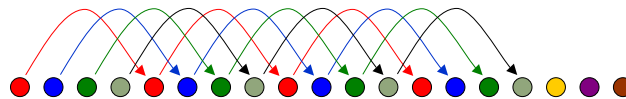
<http://www.csg.lcs.mit.edu/6.827>



## 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

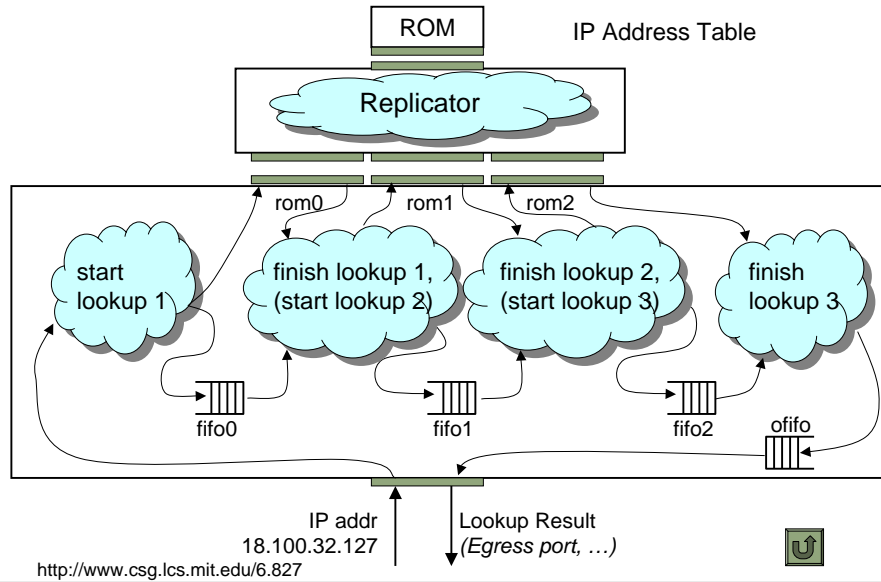


- Issues:
  - Since an LPM may take 1-3 mem accesses, unused slots may be left idle
  - May have to replan the pipeline for a different latency memory
  - Very difficult to plan if memory is also to be used for some unrelated task.

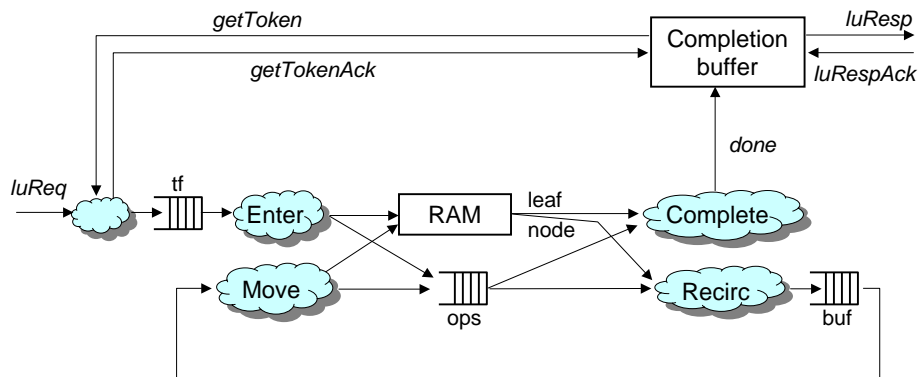
<http://www.csg.lcs.mit.edu/6.827>



## LPM: Straight Pipeline Solution



## Circular pipeline solution



## 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
  - Port replicator
- Bluespec coding for straight pipeline
- Bluespec coding for circular pipeline
- Phase 1 compilation

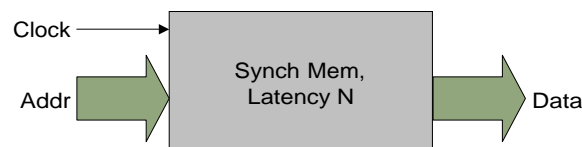
<http://www.csg.lcs.mit.edu/6.827>



## RAMs

---

- Basic memory components are "synchronous":
  - Present an 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  $J+1+N$



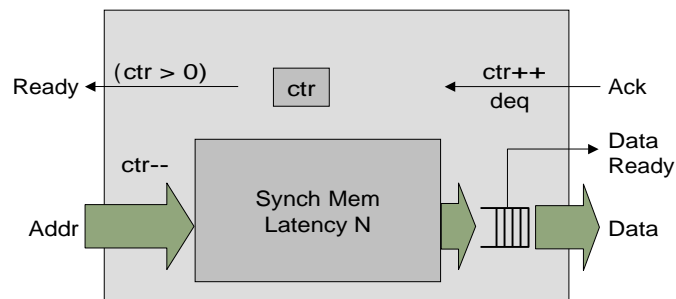
- This kind of synchronicity can pervade the design and cause complications

<http://www.csg.lcs.mit.edu/6.827>



## Asynchronous RAMs

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



<http://www.csg.lcs.mit.edu/6.827>

## RAMs: Synchronous vs Asynchronous

- The asynch mem has interface:

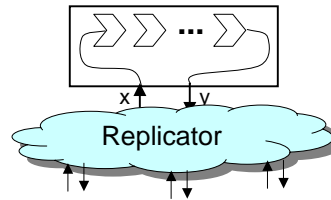
```
interface AsyncROM addr data =
  read  :: addr -> Action
  result :: data
  ack   :: Action
```

- A synch mem can be converted into an asynch mem with a Bluespec function:

```
syncToAsync :: SyncROM lat addr data ->
              AsyncROM addr data
```

<http://www.csg.lcs.mit.edu/6.827>

## A common subproblem: *port replicator*



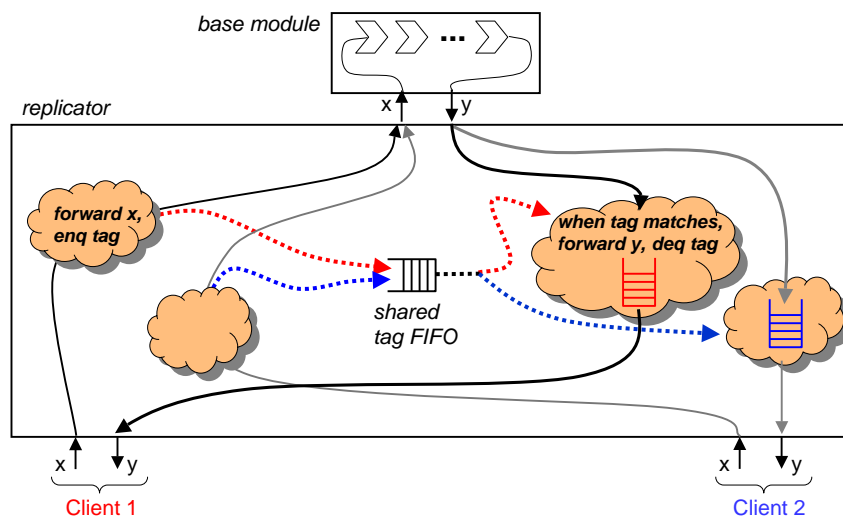
e.g., SRAM, IP lookup

- Given a *base module* s.t.:
  - It generates response  $y$  to a particular request  $x$  after some number of cycles
  - It can accept requests and produce responses on every cycle
- Construct a mechanism so that multiple *client* modules can utilize the *base module* with full utilization

<http://www.csg.lcs.mit.edu/6.827>



## A general port replicator



<http://www.csg.lcs.mit.edu/6.827>



## Port replicator code

---

```

mk3ROMports :: AsyncROM lat Adr Dta ->
  Module (AsyncROM (lat+1) Adr Dta,
          AsyncROM (lat+1) Adr Dta,
          AsyncROM (lat+1) Adr Dta)
mk3ROMports rom =
  module
    tags :: FIFO Tag <- mkSizedFIFO lat
    let mkPort :: Tag -> Module (AsyncROM (lat+1) Adr Dta)
        mkPort i =
          module
            ...

    port0 <- mkPort 0
    port1 <- mkPort 1
    port2 <- mkPort 2
    interface (port0, port1, port2)

```

*not quite legal*

<http://www.csg.lcs.mit.edu/6.827>



## Port replicator code *cont.*

---

```

mkPort :: Tag -> Module (AsyncROM (lat+1) Adr Dta)
mkPort i =
  module
    out :: FIFO Dta <- mkSizedFIFO lat
    rules
      when tags.first == i
        ==> action tags.deq
              rom.ack
              out.enq rom.result
    interface
      read a = action rom.read a
              tags.enq i
      result = out.first
      ack    = out.deq

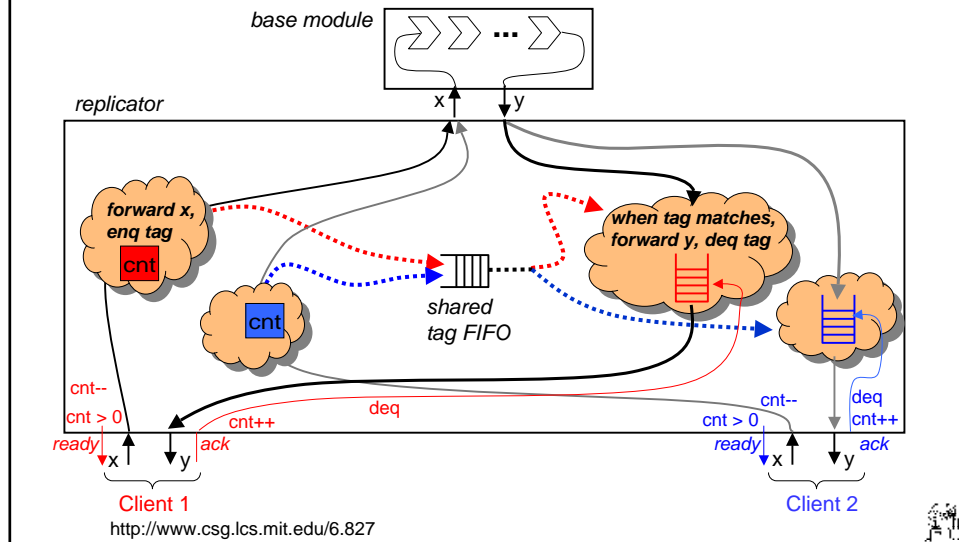
```

*Can one port's activity block another port's activity?*

<http://www.csg.lcs.mit.edu/6.827>



## A general port replicator *fixed*



## Port replicator code *fixed*

```

mkPort :: Tag -> Module (AsyncROM (lat+1) Adr Dta)
mkPort i =
  module
    out :: FIFO Dta <- mkSizedFIFO lat
    cnt :: Counter (log (lat+1)) <- mkCounter lat
    rules
      when tags.first == i
        ==> action tags.deq
              rom.ack
              out.enq rom.result
    interface
      read a = action rom.read a
                tags.enq i
                cnt.down
                when cnt.value > 0
      result = out.first
      ack    = action out.deq
                cnt.up
  
```

<http://www.csg.lcs.mit.edu/6.827>

## Some observations

---

- Port replicator can be easily generalized to n ports
- Port rules conflict with each other but serializable semantics preserves correctness



## 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
  - Port replicator
- Bluespec coding for straight pipeline ⇐
- Bluespec coding for circular pipeline
- Phase 1 compilation



## Bluespec code: Straight pipeline



```

data Mid = Lookup IPaddr | Done LuResult

mkLPM :: AsyncROM lat LuAddr LuData -> Module LPM
mkLPM rom =
  module
    (rom0, rom1, rom2) <- mk3ROMports rom

    fifo0 :: FIFO Mid <- mkFIFO
    fifo1 :: FIFO Mid <- mkFIFO
    fifo2 :: FIFO Mid <- mkFIFO
    ofifo :: FIFO LuResult <- mkFIFO

  rules

    ... for Stages 1, 2 and Completion ...

  interface
    -- Stage 0
    luReq ipa = action rom0.read (zeroExtend ipa[31:16])
                fifo0.enq (Lookup (ipa << 16))
    luResp    = ofifo.first
    luRespAck = ofifo.deq
  
```

<http://www.csg.lcs.mit.edu/6.827>



## Straight pipeline *cont.*

```

data Mid = Lookup IPaddr | Done LuResult
mkLPM rom =
  module
    ... state is rom0, rom1, rom2, fifo0, fifo1, fifo2, ofifo
  rules
    -- Stage 1: lookup, leaf
    when Lookup ipa <- fifo0.first,
      Leaf res <- rom0.result
      ==> action fifo0.deq
            rom0.ack
            fifo1.enq (Done res)

    -- Stage 1: lookup, node
    when Lookup ipa <- fifo0.first,
      Node res <- rom0.result
      ==> action fifo0.deq
            rom0.ack
            rom1.read (addr+(zeroExt ipa[31:24]))
            fifo1.enq (Lookup (ipa << 8))

  interface
  
```

<http://www.csg.lcs.mit.edu/6.827>



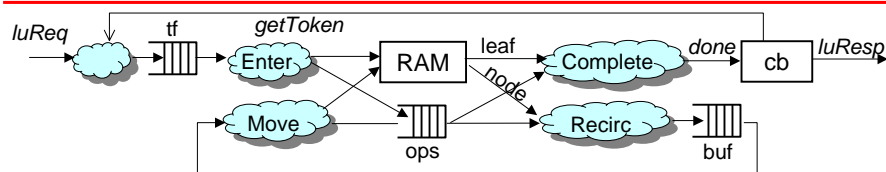
## Outline

- 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
  - Port replicator
- Bluespec coding for straight pipeline ✓
- Bluespec coding for circular pipeline ⇐
- Phase 1 compilation

<http://www.csg.lcs.mit.edu/6.827>



## Bluespec code: Circular pipeline



```

mkLPMK :: AsyncROM lat LuAddr LuData -> Module LPM
mkLPMK rom =
  module
    cb :: CBuffer NStage LuResult <- mkCBuffer
    tf :: FIFO (CBToken NStage, IPaddr) <- mkFIFO
    ops :: FIFO (CBToken NStage, IPaddr) <- mkSizedFIFO (lat+1)
    buf :: FIFO ((CBToken NStage, IPaddr), LuAddr)
          <- mkSizedFIFO (NStage+1)

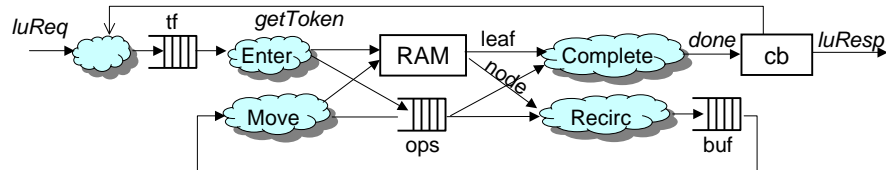
    rules
      ... rules for Entering, Completion, Recirculation & Movement...
    interface
      luReq ipa = action tf.enq (cb.getToken, ipa)
                  cb.getTokenAck

      luResp    = cb.get
      luRespAck = cb.ack
  
```

<http://www.csg.lcs.mit.edu/6.827>



## Circular pipeline rules



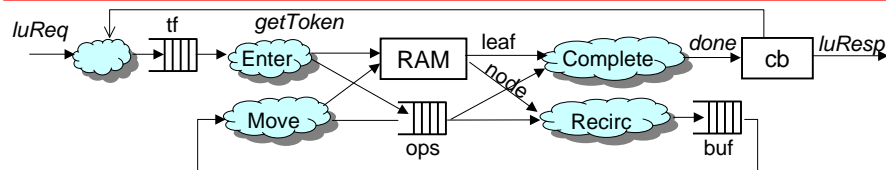
```

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

<http://www.csg.lcs.mit.edu/6.827>



## Circular pipeline rules *Continued*



```

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

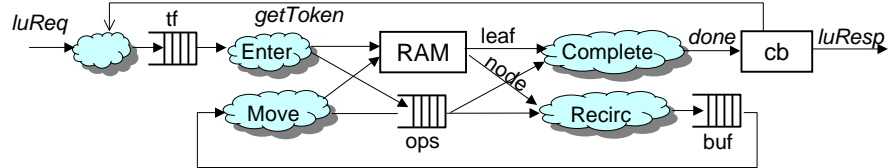
```

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

<http://www.csg.lcs.mit.edu/6.827>



## Circular pipeline rules *Continued-2*



"Move":

```
when (tokipa, addr) <- buf.first
  ==> action
    buf.deq
    rom.read addr
    ops.enq tokipa
```

"Enter":

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

notice the  
conflicts!

<http://www.csg.lcs.mit.edu/6.827>



## Some observations

- Timing Closure
  - Insert more pipeline stages (i.e. FIFO buffers)!
- Circular pipeline solution trivially extends to IPv6

<http://www.csg.lcs.mit.edu/6.827>



## 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
  - Port replicator
- Bluespec coding for straight pipeline ✓
- Bluespec coding for circular pipeline ✓
- Phase 1 compilation ⇐

<http://www.csg.lcs.mit.edu/6.827>



## LPM code structure

---

```

mkLPM rom =
  module
    (rom0, rom1, rom2) <- mk3ROMports rom
    fifo0 <- mkFIFO
    fifo1 <- mkFIFO
    fifo2 <- mkFIFO
    ofifo <- mkFIFO
    rules
      RuleStage1Leaf(fifo0, fifo1, rom0)
      RuleStage1Node(fifo0, fifo1, rom0, rom1)
      RuleStage2Noop(fifo1, fifo2)
      RuleStage2Leaf(fifo1, fifo2, rom1)
      RuleStage2Node(fifo1, fifo2, rom1, rom2)
      RuleCompletionNoop(fifo2, ofifo)
      RuleCompletionLeaf(fifo2, ofifo, rom2)
      RuleCompletionNode(fifo2, ofifo, rom2)
    interface
      luReq = EluReq(fifo0, rom0)
      luResp = EluResp(ofifo)
      luRespAck = EluRespAck(ofifo)
  
```

Free variables of the rule

<http://www.csg.lcs.mit.edu/6.827>



## Port replicator code structure

```

mk3ROMports rom =
  module
    tags <- mkSizedFIFO
    let
      mkPort i =
        module
          out <- mkSizedFIFO
          cnt <- mkCounter
          rules
            RuleTags(i, rom, tags, out)
          interface
            read = Eread(i, rom, tags, cnt)
            result = Eresult(out)
            ack = Eack(out, cnt)
        port0 <- mkPort 0
        port1 <- mkPort 1
        port2 <- mkPort 2
      interface (port0, port1, port2)

```

← substitute

<http://www.csg.lcs.mit.edu/6.827>



## Port replicator – after step 1

Step 2:  
Flatten  
the  
module

```

mk3ROMports rom =
  module
    tags <- mkSizedFIFO
    port0 <-
      module
        out <- mkSizedFIFO
        cnt <- mkCounter
        rules
          RuleTags(0, rom, tags, out)
        interface
          read = Eread(0, rom, tags, cnt)
          result = Eresult(out)
          ack = Eack(out, cnt)
      port1 <- ...similarly...
      port2 <- ...similarly...
    interface (port0, port1, port2)

```

<http://www.csg.lcs.mit.edu/6.827>



## Port replicator – after step 2

```

mk3ROMports rom =
module
  tags <- mkSizedFIFO

  out0 <- mkSizedFIFO
  cnt0 <- mkCounter
  rules
    RuleTags(0, rom, tags, out0)
  let port0 = interface
    read = Eread(0, rom, tags, cnt0)
    result = Eresult(out0)
    ack = Eack(out0, cnt0)

  port1 <- ...similarly...
  port2 <- ...similarly...
  interface (port0, port1, port2)

```

<http://www.csg.lcs.mit.edu/6.827>



## Port replicator – final step

```

mk3ROMports rom =
module
  tags <- mkSizedFIFO
  out0 <- mkSizedFIFO ; cnt0 <- mkCounter
  out1 <- mkSizedFIFO ; cnt1 <- mkCounter
  out2 <- mkSizedFIFO ; cnt2 <- mkCounter
  rules
    RuleTags(0, rom, tags, out0)
    RuleTags(1, rom, tags, out1)
    RuleTags(2, rom, tags, out2)
  let port0 = interface
    read = Eread(0, rom, tags, cnt0)
    result = Eresult(out0)
    ack = Eack(out0, cnt0)
  port1 = interface
    read = Eread(1, rom, tags, cnt1)
    ...
  port2 = interface ...
  interface (port0, port1, port2)

```

Next step:  
substitute  
mk3ROMports  
into mkLPM

<http://www.csg.lcs.mit.edu/6.827>



## Port replicator call

```

(rom0, rom1, rom2) <- mk3ROMports rom

tags <- mkSizedFIFO
out0 <- mkSizedFIFO ; cnt0 <- mkCounter
out1 <- mkSizedFIFO ; cnt1 <- mkCounter
out2 <- mkSizedFIFO ; cnt2 <- mkCounter
rules
  RuleTags(0, rom, tags, out0)
  RuleTags(1, rom, tags, out1)
  RuleTags(2, rom, tags, out2)
let port0 = interface
  read = Eread(0, rom, tags, cnt0)
  result = Eresult(out0)
  ack = Eack(out0, cnt0)
  port1 = interface ...
  port2 = interface ...

(rom0, rom1, rom2) = (port0, port1, port2)

```

substitutue  
for ports  
next

<http://www.csg.lcs.mit.edu/6.827>



## After Port replicator call substitution

```

(rom0, rom1, rom2) <- mk3ROMports rom

```

```

tags <- mkSizedFIFO
out0 <- mkSizedFIFO ; cnt0 <- mkCounter
out1 <- mkSizedFIFO ; cnt1 <- mkCounter
out2 <- mkSizedFIFO ; cnt2 <- mkCounter
rules
  RuleTags(0, rom, tags, out0)
  RuleTags(1, rom, tags, out1)
  RuleTags(2, rom, tags, out2)
let rom0 = interface
  read = Eread(0, rom, tags, cnt0)
  result = Eresult(out0)
  ack = Eack(out0, cnt0)
  rom1 = interface ...
  rom2 = interface ...

```

<http://www.csg.lcs.mit.edu/6.827>



## LPM code after flattening

---

```
mkLPM rom =
  module
    tags <- mkSizedFIFO;
    out0 <- mkSizedFIFO; cnt0 <- mkCounter;
    out1 <- mkSizedFIFO; cnt1 <- mkCounter;
    out2 <- mkSizedFIFO; cnt2 <- mkCounter;
    fifo0 <- mkFIFO; fifo1 <- mkFIFO; fifo2 <- mkFIFO;
    ofifo <- mkFIFO;
    rules
      RuleTags(0, rom, tags, out0)...
    let rom0 = interface
      read = Eread(0, rom, tags, cnt0)
      result = Eresult(out0)
      ack = Eack(out0, cnt0)
      rom1 = interface ... ; rom2 = interface ...
      RuleStageLeaf(fifo0, fifo1, rom0)...
    interface
      luReq = EluReq(fifo0, rom0)
      luResp = EluResp(ofifo)
      luRespAck = EluRespAck(ofifo)
```

<http://www.csg.cs.mit.edu/6.827>

