

# Why formal verification remains on the fringes of commercial development

Arvind

Computer Science & Artificial Intelligence Laboratory  
Massachusetts Institute of Technology

*Formal Methods, Turku, Finland*

May 28, 2008

With help from  
Nirav Dave &  
Mike Katelman

# A designer's perspective

- ◆ The goal is to design systems that meet some criteria such as cost, performance, power, compatibility, robustness, ...
- ◆ The design effort and the time-to-market matter (\$\$\$)

Can formal methods help?

# My Viewpoint

- ◆ The degree of correctness required depends upon the application

- Different applications require use of different formal

- ◆ For design verification at the system-level

The real success of a formal technique is when it is used ubiquitously without the designer being aware of it

- ◆ For design verification at the component-level, e.g., type systems

- A designer is unlikely to do any thing for the sake of helping the post design verification

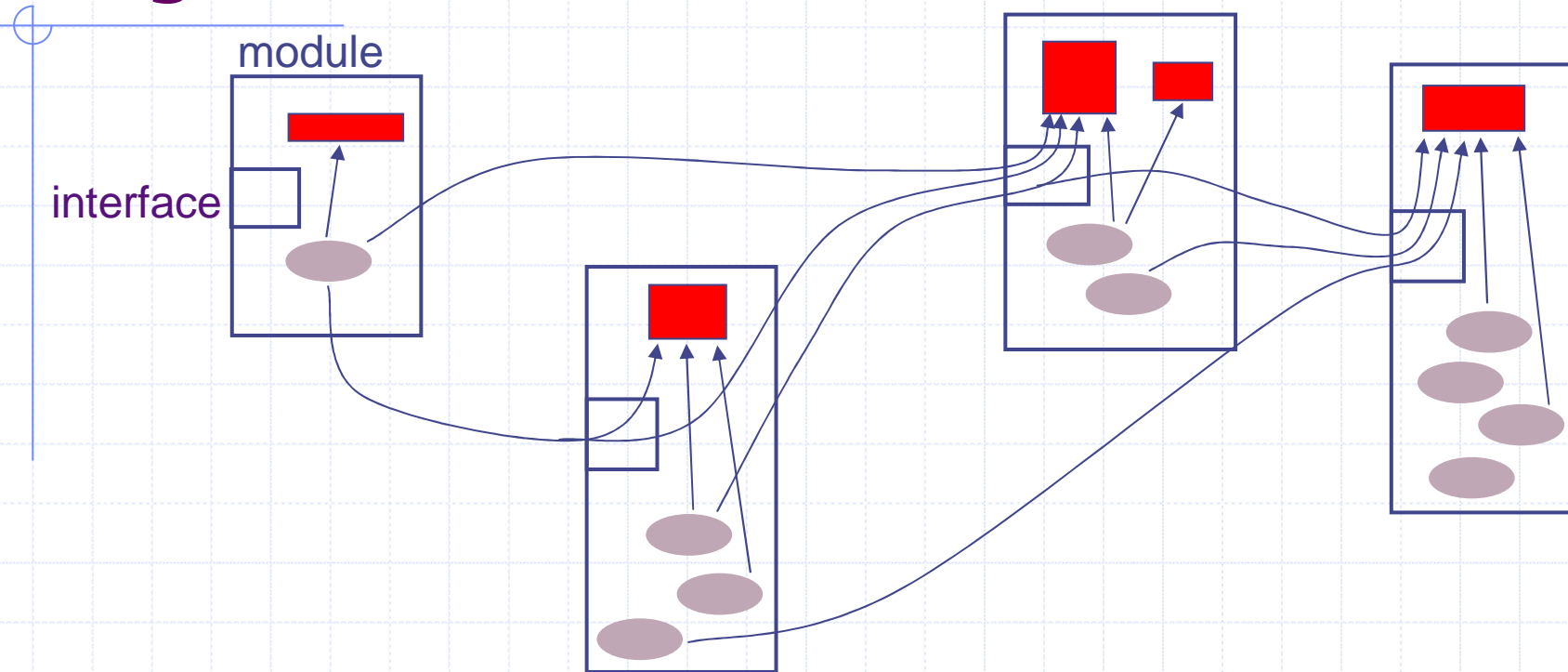
# This talk will show via examples

- ◆ How the correctness requirements differ vastly for different applications
- ◆ How Bluespec helps the designers in each case



# What is Bluespec?

# Bluespec: State and Rules organized into *modules*



All *state* (e.g., Registers, FIFOs, RAMs, ...) is explicit.  
*Behavior* is expressed in terms of guarded atomic actions on the state:      Rule: guard  $\rightarrow$  action  
Rules can manipulate state in other modules only *via* their interfaces.

# GAA Execution model

*Repeatedly:*

- Select a rule to execute
- Compute the state updates
- Make the state updates

Highly non-deterministic

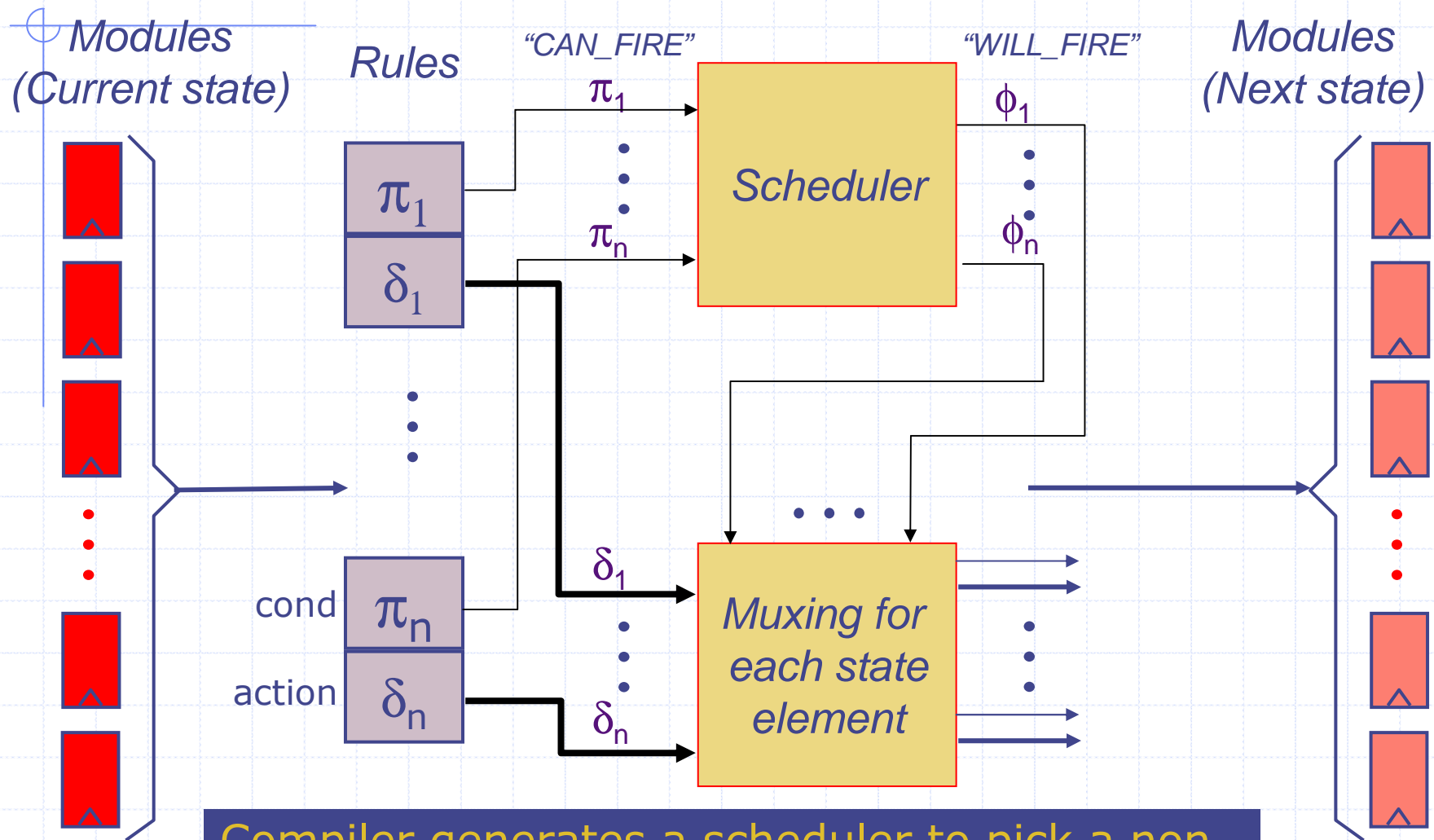
Hardware Implementation: We need to restrict this behavior to be deterministic

To get “good” hardware, we must schedule multiple rules in a cycle concurrently, while preserving the one-rule-at-a-time semantics

Unity [Chandy & Misra]

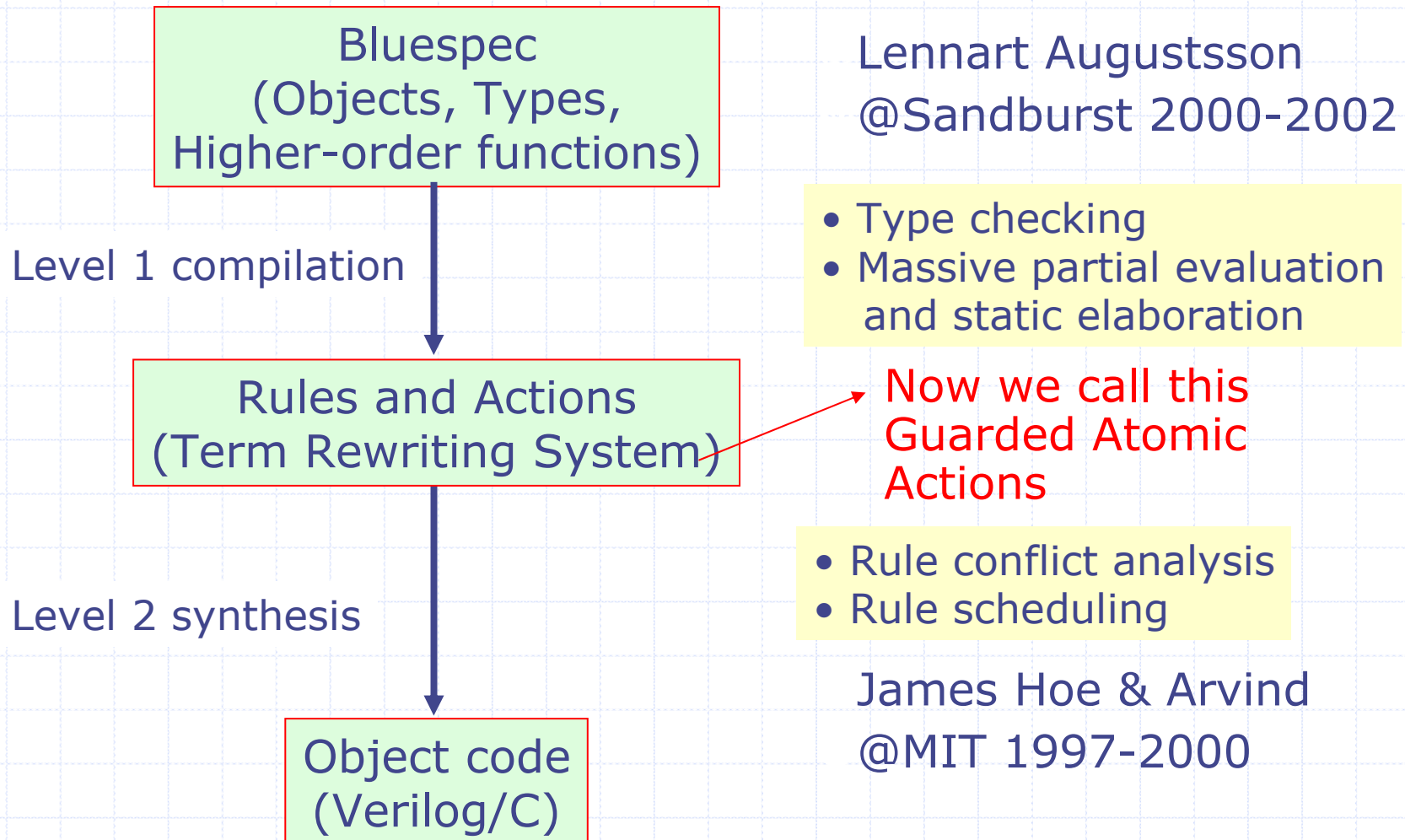
# Compiler model:

*Every rule executes in one clock cycle*



Compiler generates a scheduler to pick a non-conflicting subset of "ready" rules

# Bluespec: Two-Level Compilation

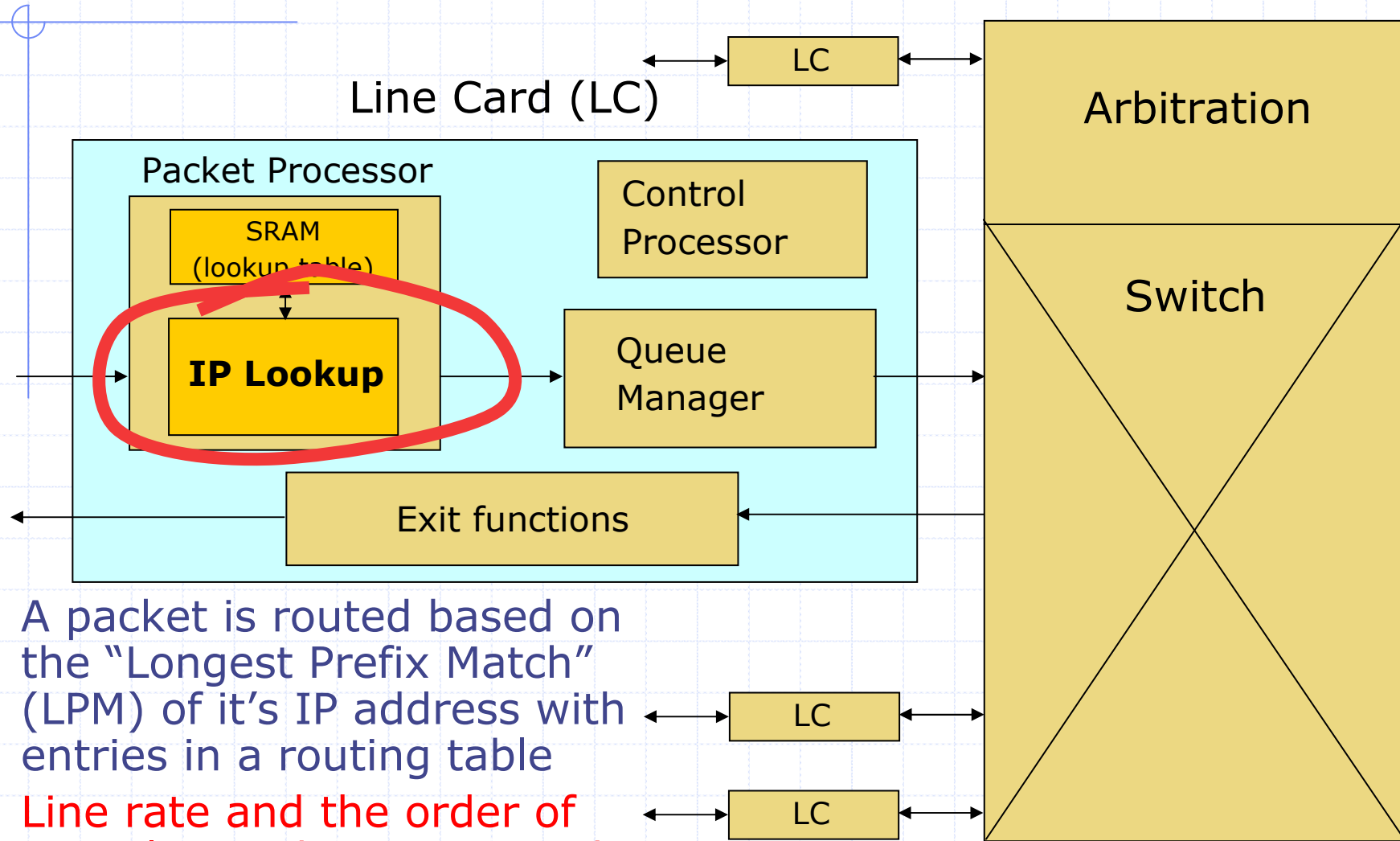




## Example 1: IP Lookup in a router

Simple, deterministic functionality  
but extreme performance  
requirement

# Internet router

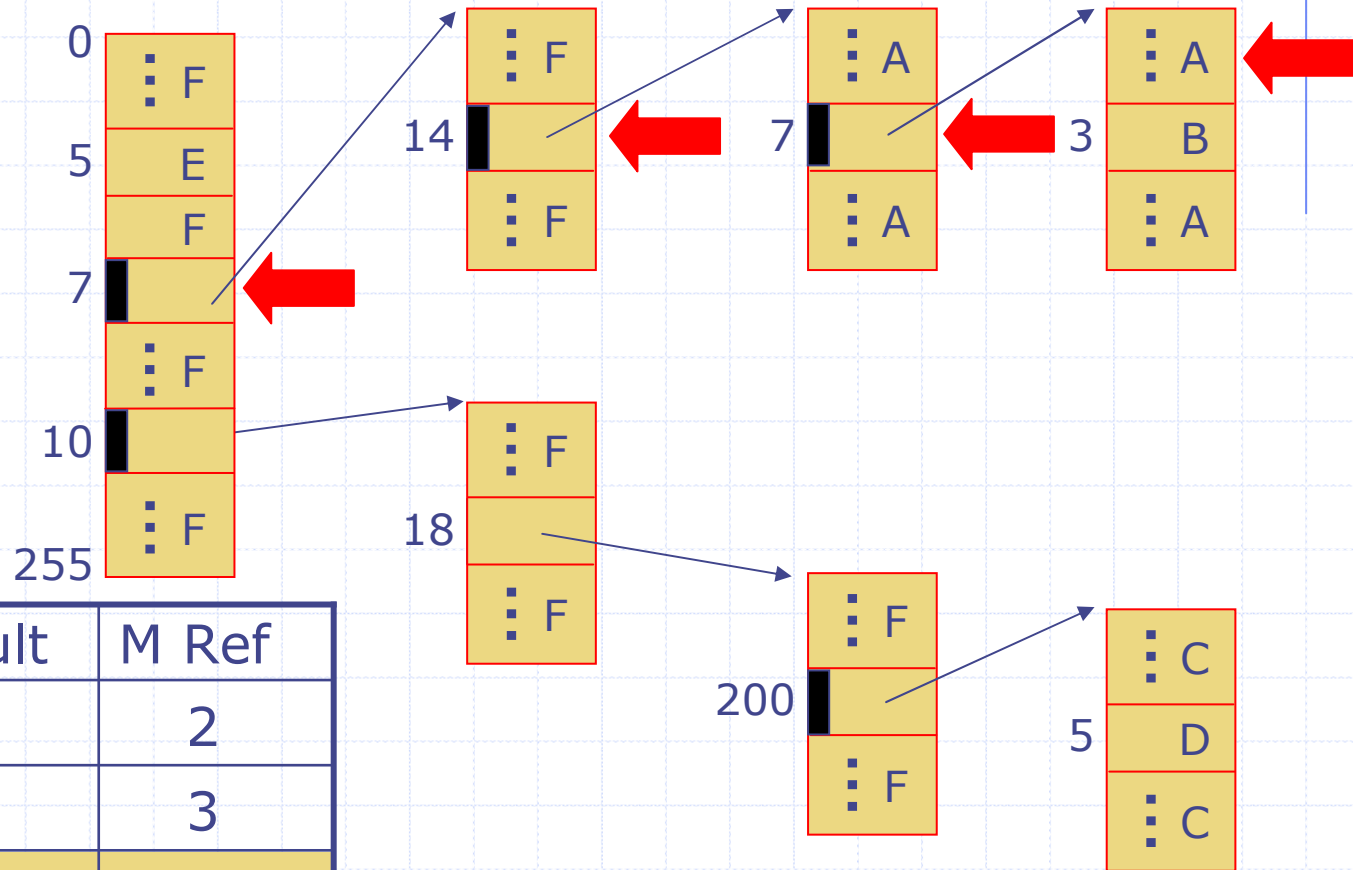


- ◆ A packet is routed based on the “Longest Prefix Match” (LPM) of it’s IP address with entries in a routing table
- ◆ **Line rate and the order of arrival must be maintained**

*line rate*  $\Rightarrow$  15Mpps for 10GE

# Routing table: Sparse tree representation

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

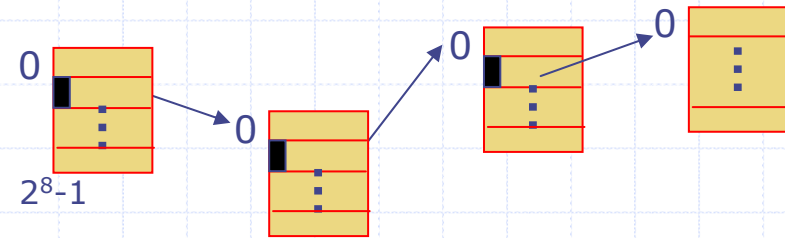


IP address	Result	M Ref
7.13.7.3	F	2
10.18.201.5	F	3
7.14.7.2	A	4
5.13.7.2	E	1
10.18.200.7	C	4

⇒ 1 to 4 memory accesses

# "C" version of LPM

```
int  
lpm (IPA ipa)  
/* 3 memory lookups */  
{ int p;  
  /* Level 0: 8 bits */  
  p = RAM [ipa[31:24]];  
  if (isLeaf(p)) return value(p);  
  /* Level 1: 8 bits */  
  p = RAM [ipa[23:16]];  
  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

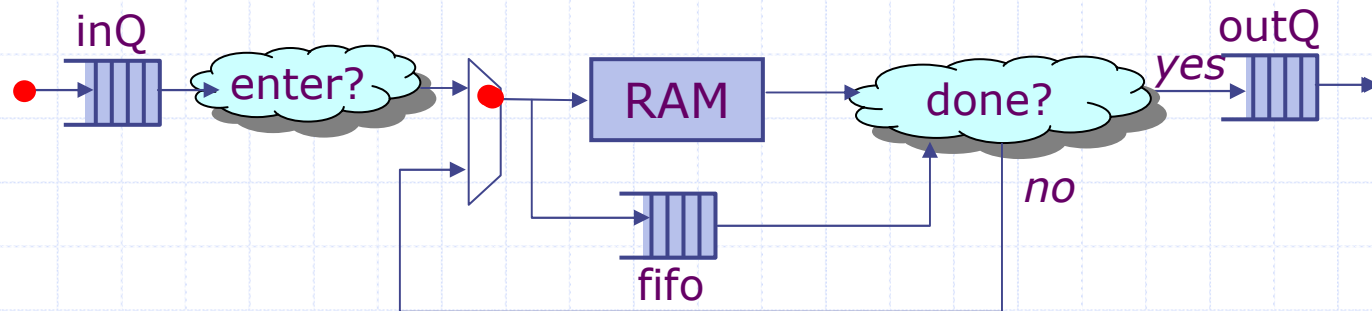
Must process a packet every  $1/15 \mu\text{s}$  or 67 ns

Must sustain 4 memory dependent lookups in 67 ns

Memory latency  
~30ns to 40ns

Real LPM algorithms are more complex

# An implementation: Circular pipeline



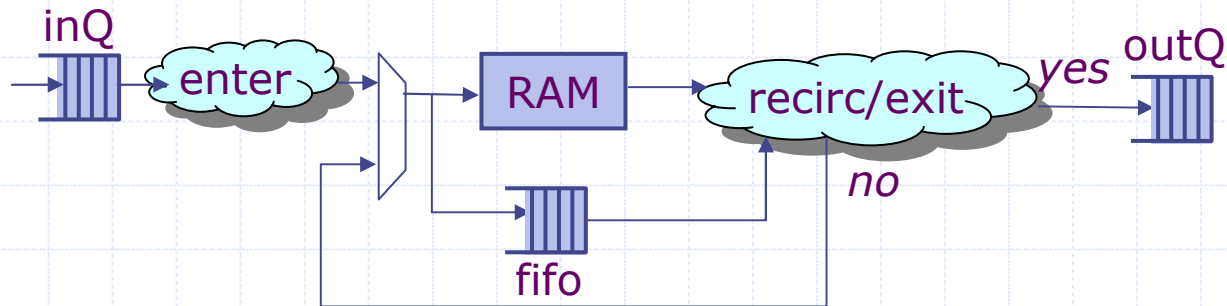
The fifo holds the request while the memory access is in progress

in order

# Is this design correct ?

- ◆ Does the look up produce the right answer?
  - Easy: check it against the C program
- ◆ Do answers come out in the right order?
  - Difficult to state formally for designers
    - ◆ Is it even possible in a given logic?
  - Alternative: The designer tags input messages and checks that the tags are produced in order
    - ◆ Challenge: verify this condition statically – can be done by someone other than the designer!
    - ◆ Code for tagging can be “erased” at any stage

# Circular pipeline: A performance issue – Dead-cycles



```
rule "enter" y = inQ.first() in  
    (inQ.deq() | mem.req(addr(y)) | fifo.enq(rest(y)))
```

```
rule "recirculate" x = mem.res() in y = fifo.first() in  
    if (!done? (x)) then ((mem.deq() ; mem.req(addr(x,y))) |  
        (fifo.deq() ; fifo.enq(shift(y))))  
    else (fifo.deq() | mem.deq() | outQ.enq(x))
```

Most performance concerns eventually  
become correctness concerns

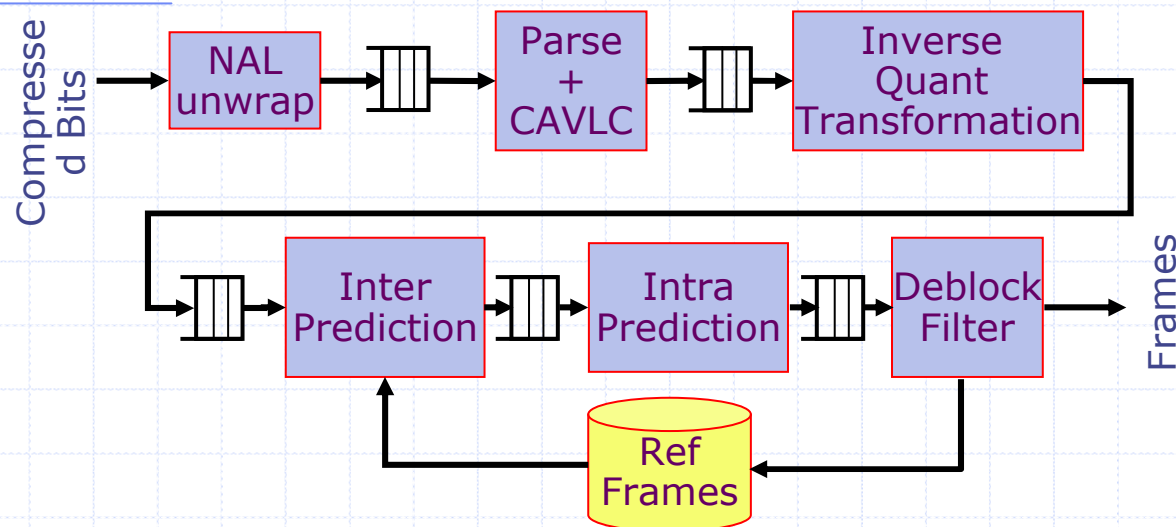
>25% slowdown!

Example 3:

## Lossy compression – H.264

The standard only specifies decoding; encoder's goal is to compress as much as possible without loss of perceptual fidelity

# H.264 Video Decoder

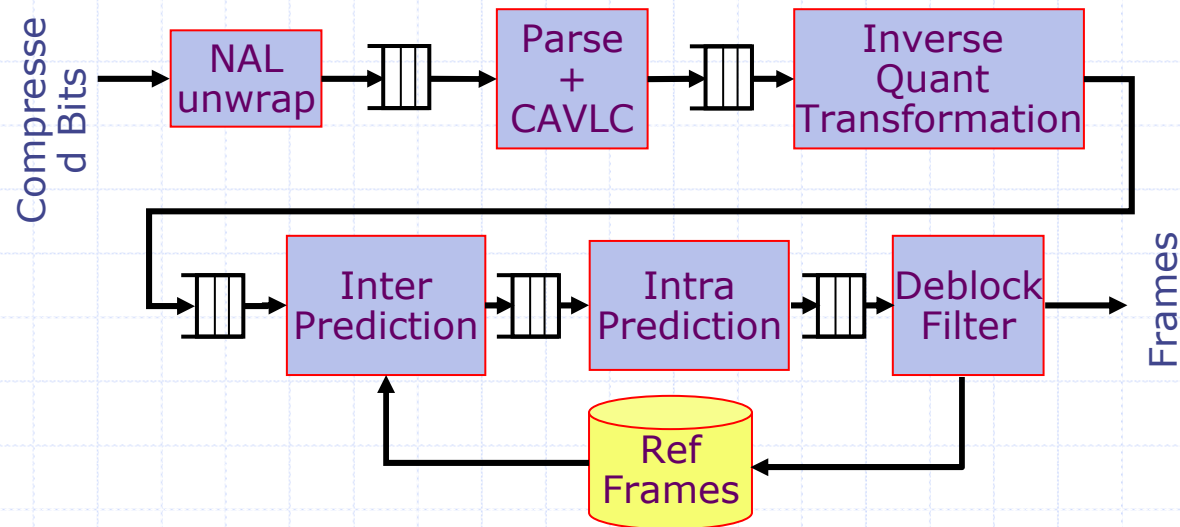


Errors don't matter much



- ◆ The standard is 400+ pages of English; the standard implementation is 80K lines of convoluted C. Each is incomplete!
- ◆ Only viable correctness criterion is bit-level matching against the reference implementation on sample videos
- ◆ Parallelization is more complicated than what one may guess based on the dataflow diagram because of data-dependencies and feedback

# H.264 Decoder: Implementation



- ◆ Different requirements for different environments
  - QVGA 320x240p (30 fps)
  - DVD 720x480p
  - HD DVD 1280x720p (60-75 fps)
- ◆ Each context requires a different amount of parallelism in different blocks
  - Modular refinement is necessary
  - Verifying the correctness of refinements requires traditional formal techniques (pipeline abstraction, etc.)

# H.264: Bluespec Implementations

- ◆ Productivity: Base profile
  - Effort: Less than one-man year
  - 8K lines of Bluespec (contrast 20k to 80K lines of C)
  - First draft decoded 720p @ ~32fps, (Available C codes do not meet this performance)
- ◆ Architectural Exploration: Many improvements made over a period of several months to increase performance and reduce area
  - Process several samples / cycle
  - Adjust FIFO depths
  - Pipeline modules: Interpolator, Deblocking filter
  - After improvements decodes 720p@ ~95fps and 1080p@ 70fps
  - Area 4.4 mm sq (180nm)

**Modular refinement is both feasible and essential**

# H.264 Takeaway

- ◆ Like most algorithmic designs bit-level matching against the C reference is totally acceptable
  - It's not clear if formal techniques would generate any additional confidence in the design
- ◆ Modular refinement is extremely important in implementations
  - Formal techniques can be of significant value

## Example 4:

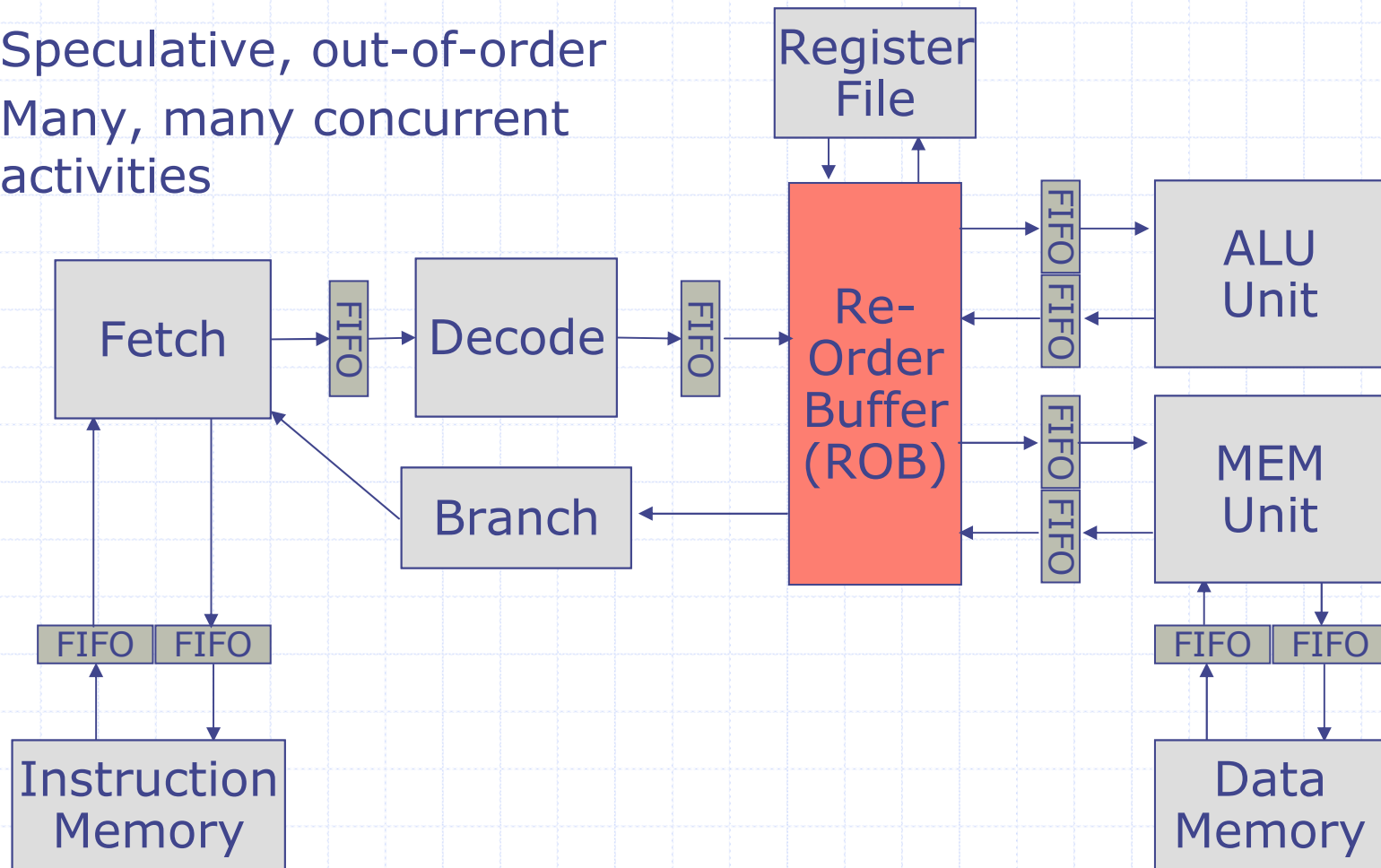
# Microprocessor design

Precise specifications are available and absolute correctness is required

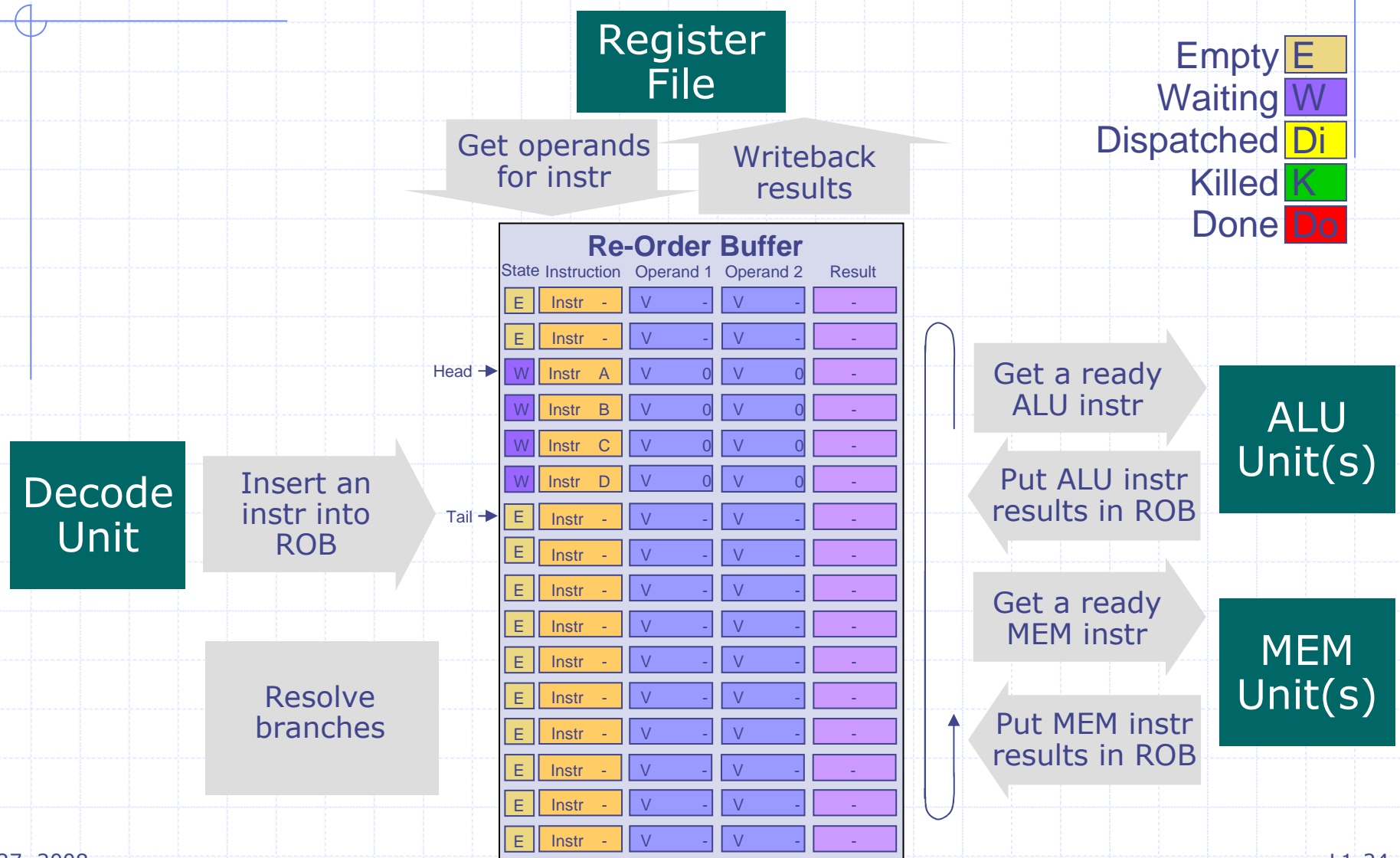
The proving ground for all verification techniques

# Microarchitecture of a modern CPU

- ◆ Speculative, out-of-order
- ◆ Many, many concurrent activities



# ROB actions



# But, what about all the potential race conditions?

- ◆ Reading from the register file at the same time a separate instruction is writing back to the same location
  - Which value to read?

- ◆ An instruction is being inserted into the ROB

simu  
inst

## Rule Atomicity

- ◆ Lets you code each operation in isolation
- ◆ Eliminates the nightmare of race conditions (“inconsistent state”) under such complex concurrency conditions

- ◆ An i  
simu  
mus  
rest  
mod

*All behaviors are explainable as a sequence of atomic actions on the state*

# USING TERM REWRITING SYSTEMS TO DESIGN AND VERIFY PROCESSORS

THE OPERATIONAL SEMANTICS OF A SIMPLE RISC INSTRUCTION SET SERVE AS AN ILLUSTRATION IN THIS NOVEL USE OF TERM REWRITING SYSTEMS TO DESCRIBE MICROARCHITECTURES.

Arvind and  
Xiaowei Shen  
Massachusetts Institute  
of Technology

..... Term rewriting systems (TRSs) offer a convenient way to describe parallel and asynchronous systems and prove an implementation's correctness with respect to a specification. TRS descriptions, augmented with proper information about the system building blocks, also hold the promise of high-level synthesis. High-level architectural descriptions that are both automatically synthesizable and verifiable would permit architectural exploration at a fraction of the time and cost required by current commercial tools.

In recent years, considerable attention has focused on formal verification of microprocessors.<sup>1-4</sup> Other formal techniques, such as Lamport's Temporal Logic of Actions and Lynch's I/O automata, also enable us to model microprocessors. While all these techniques have something in common with TRSs, we find the use of TRSs more intuitive in both architecture descriptions and correctness proofs. TRSs can describe both deterministic and nondeterministic computations. Although they have been used extensively in programming language research to give operational semantics, their use in architectural descriptions is novel.

In this article, we use TRSs to describe a speculative processor capable of register renaming and out-of-order execution. We lack space

to discuss a synthesis procedure from TRSs or to provide the details needed to make automatic synthesis feasible. Nevertheless, we show that our speculative processor produces the same set of behaviors as a simple nonpipelined implementation. Our descriptions of microarchitectures are more precise than those found in modern textbooks.<sup>5</sup> The clarity of these descriptions lets us study the impact of features such as write buffers or caches, especially in multiprocessor systems.<sup>6,7</sup> In fact, experience in teaching computer architectures partially motivated this work.

## Term rewriting systems

A term rewriting system is defined as a tuple  $(S, R, S_0)$ , where  $S$  is a set of terms,  $R$  is a set of rewriting rules, and  $S_0$  is a set of initial terms ( $S_0 \subseteq S$ ). The state of a system is represented as a TRS term, while the state transitions are represented as TRS rules. The general structure of rewriting rules is

$$s_1 \text{ if } p(s_1) \\ \rightarrow s_2$$

where  $s_1$  and  $s_2$  are terms, and  $p$  is a predicate.

We can use a rule to rewrite a term if the rule's left-hand-side pattern matches the term or one of its subterms and the corresponding

◆ Used 14 rules to describe an OOO processor and proved the correctness of the system with respect to the ISA



# Reactions (1997-98)

## ◆ IBM, Intel

- Very interesting but engineers don't prove theorems
- Maximum money is spent on tools that work on implementation descriptions as opposed to models

## ◆ Academics

- Not automated enough, no interesting decision procedures



# “Automated” Processor Verification

- ◆ Models are abstracted from (real) designs
  - UCLID – Bryant (CMU) : OOO Processor hand translated into CLU logic (synthetic)
  - Cadence SMV - McMillian : Tomasulo Algorithm (hand written model. synthetic)
  - ACL – Jay Moore: (Translate into Lisp)
  - ...
- ◆ Some property of the manually abstracted model is verified
  - Great emphasis (and progress) on automated decision procedures

**Since abstraction is not automated it is not clear what is being verified!**

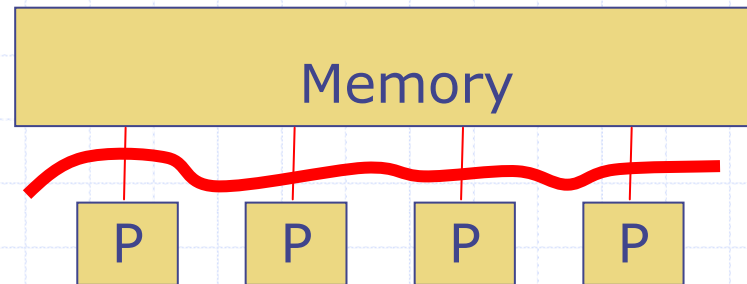
BAT[Manolios et al] is a move in the right direction

## Example 5: Cache Coherence

Dealing with nondeterministic specifications

Absolute correctness is required

# Cache Coherence Protocols



- ◆ A memory model defines what a Load instruction can return from a possible set of Store values
- ◆ Cache coherence (CC) protocols are designed to preserve a memory model in the presence of caches
  - An efficient CC protocol is difficult to design; done by experts only
    - ◆ Easy to make mistakes even in abstract protocols because of nondeterminism
  - Lot of room to make mistakes even during the implementation of a correct protocol

# A personal anecdote

- ◆ My student Xiaowei Shen designed an adaptive protocol called Caché
  - Very difficult for me to understand and be sure of its correctness
  - We used TRS to give a precise definition and TLAs to prove its correctness

*This was not enough ...*

# Realization

- ◆ The proof was so long that we could have made a mistake easily
- ◆ Nobody else was going to read the proof – “it is not interesting”

Even though the protocol correctness is of extreme importance the burden of its correctness is solely on its designers.

*Different from a math theorem*

*Mechanical theorem proving*

## Proofs of Correctness of Cache-Coherence Protocols

Joseph Stoy<sup>1</sup>, Xiaowei Shen<sup>2</sup>, and Arvind<sup>2</sup>

<sup>1</sup> Oxford University Computing Laboratory  
Oxford OX1 3QD, England  
Joe.Stoy@comlab.ox.ac.uk

<sup>2</sup> Laboratory for Computer Science  
Massachusetts Institute of Technology  
Cambridge MA 02139, USA  
xwshen, arvind@lcs.mit.edu

**Abstract.** We describe two proofs of correctness for Cachet, an adaptive cache-coherence protocol. Each proof demonstrates soundness (conformance to an abstract cache memory model CRF) and liveness. One proof is manual, based on a term-rewriting system definition; the other is machine-assisted, based on a TLA formulation and using PVS. A two-stage presentation of the protocol simplifies the treatment of soundness, in the design and in the proofs, by separating all liveness concerns. The TLA formulation demands precision about what aspects of the system's behavior are observable, bringing complication to some parts which were trivial in the manual proof. Handing a completed design over for independent verification is unlikely to be successful: the prover requires detailed insight into the design, and the designer must keep correctness concerns at the forefront of the design process.

### 1 Introduction: Memory Models and Protocols

Shared memory multiprocessor systems provide a global memory image so that processors running parallel programs can exchange information and synchronize with one another by accessing shared variables. In large-scale systems the physical memory is usually distributed across different sites to achieve better performance. Distributed Shared Memory (DSM) systems implement the shared memory abstraction with a large number of processors connected by a network, combining the scalability of network-based architectures with the convenience of shared memory programming. The technique known as caching allows shared variables to be replicated in multiple sites simultaneously to reduce memory access latency. DSM systems rely on cache-coherence protocols to ensure that each processor can observe the semantic effect of memory access operations performed by another processor.

A shared memory system implements a *memory model*, which defines the semantics of memory access instructions. An ideal memory model should allow efficient and scalable implementations while still having simple semantics

◆ It took Joe Stoy more than 6 months to learn PVS and show that some of the proofs in Xiaowei Shen's thesis were correct

This technology is not ready for design engineers

# Model Checking

- ◆ CC is one of the most popular applications of model checking
- ◆ The abstract protocol needs to be abstracted more to avoid state explosion
  - For example, only 3 CPUs, 2 addresses
- ◆ There is a separate burden of proof why the abstraction is correct
- ◆ Nevertheless model checking is a very useful debugging aid for the verification of abstract CC protocols

# Implementation

- ◆ Design is expressed in some notation which is NOT used directly to generate an implementation
  - The problem of verification of the actual protocol remains formidable
  - Testing cannot uncover all bugs because of the huge non-deterministic space

Proving the correctness of cache coherence protocol implementations remains a challenging problem

# Collective insights from these examples

- ◆ In some applications performance is paramount, even at the expense of some errors
  - 802.11, H.264
- ◆ In others, correctness is non-negotiable and performance is a secondary concern
  - IP look up, processors, caches
- ◆ Some properties are not easy to express formally but may be expressible by adding state to the design
  - Are packets processed in order?
- ◆ For some applications, specifications have to incorporate non-determinism
  - CC, speculative processors

Simulation and testing are here to stay *BUT*  
for commonly used refinements and transformations  
formal methods are preferable

# Some common threads in current formal verification

- ◆ A manually abstracted model, as opposed to the real design (implementation) is verified
- ◆ Great emphasis on automated decision procedures but not on automated model abstraction
- ◆ Stating properties often has no direct benefit for the designer

At the end of the day it is not clear if the formal verification of the model has any bearing on the verification of the real design

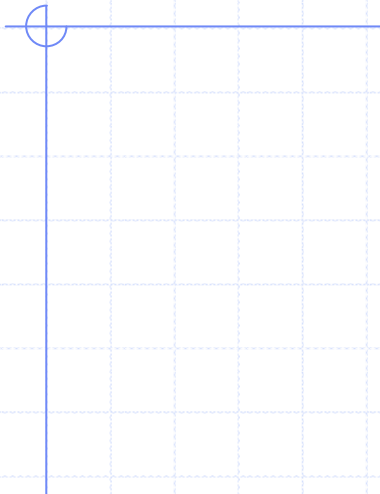
*Back to testing and debugging!*

# What is needed

- ◆ High-level notation
  - with precise semantics
  - capable of expressing nondeterminism
  - amenable to synthesis of actual implementation
- ◆ Powerful tools for proving properties of such designs

Automatic extraction of abstract models from designs expressed in Verilog or C or SystemC is a lost cause

*Thanks!*

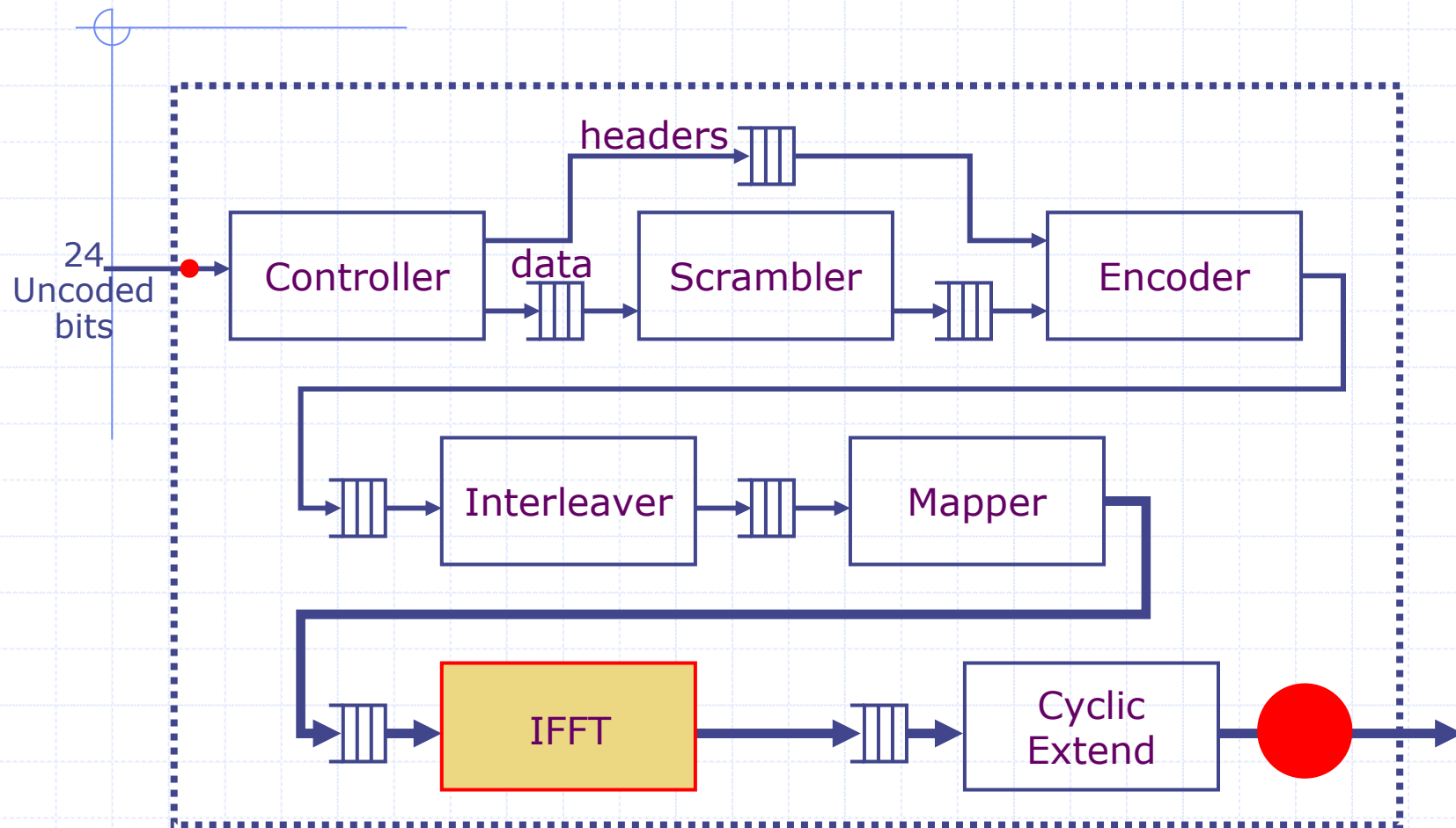




Example 2:  
**An 802.11a Transceiver**

Dealing with noise

# 802.11a Transmitter



 accounts for 85% area

Must produce one OFDM symbol  
(64 Complex Numbers) every 4  $\mu$ sec

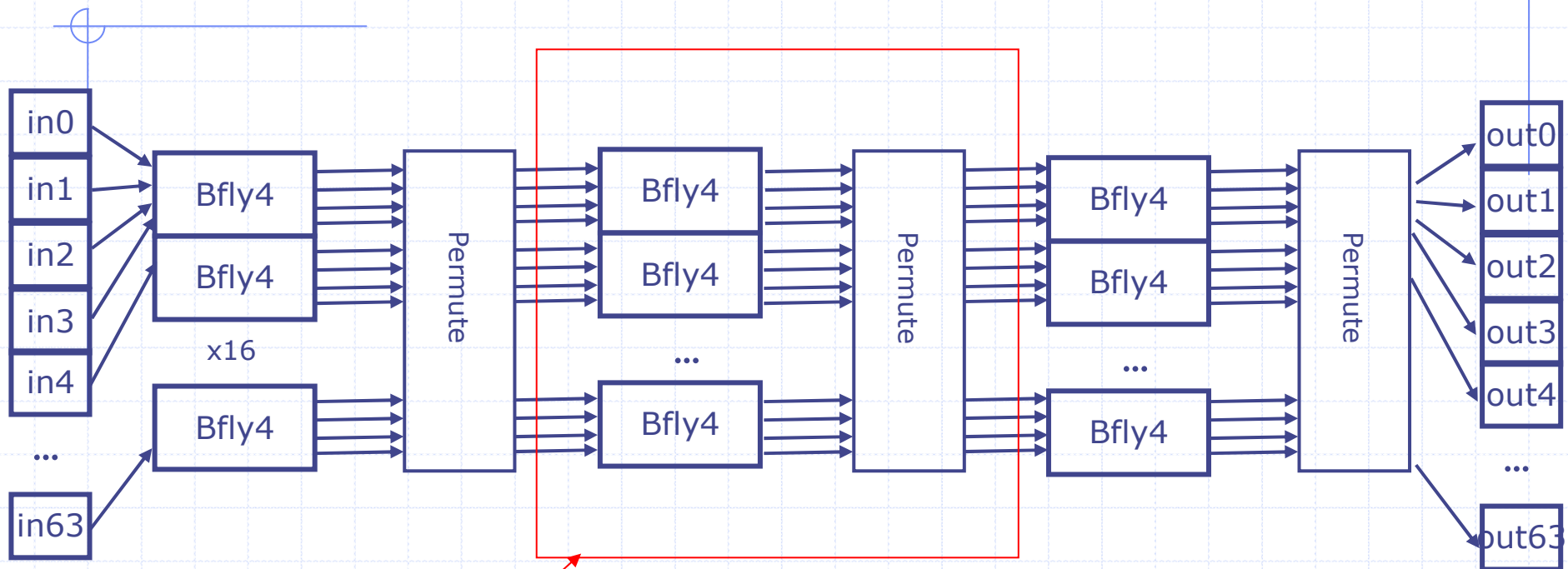
# Verification Issues

## Control is straightforward

- Small amounts of testing against the C code is sufficient, provided the arithmetic is implemented correctly
  - ◆ C code may have to be instrumented to capture the intermediate values in the FIFOs
- No corner cases in the computation in various blocks
  - ◆ High-confidence with a few correct packets

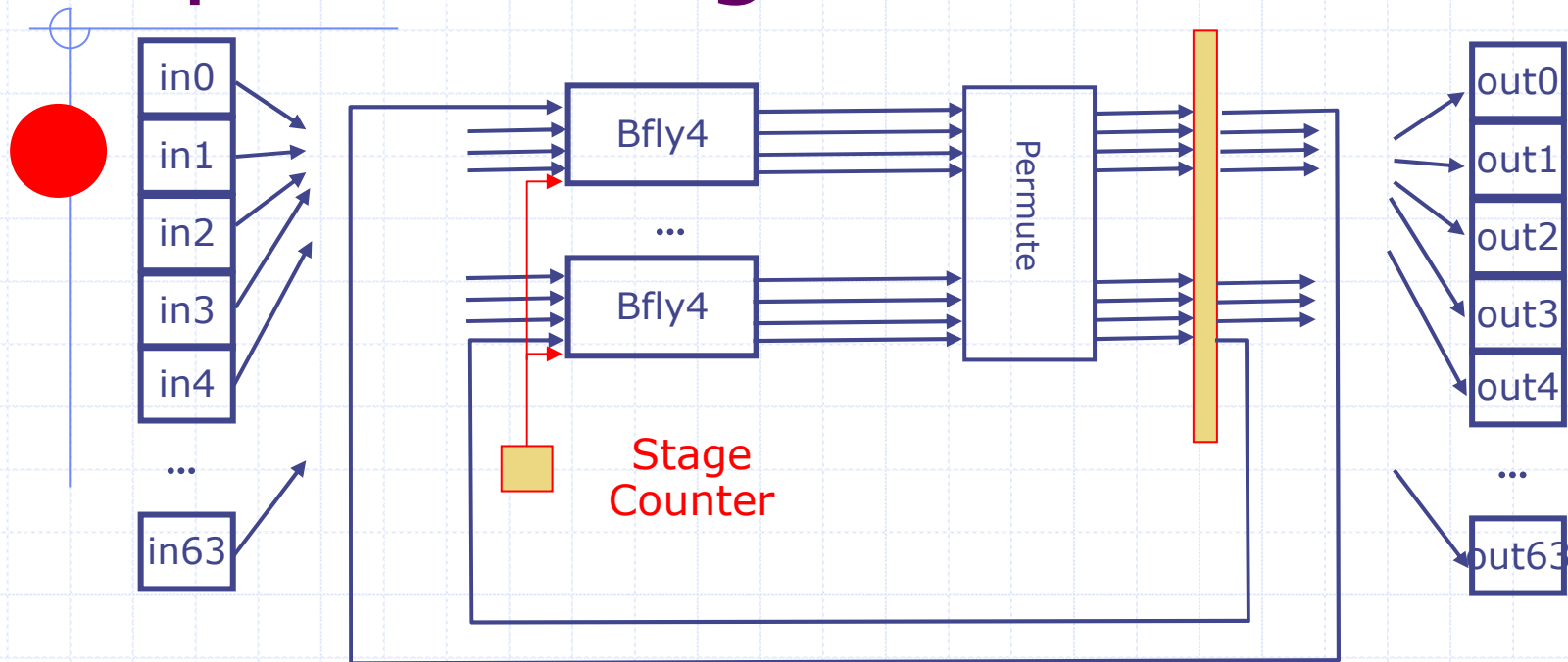
Still may be worthwhile proving that the (non standard) arithmetic library is implemented correctly

# Modular refinement IFFT

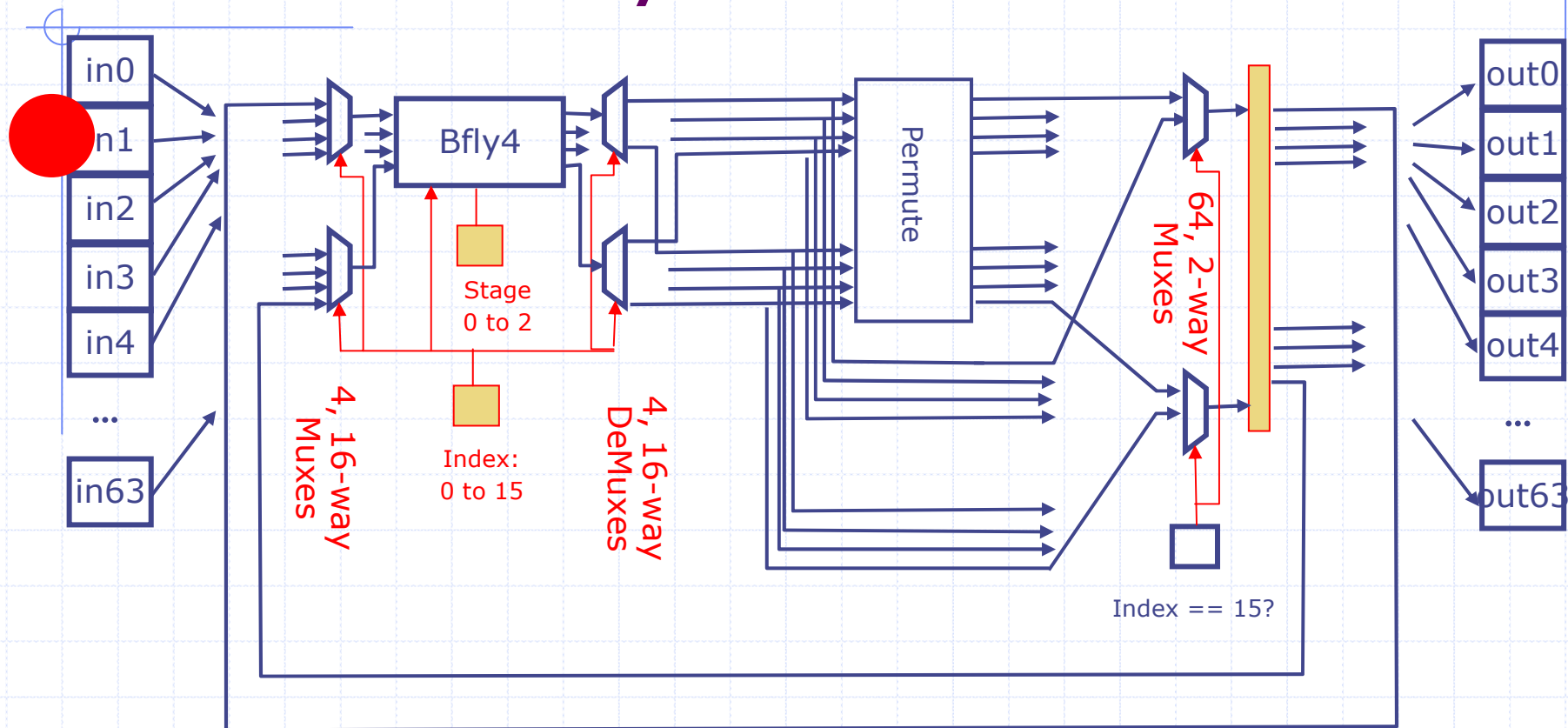


Reuse the same circuit three times  
to reduce area

# Circular pipeline: Reusing the Pipeline Stage



# Superfolded circular pipeline: Just one Bfly-4 node!



# Modular Refinement - IFFT

- ◆ When the designer changes the IFFT module, does the whole design have to be re-verified?
  - Unit verification of the refined module should be sufficient provided the designer uses an appropriate design style
- ◆ Performance and area refinements in such algorithmic blocks are often based on standard transformations for parallelism and reuse
  - Formal verification of these transformations can almost completely eliminate the need for unit testing for such refinements

*"correct-by-construction"*

802.11a transceiver:

## Higher-level correctness

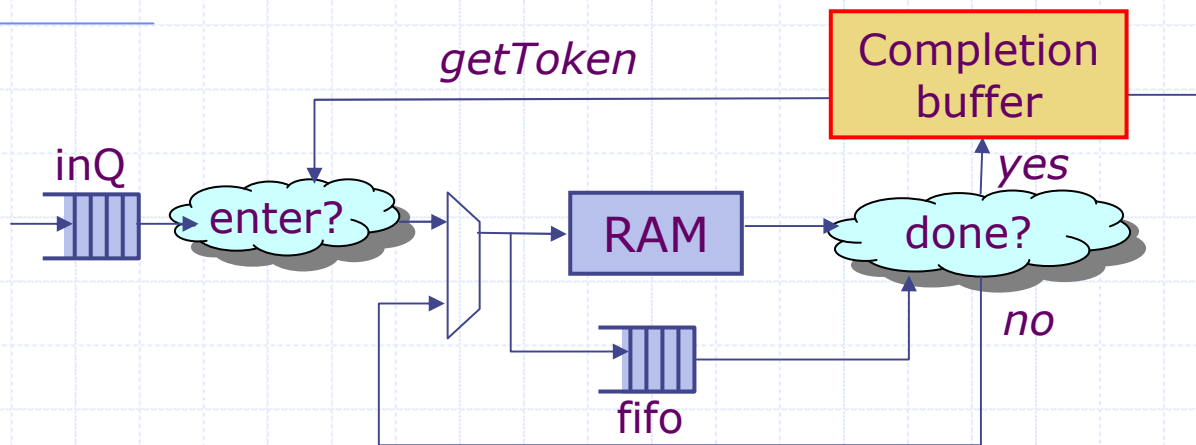
- ◆ Does the receiver actually recover the full class of corrupted packets as defined in the standard?
  - Designers totally ignore this issue
  - This incorrectness is likely to have no impact on sales

*Who would know?*

- ◆ If we really wanted to test for this, we could do it by generating the maximally-correctable corrupted traffic

***All these are purely academic questions!***

# A performance issue



- ◆ Are there any dead-cycles, i.e., can a new message enter the system in the cycle in which an old message leaves the system?
  - The system will slow down more than 25% if there is a dead cycle.
  - Formal verification?

**Most performance concerns eventually become correctness concerns**